The Physical Design of the CDF Simulation and Reconstruction Software

E. Sexton-Kennedy[1], *M. Shapiro*[2], *R. Snider*[1], *R. Kennedy*[1]

[1] Fermi National Accelerator Laboratory,USA
[2] Lawrence Berkeley National Laboratory,USA

Abstract

A good physical design is the only way to manage the complexity of a large software system. CDF's simulation and reconstruction software system is large, with about 1.3 million lines of code organized into 148 different packages in 4 different programming languages. The simulation and reconstruction applications must use a large fraction of this code in any one job. We have managed to develop guidelines for physicist-written code which keep the physical design manageable. Nevertheless we must monitor and correct mistakes in physical design daily. Proof of the success of this effort is that we are able to link all CDF offline applications statically in one pass, with no CDF libraries listed more then once. This was not possible in the Run I code system, in which users spent much of their time trying to understand how to link applications. In Run II, this process is largely automated.

Keywords: physical design, guidelines, CDF

1 Introduction

The organization and physical design of CDF's simulation and reconstruction software is largely driven by its architecture. CDF's framework package called AC++ [1] together with our Event Data Model [2] and Database system [3]. form the basis of this architecture. While it is not necessary to understand these in detail, a brief overview of them would be helpful. The smallest unit of reconstruction is a *Module*. A *Module* is a class inheriting from a specific abstract base class, AppModule. It has a minimal public interface whose members are only called by the framework and represent the different phases or states of event processing. End users then piece these units of reconstruction together to form their processing jobs. Communication between these modules can only be done with objects defined in our event data model. This requirement is essential for being able to do component testing. In this paper I will first discuss what physical design is, and then describe CDF's. I will then elaborate on the rules we use to maintain this design and what we have achieved so far.

2 Logical vs. Physical Design

Logical design deals with language constructs. It is concerned with classes, inheritance, types of data members, their visibility, and related concepts. Physical design addresses the issues involving files, directories, and libraries as well as the compile and link time dependencies between them. It answers the organizational questions like, “What files define a class?”, “Where should those files be placed?”. As systems grow larger, attention to physical design becomes critical. Classes can be physically coupled in varying degrees. There is full coupling when one class must include the header file of another class, for instance when one class inherits from another. There is also name-only coupling when a class needs only to declare the name of the other class with a forward declaration such as “class YourClass;”. Name-only coupling can be use when one class just holds a pointer to another class. From the preceding examples you can see that some logical design choices cause different levels of physical coupling. Because of this, physical design constraints can cause a logical redesign. This happens quite often at CDF. [4]

3 CDF’s System Organization

The classes which make up the CDF software system (about 70% of it) are organized into packages. These packages are layered into a hierarchical tree of dependencies. All Fortran and C code in the system (about 30 % of it) must be used by some class. For the most part this code is grouped together with the C++ code that uses it. The exceptions are at the very lowest level of the physical design hierarchy, where whole packages like readline or tcl are kept intact. The packages themselves can be grouped into categories described in the following subsections. It is critical that all of our developers be aware of these categories so that they know how to structure their code, and place it in the appropriate package. Without this the hierarchy would break down into a mire of cyclic dependencies. The following subsections are ordered such that packages at top of the hierarchy are listed first.

3.1 Binary Packages

There are now 7 of these packages whose main purpose is to provide a binary, such as the Production, visualization and test packages. Binaries that use the CDF framework are specified when a user writes a class whose only purpose is to “new” the *Modules* and *EDM Objects* that will be used in the job. This class is in turn created at the very beginning of “main” right after the framework itself is created. It is linked in as an object file and thus forms the top of the linking hierarchy tree.

3.2 Module Packages

There are 24 module packages. Some concentrate on the event reconstruction or simulation of a specific detector and others contain the reconstruction modules for physics objects like jets or electrons. These packages can not depend on each other and the only class that ever uses them is the framework itself. They in turn may use any object that is lower down in the physical hierarchy.

3.3 Algorithm Packages

There are 33 algorithm categories. These are also organized into detector and physics packages. They are used by the module packages and may use each other as long as the dependencies do not become cyclic. In general however an algorithm object should be able to use Data Object classes from the event and Database systems in order to accomplish its task.

3.4 Data Object Packages

There are 17 of these packages. Half of these objects can be passed between modules in the event and can be made persistent with ROOT [7] I/O. The other half contain geometry data objects. The infrastructure to make these objects persistent is still being developed.

3.5 Interface Packages

There are 12 packages which provide CDF-compatible interfaces to the externally developed packages which CDF uses. These include physics event generators, geant, the error logger, etc. Interfaces should only depend on the package they are an interface to and the CDF infrastructure packages described below.

3.6 Infrastructure Packages

The framework, data model, and database systems are organized into 29 different packages. These packages may use each other as long as their dependencies do not become cyclic.

3.7 Standard Utilities and Services Packages

These package live at the bottom of the physical hierarchy because they are all developed and supported outside of CDF. They include the generator packages, Zoom/CLHEP packages [5], ROOT, tcl ect. We have 26 packages in this category.

4 Rules and Guidelines

The two most important design rules for keeping the physical design clean are, "packages may not cyclically depend on each other", and "try to use name-only dependencies wherever possible". Some rules must be enforced just to be able to find components is such a large system. CDF however has also found the use of a cross-referenced code browser invaluable for this purpose. We use LXR based code browser [6] which has allowed us to be more lenient with our developers. Some rules however must be strictly followed. The following is a list of them.

- Keep class data members private.
- Avoid global data, and when possible avoid class static data.
- Avoid using preprocessor macros in header files.
- Use predictable include guards around the contents of each header file.
- Do not use "using" declarations in header files.
- Enumerations, typedefs and constants should be defined within a class scope.
- Only classes and inline functions should be defined in header files.
- Each header and source file pair should define one class, or a few closely related classes.
- If you fully depend on a class use its header file, not a local definition.
- Follow our naming conventions for files and language components.
- Do not use any C-style casts.
- Do not use any conditional compilation clauses that change the size of an object.
- Use of any 3rd party package in CDF offline software must be approved
- Class documentation should appear in the header, and must include the author's name.

A large part of our code base has not been peer-reviewed. As such, we can not say that 100% of our code follows these rules. Whenever a piece of code is reviewed however, it is usually partially redesigned and re-implemented in part to follow these rules. We are still in a rapid development phase so any mistakes that are made must be corrected immediately before they propagate. For this reason the integration leaders pay attention to every check-in message that enters the repository. Usually one can tell from the names of the classes and the names of the packages when a physical design mistake is being made. Fixing these physical design mistakes can be painful because all references to the incorrectly placed class must be changed. For this reason we have written scripts which use "find" and "sed" to examine and change include file references all over the system all at once.

5 Goals That Have Been Achieved

CDF has so far been able to avoid any cyclic package dependencies. All packages fit into a fixed hierarchy that we can capture in a makefile fragment. This means that we are able to link all CDF offline applications statically in one pass, with no CDF libraries listed more than once. Because the ordering is fixed the software is much easier for our end users to link their own jobs. With 148 different packages it is important that our users do not have to determine their own link lists. Since it is hierarchical packages only need to know their immediate dependencies. This situation is a large improvement over our Run I offline system. In that system we had fewer packages but they were poorly organized. There were many cyclic dependencies causing different jobs to have to be linked with differently ordered lists, where some libraries were listed multiple times. In Run I we attempted to create dynamically loadable shared libraries, this failed due to the physical design of that system. In Run II we expect to be able to use shared libraries in frozen releases.

6 Conclusions

Despite the large size of the CDF software system we have managed to develop a reasonable physical design for it. The design is simple enough that most of our developers understand it and help us maintain it. Most of our core developers do not knowingly violate our rules and guidelines listed section 4. The success we have had has already benefited some of our early physics users.

References

1 E. Sexton-Kennedy, “A User’s Guide to the AC++ Framework”, CDF Note 4178, see: `http://www-cdf.fnal.gov/upgrades/computing/projects/framework/`.

2 R. Kennedy, “The CDF Run II Event Data Model”, CHEP’00 presentation C201, Padova, Feb 2000.

3 J. Kowalkowski, “Mapping Auxiliary Persistent Data to C++ Objects in the CDF Reconstruction Framework”, CHEP’00 presentation C236, Padova, Feb 2000.

4 J. Lakos, “Large-Scale C++ Software Design”, Addison-Wesley, 1996

5 M. Fischler et al.,“The Fermilab ZOOM Class Libraries ”, CHEP’98 presentation 64, Chicago, Autumn 1998.

6 The CDF Run 2 Software, `http://purdue-cdf.fnal.gov/CdfCode/`

7 Root home page, `http://root.cern.ch/`