The CDF Run II Event Data Model

R. D. Kennedy[2], *D. Amidei*[4], *E. Buckley-Geer*[2], *P. Calafiura*[3], *C. Green*[6], *T. Huffman*[5], *J. Kowalkowski*[2], *A. Lee*[1], *K. McFarland*[7], *P. Murat*[2], *S. Rolli*[8], *E. Sexton-Kennedy*[2], *M. Shapiro*[3], *R. Snider*[2], *P. Tamburello*[1]

[1] Duke University, USA
[2] Fermi National Accelerator Laboratory, USA
[3] Lawrence Berkeley National Laboratory, USA
[4] University of Michigan, USA
[5] Oxford University, UK
[6] Purdue University, USA
[7] Rockefeller University, USA
[8] Tufts University, USA

Abstract

The CDF Event Data Model describes how CDF C++ and Fortran-77 software may access event data in CDF Off-line software. The CDF Event Data Model Working Group has reviewed the Run I Event Data Model which was based on YBOS data banks stored in a global array. While this model was successful for Run I analyses, it did not take advantage of the Object-Oriented design of the CDF Run II Off-line software.

In this paper, we describe the CDF Run II Event Data Model. Event data is passed via an event record amongst user-written software modules whose execution is coordinated by the AC++/Framework package. C++ classes meeting a few criteria can have instances stored in the event record. Objects in the event record are assigned a unique identifier and become write-locked, key changes in behavior from the old model. Support for storable links, collections, YBOS banks, and data access by Fortran-77 software are accommodated. The ROOT Object I/O system is used to read/write the event record to disk files. This model has simplified the effort to make event data persistent, without requiring an overwhelming effort to retrofit our large, existing code base written to the old event data model.

Keywords: CDF, event, data, model, ROOT

1 Introduction

At CDF, event data is defined to be data read from detectors as well as calculated objects based on that readout which represent the state of the physics event. This does not include calibration data, alignment data, or the detector geometry description. The CDF Event Data Model determines how individual elements of event data are accessed, how the event is passed from software component to software component, and how events are recorded in disk files or memory buffers for the Off-line, On-line, and Simulation software systems.

Event data analysis jobs at CDF are constructed from software modules whose execution is directed by Framework[1], a software package co-maintained by BaBar and CDF [3]. The repository of data files is managed by the CDF Data Handling system [6, 7]. Users specify to Framework which data files are to be used by their analysis job. Framework forwards these requests to the Data Handling system which in turn stages input files, or reserves space for output files, on a disk accessible to the analysis job. The Event Data Model implementation manages the reading and writing of events and access to the objects within an event.

The CDF Event Data Model Working Group was formed in November 1998 to review existing practice and propose a Run II Event Data Model [1]. CDF had already chosen to use the

[1]For historical reasons, Framework is often called AC++ at CDF.

ROOT[2] package's Object I/O system[4, 5] to implement our event data I/O. Most CDF software modules had been or were soon to be rewritten in C++, but they used a C++-based YBOS implementation called Trybos [2] for event data access. Modules written in Fortran-77 and adapted to the Run II Framework had to continue to be supported in order to validate newly written C++ modules. A large fraction of our developers efforts were devoted to maintaining distinct transient and persistent versions of the their object classes, due to the limited nature of the YBOS format. While we sought to support a broader variety of persistent objects consistent with using ROOT for event data I/O, we also wished to minimize the effort required of our developers to adapt our existing software to use a new Event Data Model.

2 Event Records and Storable Objects

In the CDF Run II Event Data Model, all event data is passed from module to module via an instance of the class *EventRecord*. Event data may not be passed via singletons, global variables, or common blocks. This insures that, at any module boundary, we can completely and easily capture the state of the event record. In fact, this aspect of the Event Data Model is not new, but is now enforced with greater vigilance. Event data can only be read from and written to disk files and memory buffers by special Framework modules called I/O modules.

Any class which derives from the class *StorableObject* and meets a few criteria can be appended to the event record. The class must implement a ROOT Streamer() method which directs how the object's state is converted to or from a byte stream which can then be saved to or restored from a disk file or memory buffer. A class's header file must be parsable at some level by ROOT's rootcint utility, and the class must use the ROOT preprocessor macros ClassDef and ClassImp, in order to generate the remainder of the I/O support. Storable objects are allocated on the heap and manipulated through an instance of a *Handle* class. Handles act like smart pointer classes, allowing the functionality of the underlying class to be accessed by using the C++ operator $->()$, but avoiding a time-consuming data copy when appending large objects to the event record.

Once appended to the event record, a storable object is assigned an object identifier by the event record which is guaranteed to be unique throughout the lifetime of that event record. Objects, once stored in the event record, become write-locked and owned by the event. The write-capable handle passed to the event record is nullified, and a *ConstHandle* instance is returned to the user. Users cannot modify or erase objects in the event record. Users can search for objects by common attributes, such as class name, object id, creator module, or a user-defined description string. Since each storable object knows which module created it and under what conditions the module was executed, the processing history of each object is well-defined.

Rather than erasing objects from the event record, users may specify which objects to output or not to output to persistent media. A list of objects read in the event is used to initialize a list of objects to write out. Users may add class names or object identifiers to a "keep" list or to a "drop" list. All objects referred to by the keep list and not by the drop list are output. Thus, we encourage all modules to fully label the objects they produce so that the objects can be distinguished by some other means than mere existence in the event, improving the reproducibility of results.

3 Other Components

Persistent associations are implemented by a *Link* class. Each link contains both a transient pointer to the associated object, as well as its object identifier. Links use this object identifier to save their state. After all objects in an event have been read in, each object has a postread() method called

[2]`http://root.cern.ch`

during which it may choose to restore the transient pointer in a link data member by searching the event record for that object identifier. Because of the object identifier uniqueness, an object associated with another object instance will always be associated with that exact instance. It will not become associated with a modified or relabeled instance.

A small suite of storable homogeneous containers classes have been developed to simplify the creation and storage of object collections. Each is a template class which is not directly storable in the event due to rootcint's inability to treat template classes. These are examples of *StreamableObjects*, classes which provide a Streamer() method, but cannot be parsed by rootcint. A streamable object can be data members of a storable object, and thus can indirectly be stored in an event record. In this case, a storable object class must contain a particular instantiation of the template container class. Container classes are provided for storage by value, by reference with object ownership, and by reference without object ownership, as well as for vector and list-based containers. Users are permitted access to the underlying Standard Template Library data structure, allowing Standard Library algorithms to be used on those data structures.

In order to minimize the effort to adapt bank-oriented algorithms to the new Event Data Model, and enable continued access to YBOS format data files, utilities and I/O modules have been developed to transform YBOS banks to and from instances of classes derived from the base class *StorableBank*. There is a class derived from StorableBank for each bank name present in the Off-line system, as the bank name determines the data format stored in the bank as well as the interface to the data in the bank.

We chose as our basic approach to saving and restoring events in ROOT Object I/O system a simple "sequential" event model. The entire event is stored in one *TBranch* in one *TTree*. We find this adequate for our immediate needs since both the data acquisition system and our event reconstruction processing will need to read in all objects in every event. In addition we have created the means to store events in memory buffers using the Streamer() methods for on-line applications. We expect to make use of multi-branch events in secondary processing where use-cases justify the added organizational overhead to minimized data access time.

4 Fortran-77 Module Support

Fortran-77 modules are still supported in the CDF Run II Event Data Model, though with some constraints. At the beginning of execution of a properly adapted Fortran-77 module, all storable banks in the primary event are transformed into a secondary event, a contiguous array containing YBOS banks. Storable banks have been designed to mimic the YBOS format in their type description and data sections, reducing the amount of cpu-time required to perform this transformation. At the end of a properly adapted Fortran-77 module, all YBOS banks in the secondary event are transformed back into storable banks in the primary event. While this approach, shown in Figure 1, compromises a few tenets of the new Event Data Model, such as write-locking of objects within the event, our experience has shown that these compromises are tolerable. Those Fortran-77 modules still in use today tend to be somewhat restricted in their interaction with the event record, and do not seriously violate the goals of the new Event Data Model.

5 Status

We have completed the core implementation of the Run II Event Data Model, and ported all CDF Off-line Event Reconstruction modules to be compatible with the model. At the time of this writing, the first CDF Run II Mock Data Challenge is underway, successfully feeding simulated events through the high-level on-line triggering system to the event reconstruction farms, splitting

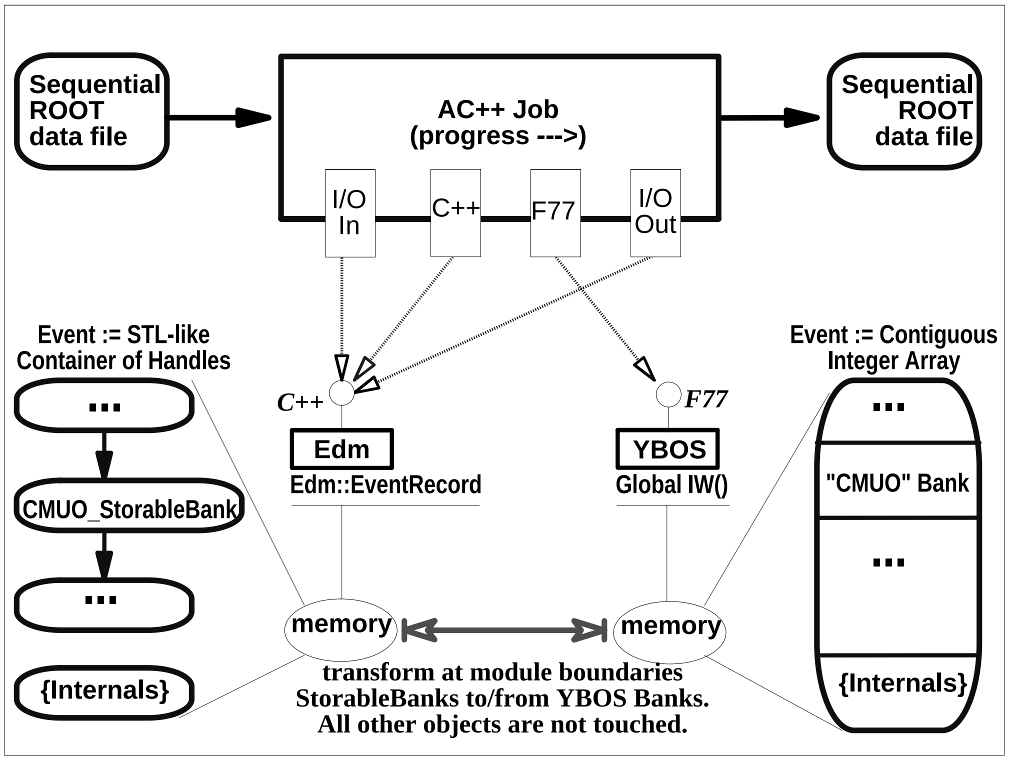
Sequential ROOT data file
AC++ Job (progress --->)
Sequential ROOT data file
I/O In
C++
F77
I/O Out
Event := STL-like Container of Handles
Event := Contiguous Integer Array
C++
F77
Edm
Edm::EventRecord
YBOS
Global IW()
...
CMUO_StorableBank
...
{Internals}
...
"CMUO" Bank
...
{Internals}
memory
memory
transform at module boundaries StorableBanks to/from YBOS Banks. All other objects are not touched.

Figure 1: Support for C++ and F77 modules in CDF Run II Event Data Model

the data into analyzable physics sub-samples. We have managed to do so without undergoing an overwhelming and risky upheaval in CDF Off-line software, with most algorithms being re-used or easily adapted to the new model, and with existing YBOS data files remaining usable. Having reached this milestone, we will begin to optimize our implementation of sequential event I/O using ROOT, adapt to a newly-arrived package which records creating modules and related conditions for each object, consider more expressive means to search for objects within the event, and improve user documentation and tutorials.

References

1 R.D. Kennedy, “The CDF Run II Event Data Model”, CDF Internal Note 4819, November 1998, `http://www-cdf.fnal.gov/upgrades/computing/projects/edm/edm.html`.

2 R.D. Kennedy, “The Trybos Project Users’s Guide”, CDF Internal Note 4174, July 1997.

3 E. Sexton-Kennedy, “A User’s Guide to the AC++ Framework”, CDF Internal Note 4178, May 1997.

4 J. Patrick, et al., “Report of the CDF Data Handling Technical Review Committee”, CDF Internal Note 5215, May 1998.

5 R.D. Kennedy and P. Murat, “Use of ROOT I/O in the CDF Run II Data Handling System”, CHEP’98 presentation 153, Chicago, September 1998.

6 S. Lammel, et al., “Overview of the CDF Run II Data Handling System”, CHEP 2000 presentation 366, Padova, February 2000.

7 P. Calafiura, et al., “The CDF Run II Data Catalog and Data Access Modules”, CHEP 2000 presentation 367, Padova, February 2000.