The CDF Run 2 Event Data Model

Von Ruden Review
07 June 1999
Rob Kennedy

- Definition of "Event Data", the CDF Run 1 Event Data Model
- Goals of the Revised CDF Event Data Model
- Overview of the CDF Run 2 EDM
- Sample Use-Cases, Examples of Implementing Object Storage
- Current Status and Future Plans
- http://www-cdf.fnal.gov/upgrades/computing/projects/edm/edm.html

CDF Run 1 Event Data Model

Event Data := the data in a physics event record which is passed from algorithm module to module and to/from permanent storage media. No calibration data, geometry, module parameters, etc.

CDF Run 1 Event Data Model := Data fields stored in YBOS banks. Banks labelled by 4 char name and a number assigned by the user. Event record was global array used to store all YBOS banks, internals, etc. YBOS maintained by CDF, few other users.

Common Usage: Get array index for first bank with name "MUON", use index plus offsets in F77 header file to read or write fields of the bank. Get array index to the next bank with same name.

Problems: Corruption of event record when writing with inappropriate offsets, change of an associated object without changing associated objects, user-defined bank numbers re-used, discomfort with using banks led to unpacking into common blocks....

☞ Model was a success. *But*, can we do better, especially with C++?

Goals of the CDF Run 2 EDM

- All data passed from algorithm module to algorithm module passed in the event record. No singletons/globals/commons allowed.

- Objects more flexible than YBOS banks can be stored/saved. One is not led to support both transient and persistent forms of objects.

- (Abstract goal) At any algorithm module boundary, we can store the state of the event record, then resume the process later. This must yield the same results as if the process had run to completion.

- Links/associations between objects should be robust. Object identity should be preserved throughout the life of an event record.

- Provide direct support for storable collections of objects in event.

- Abide by the CDF Run 2 Data Handling decision to use ROOT to implement Event I/O, beginning with sequential data files.

- Require as little change to existing C++ source as possible.

Overview of the CDF Run 2 EDM

➠ CDF’s draft 0 == D0’s Event Data Model / D0OM. Development overseen by EDM Working Group, w/reps from all facets of Offline.

➠ C++ classes derived from Edm::StorableObject can be passed in the event record. To be storable on permanent media, a class defines a Root "Streamer()" method (separates transient/persistent form).

➠ StorableObjects have ids for creating module, parameters used.

➠ StorableObjects are allocated via central memory mgmt classes (debugging, efficiency) and passed by reference using "Handles".

➠ When added to an event, objects are assigned a unique ObjectId, unique across the event record for the lifetime of the event record.

➠ Once in an event, objects are write-locked and cannot be deleted from the event until output is performed.

➠ Links (persistent pointers), StorableCollections are supported. Storable Banks (YBOS banks in ROOT format) also supported.

➠ Different File classes for ROOT event models (sequential, multi-branch, ...). These details not seen outside of I/O modules.

Sample Use-Cases....

Create object, set a data field (pseudo-C++):

```
Handle<Muon> h_muon(new Muon) ;  // Allocate muon, init handle
h_muon->set_pz(1.0) ;            // Can write into muon now
ConstHandle<Muon> ch_muon = p_record->append(h_muon) ;
// Put muon into event, non-const handle => 0, get const handle
```

Search of object in event record, read a data field

```
ConstIterator iter(ptr_to_event, "Muon") ;  // Find 1st Muon
ConstHandle<Muon> ch_muon2(iter) ;          // Create ConstHandle
string class_name = ch_muon2->class_name() ;// Query possible w/
Id     muon_oid   = ch_muon2->object_id() ; // any StorableObject
float  muon_pz    = ch_muon2->pz() ;     // *Muon*-specific query
```

Very simple Streamer() method (fields are built-in data types):

```
ToyMuon::Streamer(TBuffer& iobuffer)       // Stripped of base class support
{ if (iobuffer.IsReading())                // Stripped of schema evolution support
    { iobuffer >> _px ; iobuffer >> _py ;  // Stripped of error/sanity checks
      iobuffer >> _pz ; iobuffer >> _energy ;
    }
  else if (iobuffer.IsWriting())
    { iobuffer << _px ; iobuffer << _py ;
      iobuffer << _pz ; iobuffer << _energy ;
    }
}
```

Current Status

- Prototype Release: Complete EDM design and unoptimized implementations of all supporting elements of the EDM. Some "toy" versions of high-level physics objects implemented as tests and to demonstrate application of EDM. Only pieces missing now are the storable containers... work in progress (<1 FTE).

- ROOT event model: currently is a simple sequential collection of StorableObjects. I/O performance testing underway with that model (1 FTE). A multi-branch event model is about to be tested with efficient selective input/output, not yet in EDM package (1 FTE?).

- Improvements in ROOT: have been requested of the ROOT team to support the CDF Run 2 EDM (list on later slide). No show-stopper issues , but limited use of C++ namespaces. Some of these are being addressed in the current beta versions of ROOT, being tested in concert with EDM package (<1 FTE).

- Documentation: Soon after release, will provide updated docs....

Plans for Future

After the Prototype Release

- Begin evaluation of new EDM by Physics groups in coordination with the definition of high-level physics objects: Track, Muon, Electron, Jet, MissingEt.... Easy to code once agreement on content.

- With Physics groups: define a transition plan to agreed to production EDM. May require a "composite" YBOS/EDM2 record.

- With Physics groups: define Fortran <=> event data policy. Event data accessible to F77 in YBOS banks, via commons, via arg lists?

- Integrate use of new ROOT features appearing in v2.22, such as byte counts to detect Streamer() inconsistencies, namespaces, ...

- Develop more efficient and robust implementations of EDM package, including smarter handles/pointers, memory managers, ...

- Production Release: Expected in July, contains all of above plus doc on "getting started", "making the transition with existing code".

EDM Requests to ROOT Team

- C++ Namespaces: The ROOT I/O mechanism does not work with classes in user-defined namespaces, like Edm::Id. No estimate yet.

- Rootcint/Streamer Support for Stdlib Classes: STL, string, bitset. STL is already supported by rootcint, string support coming soon. This does not include interpretation of STL code, only the creation of Streamer methods by rootcint... defines standard storage layout.

- Improved Rootcint Robustness: rootcint fails to interpret some macro constructs, esp. complicated ones in Linux system headers.

- Byte Count Mechanism: Detects Streamer() inconsistencies on a class-by-class basis. Already implemented in beta ROOT. We request some hooks to pass errors up to our error reporting s/w.

- Yet to be Requested of ROOT Team: Improved CPU efficiency for sequential event model (are we using ROOT as efficiently as possible?). Hooks to permit better integration of ROOT "split-file" mode with CDF Data Handling system (we need to define).

The EDM and Data Processing

- *Caveat*: Plans for the complete data processing chain are not finalized. They are dependent on ROOT I/O performance tests and integration with the developing CDF Data Handling system. This is **only** one possible scenario under discussion.

- Raw data is read out into the Level 3 Trigger processes as YBOS banks (compact, efficient storage of raw data). These are translated into StorableBanks in the Level 3 reconstruction program's input module. That program fully uses the new EDM and outputs StorableObjects in a sequential ROOT event model.

- Production and on-line consumers receive this sequential ROOT data as input. The production executable uses the new EDM and outputs sequential ROOT events into one or more data streams.

- Secondary and tertiary data sets may see benefits to using the multi-branch or split-file ROOT events, depending on use patterns. Objects are collated on input/output into Branches per a mapping.

Benchmarking I/O with Run 2 EDM

- Goal 1: Define the amount of CPU required to read in or write out a GB of data (as measured inYBOS format, not ROOT format) in the sequential (and multi-branch) ROOT event model. This affects design and purchasing for both the Production Farm and the Level 3 Trigger Farm... so measurements on both IRIX and Linux, for a variety of event sizes, and for different data endian-ness.

- Goal 2: Understand the ROOT I/O model better by tuning the sequential ROOT event model to be as CPU efficient as possible relative to the F77 YBOS and C++ Trybos implementations.

Results Very Preliminary: No final numbers since tuning of model and its implementation in ROOT will continue to evolve. So far....

- ROOT read: appears slower than expected versus Trybos - YBOS tuned for sequential I/O, ROOT tuned for random access I/O? Or are we simply not using ROOT efficiently in sequential mode yet?

- ROOT write: about as efficient as Trybos, esp. with large events.