

# A Proposed EDM Upgrade Path

Marc Paterno  
CD Special Assignments, FNAL

June 7, 2000

## Abstract

This document contains a proposed path to upgrade the DØ Event Data Model (EDM). It tries to achieve two goals, to the degree which they are compatible. The first goal is to obtain an EDM that is more amenable to future upgrades and enhancements, and is more efficient; the second goal is to ease the users' transition from the current EDM to the improved EDM.

## 1 Introduction

During the meetings of the May 2000 DØ Infrastructure Week, I presented a few proposals for modification to the existing DØ Event Data Model (EDM). These talks were summarized in my talk **Updating RCP and the DØ EDM**, available on the web at <http://cdspecialproj.fnal.gov/d0/UpdatingEDM/index.htm>.

This document sketches a proposal for how we might achieve the goal of an improved EDM while allowing a significant (and adjustable) time period of backward compatibility, both for code and for previously existing Monte Carlo event samples. It is assumed that the readers are already familiar with the proposals, which the interested reader can find in the talk noted above.

## 2 Phased Obsolescence

Some of the modifications suggested below are to enhance the backward compatibility of the code, either for old MC files or for old code, or both. This will allow a more gradual move to the new EDM classes. However, it has the drawback of allowing new code to be written using the old interface. This should be strongly discouraged by the algorithms groups. In the final event, it may be that the only method of assuring compliance with the improved EDM is to remove the backwards compatibility features. The timing of this removal is a decision I happily leave to the management.

## 3 The Proposal

### 3.1 Modification to *Event*

The *Event* class is immediately modified as described in the talk. To summarize, the following changes are made:

1. *Event* no longer inherits from *d0\_Object*.
2. *Event* contains (and has sole ownership of) a single instance of *PersistentEvent* (described in section 3.2).
3. The function template `insertChunk(Event& e, auto_ptr<T> input)` is modified to create a *Provenance* from the information in the given *AbsChunk*. This introduces a space inefficiency, because some of the information stored in the *AbsChunk* class (or stored in the concrete subclasses of *AbsChunk*) will also be stored in the *Provenance* class.<sup>1</sup>

This change does not modify the requirement placed on the type used as the template parameter `T`; it must be either *AbsChunk*, or a class that inherits from *AbsChunk*. This modification will not break existing code. It allows existing code to interact with new-format *Event* objects, but at the cost of the inefficiency noted above.

4. A new *Event* friend function template is added, to support direct insertion of *Provenances* and *d0\_Objects* into the *Event*. The proposed signature is the following:

```
template <typename D0OBJ>    // this is a function template
ChunkID                    // it returns a ChunkID
insert(Event& e, const Provenance& p, auto_ptr<D0OBJ> obj);
```

Types to be used in place of the template parameter `D0OBJ` must inherit from *d0\_Object*. This function copies the *Provenance* and take ownership of the incoming *d0\_Object*, both of which are inserted into the given *Event*. Because *AbsChunk* inherits from *d0\_Object*, it is possible, in the short term, for users to switch to this syntax before modifying their “chunks”. Such use should be viewed only as a stepping-stone, on the way to reworking of the chunks themselves.

The `insertChunk` function of item 3 would make use of this function.

5. The *Event* class is given a constructor that takes an instance of *PersistentEvent*. For the sake of efficiency, it may be best for this constructor to take an `auto_ptr<PersistentEvent>`. The *Event* takes sole ownership of the given *PersistentEvent*, and over all the *d0\_Objects* therein.

---

<sup>1</sup>The data members of *AbsChunk* are one `int`, one `time_t` struct, and one `ChunkID`; these would all be replicated in *Provenance*. In addition, the collections of parent `ChunkIDs`, `RCPIDs` and `EnvIDs` stored in each subclass of *AbsChunk* would also be stored in *Provenance*.

## 3.2 Addition of *PersistentEvent*

The class *PersistentEvent* inherits from *d0\_Object*. It contains a *vector* of structs, where each struct contains one *Provenance* and one `d0_Ref<d0_Object>`.

Although the class `d0_Ref<T>` allows shared ownership of the pointed-to `T`, it is critical that there never be shared ownership of two such objects belonging to different *Events*. Since this rule of ownership is a class invariant of *Event*, it is enforced by *Event*. To assure this class invariant is kept, most manipulation of *PersistentEvent* will be done by the class *Event*. The only other classes that should need to directly manipulate *PersistentEvents* are *ReadEvent* and *WriteEvent*.

## 3.3 Responsibilities of *ReadEvent*

The class *ReadEvent* is responsible for reading both old and new data formats. Reading the old data format is more difficult and less efficient. It may be classified as a short-term responsibility, because when the decision is made to abandon the old format files this part of the code may be abandoned. Reading new-format data is, of course, a permanent responsibility.

### 3.3.1 Short Term Responsibilities

Until the old-format files can be abandoned, *ReadEvent* must be able to read old-format files, which are written as a sequence of old-format *Event* objects. To read an event, it looks inside the old *Event* object<sup>2</sup> and finds the *EventValue* object. It then iterates over all the *AbsChunks* stored in the *EventValue*, and uses the function `insert` (described above, in item 3 of section 3.1) to insert them into the *Event* object.

Note that this means the process of reading old-format data automatically translates the data into the new *Event* format. This does not magically transform an old version of a concrete chunk class to a new improved version; that transformation will have to be handled by some other mechanism, which I do not address here.

Note also that this means every `d0_Ref<AbsChunk>` is dereferenced when reading old-format data.

### 3.3.2 Permanent Responsibilities

For the long term, *ReadEvent* is responsible for reading new-format data files, which are written as a sequence of *PersistentEvent* objects. *ReadEvent* is responsible for reading the *PersistentEvent* (which is a *d0\_Object*) from the input stream, and for creating a new *Event* object, using the constructor that takes the just-read *PersistentEvent*. Note that *ReadEvent* is responsible for ensuring

---

<sup>2</sup>The new *Event* class is very different from the old *Event* class; it should not carry the responsibility for reading old-format data, and knows nothing about conversion from the old *Event* class.

that there is no shared ownership of the chunks that go into the *Event*. If *PersistentEvent* is only used to accept the stream of chunks from a file, and is given immediately to an *Event*, this should be simple.

### 3.4 Responsibilities of *WriteEvent*

The responsibility of *WriteEvent* is simple; given an *Event*, it must gain access to a *PersistentEvent*, and write it to the output. In order to make output of tagged chunks more efficient, *Event* should have member functions to allow construction of the correct *PersistentEvent* instance for the specific output requested. The *Event* will retain sole ownership of these *PersistentEvent* objects, but *WriteEvent* will be able to access them for writing.

The details of the interface of *Event* required for output must be specified by collaboration between the authors of the EDM and of the *iopackages* package.

### 3.5 Final Phase

The final phase of the modification removes the backward compatibility features. This allows some final simplification and cleanup of code, and reduces the maintenance burden of the EDM. It also ensures that all code is actually using the newer and more efficient version of the code.

The timing of implementing the final phase is a decision left to management.

1. The classes *AbsChunk* and *EventValue* are removed.
2. The function template `insertChunk(Event& e, auto_ptr<T> input)` is removed.

After this is done, it is no longer possible to read “old” Monte Carlo event files. If necessary, those files could still be transformed into new format files, subject to the ability of to transform each individual chunk class from “old” to new format. When the proposed modifications to DØØM (regarding user-controllable schema evolution) are in place, such backward compatibility will be possible. But consideration should be given to the code maintenance burden on all chunk designers when the decision regarding backward compatibility is made.

## 4 Conclusion

This document is just a sketch of a proposed method of implementing the modifications to the DØØM discussed during the May 2000 Software Infrastructure meetings. It is, I hope, a useful starting point for the production of a concrete plan of action, to be revised and extended as necessary.