CDF/DOC/COMP_UPG/PUBLIC/NNNN

CDF Run 2 Event Data Model Upgrade Proposal

R. D. Kennedy, P. Calafiura, C. Green, E. Hafen,
T. Huffman, J. Kowalkowski, A. Lee, K. McFarland, P. Murat,
S. Rolli, E. Sexton-Kennedy, P. Singh, P. Tamburello

DRAFT Version 0.2
February 1, 1999

Abstract

DRAFT DRAFT DRAFT

This note describes a proposal to upgrade the existing, but largely undocumented, CDF Run 2 Event Data Model. The Event Data Model describes how event data may be manipulated by user-written modules in CDF Offline software, through a combination of policies, programming interfaces, and object behaviors. The primary goal of this upgrade is to improve the reliability of the Offline simulation and production programs, and thus to help maximize the comprehensibility and reproducibility of physics results.

Contents

1 Introduction

We currently have an Event Data Model (EDM), though somewhat undocumented, which describes how CDF event data is organized, passed from module to module, and written/read to diskfiles for persistent storage. This model defines how C++ and F77 software accesses event data for the CDF Offline software, Level 3 and production reconstruction software, and simulation software. It is implemented by the AC++, Trybos, and Banks packages, but depends on all Offline packages obeying certain policies that are not enforced by any compiler. The event data treated in this model is just the data in a physics event record, and does not include calibration data, alignment data, and such. For historical reasons, the Event Data Model for CDF treats the persistence of module parameters, which have traditionally been stored in event datafiles.

This note proposes a potential CDF Run 2 Event Data Model. First it lists the requirements imposed on this model, and summarizes preliminary surveys of teh current state of OFfline software. This note then describes the associated policies and use-cases of the revised EDM, and provides some estimates of the effort required to implement the EDM framework and then transition Offline software packages to use that framework.

This proposal treats simple objects first, then treats associations between objects, collections of objects, ROOT Tree hierarchy amongst objects, Module Parameters Sets, and finally F77 event data access.

1.1 Nomenclature

A simple object refers to an object containing only data members of the following built-in types: char, short, int, long, uchar, ushort, uint, ulong, float, double, bool. A simple object is not associated with any objects, nor is it contained by another object.

reconstruction program = static reconstruction or simulation exe on disk, or the software used to compile and link that exe. reconstruction process = running reconstruction or simulation exe.

Algorithms are implemented by one or more AC++/Framework modules. A reconstruction or simulation process should be stoppable at any module boundary, and then restarted at the module boundary in a reproduceable fashion.

Event Data is that data which varies on an event-by-event basis. It includes amongst other things raw detector event data, reconstructed tracks and jets, and higher-level physics analsysis objects. For historical reasons, module parameter information at CDF is treated in the same manner as event data.

1.2 EDM Offline Packages

The Event Data Model is implemented throughout CDF Offline Software, but is supported in particular by a number of Infrastructure packages: Trybos, Banks, AC++/Framework. These packages are considered to be the foundation of physical design of CDF Offline software. Non-infrastructure CDF Offline packages may depend on these Infrastructure packages, but the Infrastructure may not depend any non-Infrastructure CDF Offline packages.

Refer to Trybos document for details.

Describe concepts related to Event Data Model: records, record iterators, API

Refer to Banks documentation for details.

Describe the concepts and utilities associated with Banks: Keys, base classes

Refer to AC++ document for details.

Describe concepts related to Event Data Model: modules, state diagram, etc.

Wrappers for F77 modules: what are the constraints?

1.3 The Event Data Model Working Group

In November 1998, the CDF Run II Event Data Model Working Group was formed to oversee the revision and documentation of the CDF Event Data Model. The charge of this Working Group is to:

1. define a list of use-cases which describe the minimum functionality required of the CDF Run II Event Data Model. The use-cases should include general user interactions with the event data as well as detailed interactions drawn from existing CDF software, especially those which demonstrate weaknesses or shortcomings in the current Event Data Model.
2. determine what pieces of our offline software system are consistent with our current Event Data Model and what pieces are not. What shortcomings in the current Event Data Model, if any, led to the alternative approaches taken by pieces that are not consistent with the current Event Data Model.
3. define how the implementation framework of the Event Data Model interacts with other data (Geometry and Calibration data, for example) and other Run II projects, especially the Data Handling System.
4. define a list of possible Event Data Model frameworks and an estimate of the effort to implement and transition to each.
5. recommend an Event Data Model framework which can be realized in the time allotted, achieves the best balance of functionality and maintainability versus effort to implement and transition to it, and which is consistent with existing software review recommendations.
6. periodically evaluate the realization and the performance of the revised Event Data Model. This may occur at regular intervals or at the formal request of software managers.

1.4 Summary of Preliminary Survey Results

No development is prevented by current model, though many areas of Offline would be easier to manage with improvements to the EDM. Some areas of the current EDM are no being adhered to, and should be addressed.

1.5 Requirements Imposed on Event Data Model

The EDM Working Group, working with Offline Software managers, has developed the following list of requirements to be placed upon this iterative revision of the CDF Run Event Data Model and its associated framework.

Requirement 1.5.1 *The EDM must be realizable in the time alloted.*

Clearly, it makes no sense to describe a model which cannot be achieved by the being of Run 2. All high priority items, items which yield the payoff in increased productivity given some effort, should be in place in April 1999, weeks before the May 1999 milestone for a complete production executable. All remaining items will be scheduled for completion and integration before the October 1999 milestone.

Requirement 1.5.2 *The EDM must be achieve the best balance of functionality and maintainability versus effort to implement and transition to it.*

Requirement 1.5.3 *The EDM must be consistent with existing software review recommendations.*

In particular, it must take advantage of the multi-branch, split-file mode of Root for Event I/O.

Requirement 1.5.4 *The EDM must require little or no change to existing C++ code.*

Trybos will act as an adapter layer to any new underlying implementations used. The existing Trybos API will remain intact, except where specifically noted. New approaches will generally be supported in addition to the old, rather than as an immediate replacement, giving developers time to evolve their software to the new approaches on a reasonable timescale.

Requirement 1.5.5 *The EDM must keep C++ interaction with event data as efficient as possible.*

Requirement 1.5.6 *The EDM must support Fortran access to event data at some level to be defined.*

Requirement 1.5.7 *The EDM must support the comparison of new code results with old results at some level to be defined.*

Requirement 1.5.8 *The EDM will consider changes to existing policies and use-cases to improve the robustness and reproduceability of simulation and production.*

2 Foundation

2.1 Reproduceable Results

Our ultimate goal in Offline software development is to accurately and precisely simulate and/or reconstruct physics processes detected within the CDF detector. We must do this in a controlled, reproducible fashion in order to justify the use of these reconstructed processes or events as the data sample for our scientific research. To achieve this goal, we choose to decompose our large reconstruction program into more-easily understood and managed algorithm pieces called modules. We then require that:

Policy 2.1.1 *Production results must can be fully and exactly reproduced if a production process is stopped and restarted at any module boundary versus a production process that is left to completion.*

2.2 Event Data Passed in AbsEvent in Persistable Form

Policy 2.2.1 *All event data passed from one module to another must be passed through the event data record.*

Policy 2.2.2 *All event data passed from one module to another must be persistable.*

The exact state of the event data, as well as its internal and external relationships, must captured on permanent storage when a production process is stopped, and then it must be restored when the production process is restarted.

2.3 Objects with Well-Defined State

Policy 2.3.1 *CDF event data objects should always have a complete, well-defined, and self-consistent state, one which requires no other event or algorithm-related information to interpret.*

Ideally objects should always be context-free, requiring no external knowledge in order to interpret their state. In reality, event data is never truly context-free since calibration, alignment, geometry, and other non-event data sources are often required to completely interpret the event data. Ignoring this connection to non-event data for the moment, CDF event data objects should maintain all state information internally. This insures that any algorithm which uses the objects will not have to encode assumptions about the objects, assumptions which may change over time.

Suppose Module AGenTracks generates track objects with some variations of algorithms. Subsequently, Module BFormVertices wishes to use these tracks, but only if they were generated by a particular algorthm or sequence of algorithms. BFormVertices cannot do so unless the processing history of the track objects is made an explicit part of the state information of the track objects. This requirement is imposed at all times, rather than just at module boundaries, because it is good object-oriented software practice in general.

2.4 Event Data and I/O Modules

Policy 2.4.1 *All event data input from permanent storage and output to permanent storage is performed in specialized AC++/Framwork modules called "I/O modules".*

2.5 Use-Cases

The fundamental Event Data Model use-case describes how event data is input and output by the Offline Infrastructure. The rest of the Event Data Model describes how the event data record can be manipulated by user-defined modules. The text description here actually falls somewhere between a use-case description and an activity diagram as defined in [2].

Use-Case 2.5.1 (Event Data Input and Output)

User executes a production program with a set of AC++ instructions.

AC++ interprets user's instructions, determines input and output filepaths, opens the event data files, and determines the input and output filtering criteria. AC++ passes the input file handle and filtering criteria to the input module, and the output file handle and filtering criteria to the output module.

IOModuleA performs input of that event data from input file which passes the input filtering criteria. This initializes the active event record.

UserModules manipulate the active event record.

IOModuleB performs output of that event data to output file which passes the output filtering criteria. The active event data record itself is not affected by the filtering criteria.

AC++ closes the event data files, and outputs statistics related to event data I/O to the user.

2.6 Design and Implementation

A few pieces are missing at present which would fully implement the fundamental use-case. First, output is not performed at present in an I/O module in the FrameMods package, and is not treated quite symmetrically with input. Second, no implementation of filtering criteria has been implemented at present. For Banks, we plan to use bank name sets as filtering criteria, but these are not fully implemented in Trybos, nor integrated into AC++, yet. There is no decision yet on what criteria to use for non-Banks persistable objects.

Note that this model makes explicit a change that came about over time in the Run 1 event data model. Only AC++ sets the input and output criteria. Modules do not have a say about what will be input and output. The user executing the production program will near complete control over what is input and output.

2.7 Transition

3 Simple Objects

3.1 Transient, Storable, and Persistable Objects

Objects can belong to one or more of three categories: transient, storable, and persistable. A transient object only lives as long as its enclosing C++ scope. A storable object, if stored in a CDF event data record, will outlive its enclosing scope, but not outlive the production process which created it. A persistable object, if in a CDF event data record which is saved to permanent storage, will outlive its enclosing scope and the process which created it.

There are few restrictions on the design of transient objects in the CDF Offline software system, but there significant restrictions on storable and persistable objects. Transient objects are not required to be simple objects; they can contain pointers or user-defined types. Storable and persistable objects, however, cannot contain pointers. Associations (treated later) are recorded using object links. Containment (also treated later) is recorded by saving the contained objects explicitly. In each case, the user must provide some mapping of the transient object's state to and from the storable and persistable object representation.

3.2 Intermodule Communication

Currently, the only storable objects in CDF event data are those which are derived from TRY_Generic_Bank. To make an object storable, the user must create a sub-class of TRY_Generic_Bank, with the ability to "flatten" and "restore" the state of a transient object to and from the flat integer array storage of a YBOS bank.

Now) All objects stored in AbsEvent are identified by a composite bank key, consisting of a bank name (a string of 4 printable characters) and an individual bank identifier (a 32-bit integer number ¿= 0). Each bank key is unique in the AbsEvent at any instant of time.

Future) All objects stored in AbsEvent are identifed by a composite key, consisting of a name string identifying the class of the object and an individual 32-bit integer identifier. Each composite key is unique in the AbsEvent throughout the lifetime of the AbsEvent.

*) Once an object is stored in AbsEvent, it should no longer be modified in place. Rather, a copy should be made, fields editted, and the editted copy stored in the record with a distinct identifier number. Explicit write-locking of stored objects will be considered if this policy is not implemented by changes to the Trybos/Banks APIs.

3.3 Permanent Storage

Currently, the only persistable objects in CDF event data are those which are derived from TRY_Generic_Bank. To make an object storable, the user must create a sub-class of TRY_Generic_Bank, with the ability to "flatten" and "restore" the state of an object to and from the flat integer array storage of a YBOS bank.

How to make them persistable? (now =¿ Banks)

The saving and restoring of event data is performed by, and only by, user-selectable I/O modules.

Bank name sets for I/O

3.4 Which Objects Should Be Storable and Persistable?

In the course of a reconstruction process, many objects are created and destroyed, some without being saved to permanent storage (transient objects), some with information sufficient to reconstiture them being saved to permanent storage (persistent objects). A user must decide whether or not there is cause to invest the extra development effort in order to make a particular object class persistent-capable (persistable) or leave the class as is.

Recommendation 3.4.1 *Any datum which is to be passed from one module to another, or compared to another datum arrived at by alternative algorithms or parameters, should be made persistable.*

Note that the recommendation above seems to mix the notion of intermodular communication and persistence. This mixing is due to the requirement that the reconstruction process be interruptible at any module boundary (in the midst of intermodule communication), its state saved by running an I/O module (persistence), the process restarted and its state restored by running an I/O module (persistence), continue processing at the next module (complete the interrupted intermodular communication), and reproduce the same results as an un-interrupted reconstruction process.

3.5 C++ Object Ownership and History

bookkeeping information, similar to D0 EDM. What module, parameters created the object? What is the object's post-creation processing history (We hope for no additional history, but may be some).

Now) Currently there is no explicit support for storing the originator of a CDF persistable object, nor for linking to a AbsParmSet representing the parameters driving the originating module. Users are responsible at present for saving this information themselves, as they have had to do in the past.

3.6 Use-Cases

As described in [1], there are four fundamental activities that can be performed by a user on elements of a data store: create, retrieve, update, and delete.

3.7 Design and Implementation

3.8 Transition

4 Associations

The simplest relationship between objects is an association, represented in Offline software as object links. Currently an object link is simply a bank number (incomplete instance association), or a bank name (class association), or both (complete instance association). These links are intended to be valid throughout the lifetime of an event. This is not currently enforced, however, and is an area which we seek to improve robustness.

4.1 Instance Association

Instance associations relate the object containing the link to a specific instance of another bank. The bank name is often not stored, but rather is assumed from the format of the bank data.

4.2 Class Association

Class associations relate the object (or its class) to another entire class.

4.3 Bank Numbers and Link Validity

Common use pattern: change a bank in place, leave with same number. Then anything linking to it would have a potentially stale link. The event data is no longer internally inconsistent.

4.4 Use-Cases

4.5 Design and Implementation

4.6 Transition

5 Collections

5.1 Homogenous Collections

recommendations, support tools, links to an element within a collection

5.2 Heterogenous Collections

In general, there is no support for heterogenous collections in the EDF Run 2 EDM. One can, however, store a collection summary object containing links to each the elements in the heterogenous collection.

If classes are cousins, that is if they share a common base class, and if the state information resides entirely in the shared base class, then the same tools available for heterogenous collections may be used provided each object also saves something to indentify its actual child class, rather than identify the shared base class.

5.3 Use-Cases

5.4 Design and Implementation

5.5 Transition

6 Root Branches

hierarchy within event, fixed versus user-defined (or physics group-defined) organization

6.1 Basic Tree-Branch Hierarchy of Event

6.2 Physics Group-Definable Branch Hierarchy

6.3 User-Manipulable Branch Hierarchy

6.4 Use-Cases

6.5 Design and Implementation

6.6 Transition

7 Parameter Sets

grouped with EDM for historical reasons. LSK will do this, identifiers needed which are unique within some space.

7.1 Creating a Parameter Set

7.2 Retrieving a Parameter Set

7.3 Updating a Parameter Set

7.4 Referring to a Parameter Set

7.5 Use-Cases

7.6 Design and Implementation

7.7 Transition

8 Fortran Use

8.1 Policy

Use of Fortran software in the reconstruction program is supported in a limited fashion, to permit the re-use of existing Fortran algorithm software without hindering C++ software development. In this discussion, modules are referred to as Fortran modules if they access the event data directly from Fortran, C++ modules if the event data is accesses directly from C++, and mixed-language modules if the event data is accessed from both Fortran and C++. This description only refers to the language of the software which accesses event data.

In any one module in an Offline software, only one language (F77 or C++) may access the event data directly. In other words, there is no support for mixed-language modules. Also, there will be significant CPU overhead per Fortran module when accessing the event data from Fortran, due to the entire event be copied and/or transformed to a Fortran-compatible format, and then copied/transformed back upon leaving the Fortran module.

Neither Fortran nor mixed-language modules are supported in the reconstruction program. All event data access is from C++. This is a policy change relative to past statements. Some changes to existing reconstruction software will be required to implement this policy. Recent surveys indicate that there are only a few Fortran modules remaining, and that these are scheduled to be replaced by C++ modules at some point in the future.

This does not mean that one cannot use Fortran in the reconstruction program, just that some other way of passing the necessary event data between these languages must be implemented by the developer. One can pass data to Fortran from C++, for instance, via subroutine arguments or common blocks.

8.2 Support for Accessing F77 "Objects" from C++

Banks

8.3 Support for Accessing C++ Objects from F77

NOT in general. Exceptions on a case-by-case basis, with support implemented on approval

8.4 Use-Cases

8.5 Design and Implementation

8.6 Transition

9 Conclusion

Form a group to explore how much effort needs to be expended to develop and adapt to elements of the future event data model. Based on this, the group would propose a future event data model along with the estimate of cost and time to achieve it.

References

[1] Scott A. Ambler, *Building Object Applications That Work*, Managing Object Technology (Cambridge University Press, Cambridge, 1998), ISBN 0-521-64826-2.

[2] Hans-Erik Eriksson and Magnus Penker, *UML Toolkit*, (John Wiley & Sons, New York, 1998), ISBN 0-471-19161-2.