ACCESS TO NON-EVENT DATA FOR CMS

C. D. Jones, Cornell University, Ithaca, NY 14853, USA

Abstract

In order to properly understand the data taken for an HEP Event, information external to the Event must be available. Such information includes geometry descriptions, calibrations values, magnetic field readings plus many more. CMS has chosen a unified approach to access to such information via a data model based on the concept of an 'Interval of Validity', IOV. This data model is organized into Records which hold data that have the same IOV and an EventSetup which holds all Records whose IOV overlaps with the Event that is being studied. The model also allows dependencies between Records and guarantees that child Records have IOVs which are intersections of the parent Records' IOVs. The implementation of this model allows the data from a Record to either be created from a persistent store (such as a database) or from an algorithm, where the choice is made by the physicist at job configuration time. The client code that uses the data from a Record is completely uneffected (relinking is not even necessary) by the mechanism used to create the data.

INTRODUCTION

To properly process an HEP Event requires access to data that is not directly part of the Event. Such data items are

- **Conditions**: data measured during data taking, e.g., the magnetic field.
- **Calibrations**: data used to properly interpret the measurements from the detector: e.g., pedestal values for the silicon pixel detectors.
- **Geometry**: the physical position of the various detector components, e.g., the perfect (design) position and shapes of all materials, or the measured position of the tracking system.

The system for delivering the non-Event data must be capable of returning the appropriate data for the particular Event being processed by the data processing framework. This paper will describe the system employed by the new CMS data processing framework [1] to deliver the non-Event data. The part of the framework that handles the non-Event data is known as the EventSetup system.

MENTAL MODEL

Interval of Validity

The Interval of Validity, IOV, is the central concept for the design. An Interval of Validity is the time period for which a datum is valid. For example, the ECal pedestals version 123 is valid from run 12 through run 14.

Data is versioned by forming a graph of non-overlapping Intervals of Validity. Although overlaps are not allowed within one version, gaps in time when there is no valid datum are allowed. Figure 1 shows two version graphs for the same data. Version graph 1 has two different versions of the data (A and B) with a gap in time between the two data versions. Version graph 2 also uses the same two data versions but it has truncated the valid time range for version A and added a new version, C, between the original data versions.

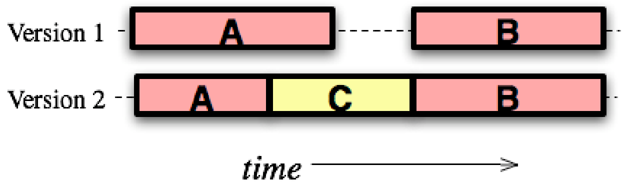

Figure 1: Two Version graphs made by chaining different Intervals of Validity for different versions of data

EventSetup Model

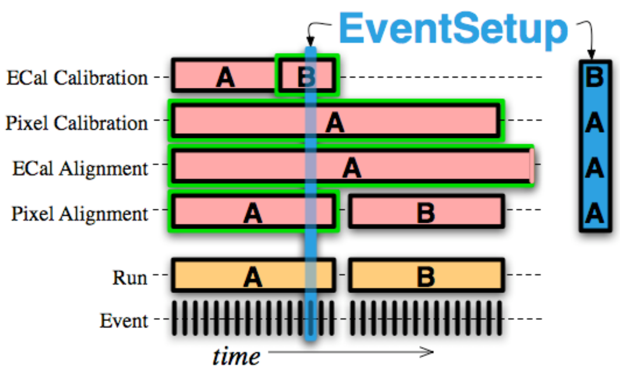

Figure 2: The EventSetup is formed from the Records that have an *IOV* that overlaps with the moment in time that is being studied

The EventSetup provides a uniform access mechanism to all data/services constrained by an IOV. The main concepts for the EventSetup are shown in Figure 2:

- **Record**: holds data and services that have identical IOVs.
- **EventSetup**: provides a 'snapshot' of the experiment at an instant in time. Formed by all Records that have an IOV which overlap with the 'time' of the Event being studied.

This is not a new idea, the CLEO experiment has been using the same idea for nearly a decade [2]. The CMS implementation of this system is directly based on the work done for CLEO.

IMPLEMENTATION

EventSetup

The EventSetup class provides type-safe access to the various Records it contains. This access is done through the EventSetup's *get<RecordT>()* method. If the requested Record is not available, an exception will be thrown. There is also an interface to get data directly from an EventSetup instead of from a Record for the case where the data type has been assigned a 'default' Record. The direct data access interface is discussed later.

In addition to access to Records, the EventSetup has a method *iovSyncValue()* which return information about the 'instance in time' for which the EventSetup is describing.

Records

Each Record is a separate C++ class inheriting from a common base class. This allows the compiler to catch name typos and is used to do C++ template meta-programming (described later). Records provide type-safe access to the objects it contains. The access is handled through the Record's *get(ESHandle<T>&)* method. This is analogous to data access from the Event.

A Record also provides access to its interval of validity (IOV) through its *validityInterval()* method.

Contents of a Record

The EventSetup system sets no requirements on the C++ class type of an object that may be placed in a Record. The only restriction is the lifetime of the objects within a Record is only guaranteed to be as long as the IOV for which the Record is appropriate. This does not mean that an object within a Record cannot be reused across an IOV transition; it only means code that reads the object from a Record should not assume that it will be reused.

In the case where a data/service C++ type is only meant to come from one Record type, then the 'default' Record type can be declared at compile time. If a 'default' Record has been declared for a data type, then users can access that data directly from the EventSetup via the *get(ESHandle<T>&)* method.

USAGE

A const instance of the EventSetup is passed as an argument to the Event system components processing method. An example is shown below.

```
void Example::produce(
  Event& event, const EventSetup& eventSetup ) {
  ESHandle<TrackerGeometry> geomPtr;
  eventSetup.get<TrackerAlignmentRecord>()
       .get( geomPtr );
```

In the example we are requesting the *TrackerGeometry* object. The smart pointer *ESHandle<>* will hold this object. We first ask the EventSetup for the proper Record (in this case *TrackerAlignmentRecord*) and then get the data from the Record by passing the *ESHandle<>*. The system determines the type of the data being requested based on the template argument of the *ESHandle<>*. The type is then used to lookup the data from within the Record. If a default Record has been assigned to the data at compile time, then the data can be directly requested from the EventSetup:

```
eventSetup.getData( geomPtr );
```

The use of a 'smart pointer' to hold the results of the data rquest is completely analogous to how the Event system operates.

```
Handle<PixelDigi> digiPtr;
Event.getByLabel("barrel", digiPtr)
```

The similarity between the Event and EventSetup data access should make it easier for physicists to use both systems.

SYSTEM COMPONENTS

The EventSetup system design, uses two categories of dynamically loadable components to do the work of creating or reading the data: ESSource and ESProducer.

ESSource

An ESSource is responsible for determining the IOV of a Record (or a set of set of Records). The ESSource may also deliver data/services. An ESSource normally reads its information from a 'persistent store' (e.g., a database) although it is not required to do so. For example, one could write an ESSource which reads calibration information from a database using the run number as an index.

ESProducer

Conceptually, an ESProducer is an algorithm whose inputs to its algorithm are dependent on data with IOVs. This data is obtained by getting it from EvenetSetup Records. For example, an ESProducer could create the tracking geometry based on alignment values and structural information obtained from different Records.

CONFIGURATION

EventSetup components are configured using the same configuration mechanism as their Event counterparts, i.e., via the ParameterSet system. An example of the ParameterSet configuration language is shown below:

```
es_source = GeometryFileSource {
        string filename = "geom.xml" }
es_source = CalibrationDBSource {
           using standardDBConfiguration
           string version = "v1"
           string tag = "RECO" }
es_module = TrackerGeometryProducer { }
```

The *es_source* keword states that the component to be loaded is an ESSource, while the *es_module* keyword states that this is an ESProducer (this matches the use of *source* and *module* used to configure components that deal with Events). The name after the equals sign is the

C++ type of the component that should be dynamically loaded. Within the curly braces are the various parameter names and values used to configure the component.

Once configuration is complete, which is before the first Event is processed, all EventSetup (and Event) components that will be used in the job will have been dynamically loaded. This guarantees that any load problems will have been caught before any time-consuming event processing has occurred.

DATA RETRIEVAL

To a physicist, the EventSetup appears to contain all the data necessary for the Event being processed. However, preloading or computing all data associated with an Event could be costly, especially if some of that data is never used. Therefore the EventSetup system employs 'data on demand', i.e., the data is only loaded or computed the first time it is requested. Because ESProducrs can be dependent on other data in the EventSetup, it is possible for a data request to start a whole chain of data processing. An example is shown in Figure 3. In the example, the EDProducer (which is an Event processing component) named Tracker requests the tracking geometry description. The request is dispatched to the TrackerGeometry ESProducer which then requests the 'perfect' geometry description from the Geometry Source and the alignment displacements from the Alignment DB. Once the new tracking geometry has been created, it is returned to the Tracker module.

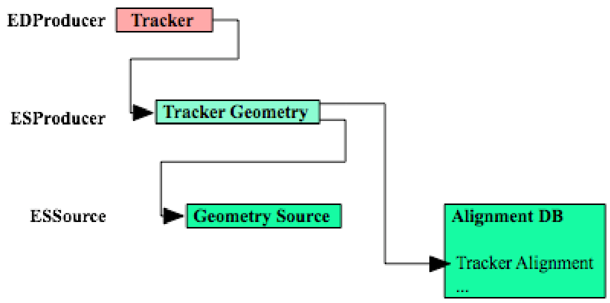

Figure 3: Example of data on demand

IOV DEPENDENCIES

Sometimes an algorithm in the EventSetup is dependent on data coming from more than one Record. For example, the tracking geometry is dependent on the 'ideal geometry' and on the tracking alignment values. In such a case the Record used by that algorithm needs to be declared 'dependent' on the other Records. This dependency declaration is done by having the dependent Record inherit from *DependentRecordImplementation<T,List>*. The template parameter *TList* is a compile time list of the Records upon which this Record is dependent.

Dependent Records allow access to the Records to which they are dependent via the *getRecord<T>()* method. The *TList* is used at compile time to check that the *T* used in the method is actually in the list.

The IOV of a dependent Record is the intersection of the IOV of all the Records to which it depends. The EventSetup system guarantees that the proper relationships between the IOVs are preserved.

CREATING A COMPONENT

To better understand what is involved in creating a component, an example of an ESProducer is shown below.

```
MyProd::MyProd( const ParameterSet& iPS ) {
  setWhatProduced( this );
}

auto_ptr<MyData>
MyProd::produce( const MyRecord& iRecord ) {
   ESHandle<OtherData> otherPtr;
   iRecord.get( otherPtr );
   auto_ptr<MyData> myPtr( ... );
   return myPtr; }
```

The call to *setWhatProduced* in the constructor deduces what data the MyProd produces (in this case a MyData) as well as what Record that data should be placed (in this case the MyRecord) from the return value and arguments to the *produce* method. The data and Record type are used to register what the ESProducer can do so that the data retrieval can work. This allows developers to just change the *produce* method and not have to worry about updating the registration information. Because the produce method only has access to one Record type (or to the Records that have been declared as dependencies of that Record) the data produced will automatically have the proper IOV.

CONCLUSION

CMS' new data processing framework employs a unified data model for all non-Event data. This data model is based on the concept of an Interval of Validity and is accessed via Records held by the EventSetup. This unified data model means physicists only have to learn one set of rules to access all non-Event data.

The EventSetup system design employs C++ type safe data access, 'data on demand', guaranteed consistency between related IOVs and C++ object lifetime management. The actual work of creating the non-Event dat is handled by dynamically loaded components.

Based on the experience from the CLEO experiment, the EventSetup system should be sufficient to provide for all CMS' non-Event data needs.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation and the U.S. Department of Energy.

REFERENCES

[1] C. Jones, M. Paterno, J. Kowalkowski, L. Sexton-Kennedy, W. Tanenbaum "The New CMS Event Data Model and Framework", International Conference on Computing in High-Energy Physics and Nuclear Physics, Mumbai, India (2006).

[2] C. Jones, S. Patton, M. Lohner, P. Avery, “Design and Implementation of the CLEO III Data Analysis Model”, International Conference on Computing in High-Energy Physics and Nuclear Physics, Berlin, Germany (1997).