Access to Non-Event Data for CMS

Christopher Jones
Cornell University

Overview

Purpose

Mental Model

Implementation
- Getting Data
- Component Architecture

Conclusion

Purpose

Provides access to 'non-Event' data needed to process an event

Conditions
- *e.g. magnetic field readings*

Calibrations
- *e.g., pixel pedestals*

Geometry
- *e.g., perfect full detector description*
- *e.g., aligned tracking geometry*

Guarantees that the data returned is appropriate for the particular Event being processed by the framework

Manages the 'non-Event' data C++ objects' life-time

Interval of Validity

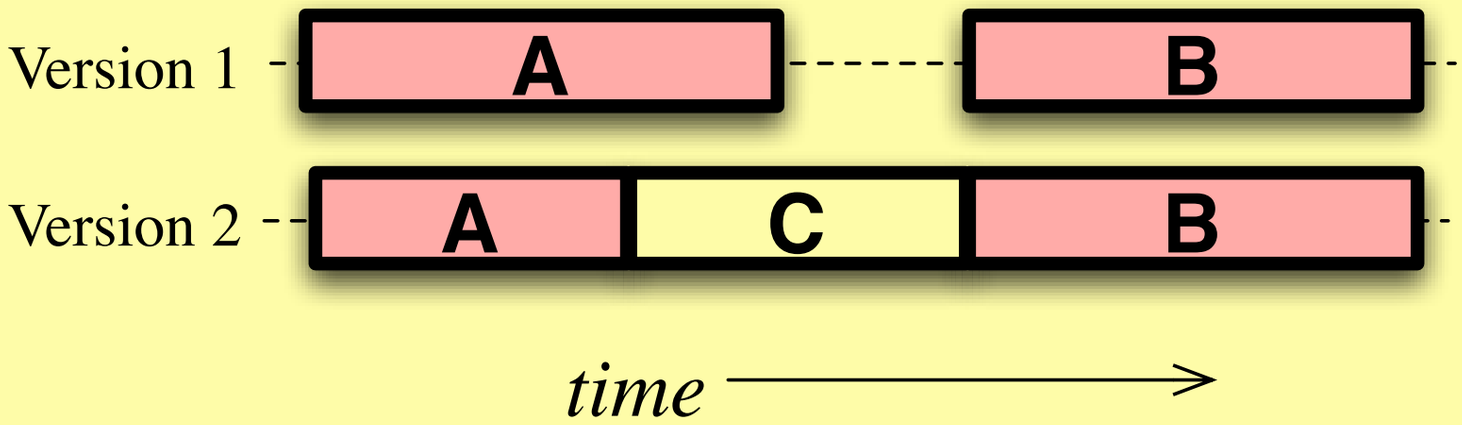
Central concept: Interval of Validity (IOV)

Time period for which a datum is correct

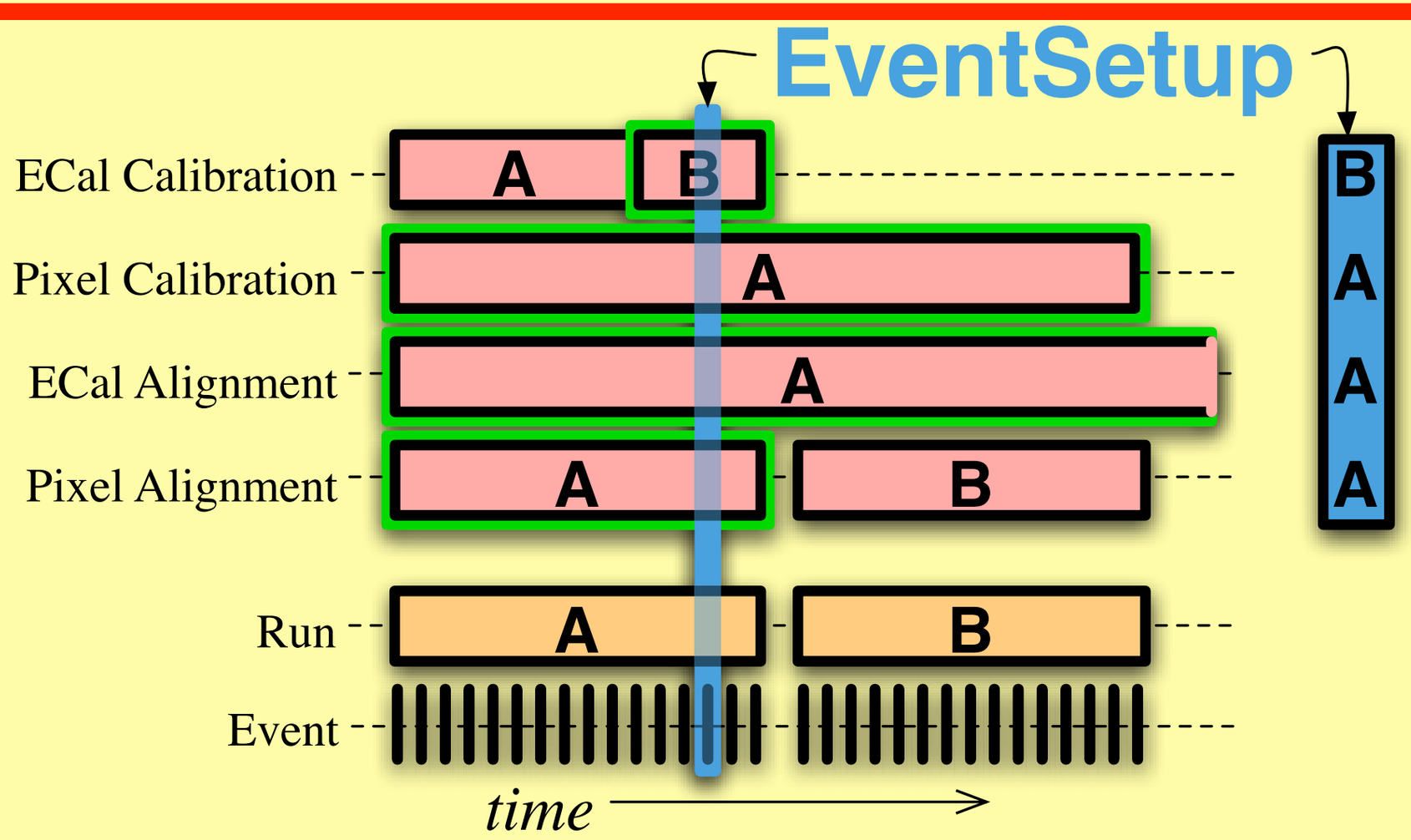
e.g., ECal pedestal version 123 is appropriate for run 12 through run 14

Version graph formed by chaining non-overlapping IOVs

Gaps in time are permitted



Mental Model



Provides a unified access mechanism for non-Event data

Record holds data with same interval of validity

EventSetup "snapshot" of detector at an instant in time

Not a new idea: has been used by CLEO experiment since 1998

Implementation

Philosophy: catch as many problems as possible at compile time

Each Record is a unique C++ type
- Compiler catches typos
- Record type can be used for C++ template meta-programming

EventSetup class returns a Record in a type-safe manner

Records can hold any C++ object
- Data returned from a Record in a type-safe manner
- Lookup data in Record by C++ type or 'C++ type and string'

Use in Event Processing

EventSetup is passed as argument to Event module's method

```
void Example::produce(Event& event,
                      const EventSetup& eventSetup ) {

   ESHandle<TrackerGeometry> geomPtr;
   eventSetup.get<TrackerAlignmentRecord>().get( geomPtr );


  //the above is the same as the following
  // (default Record was set at compile time)
   eventSetup.getData( geomPtr );


    //very similar to Event access
   Handle<PixelDigi> digiPtr;
   event.getByLabel("barrel", digiPtr );
```

Components

Components do the work of actually creating/reading the data

The EventSetup supports two types of dynamically loaded components

ESSource

reads data from disk
sets the 'interval of validity' for data in a Record
e.g., read calibration information from a database for a particular run range

ESProducer

creates data by running an algorithm
obtains data needed by the algorithm from Records in the EventSetup
e.g., create tracking geometry by combining alignment shifts and perfect positioning of material

Configuration

EventSetup configuration is done in the job configuration file the same way it is done for the Event component configuration

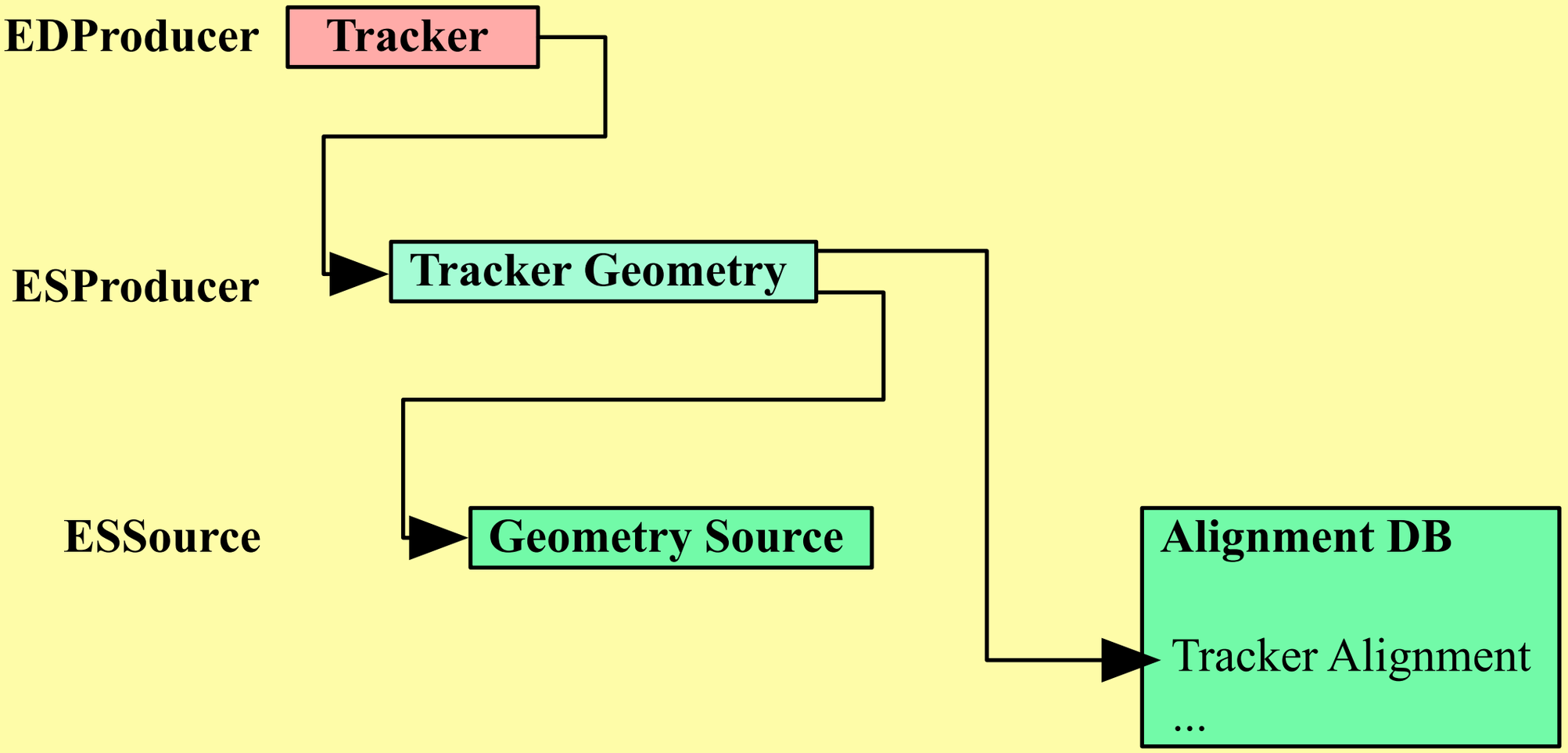
```
process RECO = {
    es_source = GeometryFileSource {
                  string filename = “geom.xml”
                }

    es_source = CalibrationDBSource {
                  using standardDBConfiguration
                  string version = “v1”
                  string tag = “RECO”
                }

  es_module = TrackerGeometryProducer { }
```

After configuration is complete, all EventSetup components will have been loaded into the application

Data Retrieval

To a user, EventSetup appears to have all its data loaded

To avoid unnecessary computation, data is retrieved on the first request



NOTE: an EDProducer is an Event module

Record Interdependency

ESProducer's may need to get data from other Records

If data in Record A depends on data in Record B then when Record B's validity interval changes Record A's validity must also change

System automatically handles validity dependencies

Record interdependencies set and checked at compile time

```
class TrackerGeometryRecord :
  public DependentRecord<mpl::vector<GeometryRecord,
                                     TrackerAlignmentRecord>
                         >
{ };
```

Creating a Component

An ESProducer component creates data via an algorithm

The data produced has the 'validity interval' of its Record
produce method called on first data request after validity interval change

The data and Record type are deduced from the signature of the produce method

```
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; }
```

Conclusion

CMS has a unified data model for all non-Event data

Based on an 'Interval of Validity' accessed through an EventSetup

Physicists only have to learn one set of rules to access all such data

Design uses

type-safe data access

'on demand' data retrieval or computation

automatic consistency between related 'Intervals of Validity'

C++ object lifetime management

Implemented using dynamically loadable components

Client code does not need to be recompiled or relinked when configuration changes

Based on experience from CLEO, the system should provide for all of CMS's future needs