Software for e+e- analysis

3rd FCC-France / Higgs & ElectroWeak Factory Workshop, Annecy

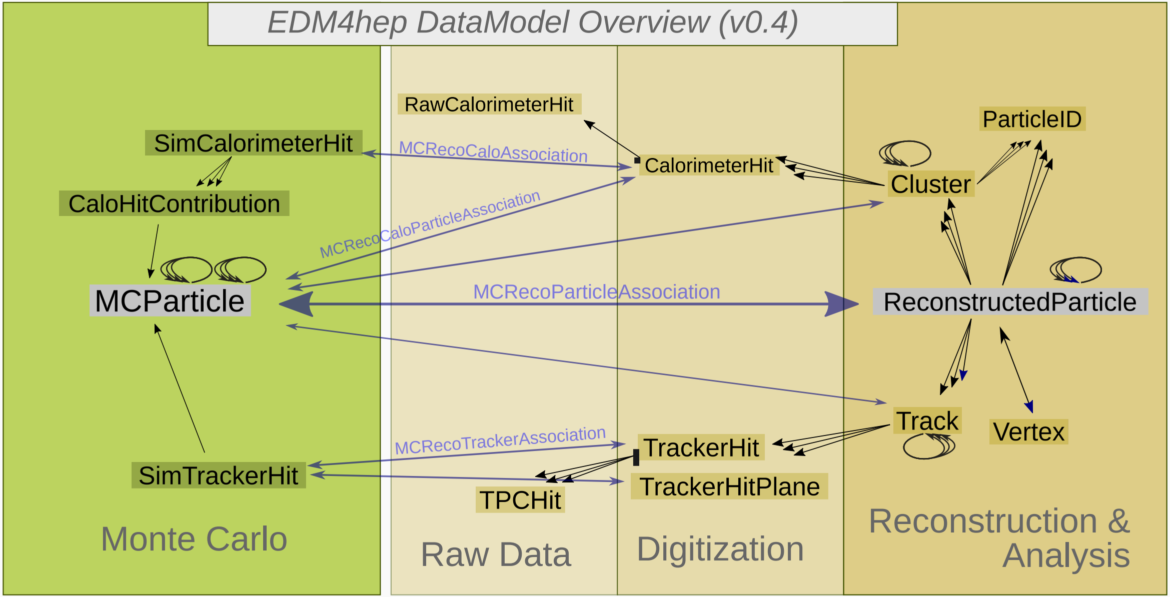
Thomas Madlener
for the Key4hep team

Nov 30, 2021

Goals for EDM4hep

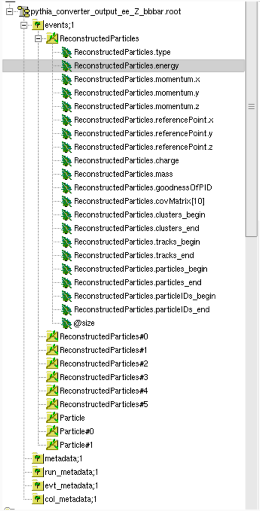
- The Key4hep project aims to define a common software stack for all future collider projects
  - see talks by Valentin and Clement
- **EDM4hep** is the **common EDM** that can be used by **all communities** in the Key4Hep project
  - ILC, CLIC, FCC-ee & FCC-hh, CEPC, ...
- Support different use cases from these communities
- Efficiently implemented, support multi-threading and potentially heterogeneous resources
  - Generated by **`podio`**
- Use experience from LCIO and FCC-edm

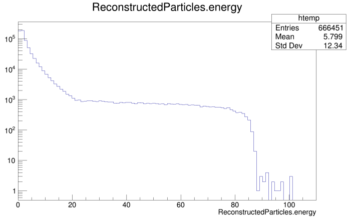
EDM4hep schema

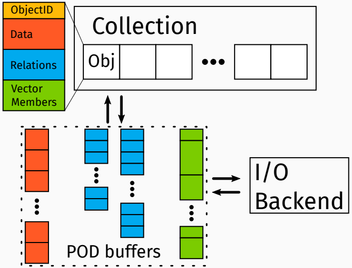


EDM4hep supports different I/O backends

- Default **ROOT** backend
  - POD buffers are stored as branches in a **`TTree`**
  - Can be used in **`RDataFrame`** or with **`uproot`**
  - Files can be interpreted **without EDM4hep library**
- Alternative **SIO** backend
  - Persistency library used in **`LCIO`**
  - Complete events are stored as binary records
- Adding more I/O backends is possible







Different ways to work with EDM4hep

- C++ interface
  + Easy access to all relations
  + Transparently handle I/O backends
  - Need to compile against existing installation
- Python interface
  - Very similar to C++ interface by design
  - Using PyROOT and a few dedicated python wrappers
- Reading files with uproot or RDataFrame
  + Can work without `edm4hep` shared library
  + Data is already stored “columnar”, no need to produce “flat tuples” first
  - Relation handling can be cumbersome
  - Somewhat relies on implementation details
  - Only possible from root files

C++ & Python interface

```cpp
using namespace edm4hep;

auto reader = podio::ROOTReader();
reader.openFile("events.root");

auto store = podio::EventStore();
store.setReader(&reader);

for (size_t i = 0; i < reader.getEntries(); ++i) {
  auto& recos =
    store.get<ReconstructedParticleCollection>("recos");

  for (auto rp : recos) {
    // get associated tracks and clusters
    auto tks = rp.getTracks();
    auto clus = rp.getClusters();

    // Loop over decay products
    for (auto dp : rp.getParticles()) {
      std::cout << dp.getMass() << std::endl;
    }
  }
}
```

```python
# alignment



store = Eventstore('events.root')



for event in store:
  recos = event.get('recos')


  for rp in recos:
    # get associated tracks and clusters
    tks = rp.getTracks()
    clus = rp.getClusters()

    # Loop over decay products
    for dp in rp.getParticles():
      print(dp.getMass())


# alignment
```

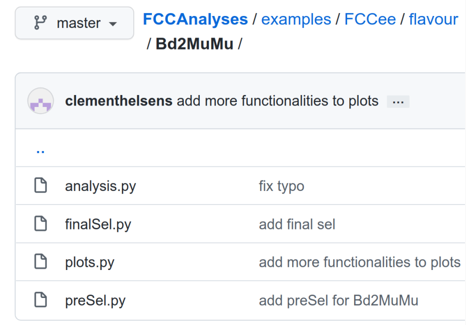
- Essentially the same interface in python and C++
- Can also be useful for quick prototyping or debugging

FCCAnalyses - RDataFrame & EDM4hep

- `FCCAnalyses` is a python analysis framework based on `RDataFrame`
  - "Builtin multithreading"
  - Comes with high level reco functionality
  - Extensible via C++
- **Not specific to FCC!** Rather to the EDM4hep input format
- Declarative style of analysis
  - Describe what you want
  - Framework deals with the details of how exactly
- HEP-FCC/FCCAnalyses

FCCAnalyses - The basic building blocks

- Each analysis needs to define 4 python modules/scripts
  - `analysis.py` defines the analyzers and filters to run as well as the available output variables
  - `preSel.py` defines the samples to use, number of CPUs and produces a local ntuple file
  - `finalSel.py` defines the final cuts and variables that are used for plotting
  - `plots.py` defines which samples to plot and the cosmetics of the produced plots



Analysis steps

Run high level reconstruction and first stage selection

```
python examples/FCCee/flavour/Bd2MuMu/preSel.py
```

Defining analyzers and filters `analyis.py`

```python
    def run(self):
        df2 = (self.df
               ############################################
               ##          Aliases for # in python       ##
               ############################################
               .Alias("MCRecoAssociations0", "MCRecoAssociations#0.index")
               .Alias("MCRecoAssociations1", "MCRecoAssociations#1.index")
               .Alias("Particle0", "Particle#0.index")
               .Alias("Particle1", "Particle#1.index")

               ############################################
               ##               Build MC Vertex          ##
               ############################################
               .Define("MCVertexObject", "myUtils::get_MCVertexObject(Particle, Particle0)")

               ############################################
               ##              Build Reco Vertex         ##
               ############################################
               .Define("VertexObject", "myUtils::get_VertexObject(MCVertexObject,ReconstructedParticles,EFlowTrack_1,MCRecoAssociations

               ############################################
               ##          Build PV var and filter       ##
               ############################################
               .Define("EVT_hasPV",     "myUtils::hasPV(VertexObject)")
               .Define("EVT_NtracksPV", "myUtils::get_PV_ntracks(VertexObject)")
               .Define("EVT_NVertex",   "VertexObject.size()")
               .Filter("EVT_hasPV==1")
```

- Branch naming not yet ideal
- Relation handling requires a bit of "inside knowledge"

- Defining new variables and filtering on them is easy
- Event loop is only run once!

Defining analyzers and filters

analysis.py

```
                ############################################
                ##         Build B0 -> MuMu candidates      ##
                ############################################
                .Define("Bd2MuMuCandidates",         "myUtils::build_Bd2MuMu(VertexObject,RecoPartPIDAtVertex)")

                ############################################
                ##        Filter B0 -> MuMu candidates      ##
                ############################################
                .Define("EVT_NBd2MuMu",              "float(myUtils::getFCCAnalysesComposite_N(Bd2MuMuCandidates))")
                .Filter("EVT_NBd2MuMu==1")

                ############################################
                ##    Get the B0 -> MuMu candidate mass     ##
                ############################################
                .Define("Bd2MuMu_mass",    "myUtils::getFCCAnalysesComposite_mass(Bd2MuMuCandidates)")
```

```
ROOT::VecOps::RVec<FCCAnalysesComposite2> build_Bd2MuMu(ROOT::VecOps::RVec<VertexingUtils::FCCAnalysesVertex> vertex,
                                                        ROOT::VecOps::RVec<edm4hep::ReconstructedParticleData> recop);
```

Dedicated code defined in analyzers/dataframe/myUtils.h and analyzers/dataframe/myUtils.cc

Analysis steps

Run high level reconstruction and first stage selection

```
python examples/FCCee/flavour/Bd2MuMu/preSel.py
```

Run final selection and fill histograms

```
python examples/FCCee/flavour/Bd2MuMu/finalSel.py
```

Final selection

finalSel.py

```python
from config.common_defaults import deffccdicts
import sys, os
import ROOT

###Input directory where the files produced at the pre-selection level are
baseDir  ="/eos/experiment/fcc/ee/analyses/case-studies/flavour/Bd2MuMu/flatNtuples/spring2021/Batch/'

###Link to the dictonary that contains all the cross section informations etc...
procDict = os.path.join(os.getenv('FCCDICTSDIR', deffccdicts), '') + "FCCee_procDict_spring2021_IDEA.

process_list=['p8_ee_Zbb_ecm91_EvtGen_Bd2MuMu',
              'p8_ee_Zbb_ecm91'
              ]

define_list={}

###Dictionnay of the list of cuts. The key is the name of the selection that will be added to the out
cut_list = {"sel0":"Bd2MuMu_mass>0"}

###Dictionary for the ouput variable/hitograms. The key is the name of the variable in the output fil
variables = {
    "EVT_CandMass"      :{"name":"Bd2MuMu_mass","title":"mass [GeV]","bin":300,"xmin":0,"xmax":6.},
    "EVT_CandMass_zoom" :{"name":"Bd2MuMu_mass","title":"mass [GeV]","bin":100,"xmin":5.,"xmax":6.}
}
```

- Define additional cuts
- Use variables defined previously to fill histograms

Analysis steps

Run high level reconstruction and first stage selection

```
python examples/FCCee/flavour/Bd2MuMu/preSel.py
```

Run final selection and fill histograms

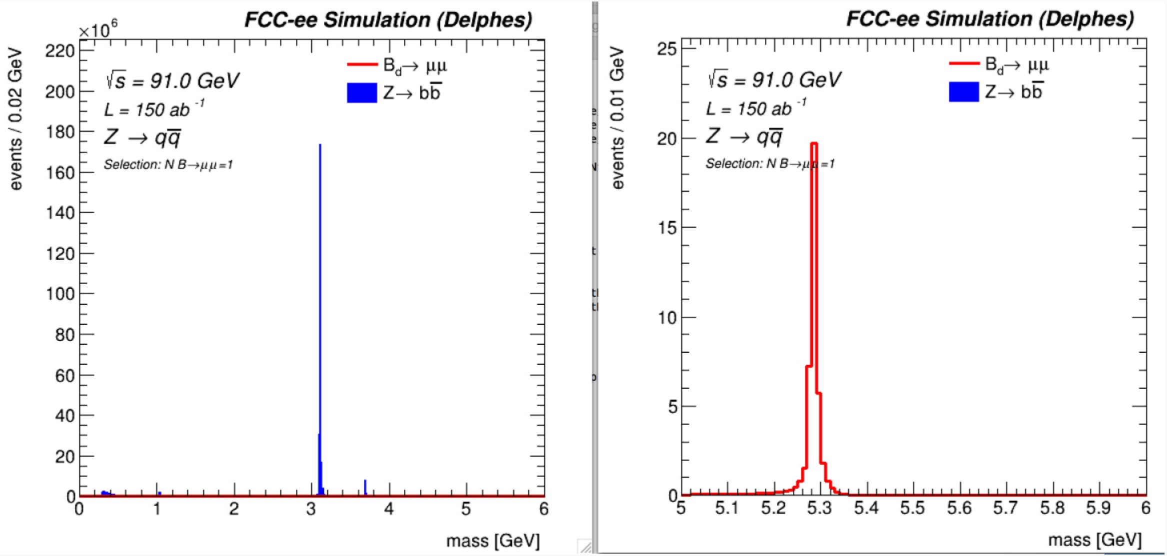
```
python examples/FCCee/flavour/Bd2MuMu/finalSel.py
```

Produce plots

```
python config/doPlots.py examples/FCCee/flavour/Bd2MuMu/plots.py
```

```python
# global parameters
intLumi        = 150000000. #in pb-1
ana_tex        = "Z #rightarrow q#bar{q}"
energy         = 91.0
collider       = "FCC-ee"
customLabel    = "Preliminary"
inputDir       = "/eos/experiment/fcc/ee/analyses/case-studies/flavour/Bd2MuMu/flatNtuples/spring2021/Batch/"
formats        = ['png','pdf']
yaxis          = ['lin','log']
stacksig       = ['nostack','stack']
outdir         = 'plots_Bd2MuMu/'
variables      = ["EVT_CandMass","EVT_CandMass_zoom"]
legendCoord    = [0.68,0.76,0.96,0.88]
scaleSig       = 1.

###Dictonnary with the analysis name as a key, and the list of selections tc
###The name of the selections should be the same than in the final selectior
selections = {}
selections['Bd2MuMu']       = ["sel0"]

extralabel = {}
extralabel['sel0'] = "Selection: N B#rightarrow#mu#mu=1"

colors = {}
colors['Z_bb']  = ROOT.kBlue
colors['Z_Bd']  = ROOT.kRed
colors['Z_Bd2']  = ROOT.kBlue
colors['Z_Bd3']  = ROOT.kRed

plots = {}
plots['Bd2MuMu'] = {'signal':{'Z_Bd':['p8_ee_Zbb_ecm91_EvtGen_Bd2MuMu'],'Z_Bd2':['p8_ee_Zbb_ecm91_EvtGen_Bd2MuMu'],'Z_Bd3':['p8_ee_Zbb_ecm91_EvtGen_Bd2MuMu']},
                    'backgrounds':{'Z_bb':['p8_ee_Zbb_ecm91']}
```

- Choose what to plot
  - Output format, variables, ...
- Define cosmetics of plots
  - Axis scaling, colors, ...



courtesy of C. Helsens

Outlook & Currently ongoing work

- Move core functionality and utilities to key4hep/k4Analysis
    - Keep FCC specific parts in HEP-FCC/FCCAnalyses
- Currently collaborating with the RDataFrame developers
    - Find performance bottlenecks
    - Improve handling of inter-object relations
    - First version of a `RNTuple` based I/O backend on the way
- Finalize schema of EDM4hep v1.0
    - Still some work to do on the technical side

Summary

- **EDM4hep** is the **common EDM of the Key4hep project**
  - Already actively used for physics studies in different communities
- FCCAnalyses/k4Analysis offers an easy to use analysis framework based on RDataFrame
  - Very flexible and powerful
  - Comes with high level reconstruction functionality
  - Already used for large scale FCC-ee productions
- Still under active development for improved performance and usability

Pointers to software (re)sources

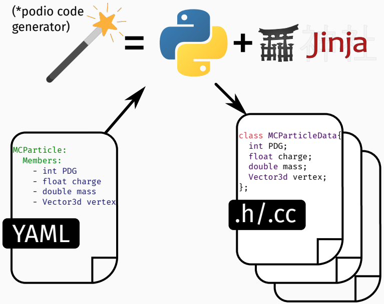
- Key4hep
  - key4hep.github.io/key4hep-doc
  - key4hep - github organisation
- EDM4hep
  - key4hep/EDM4hep
  - cern.ch/edm4hep
- podio
  - AIDASoft/podio
- FCCAnalyses
  - HEP-FCC/FCCAnalyses

xkcd.com/138

Backup

podio as generator for EDM4hep

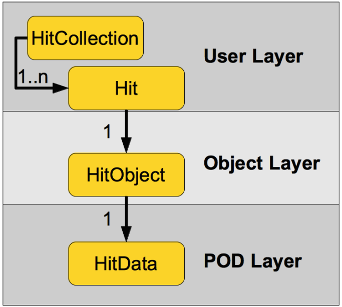
- Original HEP c++ EDMs are heavily Object Oriented
  - Deep inheritance structures
  - Thread-safety can be hard
  - Objects scattered in memory
- Data access can be slow with these approaches
- Use `podio` to generate thread safe code starting from a high level description of the desired EDM
  - Users are isolated from implementation details
- Provide an easy to use interface to the users
  - Users should not need to worry about resource management
  - Treat python as first class citizen and allow "pythonic" usage



AIDASoft/podio

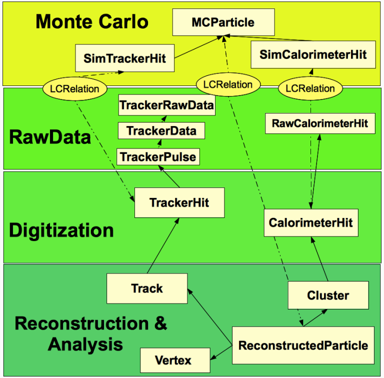
The three layers of podio

- podio favors **composition** over inheritance and uses **plain-old-data (POD)** types wherever possible
- **User Layer** consists of handles to the EDM objects and offers the full functionality
- The **Object Layer** handles resources and references to other objects
- The actual PODs live in the **POD Layer**
- Layered design allows for efficient memory layout and performant I/O implementation
    - Possible to support different formats and I/O backends

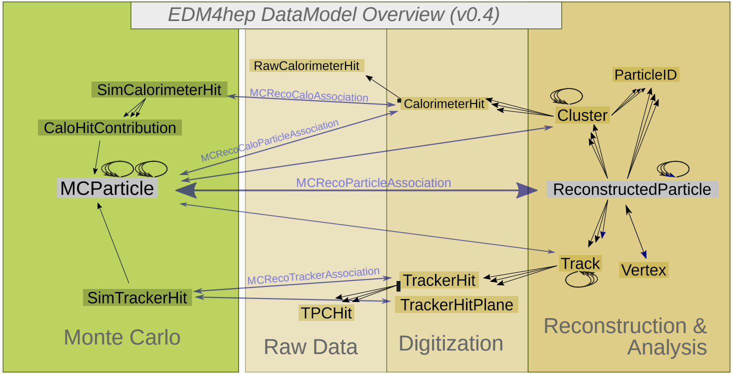


LCIO vs EDM4hep

LCIO



EDM4hep



- Since EDM4hep is based on LCIO the high-level structure is very similar
- Largest differences between the two are due to their implementations
- LCIO has over 15 years of usage. A lot of time to develop tools for it.
    - Not nearly as far with EDM4hep

From LCIO to EDM4hep - The easy parts

LCIO

```cpp
auto* coll = new LCCollectionVec(MCPARTICLE);
auto* mc = new MCParticleImpl;
coll->addElement(mc);

mc->setMass(3.096);

auto* mc2 = static_cast<MCParticle*>(
  coll->getElementAt(0));
auto mass = mc2.getMass();

for (auto* p : mc2.getParents()) { /**/ }
```

EDM4hep

```cpp
auto coll = MCParticleCollection();
auto mc = coll.create();


mc.setMass(3.096);


auto mc2 = coll[0];
auto mass = mc2.getMass();

for (auto p : mc2.getParents()) { /**/ }
```

- The most common use cases work very similarly with mainly syntactic differences
    - pointer vs. value semantics
- Differences in reader/writer handling are “hidden” by framework code

From LCIO to EDM4hep - The parts that still require work

LCIO

```
auto recoMCNav = LCRelationNavigator(
  evt->getCollection("RecoMCTruthLink"));


auto relRecos =
  recoMCNav->getRelatedToObjects(mc);



//
```

EDM4hep

```
auto recoMCAssoc =
  store.get<MCRecoParticleAssocCollection>(
    "RecoMCTruthLink");

std::vector<ReconstructedParticle> relRecos;
for (const auto assoc : relMCAssoc) {
  if (assoc.getSim() == mc) {
    relRecos.push_back(assoc.getRec());
  }
}
```

- LCIO has a 15 years head start in tooling, hiding some of the complexities
- Port the tooling to EDM4hep as we go along and as necessary