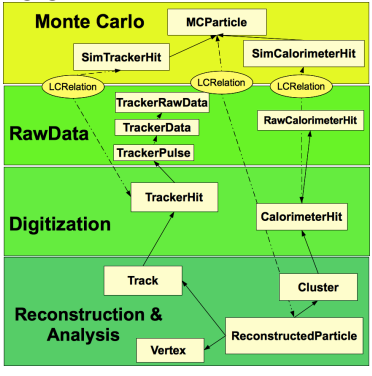
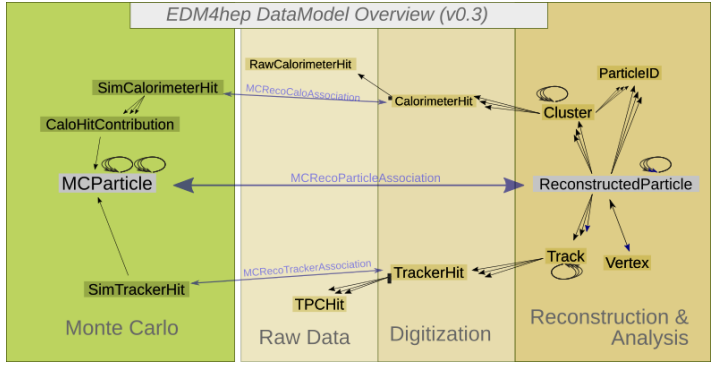
LCIO vs EDM4hep - Overview

LCIO



EDM4hep



- On a high-level almost just a “90 degree flip”
- Some usage differences due to the different choices in implementation
- Some more usage differences due to different levels of tooling

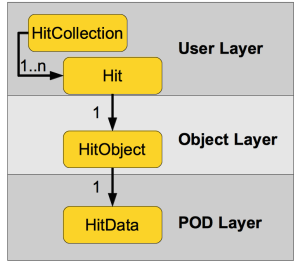
LCIO vs EDM4hep - Implementation

LCIO

- Based on (virtual) inheritance
- Each objects inherits from `LCObject`
- Collections inherit from `LCCollection`
- “pointer semantics”

EDM4hep

- Generated via podio



- Each object is its own (c++) type
- “value semantics” in the User Layer

Getting some experience with LCIO

```
62
63  class DelphesEDM4HepConverter {
64  public:
65
66    DelphesEDM4HepConverter(std::string filename_delphescard);
67
68    DelphesEDM4HepConverter(const std::vector<BranchSettings>& branches,
69                            OutputSettings const& outputSettings, double magFieldBz);
70
71    void process(TTree* delphesTree);
72
73    inline std::unordered_map<std::string_view, podio::CollectionBase*> getCollections() { return m_collections; }
74
75
76    void processParticles(const TClonesArray* delphesCollection, std::string_view const branch);
77    void processTracks(const TClonesArray* delphesCollection, std::string_view const branch);
78    void processClusters(const TClonesArray* delphesCollection, std::string_view const branch);
79    void processJets(const TClonesArray* delphesCollection, std::string_view const branch);
80    void processPhotons(const TClonesArray* delphesCollection, std::string_view const branch) {
81      fillReferenceCollection<Photon>(delphesCollection, branch, "photon");
82    }
83
84    void processMissingET(const TClonesArray* delphesCollection, std::string_view const branch);
85    void processScalarHT(const TClonesArray* delphesCollection, std::string_view const branch);
86
87    void processMuons(const TClonesArray* delphesCollection, std::string_view const branch) {
88      fillReferenceCollection<Muon>(delphesCollection, branch, "muon");
89    }
90    void processElectrons(const TClonesArray* delphesCollection, std::string_view const branch) {
91      fillReferenceCollection<Electron>(delphesCollection, branch, "electron");
92    }
93
94  private:
95
```

key4hep/k4SimDelphes

- Use the structure of the `k4SimDelphes::DelphesEDM4HepConverter` and replace the EDM4hep specific parts with LCIO
- Probably doesn't cover all of the use cases, but hopefully the majority that the "average" user encounters
- Bonus: Now we are (almost) able to do benchmarks that can be directly compared to the podio benchmarks with this

Contents of the next slides

- Each slide tries to show one use case to highlight the differences between the two approaches
- The examples aim to be pretty self-contained and should ideally compile standalone
  - They do not necessarily do meaningful things at runtime, but they should compile
  - In some cases the setup is not complete for space reasons (these will then also not compile without additional setup)
  - All examples combined should give an almost complete guide to either approach
- The intent is to highlight some of the differences in more detail to give some hints to what has to be done during a migration
  - Potentially serve as a starting point for a migration guide
- The list is most probably not complete

Writing infrastructure

```cpp
using namespace lcio;
using lcio::LCIO::MCPARTICLE;
// ------ Setup before event loop ---
auto writer = std::unique_ptr<LCWriter>(
LCFactory::getInstance()->createLCWriter());



// ------ Event loop ----------------
auto* evt = std::make_unique<LCEventImpl>();
auto* coll = new LCCollectionVec(MCPARTICLE);
evt->addCollection(coll, "mc");
auto* mc = new MCParticleImpl;
coll->addElement(mc);

writer->writeEvent(event.get());


// ------ After event loop ----------
writer->close();
```

```cpp
using namespace podio;
using namespace edm4hep;
// ------ Setup before event loop ---
EventStore store;
ROOTWriter writer(outputFile, &store);

auto& coll = store.create<MCParticleCollection>("mc");
writer.registerForWrite("mc");

// ------ Event loop ----------------
auto mc = coll.create();



writer.writeEvent();
store.clearCollections();

// ------ After event loop ----------
writer.finish();
```

Reading infrastructure

```cpp
using namespace lcio;

// ------ Setup before event loop ---
auto reader = std::unique_ptr<LCReader>(
LCFactory::getInstance()->createLCReader());
reader->open(filename);



// ------ Event loop ----------------
auto* evt = reader->readNextEvent();
LCIterator<MCParticle> iter(evt, "mc");
while (auto* mc = iter.next()) {
  // mc --> MCParticle*
}



// ------ After event loop ----------
// nothing to do
```

```cpp
using namespace podio;
using namespace edm4hep;
// ------ Setup before event loop ---
ROOTReader reader;

reader.openFile(filename);
EventStore store;
store.setReader(&reader);

// ------ Event loop ----------------

auto& coll = store.get<MCParticleCollection>("mc");
for (auto mc : coll) {
  // mc --> ConstMCParticle
}

store.clear()
reader.endOfEvent();

// ------ After event loop ----------
// nothing to do
```

Working with data types - getting/setting data members

```cpp
using namespace lcio;

auto* mc = new MCParticleImpl;

// ------ Get and set mass ----------
mc->setMass(3.096);
auto mass = mc->getMass(); // == 3.096
```

```cpp
using namespace edm4hep;

auto mc = MCParticle();

// ------ Get and set mass ----------
mc.setMass(3.096);
auto mass = mc.getMass(); // == 3.096
```

- In LCIO have to remember to use the `Impl` classes, otherwise there are no setter functions (i.e. will fail at compile time)
- If all the data members are the same this migration can almost be done via `s/\./->/g`

Working with the data types - relations

```cpp
using namespace lcio;

auto* mc = new MCParticleImpl;
auto* mc2 = new MCParticleImpl;

// ------ Add a relation ------------
mc->addParent(mc2);

// ------ Use relations -------------
const auto& parents = mc->getParents();

for (const auto p : parents) {
  // p --> const MCParticle*
}

auto p1 = parents[0];
```

```cpp
using namespace edm4hep;

auto mc = MCParticle();
auto mc2 = MCParticle();

// ------ Add a relation ------------
mc.addToParents(mc2);

// ------ Use relations -------------
const auto parents = mc.getParents()

for (auto p : parents) {
  // p --> ConstMCParticle
}

auto p1 = mc.getParents(0);
```

- `lcio::MCParticle::getParents()` retuns a `std::vector<lcio::MCParticle*>`, hence indexed access is supported by default.
- `edm4hep::MCParticle::getParents()` returns a `podio::RelationRange<edm4hep::ConstMCParticle>`, which does not yet support indexed access.

Working with collections

```cpp
using namespace lcio;
using lcio::LCIO::MCPARTICLE;

auto coll = new LCCollectionVec(MCPARTICLE);

// ------ Adding elements -----------
auto* mc = new MCParticleImpl;
coll->addElement(mc);


// ------ Indexed access ------------
auto* p = static_cast<MCParticle*>(
  coll->getElementAt(0));

// ------ Looping -------------------
LCIterator<MCParticle> iter(coll);
while (auto p = iter.next()) { /**/ }
```

```cpp
using namespace edm4hep;


auto coll = MCParticleCollection();

// ------ Adding elements -----------
auto mc = MCParticle();
coll.push_back(mc);

auto mc2 = coll.create(); // preferred!

// ------ Indexed access ------------
auto p = coll[0];


// ------ Looping -------------------

for (auto p : coll) { /**/ }
```

Reference collections

```cpp
using namespace lcio;
using lcio::LCIO::MCPARTICLE;

auto* coll = new LCCollectionVec(MCPARTICLE);
// ... fill collection

// ------ filling ------------------
auto* refColl = new LCCollectionVec(MCPARTICLE);
refColl->setSubset(true);

refColl->addElement(coll->getElementAt(0));

// ------ using ---------------------
LCIterator<MCParticle> iter(refColl);
while (auto p = iter.next()) {
  // nothing special here
  // behaves like a regular collection
}
```

```cpp
using namespace edm4hep;

auto coll = MCParticleCollection();
// ... fill collection

// ------ filling ------------------
auto refColl = MCParticleRefCollection();

auto ref = refColl.create();
ref.setParticle(coll[0]);

// ------ using ---------------------

for (auto r : refColl) {
  auto p = r.getParticle();
  // now p is a MCParticle
}
```

- `edm4hep::MCParticleRef` does not exist!. But the example highlights the issue.
- A LCIO subset collection works basically the same as any other collection
- EDM4hep needs a clumsy workaround at the moment

MC - reco associations - filling

```cpp
using namespace lcio;
using lcio::LCIO::MCPARTICLE;
using lcio::LCIO::RECONSTRUCTEDPARTICLE;

auto* reco = new ReconstructeParticleImpl;
auto* mc = new MCParticleImpl;

// ------ filling -------------------
LCRelationNavigator recoToMC(
  RECONSTRUCTEDPARTICLE, MCPARTICLE);

recoToMC.addRelation(reco, mc);

// ------ writing -------------------
evt->addCollection(
  recoToMC->createLCCollection(),
  "RecoMCTruthLink");
```

```cpp
using namespace edm4hep;

auto reco = ReconstructedParticle();
auto mc = MCParticle();

MCRecoParticleAssociationCollection mcRecoAssocs();

// ------ filling -------------------
auto relation = mcRecoAssocs.create();
relation.setSim(mc);
relation.setReco(reco);


// ------ writing -------------------
// if the mcRecoAssocs collection has been
// created via the EventStore nothing has
// to be done here
```

- For both cases all the objects also have to be actually stored in some collection for this to persist correctly
- LCIO comes with some useful tooling for handling relations, especially for actually using them (next slide)

MC - reco associations - reading

```cpp
using namespace lcio;



LCIterator<ReconstructedParticle> recos(
  evt, "reco"));
auto* reco = recos.next();

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

// ------ reading -------------------
for (auto p :
  recoToMC.getRelatedToObjects(reco)) {
  // p --> MCParticle*
}




// -alignment-
```

```cpp
using namespace edm4hep;
using namespace podio;

auto st = EventStore();
auto& recos =
 st.get<ReconstructedParticleCollection>("reco");
auto reco = recos[0]

auto& mcRecoAssocs =
 st.get<MCRecoParticleAssociationCollection>("assoc");

// ------ reading -------------------
std::vector<ConstMCParticle> relMCs;
for (const auto assoc : mcRecoAssocs) {
  if (assoc.getSim() == mc) {
    relMCs.push_back(mc);
  }
}

for (auto p : relMCs) {
  // p --> ConstMCParticle
}
```

Summary / other differences

LCIO

- Self contained `LCEvent` that gives access to all collections in an event
- Selection of the collections that are persisted can be different for each event
- “Natively” supports subset (or reference) collections which do not store the data of their objects, since that is stored already in another collection
- Utilities exist that make some of the things a bit easier
- ParticleID is managed by the collection it is used in

EDM4hep

- The `EventStore` gives access to the collections of an event
- Collections that need to be persisted need to be registered before the first event is processed (podio limitation at this point)
- No support for reference collections (yet). Need to introduce an additional data type `RecoParticleRef` to work around this.
- No real utility functionality yet and you have to deal with the “bare metal” and all its problems / opportunities
- ParticleID is a separate collection that uses the same relation mechanism as other objects


IF YOU COULD GIVE ME MORE DETAILS
THAT WOULD BE GREAT
makeameme.org

Converter differences

LCIO

- No need to store collections internally, can create them as necessary for each Event

EDM4hep

- Collection pointers stored internally, because they are managed by the `EventStore` (outside the converter)

```
// LCIO main interface
void process(TTree* delphesTree, lcio::LCEventImpl* evt);
// calls a bunch of
using ProcessFunction =
void (DelphesLCIOConverter::*)(const TClonesArray*, lcio::LCEventImpl*, const std::string&);

// EDM4hep main interface
void process(TTree* delphesTree);
// calls a bunch of
using ProcessFunction =
void (DelphesEDM4HepConverter::*)(const TClonesArray*, std::string_view const);
```