Reconstruction in Key4hep using Gaudi

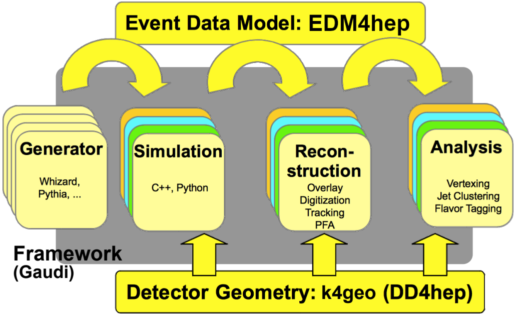
Juan Miguel Carceller j.m.carcell@cern.ch

CERN, EP-SFT

October 23, 2024

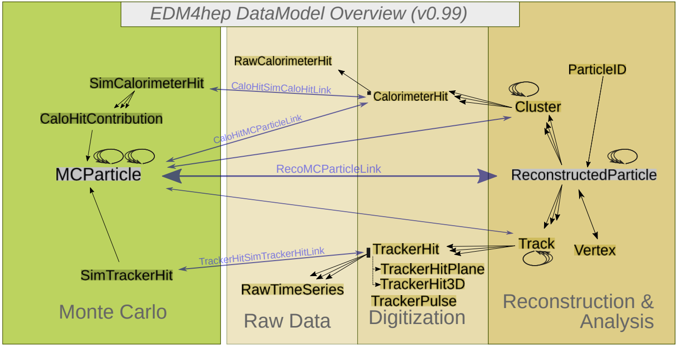
Key4hep

- Turnkey software for future colliders
- Share components to reduce maintenance and development cost and allow everyone to benefit from its improvements
- Complete data processing framework, from generation to data analysis
- Community with people from many experiments: FCC, ILC, CLIC, CEPC, EIC, Muon Collider, etc.
- Open biweekly meetings



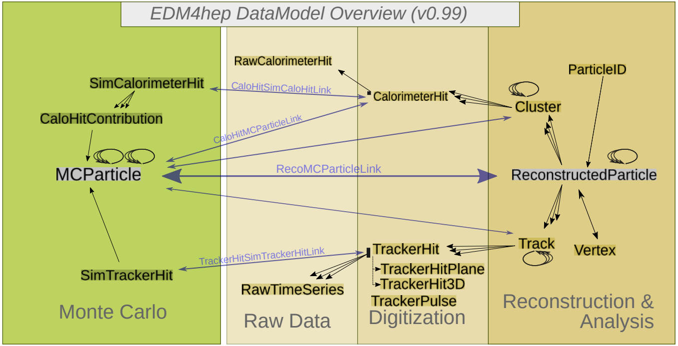
The Event Data Model in Key4hep: EDM4hep

- Data Model used in Key4hep, it is the language that all components must speak
- Classes for physics objects, like `MCParticle`, with possible relations to other objects
- Links between objects
- Objects are grouped in collections, like `MCParticleCollection`



The Event Data Model in Key4hep: EDM4hep

- Data Model used in Key4hep, it is the language that all components must speak
- Classes for physics objects, like `MCParticle`, with possible relations to other objects
- Links between objects
- Objects are grouped in collections, like `MCParticleCollection`



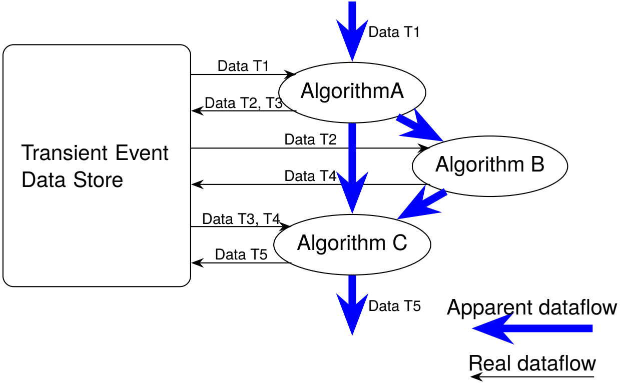
T. Madlener's talk on Thursday

The Key4hep Framework

- Gaudi based core framework:
  - k4FWCore provides the interface between EDM4hep and Gaudi
  - k4Gen for integration with generators
  - k4SimDelphes for integration with Delphes
  - k4MarlinWrapper to call Marlin processors
  - Algorithms for trackers, calorimeters
  - Algorithms ported from the linear collider community
  - …

Gaudi

- Event processing framework
- Algorithms are written in C++ and are configured with steering files in python
- Data is passed between algorithms using a Transient Event Data Store
- Lots of services for histogramming, logging, etc.

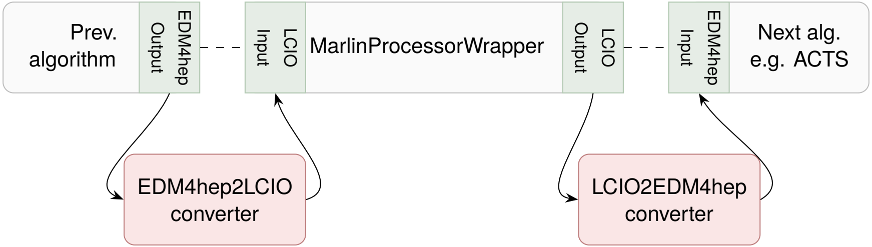


Running reconstruction algorithms

- **1. Algorithms that do not use EDM4hep as their event data model**
- 2. Algorithms that use EDM4hep as their event data model

LCIO Converters

- LCIO is the EDM in the linear collider community
- Marlin processors (algorithms from the linear collider community) can be used in Gaudi using the `MarlinProcessorWrapper`
- EDM4hep input and output can be used; Marlin processors take LCIO input and give LCIO output
- Standalone converter `lcio2edm4hep` to convert files



Running reconstruction algorithms

- 1. Algorithms that do not use EDM4hep as their event data model
- **2. Algorithms that use EDM4hep as their event data model**

Current status

- Most algorithms are implementing `Gaudi::Algorithm`
- Using a custom data service: `PodioDataSvc` and custom algorithms for input and output: `PodioInput` and `PodioOutput`
- Issue: No support for multithreading, unable to use Gaudi `HiveWhiteBoard`

Current status

- Most algorithms are implementing `Gaudi::Algorithm`
- Using a custom data service: `PodioDataSvc` and custom algorithms for input and output: `PodioInput` and `PodioOutput`
- Issue: No support for multithreading, unable to use Gaudi `HiveWhiteBoard`
- New developments in Key4hep
  - Support for functional algorithms
  - Support for multithreading

Functional algorithms in Gaudi

- `Gaudi::Functional` algorithms
  - Multithreading friendly, no internal state
  - Leave details of the framework to the framework

```
class MySum : public TransformAlgorithm<OutputData(const Input1&, const Input2&)> {
     MySum(const std::string& name, ISvcLocator* pSvc)
      : TransformAlgorithm(name, pSvc, {
            KeyValue("Input1Loc", "Data1"),
            KeyValue("Input2Loc", "Data2") },
         KeyValue("OutputLoc", "Output/Data")) { }

    OutputData operator()(const Input1& in1, const Input2& in2) const override {
        return in1 + in2;
    }
```

- Adapted to work in Key4hep with EDM4hep

Functional algorithms in Key4hep

- New service, `IOSvc`, supports multithreading and reading and writing ROOT TTrees and ROOT RNTuples
  - Reading detects automatically if it's a TTree or RNTuple
- Two input/output algorithms: `Reader` and `Writer`
  - `Reader` will ask `IOSvc` to read and then will push itself the collections to the store
  - `Writer` will write the collections to a file
- Take into account collection ownership
- Easily change to multithreading by using Gaudi's `HiveWhiteBoard`

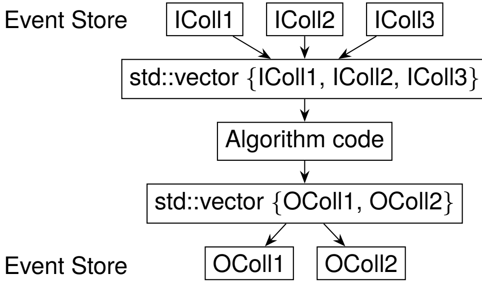
```
svc = IOSvc("IOSvc")
svc.Input = "input.root"
svc.Output = "output.root"
svc.OutputType = "RNTuple"
```

Functional algorithms in Key4hep: Features

- Support for having as input or output an arbitrary number of collections with `std::vector`
- Algorithms should work for all detectors (with different number of subdetectors)
- Reimplemented the `Consumer`, `Transformer` and `MultiTransformer` from Gaudi
  - `k4FWCore::Consumer`, `k4FWCore::Transformer` and `k4FWCore::MultiTransformer`
- Algorithms can now:
  - Pick up multiple collections and store them in a `std::vector` when reading
  - Iterate over the collections and push them individually when pushing a `std::vector`
  - Abstracted into a common function for reading and a common function for pushing

Functional algorithms in Key4hep: Features

- Support for having as input or output an arbitrary number of collections with `std::vector`
- Algorithms should work for all detectors (with different number of subdetectors)
- Reimplemented the `Consumer`, `Transformer` and `MultiTransformer` from Gaudi
  - `k4FWCore::Consumer`, `k4FWCore::Transformer` and `k4FWCore::MultiTransformer`
- Algorithms can now:
  - Pick up multiple collections and store them in a `std::vector` when reading
  - Iterate over the collections and push them individually when pushing a `std::vector`
  - Abstracted into a common function for reading and a common function for pushing



Example with an arbitrary number of collections

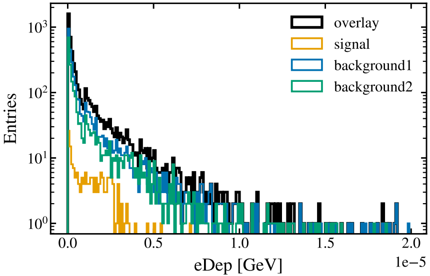
```
struct ExampleAlgorithm final
    : k4FWCore::Transformer<std::vector<edm4hep::MCParticleCollection>(
          const std::vector<const edm4hep::MCParticleCollection*>& input)> {
  ExampleAlgorithm(const std::string& name, ISvcLocator* svcLoc)
      : Transformer(name, svcLoc, {KeyValues("InputCollections", {"MCParticles"})},
                    {KeyValues("OutputCollections", {"MCParticles"})}) {}

  std::vector<edm4hep::MCParticleCollection> operator()(
      const std::vector<const edm4hep::MCParticleCollection*>& input) const override {
    std::vector<edm4hep::MCParticleCollection> outputCollections;
    for (size_t i = 0; i < input.size(); ++i) {
    ...
    }
    return outputCollections;
  }
};
```

- In the steering file multiple collections are passed in a list

```
consumer = ExampleFunctionalConsumerRuntimeCollections(
    "Consumer",
    InputCollections=["MCParticles0", "MCParticles1", "MCParticles2"],
    OutputCollections=["NewMCParticles0", "NewMCParticles1", "NewMCParticles2"],
)
```

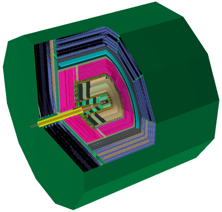
Example algorithm: Overlay

- Ported from the original `Overlay Timing` from iLCSoft
- Reads collections from background files and overlays them on top of the signal
  - `edm4hep::MCParticle`: all particles with a time offset for background
  - `edm4hep::SimTrackerHit`: only hits within a configurable time-window
  - `edm4hep::SimCalorimeterHit`: only if they have any `edm4hep::CaloHitContribution` within a specific time window. Hits with the same cellID are merged



Example algorithm: Overlay

- Feature: the Overlay algorithm takes any number of collections as input
- As many output collections as input collections



```
overlay = OverlayTiming()
overlay.MCParticles = ["MCParticles"]
overlay.BackgroundMCParticleCollectionName = "MCParticle"
overlay.SimTrackerHits = ["VertexBarrelCollection", "VertexEndcapCollection"]
overlay.OutputSimTrackerHits = ["NewVertexBarrelCollection", "NewVertexEndcapCollection"]
...
```

- For a different detector the number and names of the collections can be changed

Other features

- Support for reading and writing metadata from algorithms
- Convenience python wrappers for `IOSvc` and `ApplicationMgr`
  - if input/output is specified for `IOSvc` then a `Reader`/`Writer` algorithm is added. `Reader` and `Writer` don't have data dependencies but the wrapper correctly wraps them in a sequencer
  - To create a `MetadadataSvc` if it's not there
- Backwards compatibility with existing algorithms

Summary and Outlook

- Reconstruction algorithms in Key4hep
  - Using EDM4hep natively
  - Using LCIO, from the Linear Collider community, with the `MarlinWrapper`
- Support added for functional algorithms in Key4hep
  - New IOSvc, with support for multithreading and reading and writing TTrees and RNTuples
  - Motivated by the lack of multithreading support
  - Algorithms support reading and pushing arbitrary number of collections
  - New algorithms are being implemented as functional algorithms
  - Work on integrations with other software, like ACTS and Pandora

Backup

Past (and present)

- Using exclusively `GaudiAlg`
- Custom `DataHandle` class
- A custom `DataWrapper` is pushed to the store, thin wrapper of a pointer to a collection
- Two algorithms for IO: `PodioInput` and `PodioOutput` and an IO service: `PodioDataSvc`
- How it works:
  - `PodioDataSvc` holds a `podio::Frame` (Frame = event) and some metadata. This `Frame` owns all the collections
  - `PodioInput` will ask `PodioDataSvc` to read and register the collections
  - [Algorithm execution]. . .
  - `PodioOutput` will use the `podio::Frame` to write the collections to a file (only those that we want to write)
- Multiple issues
  - Not designed for multithreading
  - `PodioDataSvc` isn't an implementation of `IHiveWhiteBoard`

podio::Frame

- The Frame (from podio) is a data container where collections can be stored
- Support for multithreading
- Typically represents an event but can be anything else
- A backend decides how it is written to a file (ROOT TTrees most of the time, but can also be RNTuples)
- Takes ownership of the collections

Simple interface with `get` and `put`

```
frame.get("MCParticleCollection");
frame.put(std::move(coll), "NewCollection");
```

Also in python:

```
from podio.root_io import Reader
reader = Reader('myfile.root')
events = reader.get('events')
for frame in events:
    coll = frame.get('MCParticleCollection')
```

Functional algorithms

- Example: producer of an arbitrary number of collections

```
struct ExampleFunctionalProducerRuntimeCollections final
    : k4FWCore::Producer<std::vector<edm4hep::MCParticleCollection>()> {
  ExampleFunctionalProducerRuntimeCollections(const std::string& name, ISvcLocator* svcLoc)
      : Producer(name, svcLoc, {}, {KeyValues("OutputCollections", {"MCParticles"})}) {}

  std::vector<edm4hep::MCParticleCollection> operator()() const override {
    const auto locs = outputLocations();
    std::vector<edm4hep::MCParticleCollection> outputCollections;
    for (size_t i = 0; i < locs.size(); ++i) {
      info() << "Creating collection " << i << endmsg;
      auto coll = edm4hep::MCParticleCollection();
      coll.create(1, 2, 3, 4.f, 5.f, 6.f);
      coll.create(2, 3, 4, 5.f, 6.f, 7.f);
      outputCollections.emplace_back(std::move(coll));
    }
    return outputCollections;
  }
};
```

Functional algorithms in Key4hep: IOSvc

- Example of a steering file

```
from Gaudi.Configuration import INFO
from Configurables import ExampleFunctionalTransformer
from Configurables import EventDataSvc
from k4FWCore import ApplicationMgr, IOSvc

svc = IOSvc("IOSvc")
svc.Input = "input.root"
svc.Output = "output.root"

transformer = ExampleFunctionalTransformer(
    "Transformer", InputCollection=["MCParticles"], OutputCollection=["NewMCParticles"]
)

mgr = ApplicationMgr(
    TopAlg=[transformer],
    EvtSel="NONE",
    EvtMax=-1,
    ExtSvc=[EventDataSvc("EventDataSvc")],
    OutputLevel=INFO,
)
```

Functional algorithms in Key4hep: IOSvc

- For multithreading, add

```
evtslots = 6
threads = 6

whiteboard = HiveWhiteBoard("EventDataSvc", EventSlots=evtslots)
slimeventloopmgr = HiveSlimEventLoopMgr("HiveSlimEventLoopMgr")
scheduler = AvalancheSchedulerSvc(ThreadPoolSize=threads)
```

- Pass it to the `ApplicationMgr`

```
mgr = ApplicationMgr(
    TopAlg=[transformer],
    EvtSel="NONE",
    EvtMax=-1,
    ExtSvc=[whiteboard],
    EventLoop=slimeventloopmgr,
    OutputLevel=INFO,
)
```

Functional algorithms in Key4hep: backwards compatibility

- Existing algorithms are based on `DataHandle` and `PodioDataSvc` for reading and writing
- Question: can we mix old `DataHandle` based algorithms with new functional algorithms?
- Code has been implemented
  - `DataHandle` based algorithms can fetch data produced by functional algorithms
  - Functional algorithms can fetch data produced by `DataHandle` based algorithms
- Mixing of algorithms is possible
- Multithreading won't work unless using the new IOSvc

Functional algorithms in Key4hep: Example

- Using `k4FWCore::Consumer`
- Does not have any outputs

```
struct ExampleFunctionalConsumer final : k4FWCore::Consumer<void(const edm4hep::MCParticleCollection& input)> {
  ExampleFunctionalConsumer(const std::string& name, ISvcLocator* svcLoc)
      : Consumer(name, svcLoc, KeyValues("InputCollection", {"MCParticles"})) {}

  void operator()(const edm4hep::MCParticleCollection& input) const override {
    if (input.size() != 2) {
      throw std::runtime_error("Wrong size of the MCParticle collection");
    }
  }
};
```

Functional algorithms in Key4hep: Example

- Producer, does not have any inputs

```
struct ExampleFunctionalProducer final : k4FWCore::Producer<edm4hep::MCParticleCollection()> {
  ExampleFunctionalProducer(const std::string& name, ISvcLocator* svcLoc)
      : Producer(name, svcLoc, {}, KeyValues("OutputCollection", {"MCParticles"})) {}

  edm4hep::MCParticleCollection operator()() const override {
    auto coll = edm4hep::MCParticleCollection();
    coll.create(1, 2, 3, 4.f, 5.f, 6.f);
    coll.create(2, 3, 4, 5.f, 6.f, 7.f);
    return coll;
  }
};
```