Reconstruction in Key4hep using Gaudi

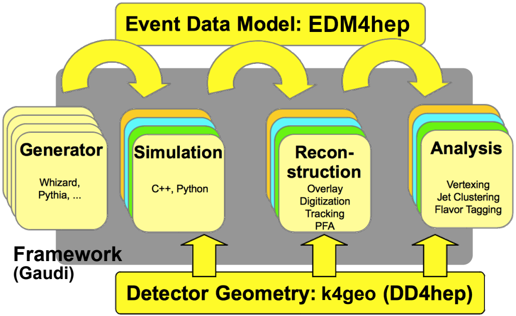
Juan Miguel Carceller

CERN, EP-SFT

October 23, 2024

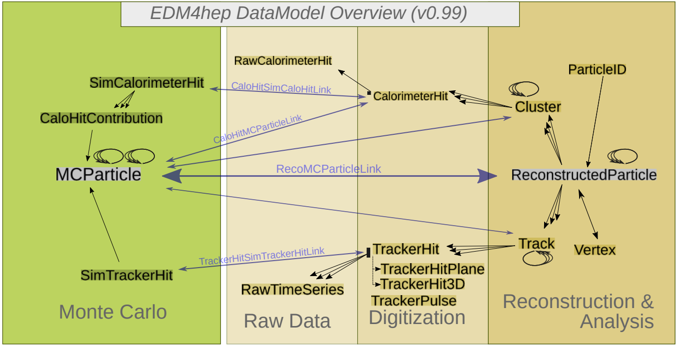
Key4hep

- 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 talks with all stakeholders



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 `MCParticleColleciton`



Podio

- Podio is the tool used to generate the C++ code for EDM4hep
- The specification is written in YAML

```yaml
edm4hep::MCParticle:
  Description: "The Monte Carlo particle - based on the lcio::MCParticle."
  Members:
    - int32_t PDG                                   // PDG code of the particle
    - int32_t generatorStatus                       // status of the particle as defined by the generator
    - int32_t simulatorStatus                       // status of the particle from the simulation program
    - float charge                                  // particle charge
    - float time [ns]                               // creation time of the particle in wrt. the event
    - double mass [GeV]                             // mass of the particle
    - edm4hep::Vector3d vertex [mm]                 // production vertex of the particle
    - edm4hep::Vector3d endpoint [mm]               // endpoint of the particle
    - edm4hep::Vector3d momentum [GeV]              // particle 3-momentum at the production vertex
    - edm4hep::Vector3d momentumAtEndpoint [GeV]    // particle 3-momentum at the endpoint
    - edm4hep::Vector3f spin                        // spin (helicity) vector of the particle
    - edm4hep::Vector2i colorFlow                   // color flow as defined by the generator
  OneToManyRelations:
    - edm4hep::MCParticle parents // The parents of this particle
    - edm4hep::MCParticle daughters // The daughters this particle
```

- Podio uses Jinja templates to transform this to C++ code

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

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:

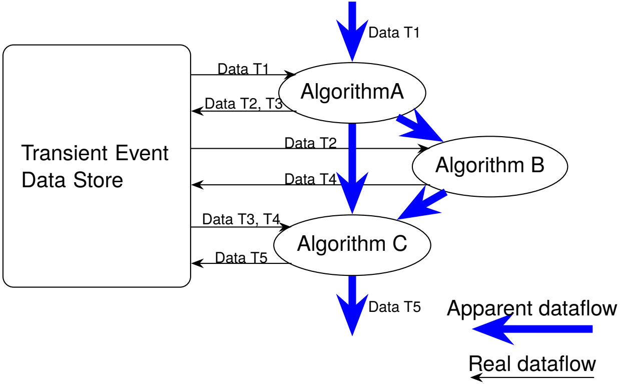
```
from podio.root_io import Reader
reader = Reader('myfile.root')
events = reader.get('events')
for frame in events:
    coll = frame.get('MCParticleCollection')
```

The Key4hep Framework

- Gaudi based core framework:
  - k4FWCore provides the interface between EDM4hep and Gaudi
  - k4Gen for integration with generators
  - k4SimGeant4 for integration with Geant4
  - k4SimDelphes for integration with Delphes
  - k4MarlinWrapper to call Marlin processors
  - …

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.



Gaudi in Key4hep

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 will detect 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
- Collections are wrapped in a `std::unique_ptr` and pushed to the store
  - If there is a `podio::Frame`, some collections may be removed from the store to avoid double deletions
- Easily change to multithreading by using the Gaudi's `HiveWhiteBoard`

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: Features

- Support for having as an input an arbitrary number of collections through a `std::vector` of collections was required
- Reimplemented the `Consumer`, `Transformer` and `MultiTransformer` from Gaudi
  - `k4FWCore::Consumer`, `k4FWCore::Transformer` and `k4FWCore::MultiTransformer`
- Algorithms have now to:
  - 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: 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;
  }
};
```

Example with an arbitrary number of collections

- Example: consumer of an arbitrary number of collections

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

  void operator()(const std::vector<const edm4hep::MCParticleCollection*>& input) const override {
    if (input.size() != 3) {
      throw std::runtime_error("Wrong size of the input vector, expected 3, got " + std::to_string(input.size()));
    }
  }
};
```

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

```
consumer = ExampleFunctionalConsumerRuntimeCollections(
    "Consumer",
    InputCollection=["MCParticles0", "MCParticles1", "MCParticles2"],
    Offset=0,
)
```

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: Usage

- Most existing algorithms use `DataHandles` and have internal state, so they can't be run multithreaded
- More algorithms being implemented as functional algorithms
- Background Overlay: overlay background events on top of signal events, takes an arbitrary number of input collections and returns an arbitrary number of collections
- Other ported algorithms from the linear collider community: digitizer, Pandora algorithms for Particle Flow algorithm
- Algorithms for trackers and calorimeters used by FCC people

Summary

- New IOSvc, with support for multithreading and reading and writing TTrees and RNTuple
- Support added for functional algorithms in Key4hep
  - New algorithms are being implemented as functional algorithms
  - Algorithms support reading and pushing arbitrary number of collections
  - Already being used in several places

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`

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;
  }
};
```