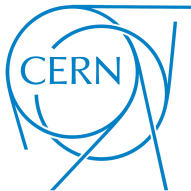


Gaudi in Key4hep

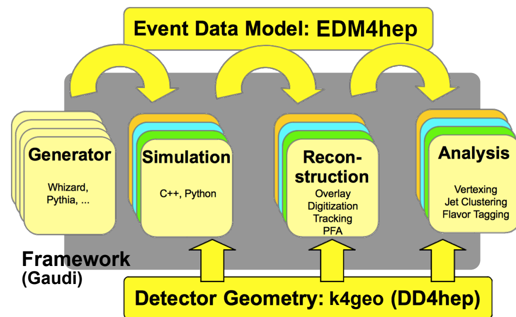
Juan Miguel Carceller

CERN

July 10, 2024

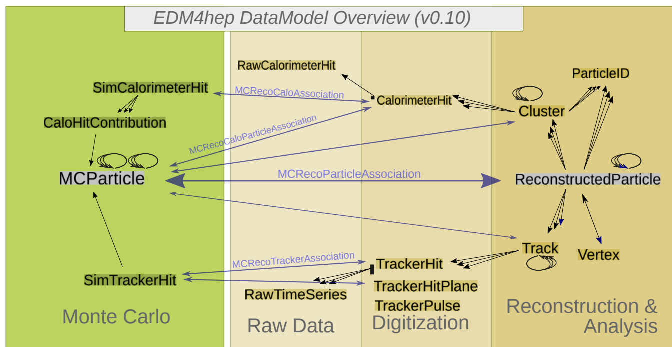
Key4hep

- Turnkey software for future accelerators
- 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 different experiments: FCC, CEPC, CLIC, EIC, ILC, Muon Collider, etc.
- Open biweekly talks with all stakeholders



The Key4hep Event Data Model: 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
- Associations between objects
- Objects are group in collections, like `MCParticleColleciton`



Podio

- Podio is 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 template 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 files with ROOT TTrees most of the time, but can also be RNTuple)
- 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 in Key4hep

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

- Recently added support for functional algorithms
- New service, `IOSvc`
- Two algorithms `Reader` and `Writer`
    - `Reader` will ask `IOSvc` to read (locked) and then will push itself the collections
    - `Writer` will write the collections to a file
- Collections are wrapped in a `std::shared_ptr<podio::CollectionBase>` and pushed to the store
- Use ‘EventDataSvc‘ directly or ‘HiveWhiteBoard‘ instead of having our own implementation of the data service

Functional algorithms

- Nice interface, the existence of `std::shared_ptr` is hidden for users

```
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) {
      fatal() << "Wrong size of MCParticle collection, expected 2 got " << input.size() << endmsg;
      throw std::runtime_error("Wrong size of MCParticle collection");
    }
  }
};
```

Functional algorithms

- Nice interface, the existence of `std::shared_ptr` is hidden for users

```cpp
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;
  }
};
```

Functional algorithms

- Requested feature: have as input and / or output an arbitrary (known at runtime) number of collections
- Example use-case: Overlay algorithm
- Reimplementation of `Consumer`, `Transformer` and `Multitransformer` that use a vector with actual collections
- In the end not so much work, since the way inputs are read or outputs are written is the same
  - Extracted to a common function that all use

Functional algorithms

- 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 map, expected 3, got " + std::to_string(input.size()));
    }
  }
};
```

Functional algorithms

- Example: producer of an arbitrary number of collections

```cpp
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;
  }
};
```

Summary

- Previously using `GaudiAlg` and `PodioDataSvc` for reading and writing
  - Moved many algorithms to use `Gaudi::Algorithm`
  - Still using GaudiTool from GaudiAlg
- Support added for functional algorithms
- Reimplemented `Consumer`, `Transformer` and `MultiTransformer` to support an arbitrary number of collections
  - No plans on reimplementing others, no usage for example for a Filter that can filter an arbitrary number of collections