Examples for Data Model Usage

Wouter Deconinck

What is a Data Model?

- **The set of standardized data structures that we collectively agree to use to pass information between reconstruction algorithms**
- Example: The information we talk about when we say “a hit in a tracking detector,” such as channel number, energy deposition, time, position, etc…

What is **not** included in this discussion of the data model?

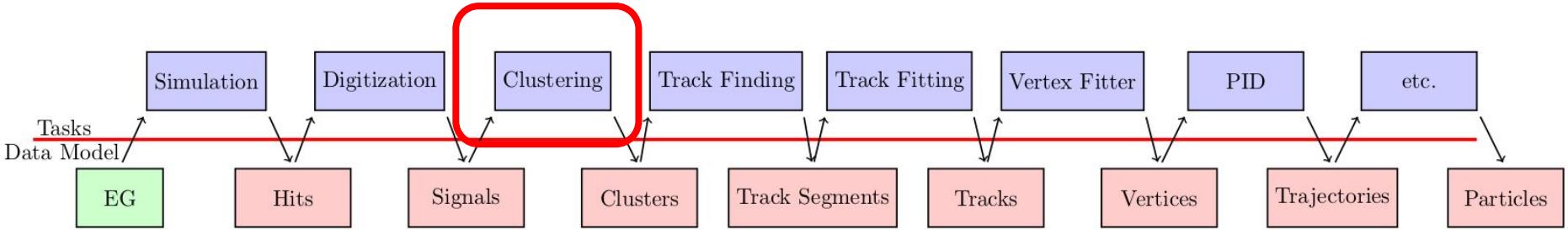
- Decisions about input/output file/memory formats, physical data storage medium: **we aim for flexibility through our choice of data model**.
- Example: Our choice of data model does not require storage in ROOT files (but can be written to ROOT files, HDF5 files, protobuf, and many others), does not require C++ (or Python), does not require row-oriented memory layouts (may allow for GPU processing), etc…

The Motivation Behind a Standardized Data Model

Use of **standard interfaces** between individual simulation, reconstruction, and analysis tasks **creates modularity** that allows **easy exchange of components**.

Example: Multiple clustering algorithms can be swapped out, as long as they adhere to the data model interfaces.

This modularity extends beyond the EIC, since many data structures are common across collider experiments worldwide.



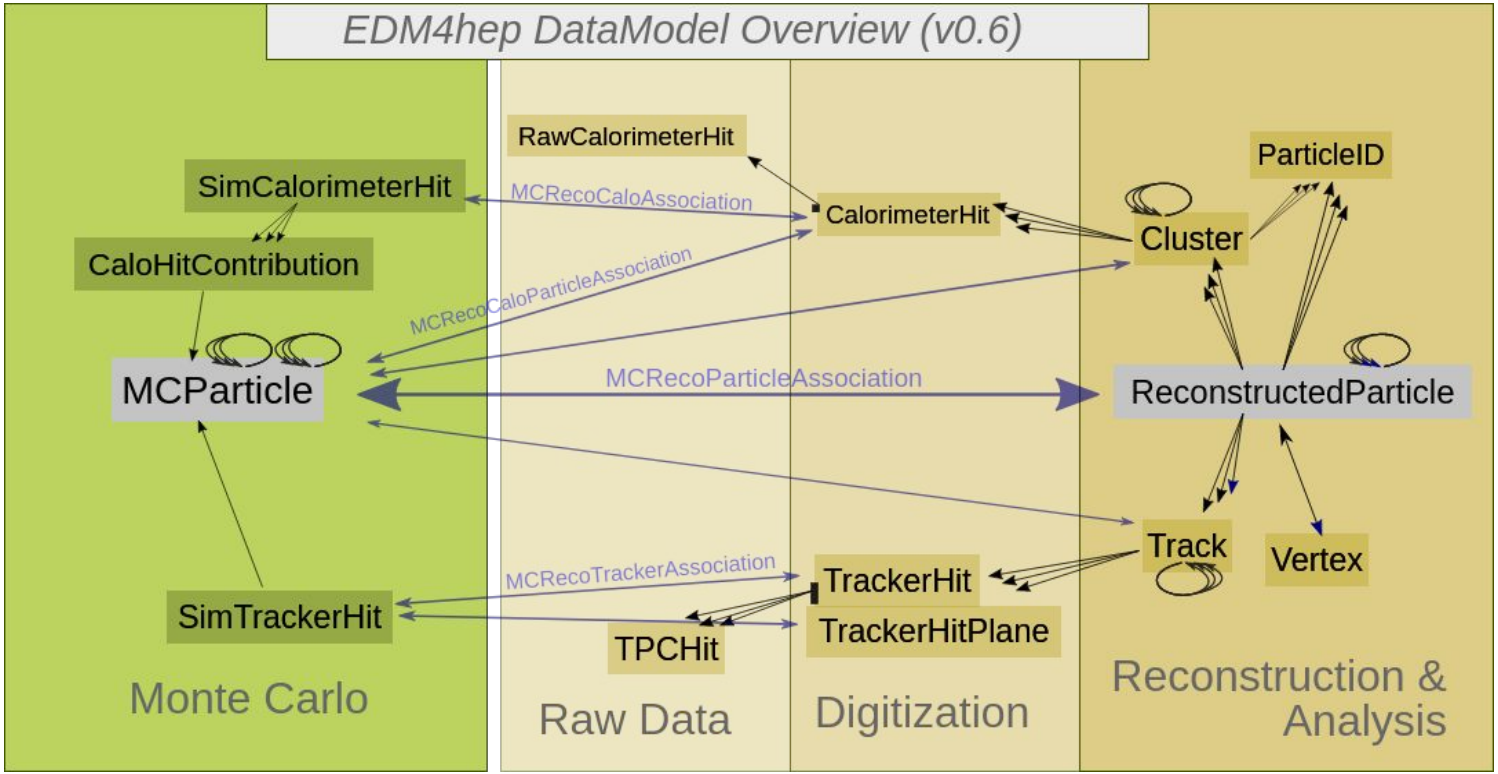
Podio: Plain-Old-Data I/O

Example: human-readable data model definition

```
edm4hep::SimTrackerHit:
  Description: "Simulated tracker hit"
  Author : "F.Gaede, DESY"
  Members:
    - uint64_t cellID    // ID of the sensor that created this hit
    - float EDep         // energy deposited in the hit [GeV].
    - float time         // proper time of the hit in the lab frame in [ns].
    - float pathLength   // path length of the particle in the sensitive material.
    - int32_t   quality  // quality bit flag.
    - edm4hep::Vector3d position // the hit position in [mm].
    - edm4hep::Vector3f momentum // the 3-momentum of the particle at the hits position in [GeV]
  OneToOneRelations:
    - edm4hep::MCParticle MCParticle // MCParticle that caused the hit.

  #etc
```

EDM4hep: Event Data Model for HEP



Ref: https://cern.ch/edm4hep

EDM4eic: Adding EIC Physics to EDM4hep

By request of the EIC community, podio supports extensions of data models.

We have been using this to define data types on top of EDM4hep.

```
edm4eic::InclusiveKinematics:
  Description: "Kinematic variables for DIS events"
  Author: "S. Joosten, W. Deconinck"
  Members:
    - float             x                 // Bjorken x (Q2/2P.q)
    - float             Q2                // Four-momentum transfer squared [GeV^2]
    - float             W                 // Invariant mass of final state [GeV]
    - float             y                 // Inelasticity (P.q/P.k)
    - float             nu                // Energy transfer P.q/M [GeV]
  OneToOneRelations:
    - edm4hep::ReconstructedParticle e    // Associated scattered electron (if identified)
```

Design Criteria of our Data Model

- The **smallest** number of **unique** data structures to represent our data.
- Internal consistency, predictability: units, naming scheme, components vs data types, relations vs vectors, truth associations.
- Minimal redundancy, minimal repetition.
- Clear and unambiguous definitions.

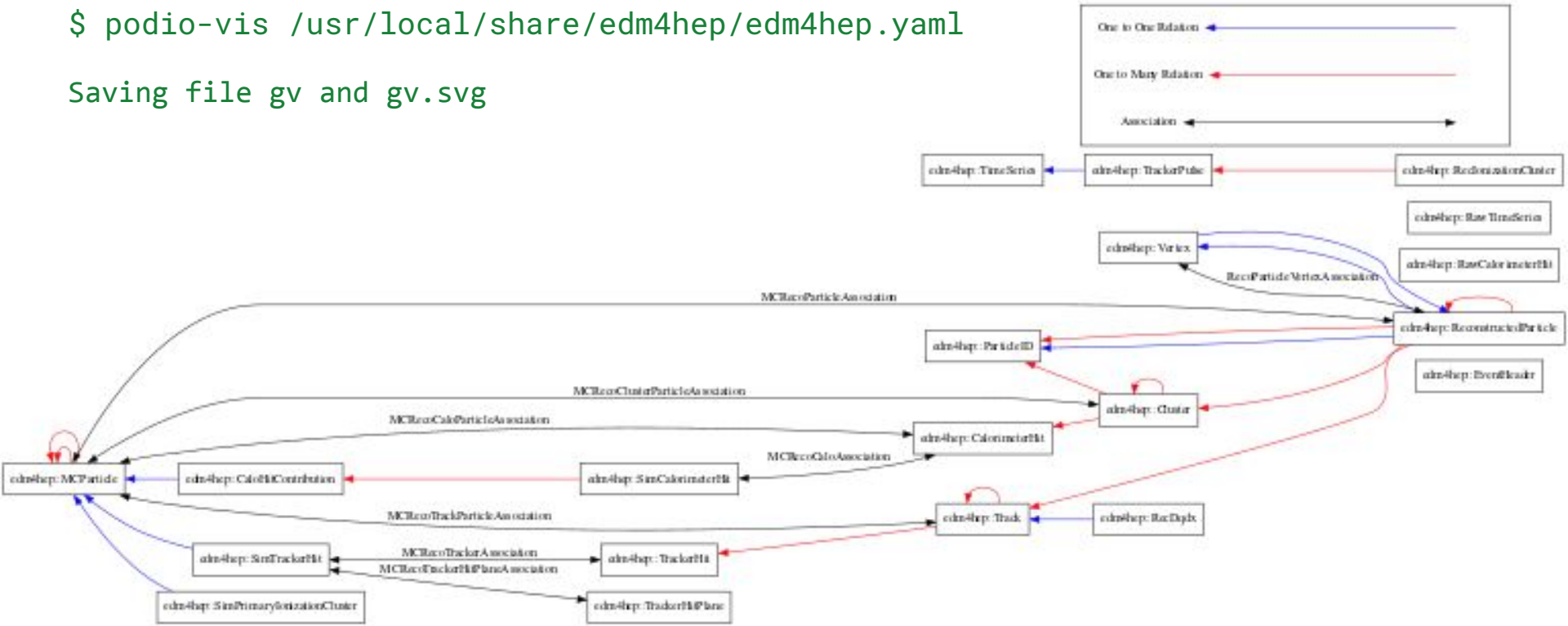
Managing Data Objects in Collections

- Objects should be created as part of collections:
  - Only objects in collections have `getObjectID()` (which refers to `collectionID` and `index`).
  - Ownership of objects in collections is clear.
- Objects in collections should be considered immutable.
  - Collections represent input data, which can be on an immutable medium.
  - Collections with output data can be added to, but once sent off they are immutable.
- References to objects in other collections are by `collectionID` and `index`.
  - `collectionID` is 32-bit hash of name, e.g. `2714477136` for `MCParticles`.
  - `index` is a signed int (-1 for untracked objects, outside collections).
- References are either called *Relations* (one-to-one, one-to-many) or *Associations* when data type contains two one-to-one relations and a weight.

Generic Tools to Access and Debug

```
$ podio-vis /usr/local/share/edm4hep/edm4hep.yaml

Saving file gv and gv.svg
```



Generic Tools to Access and Debug

```
$ podio-dump file.edm4hep.root

input file:
sim_dis_18x275_minQ2=1000_craterlake.ed
m4hep.root

datamodel model definitions stored in
this file: edm4hep

Frame categories in this file:
Name                    Entries
------------------------------
runs                    1
metadata                1
events                  100
```

```
$ edm4hep2json --events 1,2,10
--coll-list MCParticles
sim_dis_18x275_minQ2=1000_craterlak
e.edm4hep.root

$ jq '.[].MCParticles.collection'
sim_dis_18x275_minQ2=1000_craterlak
e.edm4hep.json
```

Supports primarily `edm4hep` data model, but approach could be replicated in `edm4eic`.

Components and Data Types

Components have only data in them (struct), not possible by itself inside a collection:

```
components:
  Vector3f:
    Members:
      - float x
      - float y
      - float z
```

Data types can refer to components or other data types, and are stored in collections:

```
datatypes:
  TrackerHit:
    Members:
    - Vector3f position
```

Python

```
import edm4hep
v = edm4hep.Vector3f()
print(v.x)

h = edm4hep.TrackerHit
v = h.getPosition()
print(v.x)
```

Vector Members

Vector members are arrays of data or components (not often in simulated data).

```
edm4hep::EventHeader:
  VectorMembers:
    - double weights // edm4hep v1.0

edm4eic::ProtoCluster:
  VectorMembers:
    - float weights
```

These are `std::vector`-like quantities (variable length), and technically break POD-ness. They are harder to feed to fixed-length AI/ML training algorithms.

C++ (with edm4hep v1.0)

```
#include <edm4hep/EventHeaderCollection.h>
#include <podio/Frame.h>
#include <podio/ROOTFrameReader.h>

podio::ROOTFrameReader r;
r.openFile(argv[1]);
auto f =
podio::Frame(r.readNextEntry(podio::Category:
:Event));
auto& h =
f.get<edm4hep::EventHeaderCollection>("EventH
eader");
auto& w = h.at(0).getWeights();
std::cout << w.size();
```

One-to-One Relations

Relations are references to one objects in another collection.

```
edm4hep::SimTrackerHit:
  OneToOneRelations:
    - edm4hep::MCParticle MCParticle
      // (‘particle’ in edm4hep v1.0)
```

E.g. `B0TrackerHits` contains `edm4hep::SimTrackerHit`, and the one-to-one relations is in:
`_B0TrackerHits_MCParticle.collectionID`
`_B0TrackerHits_MCParticle.index`

Transparent access when using `podio` tools.

C++

```
#include <edm4hep/SimTrackerHitCollection.h>
#include <podio/Frame.h>
#include <podio/ROOTFrameReader.h>

podio::ROOTFrameReader r;
r.openFile(argv[1]);
auto f =
podio::Frame(r.readNextEntry(podio::Category:
:Event));
auto& h =
f.get<edm4hep::SimTrackerHitCollection>("B0Tr
ackerHits");
std::cout <<
h.at(0).getMCParticle().getPDG();
```

One-to-Many Relations

Relations are references to multiple objects in other collections. This requires an intermediate table.

```
edm4hep::MCParticle:
    OneToManyRelations:
      - edm4hep::MCParticle daughters
```

All different `daughters` are referenced in `_MCParticles_daughters`, as for one-to-one relations.

`MCParticles.daughters_begin` and `MCParticles.daughters_end` indicate range of references that should be used.

Python

```
import edm4hep
from podio import Frame
from podio.reading import get_reader

reader = get_reader("file.edm4hep.root")
frames = reader.get("events")
frame = frames[0]
p = frame.get("MCParticles")
d = p[0].getDaughters()
print(d.size())
```

```
auto& d = p.at(0).getDaughers();
std::cout << d.size();
```

Direct Access through ROOT Storage Layer

Requires explicitly resolving the relations collection IDs and index.

Python

```
import uproot as up

events =
up.open(“file.edm4hep.root”)["events"]
daughters_begin =
events["MCParticles.daughters_begin"].a
rray()
```