EDM4hep - The common event data model for the Key4hep project

Thomas Madlener
for the Key4hep developers

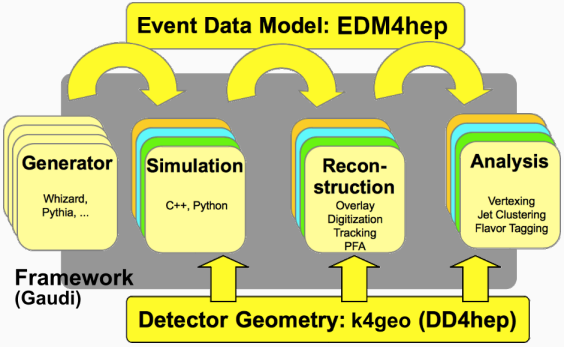
CHEP 2024

Oct 24, 2024

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under grant agreement No 101004761.

The EDM at the core of HEP software

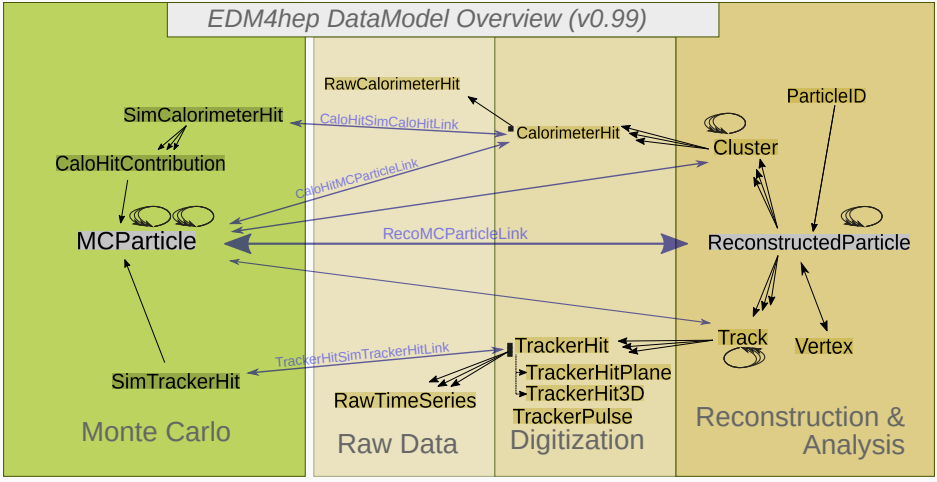
- Key4hep aims to provide a common SW stack for future collider projects



See J. Carcellers talk for Key4hep reco overview

- Different components of experiment software have to exchange data
- The event data model defines structure and language - also for users

EDM4hep - The EDM for Key4hep

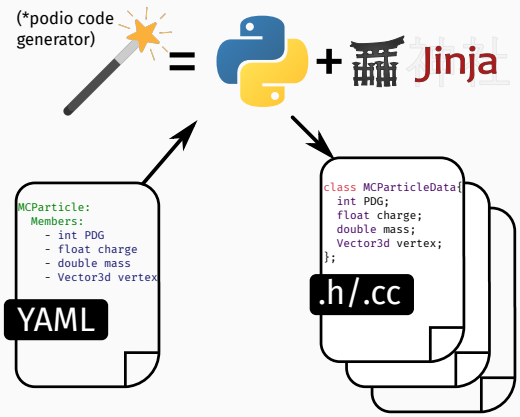


- Heavily inspired by *LCIO* and *FCC-edm*
- Focus on usability in reconstruction and analysis

key4hep/EDM4hep
edm4hep.web.cern.ch

The podio EDM toolkit

- Implementing a performant event data model (EDM) is non-trivial
- Use `podio` to generate code starting from a high level description
- Provide an easy to use interface to the users
- **v1.0 available!** 🎉

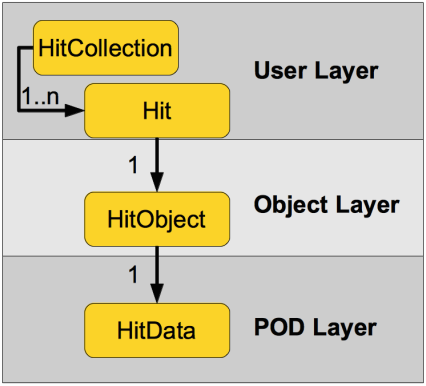


AIDASoft/podio

key4hep.web.cern.ch/podio

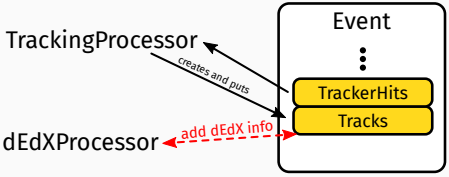
The three layers of podio

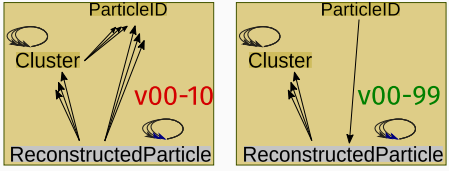
- podio favors **composition over inheritance** and uses **plain-old-data (POD)** types wherever possible
- Layered design allows for efficient memory layout and performant I/O implementation



Shedding some LCIO heritage

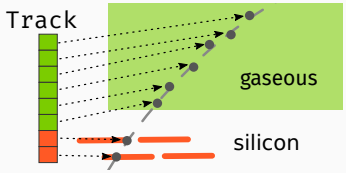
- LCIO designed without focus on multithreading
  - Some inconsistencies in mutability concept
- EDM4hep much more stringent
  - *Can't mutate what isn't yours*
- Harmonized EDM4hep to have a consistent mutability concept
  - Overhauld some relations (directions)
  - Moved data members into dedicated types
- Utilities to simplify navigation
- **Breaking changes, no schema evolution!**






```cpp
auto pidH = PIDHandler::from(event, metadata);
// Get all related ParticleIDs
auto pids = pidH.getPIDs(reco);
// For a specific ParticleID algorithm
auto algoType = pidH.getAlgoType("TOF").value();
auto tofPID = pidH.getPID(reco,
                          algoType).value();
```

Interface types and their use in EDM4hep




```yaml
interfaces:
  edm4hep::TrackerHit:
    Types: [edm4hep::TrackerHit3D, edm4hep::TrackerHitPlane]
    Members:
      - edm4hep::Vector3f position [mm] // hit position

datatypes:
  edm4hep::Track:
    OneToManyRelations:
      - edm4hep::TrackerHit trackerHits // hits of this track
```

```python
auto track = edm4hep::Track{};
track.addHit(edm4hep::TrackerHit3D{});
track.addHit(edm4hep::TrackerHitPlane{});

const auto hits = track.getHits();
hits[0].isA<edm4hep::TrackerHit3D>();  // <-- true
hits[0].as<edm4hep::TrackerHit3D>();   // <-- "cast back"
hits[1].isA<edm4hep::TrackerHit3D>();  // <-- false
hits[1].as<edm4hep::TrackerHit3D>();   // <-- exception!
```

- General interface can be useful to “gloss over some details”
- Handles prevent inheritance based approach
  - Pointers break consistency
  - No base class to inherit from
- Use *type erasure* for implementation
- Introduce *interfaces* as new category in YAML definition
  - Define desired functionality
  - Use like normal *datatypes*
  - No collections
  - “Casting back” is possible

Links (formerly known as Associations)

- *External links* useful for Sim - Reco bridging
  - Some boilerplate in EDM4hep
- Switch to C++ template implementation
  - Arbitraty links in memory
  - Streamlined I/O datatypes
  - Improved user interface
  - Simpler utilities / tools
- Introduce `links` category for YAML to have full EDM in YAML
- Will be **transparent switch for users!**

```yaml
datatypes:
  MCRecoParticleLink:
    Description: "..."
    Authors: "..."
    Members:
      - float weight // weight
    OneToOneRelations:
      - RecoParticle from // rec
      - MCParticle to     // sim
```

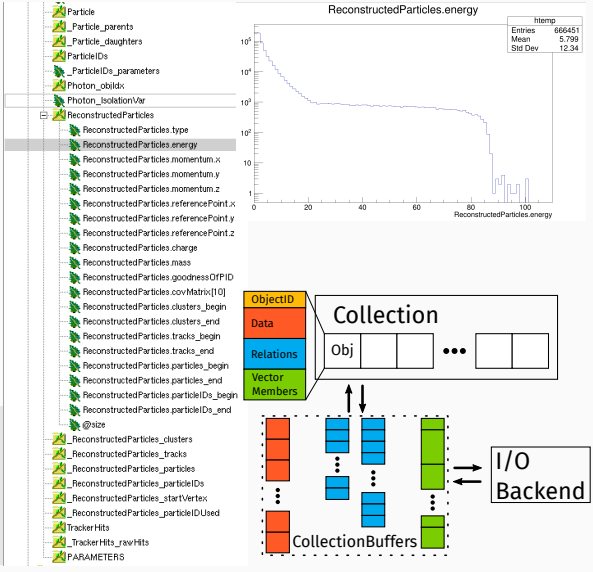
```yaml
links:
  MCRecoParticleLink:
    Description: "..."
    Authors: "..."
    From: RecoParticle
    To: MCParticle
```

```cpp
#include <podio/LinkCollection.h>

// Link arbitrary podio generated datatypes
using McRecoParticlLinkCollection = podio::LinkCollection<
    edm4hep::ReconstructedParticle,
    edm4hep::MCParticle>;

// Enable I/O
PODIO_DECLARE_LINK(edm4hep::ReconstructedParticle,
                   edm4hep::MCParticle)

// Conventional access
auto mcP = link.getFrom();

// Templated / tuple like access
mcP = link.get<edm4hep::MCParticle>();
mcP = link.get<2>();
auto& [rp, mp, w] = link; // <-- structured bindings!
```

Reminder: Basics of podIO

- Default ROOT backend with effectively flat `TTree` / `RNTuple`
  - Files can be used **without EDM library**
  - See P. Matos poster for Julia
  - Prototype support in coffea
- Relation handling done in podio generated code during reading / writing

```
d = ROOT.RDataFrame('events', 'events.root')
h = (d.Define('abs_pdg', 'abs(Particle.PDG)')
      .Define('mu_sel', 'abs_pdg == 13')
      .Define('mu_px',
              'Particle.momentum.x[mu_sel]')
      .Histo1D('mu_px'))
h.DrawCopy()
```



RDataSource for podio generated EDMs

- Relation handling cumbersome on branches alone
  - Need to know quite a few podio details
- **podio::DataSource** allows to use user facing classes
  - Provides schema evolution through podio
  - Supports reading from all backends
- Still some optimization potential

```
auto get_mothers(RVec<MCParticleData> mcps, RVec<int> idcs) {
  RVec<RVec<MCParticleData>> result{};
  for (const auto& mc : mcps) {
    RVec<MCParticleData> mothers{}
    for (auto i = mc.parents_begin; i != mc.parents_end; ++i) {
      mothers.push_back(mcps[idcs[i]]);
    }
    result.push_back(mothers);
  }
  return result;
}
```

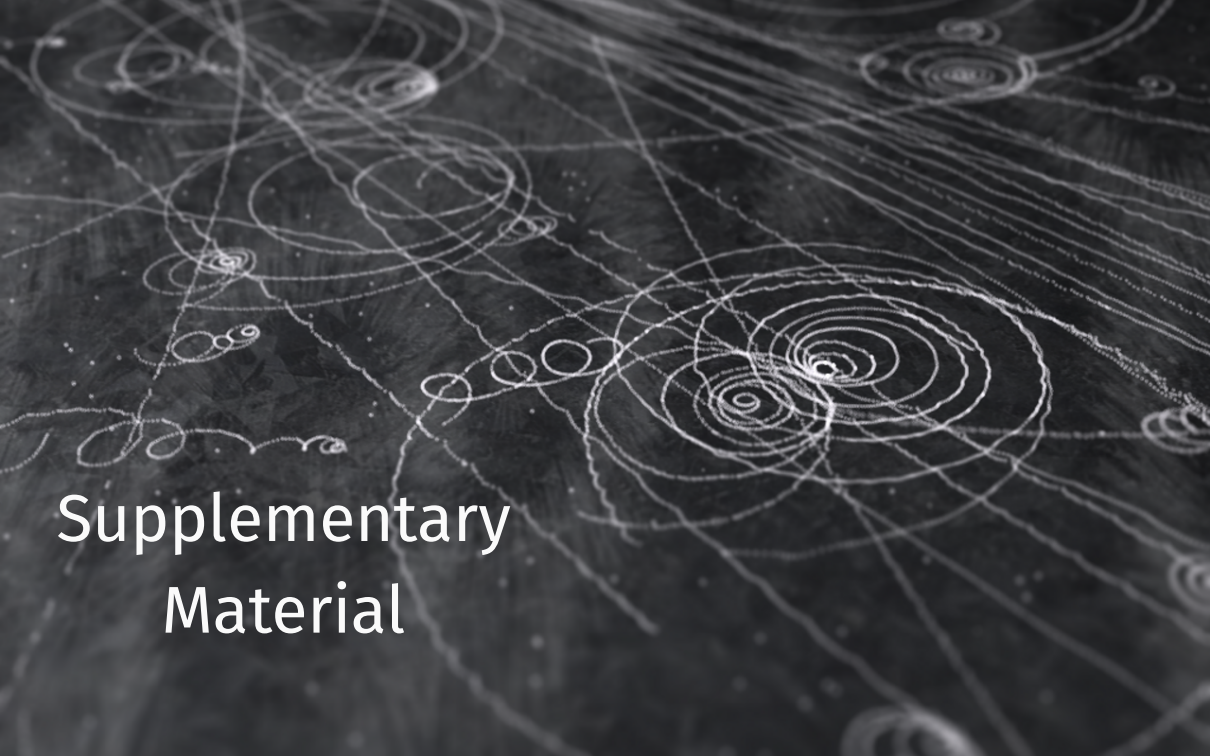
```
rdf = RDataFrame("events", "input-file.root")
rdf.Define("mc_mothers",
           "get_mothers(MCParticles, _MCParticles_parents.index)")
```

```
auto get_mothers(MCParticleCollection mcps) {
  RVec<RVec<MCParticle>> result;
  for (const auto mc : mcps) {
    RVec<MCParticle> mothers(mc.getParents().begin(),
                             mc.getParents().end());
    result.push_back(mothers);
  }
}
```

```
rdf = podio.CreateDataFrame("input-file.root")
rdf.Define("mc_mothers", "get_mothers(MCParticles)")
```

Summary & Outlook

- podio v01-00 released earlier this year
  - Some new features required by EDM4hep in v01-01
- Breaking changes for EDM4hep for consistent usability in multithreaded contexts
  - Deliberately without schema evolution
- Many smaller developments not shown in this presentation
- EDM4hep definition for v01-00 (almost) finalized
  - Some cleanup and documenation work still to be done
- Backwards compatibility already now!
- Stable basis for future developments in Key4hep


Supplementary
Material

Other miscellaneous developments for EDM4hep v01-00

- Addition of datatypes to store some generator meta information
  - Outcome of generator technical benchmark efforts for ECFA study (see e.g. this talk)
- Harmonizing a few member names to be internally consistent
- Tests to ensure backwards compatibility for reading
- Renaming of `Associations` to `Links`
- Introduction of constant `edm4hep::labels` to have consistent names for collections
- Dropped c++17 support from v00-99 onwards (following general Key4hep development)
- Make CI produce an output file with dummy values for all data members and relations

Other miscellaneous developments in podio

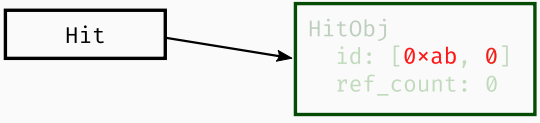
- Making `Frame::getParameter` return `std::optional`s to differentiate between empty and unavailable parameters
- `Reader` and `Writer` type-erased "interfaces" for backend agnostic I/O
- *pythonizations* that generated datamodels can use to opt-in to certain behaviors
  - *freezing of attributes*, `IndexError` for out-of-bounds accesses
- Hook for injecting the *datamodel version* (different from *schema version*!) of generated EDMs
- Introduction of `MaybeSharedPtr` to simplify internal management of `Obj` pointers

Handles and lifetimes

- Handles offer unique opportunities for lifetime issues
- *use-after-free* flagged by ASan immediately
  - No noticeable issues in actual use
- `MaybeSharedPtr` disentangles lifetimes of managed objects and control block
  - Fixes object destruction issues
- Still possible to create dangling handles
  - Document assumptions around handles

```
// Old approach of managing resources
// Common base class for resource management
struct ObjBase {
  ObjectID id; // tracks if part of collection
  atomic<unsigned> ref_count;
};
struct HitData { /* all the hit data members */ };
struct HitObj : public ObjBase { HitData data; };
class Hit { HitObj* m_obj; };
class HitCollection { vector<HitObj*> entries; };
```

```
Hit hit{};
{
  HitCollection hitColl{};
  hit = hitColl.create();
} // ~HitCollection deletes all HitObj*
// HitObj* in hit is now dangling
// ~Hit will be use-after-free
```



MaybeSharedPtr implementation

```
struct MarkOnwedTag {};
constexpr static auto MarkOwned = MarkOnwedTag{};

template<typename T>
class MaybeSharedPtr {
  struct CtrlBlock { unigned count{1}; bool owned{true}; };
  T* m_ptr;
  CtrlBlock* m_ctrlBlock{nullptr};
  /// Constructor with ownership
  MaybeSharedPtr(T* p, MarkOwnedTag) :
    m_ptr(p), m_ctrlBlock(new CtrlBlock()) {}
  /// Copy constructor increases ref-count if necessary
  MaybeSharedPtr(const MaybeSharedPtr& other) :
   m_ptr(other.m_ptr), m_ctrlBlock(other.m_ctrlBlock) {
     m_ctrlBlock && m_ctrlBlock->count++;
  }
  /// Destructor does cleanup only when necessary
  ~MaybeSharedPtr() {
    if (m_ctrlBlock && --m_ctrlBlock.count == 0)
      if (m_ctrlBlock->owned) delete m_ptr;
    delete m_ctrlBlock;
  }
  /// release only changes ownership flag
  T* release() {
    if (m_ctrlBlock) m_ctrlBlock->owned = false;
    return m_ptr;
  }
};
```

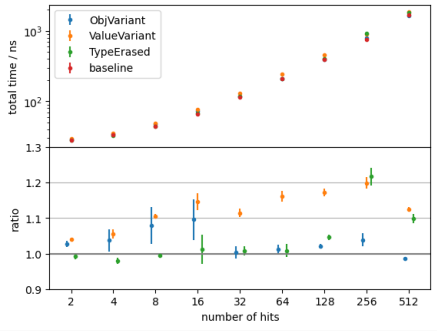
- Separate control block and managed pointer to disentangle their lifetimes
- Can only relinquish ownership of the managed pointer, never acquire it
- Reference count keeps track of control block lifetime
- Either created with or without a control block
  - `release` only changes the `owned` flag
- Only destructor destroys the control block!

Interface types implementation

- Several options possible
  - *Type erasure* on handles
  - Wrapping `std::variant` of handles
  - Wrapping `std::varaint` of `Obj*`
- Choice (almost) transparent for users
- *Type erasure* on handles makes usage of *ExtraCode* possible
  - Swallow the increased costs of assignment
  - (Artificially) restrict to interfaced types

See more details here

```
auto trackLenght(const edm4hep::Track& track) {
  double len = 0;
  edm4hep::Vector3d lastPos = {0, 0, 0};
  for (const auto& hit : track.getTrackerHits()) {
    const auto pos = hit.getPosition();
    const auto diff = pos - lastPos;
    len += std::sqrt(diff * diff);
    lastPos = pos;
  }
  return len;
}
```



Links - implementation details

```
struct LinkData { float weight; };

template<typename FromT, typename ToT>
struct LinkObj {
  podio::ObjectID id;
  LinkData data;
  std::unique_ptr<FromT> m_from;
  std::unique_ptr<ToT> m_to;
};

template<typename FromT, typename ToT, bool Mutable>
class LinkT {
  MaybeSharedPtr<LinkObj> m_obj;

  static_assert(std::is_same_v<GetDefaultHandleType<FromT>, FromT);

  template <typename FromU, typename = std::enable_if_t<
              Mutable &&
              std::is_same_v<GetDefaultHandleType<FromU>, FromT>>>
  void setFrom(FromU value);
};

template<typename FromT, typename ToT>
using Link = LinkT<FromT, ToT, false>;

template<typename FromT, typename ToT>
using MutableLink = LinkT<FromT, ToT, true>;
```

- Non-user facing classes very close to what podio code generation yields
- `LinkData` defined for simpler transition from explicitly declared datatypes
- User facing classes also templated on *mutability* to avoid code duplication
  - *SFINAE* using `enable_if` to ensure correct behavior at compile time
  - `static_assert`s at the entry points to simplify some of the `enable_if` statements

`podio::DataSource` current implementation

```
using namespace podio;
using namespace std;

class DataSource : public ROOT::RDF::RDataSource {

  // ...

  vector<string>                       m_collNames;
  vector<unsigned>                     m_activeColls;
  vector<vector<const CollectionBase*> m_colls;
  vector<unique_ptr<Reader>>           m_readers;
  vector<unique_ptr<Frame>>            m_frames;

  // ...

  bool SetEntry(unsigned slot, ULong64_t entry) override {
    m_frames[slot] = make_unique<Frame>(
        m_readers[slot]->readFrame("events", entry));

    for (auto& idx : m_activeColls) {
      m_colls[idx][slot] = m_frames[slot]->get(m_collNames[idx]);
    }
    return true;
  }

  // ...
}
```

- One pair of `Reader` and `Frame` per slot
  - Usage of `Reader` allows to use non ROOT backends
- Eagerly read all required collections
- Some additional setup work for book keeping and type information
- Relying fully on `RDataFrame` (external) synchronization
- Investigating ways to optimize

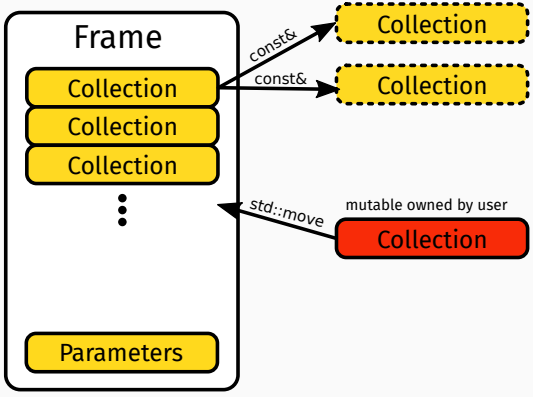
Backend agnostic reading / writing

```
class Reader {
  struct ReaderConcept {
    virtual podio::Frame readFrame(const std::string&, size_t) = 0;
  };

  templat<typename T>
  struct ReaderModel final : ReaderConcept {
    std::unique_ptr<T> m_reader;

    podio::Frame readFrame(const std::string& name,
                           size_t idx) override {
      return podio::Frame(m_reader->readEntry(name, idx);
    }
  };

  std::unique_ptr<ReaderConcept> m_self;
public:
  template<typename T>
  Reader(std::unique_ptr<T> reader) :
    m_self(std::make_unique<ReaderModel<T>>(std::move(reader))) {}

  podio::Frame readFrame(const std::string& name, size_t idx) {
    return m_self->readFrame(name, idx);
  }
};
```

- Wanted an easy way to decide at runtime which backend to use
- Didn't want to introduce a "classical abstract interface"
  - Keep a consistent feel to podio API
- Type erasure (again)
  - Not really complicated, but boilerplate-y
  - *Infectious for a code base*

The `Frame` - A generalized (event) data container

- *Type erased* container aggregating all relevant data
- Defines an *interval of validity* / category for contained data
  - Event, Run, readout frame, ...
- Easy to use and thread safe interface for data access
  - Immutable read access only
  - Ownership model reflected in API
- Decouples I/O from operating on the data




```
template<typename CollT>
const CollT& get(const std::string& name) const;

template<typename CollT, /*enable_if*/>
const CollT& put(CollT&& collection,
                 const std::string& name);
```