EDM4hep - The common event data model for the Key4hep project

Juan Miguel Carceller[1], *Mateusz Jakub* Fila[1], *Brieuc* Francois[1], *Frank* Gaede[2], *Benedikt* Hegner[1], *Thomas* Madlener[2*], *Juraj* Smiesko[1], and *André* Sailer[1]

[1]CERN, Geneva, Switzerland
[2]Deutsches Elektronen-Synchrotron DESY, Germany

Abstract. The common and shared event data model EDM4hep is a core part of the Key4hep project. It is the component that is used to not only exchange data between the different software pieces, but it also serves as a common language for all the components that belong to Key4hep. Since it is such a central piece, EDM4hep has to offer an efficient implementation. On the other hand, EDM4hep has to be flexible enough in order to allow for new developments in detector technology and reconstruction. In order to meet these challenges EDM4hep is using the podio EDM toolkit to generate its implementation from a high level description.
In this talk we give an overview of EDM4hep emphasizing the most recent developments that were tackled on the way to a first stable release. We use this opportunity to also highlight the latest developments in the podio toolkit that were required by the latest EDM4hep features. These include the introduction of type erased interface types, and a new generic RDataSource to support the full data model API in RDataFrame.

1 Introduction

The goal of the Key4hep project is the provision of a ready-to-use software stack for all currently discussed future collider options [1] and has recently reached its five year anniversary [2]. An essential component for Key4hep is a common and shared event data model (EDM) in the form of EDM4hep. It sits at the core of the Key4hep stack defining not only the data format that the different software components and packages use to exchange data but also the language that physicists have at their disposal for expressing their ideas for reconstruction and analysis.

Given its central role a performant implementation is imperative. Nevertheless EDM4hep has to be flexible enough to accommodate novel developments in detector technology and reconstruction. We address both of these challenges via the usage of the podio EDM toolkit [3, 4]. The main feature of the podio toolkit is the generation of a cache-friendly and thread safe implementation of an EDM from a high level description in YAML format. It allows to stack EDMs, such that datatypes can be shared, allowing for a streamlined approach to prototyping datatypes for new detector concepts or reconstruction techniques. Once these

*e-mail: thomas.madlener@desy.de

prototypes have matured, they can then be easily integrated into EDM4hep, since they are already defined in the proper format.

EDM4hep and its usage as well as the basic functionality provided by podio have previously also been discussed in [5]. In these proceedings we focus on the latest developments of EDM4hep towards a first stable release with backwards compatibility guarantees. Most importantly, we addressed some of the conceptual inconsistencies that were inherited from the initial version that was largely based on the LCIO EDM [6] used by the linear collider communities before Key4hep. These developments will be presented in Section 2.

Some usability improvements that were necessary for a coherent definition of EDM4hep also required new features in the podio toolkit. These include the introduction of a new category of types, so called *interface types*, as well as a generalization of the possibility to link objects of EDMs generated by podio. They will be described in more detail in Section 3 and Section 4.

Finally, we have also put together a first version of an *RDataSource* to facilitate the usage of event data in EDM4hep format via RDataFrames [7]. We report on some first experiences and potential improvements in Section 5.

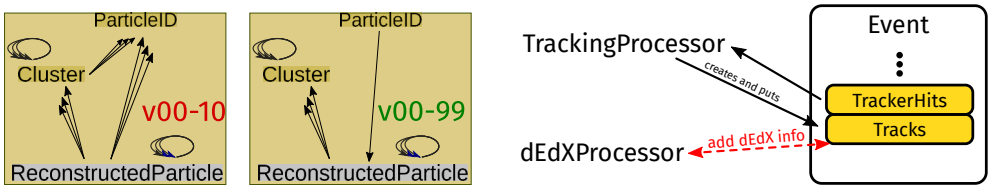
2 A consistent multi-threading concept

The LCIO EDM has initially been designed and developed two decades ago at a time when thread safety and cache friendliness were not yet considered as inherently important as they are today. Hence, some conceptual inconsistencies related to mutability in a multi-threaded context are present. These have also partially permeated into the design of some parts of the reconstruction and analysis algorithms developed by the linear collider communities. In the majority of cases, they are related to the possibility of updating data in objects after they have been handed to the event processing framework. Since EDM4hep allows mutating operations only as long as the data objects are still in control of the user, such an update mechanism is obviously not possible and would require cumbersome workarounds like creating and storing an updated copy.

To mitigate these issues we have harmonized the definition of EDM4hep to have a consistent mutability concept. In some cases, this amounted to reversing the direction of certain relations, e.g. for adding `ParticleID` objects to `ReconstructedParticle`s, as shown in Figure 1(a). In other cases new datatypes had to be introduced and defined, lifting data members that were not easily computable in the same algorithm that creates the object itself. As an exemplary case, we show the computation and storage of the specific ionization loss in a gaseous detector, dE/dx, which in the LCIO case was done by simply updating the corresponding data member in a separate processor (i.e. algorithm) after the initial tracking processor has been previously run, see Figure 1(b). Here a new `RecDqdx` datatype with a one-to-one relation to a `Track` object from which it was computed was introduced.

In almost all of the cases, the changes resulted in a more hierarchical picture of EDM4hep, where the relationships between different datatypes are now much more structured in a "is-produced-from" fashion. As another benefit, this also has a positive effect on the provenance of objects, where it is now easily possible to assign a single algorithm as the producer of each object, which was impossible before with several algorithms mutating the same object. In some cases, it might still be advantageous to be able to easily access related objects in the reverse direction, e.g. to get all ParticleID objects related to a ReconstructedParticle. To facilitate and enable these use cases we also introduced some utilities that allow for an easy reversal of the relations.

Almost all of these developments resulted in breaking changes for EDM4hep. Given the small available person power and the foreseeable technical challenges of providing backwards



(a) (b)

Figure 1: (a) Changes to the relation structure between `ParticleID` objects and `Cluster` and `ReconstructedParticle` objects between the previous version of EDM4hep and the current pre-release version. (b) Example of an inconsistent mutability concept from tracking that worked with the LCIO EDM but would no longer be possible with EDM4hep.

compatible schema evolution functionality for these changes, we decided to adopt a "before and after" approach to reduce the time to a pre-release version of EDM4hep. Hence, data that have been written with the v00-10 series of EDM4hep will only be readable with the EDM4hep v00-99 pre-release version in very limited cases.

3 Interface types and their use in EDM4hep

In some cases, the possibility to identify related datatypes by common functionality or data contents while ignoring some specific differences can be extremely useful. One such case in EDM4hep is related to tracker hits, where we currently have a `TrackerHit3D` and a `TrackerHitPlane`. The former represents a 3D measurement, as can for example be provided by a gaseous tracking detector, while the latter is a typical representation of a local 2D measurement obtained from a silicon tracking device, where the global position is defined by the placement of the sensor. Since detector concepts can use subdetectors of different technologies, a `Track` object needs to be able to hold references to both. One approach would be to introduce several, type-specific, one-to-many relations. However, that would require additional data members to keep track of the order of the hits. Ideally, we would be able to use an *interface* to simply refer to both of them simultaneously.

In a classic EDM like LCIO this was solved with a virtual inheritance structure and pointers of the base class to the different objects, but this approach is impossible in podio generated EDMs like EDM4hep. First, introducing pointers into the user interface would break consistency with the value semantics approach of podio. Secondly, and more importantly, there is no base class to inherit from in the first place. Hence, we use the *type erasure* technique to implement interface functionality using value semantics.

We have implemented the necessary code generation in podio and interface types can now be easily declared in a new `interface` category in the YAML files. The minimal definition of an interface type in YAML requires a list of participating types and optionally a list of the functionality that should be provided directly via the interface. The types that are used in an interface can then either provide this functionality directly via one of their member variables or, alternatively via the declaration of *ExtraCode*. The resulting interface objects are usable in algorithms effectively like any other datatype generated by podio. The main difference is that interface types do not provide a collection, as the actual data is stored via collections of the dedicated datatypes. As shown in Listing 1 the interface types blend in very well in their typical usage and also provide the possibility to query about and down-cast to the underlying type if necessary.

```
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!
```

Listing 1: Example usage of the `edm4hep::TrackerHit` interface to add hits of different types to a track and how to query an interface object about the current type that is being stored as well as how to down-cast to the actual type.

We have used this mechanism to introduce the `TrackerHit` interface in EDM4hep and use that in the one-to-many relations of a track. Currently, the existing `TrackerHit3D` and `TrackerHitPlane` are usable with this interface.

3.1 Alternative implementation strategies

EDMs generated by podio are implemented using a layered structure [3], where the bottom layer only contains simple plain-old-data (POD) structs. On top of that layer is the so called object layer that is responsible for resource management and relation handling. Finally, the topmost layer, which is the only one with which users interact, is the *user layer* that consists mainly of thin, cheap to copy, handles to the objects living in the object layer. This makes it possible to explore several alternative implementation strategies for interface types that are possible within the constraints of a podio generated EDM.

Before we settled on the implementation using type erasure we also investigated implementations based on `std::variant`[1], a type-safe union available since C++17. The two options that are possible with `std::variant` are either to use the handles of the user layer or the objects of the object layer. In Figure 2 we show the results of a simplified but real-world use case inspired micro-benchmark for the two variants based options as well as the type erased implementation strategy. For the benchmark, we calculate the length of a track by summing the distance between individual hits by looping over all of them. As can be seen, the performance of the variant implementation using the object layer objects (*ObjVariant*) and the type erasure based implementation is on a comparable level for this benchmark, while the variant implantation based on handles (*ValueVariant*) is significantly slower.

In the end, the deciding factor for using the type erased implementation strategy was the fact, that it trivially supports the use of functionality defined in ExtraCode, which would have required significant amounts of work with the ObjVariant approach. However, we would also like to point out that the exact implementation mechanism of interface types is indeed an implementation detail that could be refined if necessary.

4 Templated links between podio objects

In EDM4hep *external links* are used to bridge the gap between the objects created during event generation and detector simulation and the ones produced by the subsequent reconstruction. This is in contrast to the internal relations that are used to build the hierarchy of objects on either side and that allow easy navigation and access. On a technical level both of

[1] See https://en.cppreference.com/w/cpp/utility/variant for documentation and examples

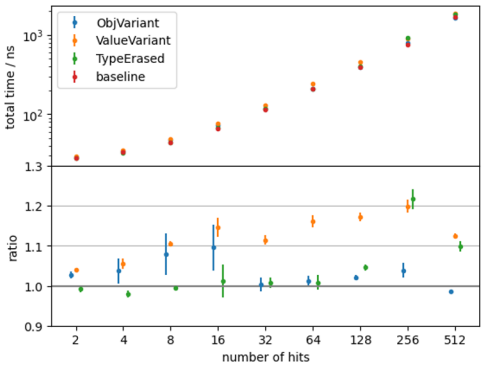


Figure 2: Results for a real-world use case inspired micro-benchmark described in the text. The different colors show different implementation strategies for interface types. The top panel shows the run time of the benchmark function as a function of the number of hits in the track. The baseline (red) values are obtained using a concrete datatype as tracker hit. The bottom panel shows the ratio of the discussed implementation options w.r.t. the baseline.

these are implemented using exactly the same mechanism in podio. In the initial definition of EDM4hep these links were called *associations* and were implemented as dedicated datatypes with effectively identical structure, providing a weight member as well as two one-to-one relations to either side of the link.

Again inspired by LCIO we would like to make it possible for users to link effectively arbitrary datatypes if necessary. However, the solution should provide more type-safety as was possible in LCIO, where only very limited guarantees could be given. Given the simple structure of such links we implemented a template based solution directly in podio. Users can then trivially create links between arbitrary datatypes in-memory and can opt-in to I/O functionality via the use of a single macro, as shown in Listing 2. The macro only hides some boilerplate code related to registering the datatype with the I/O system of podio.

```
// Link arbitrary podio generated datatypes
using RecoMCParticleLinkCollection = podio::LinkCollection<edm4hep::ReconstructedParticle,
                                                           edm4hep::MCParticle>;
// Enable I/O (only necessary for links not defined in YAML)
PODIO_DECLARE_LINK(edm4hep::ReconstructedParticle, edm4hep::MCParticle)
// Conventional access
auto mcP = link.getTo();
// Templated / tuple like access
mcP = link.get<edm4hep::MCParticle>();
mcP = link.get<2>();
auto& [rp, mp, w] = link; // <-- structured bindings!
```

Listing 2: Example declaration and use of a templated link collection that connects `ReconstrucedParticles` and `MCParticles` in EDM4hep. The usage of the `PODIO_DECLARE_LINK` enables I/O for this specific link type. The usage examples in the bottom half show the possibilities for accessing the linked objects as well as the weight.

Compared to the manually defined links as dedicated datatypes, templated links offer some additional functionality. The main goal was to further instill the point that even though the template parameters indicate a direction of the link, they are in principle undirected, i.e.

there is no need to define two different datatypes to go from simulation to reconstruction and vice versa. As a result links now also provide `get` and `set` functionality which is templated on the datatypes that are linked. This makes it possible to completely omit the direction in usage as shown in Listing 2.

In order to keep the possibility of having the full definition of a datamodel in the canonical YAML file, we also introduced a new `links` category, which also takes care of generating all the necessary code to enable I/O functionality. Thus, this is the recommended way of defining links if they should be persist-able.

With the aim of trying to have backwards compatibility already for the v00-99 pre-release series, we have renamed the initial associations before the pre-release. With the correct names in place, introducing the templated links was possible in a completely transparent way for users.

5 An RDataSource for podio generated EDMs

The default backend of podio is based on ROOT [8, 9] and we offer I/O functionality based on TTrees and the newly developed RNTuple [10–12] format. As a result of the layered implementation approach of podio we are able to write the event data as effectively flat, contiguous data buffers [4] via ROOT, which makes them easily accessible for columnar analysis, e.g. using RDataFrame [7], or in completely independent implementations, e.g. via Julia [13]. On the other hand, the in-memory relations between objects is also persisted as one buffer of effectively indices per relation and event data collection. As a consequence navigating these relations in an RDataFrame based analysis becomes cumbersome and error prone, especially if one-to-many relations are involved, or if retrieving the necessary information requires the use of more than one relation. Additionally, implementing functions that can be used for the analysis requires one to have knowledge about implementation details of podio, like the fact that the POD structs are suffixed with *Data* and that some information necessary for handling one to many relations is stored in these PODs.

In Listing 3 we show a rather simple example of how this looks for retrieving the mother particles of MC particles both on the implementation side as well as on the user side. Here another implementation detail of podio shines through, namely the names of the branches where relation indices are stored are required to be known and used as inputs for calling this function.

```
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)")
```

Listing 3: Definition of a function usable in RDataFrame for retrieving the mothers of MC particles and its subsequent usage without the newly implemented RDataSource for podio.

To allow for the usage of the full podio generated user interface also in columnar analysis we have implemented the `podio::DataSource` via inheriting from the `ROOT::RDF::RDataSource`. Instead of giving access to the raw data buffers, it routes the file access through a podio based reader, which also takes care of resolving all relations on the fly. Thus, it becomes possible to define utility functions using the full expressivity of the EDM interface, as visible in Listing 4. The implementation of the `get_mothers` function is greatly simplified compared to the one shown in Listing 3. Additionally, the call site of the function has become much simpler as well, where now only the name of the collection is required. All the details of the podio based implementation are completely hidden.

```
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)")
```

Listing 4: Definition of a function usable in RDataFrame for retrieving mothers of MC particles and its subsequent usage using the newly available `podio::DataSource`.

The implementation of the `podio::DataSource` transparently provides reading support for all available I/O backends of podio [4] and, more importantly, also has the full schema evolution machinery at its disposal. We consider the interface of the `podio::DataSource` stable enough for release at this point, but also acknowledge that we have not yet done any optimizations. Hence, especially for simple analysis event loops, it performs significantly worse, both in runtime and memory usage. We intend to spend some time to optimize both aspects in the near future.

6 Conclusions & Outlook

The common and shared EDM of the Key4hep project, EDM4hep, is close to a first stable release with strong backwards compatibility guarantees for reading event data. After some important developments that we have reported here, namely the harmonization of the datatypes for use in multi-threaded contexts, the introduction of a tracker hit interface and templated links, as well as a newly developed RDataSource for podio, we have made a v00-99 pre-release series for extended user testing.

The final changes that still need to be applied to the definition of EDM4hep are mostly related to a small overhaul of contents related to MC generators and the MC particle. The changes are mostly related to removing some data members that are not reasonably usable without additional context from a generator. Given that HepMC3 [14] is the common format for data exchange between generators, we opted to remove these data members to avoid leaking partial information from generator internals to the analysis stage. We plan to implement these changes in a backwards compatible fashion to keep files that have been written with the v00-99 pre-release readable with v01-00 as well.

Acknowledgments

This work benefited from support by the CERN Strategic R&D Programme on Technologies for Future Experiments (CERN-OPEN-2018-006). This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 101004761.

References

[1] P. Fernandez Declara et al., The Key4hep turnkey software stack for future colliders, PoS **EPS-HEP2021**, 844 (2022). 10.22323/1.398.0844

[2] J.M. Carceller et al., Five years of Key4hep - Towards production readiness and beyond, PoS **ICHEP2024**, 1029 (2025). 10.22323/1.476.1029

[3] J.M. Carceller, F. Gaede, G. Ganis, B. Hegner, C. Helsens, T. Madlener, A. Sailer, G.A. Stewart, V. Volkl, Towards podio v1.0 - A first stable release of the EDM toolkit, EPJ Web Conf. **295**, 06018 (2024), 2312.08206. 10.1051/epjconf/202429506018

[4] F. Gaede, G. Ganis, B. Hegner, C. Helsens, T. Madlener, A. Sailer, G.A. Stewart, V. Volkl, J. Wang, EDM4hep and podio - The event data model of the Key4hep project and its implementation, EPJ Web Conf. **251**, 03026 (2021). 10.1051/epjconf/202125103026

[5] F. Gaede, T. Madlener, P. Fernandez Declara, G. Ganis, B. Hegner, C. Helsens, A. Sailer, G. A. Stewart, V. Voelkl, EDM4hep - a common event data model for HEP experiments, PoS **ICHEP2022**, 1237 (2022). 10.22323/1.414.1237

[6] F. Gaede, T. Behnke, N. Graf, T. Johnson, LCIO: A Persistency framework for linear collider simulation studies, eConf **C0303241**, TUKT001 (2003), physics/0306114. 10.48550/arXiv.physics/0306114

[7] D. Piparo, P. Canal, E. Guiraud, X. Valls Pla, G. Ganis, G. Amadio, A. Naumann, E. Tejedor, RDataFrame: Easy Parallel ROOT Analysis at 100 Threads, EPJ Web Conf. **214**, 06029 (2019). 10.1051/epjconf/201921406029

[8] R. Brun, F. Rademakers, P. Canal, A. Naumann, O. Couet, L. Moneta, V. Vassilev, S. Linev, D. Piparo, G. GANIS et al., root-project/root: v6.18/02 (2019), https://doi.org/10.5281/zenodo.3895860

[9] R. Brun, F. Rademakers, ROOT: An object oriented data analysis framework, Nucl. Inst. & Meth. **A389**, 81 (1997). 10.1016/S0168-9002(97)00048-X

[10] J. Blomer, P. Canal, A. Naumann, D. Piparo, Evolution of the ROOT Tree I/O, EPJ Web Conf. **245**, 02030 (2020), 2003.07669. 10.1051/epjconf/202024502030

[11] J. Lopez-Gomez, J. Blomer, RNTuple performance: Status and Outlook, J. Phys. Conf. Ser. **2438**, 012118 (2023), 2204.09043. 10.1088/1742-6596/2438/1/012118

[12] J. Blomer, ROOT RNTuple: Next Generation Event Data I/O for HENP, in *CHEP 2024* (Krakow, Poland, 2024), https://indico.cern.ch/event/1338689/contributions/6005282/

[13] P. Mato, EDM4hep.jl: Analysing EDM4hep files with Julia, in *CHEP 2024* (Krakow, Poland, 2024), https://indico.cern.ch/event/1338689/contributions/6016139/

[14] A. Buckley, P. Ilten, D. Konstantinov, L. Lönnblad, J. Monk, W. Pokorski, T. Przedzinski, A. Verbytskyi, The HepMC3 event record library for Monte Carlo event generators, Comput. Phys. Commun. **260**, 107310 (2021), 1912.08005. 10.1016/j.cpc.2020.107310