Towards podio v1.0 - A first stable release of the EDM toolkit

Juan Miguel Carceller[1], *Frank* Gaede[2], *Gerardo* Ganis[1], *Benedikt* Hegner[1], *Clement* Helsens[1,3], *Thomas* Madlener[2,*], *André* Sailer[1], *Graeme A* Stewart[1], and *Valentin* Volkl[1]

[1]CERN, Switzerland
[2]Deutsches Elektronen-Synchrotron, Germany
[3]KIT, Germany

Abstract. A performant and easy-to-use event data model (EDM) is a key component of any HEP software stack. The podio EDM toolkit provides a user friendly way of generating such a performant implementation in C++ from a high level description in yaml format. Finalizing a few important developments, we are in the final stretches for release v1.0 of podio, a stable release with backward compatibility for datafiles written with podio from then on. We present an overview of the podio basics, and go into slighty more technical detail on the most important topics and developments. These include: schema evolution for generated EDMs, multithreading with podio generated EDMs, the implementation of them as well as the basics of I/O. Using EDM4hep, the common and shared EDM of the Key4hep project, we highlight a few of the smaller features in action as well as some lessons learned during the development of EDM4hep and podio. Finally, we show how podio has been integrated into the Gaudi based event processing framework that is used by Key4hep, before we conclude with a brief outlook on potential developments after v1.0.

1 Introduction

A typical high energy physics (HEP) analysis workflow comprises many steps, where each is usually in charge of producing a specific result or of linking these results into a more comprehensive picture of the underlying physics event. A core part of such a software stack is the so called event data model (EDM) that enables the flow of information between the different components and also defines the language of this communication. Crucially, it is also the language in which users express their ideas. Defining the contents of such an EDM is one aspect, the other one is the efficient implementation of the schema that users come up with. The podio EDM toolkit [1–3] addresses the latter part, while EDM4hep [4, 5] defines the common EDM of the Key4hep project [6, 7] using podio.

In these proceedings we focus on recent developments in podio towards a first stable release, which we plan to announce in the near future after finishing the necessary developments. We will discuss these final missing pieces in the text where appropriate.

The structure of these proceedings is as follows: After a brief recap of the basics of podio in Section 2 we describe the Frame concept and how it relates to the multithreading concept

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

of podio in Section 3. In Section 4 we dive into some of the challenges and solutions we found in the context of implementing a schema evolution mechanism for EDMs generated by podio, before we briefly talk about other recent developments in Section 5. These include the addition of a new RNTuple [8, 9] based I/O backend, as well as the integration of Frame based I/O into the core framework of Key4hep [10]. We close with a discussion of the open issues that we still want to address before actually releasing version 1.0 of podio.

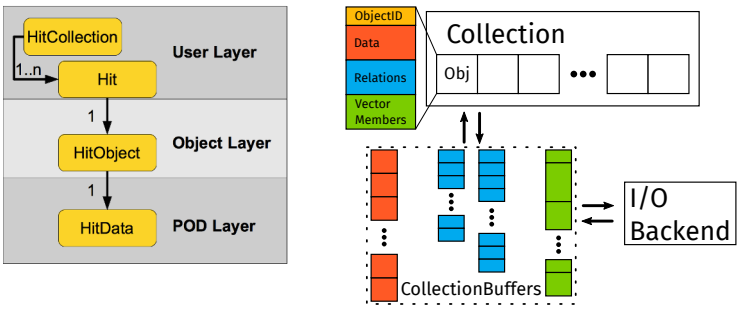
2 The basics of podio

Since the basic ideas of podio as well as the features of the generated code have been covered in more detail in previous publications [1, 2, 4], we will focus on those parts that are most relevant for the technical discussions later.

One of the key features of podio is its code generator that reads a high level description of an EDM in YAML format and generates performant C++ code which is then compiled into a shared library. The generated code favors composition over inheritance and uses plain-old-data (POD) types wherever possible [2]. The design leverages three different layers which allows for a performant I/O layer and cache friendly memory layout. These layers are shown in Figure 1a and organize the generated classes into the following:

- The *User Layer* which is the only one with which users interact, consisting mainly of thin handles.
- The *Object Layer* that manages resources and inter-object relations.
- The *POD* or *Data Layer* where the data of all objects live as simple POD structs.

Each object in the Object Layer is uniquely identified by an `ObjectID` consisting of a collection ID and an index in that collection. The collections contain these objects and give access to the data via thin handle classes. These handles come in two varieties; the default handles are immutable and lack any functionality that would allow one to alter the state of the underlying object. On the other hand the collections also serve as factories for mutable handles that can be used to fill the collections with meaningful content. An implicit conversion of mutable to immutable handles makes sure that interfaces consuming EDM datatypes only need to be defined once.



(a) podio layers (b) schematic collection I/O

Figure 1: (a) The three layers of podio for an example *Hit* class and (b) the (un)packing of Collections (from) into `CollectionBuffers` for I/O.

Data persistency in podio generated EDMs is based on collections of objects. Collections can be created from or grant access to their data in the form of `CollectionBuffers`. These hold all the necessary data to (de)serialize a collection and they are comprised solely of contiguous arrays of PODs (in *array-of-struct (AoS)* layout) as shown in Figure 1b. The necessary functionality to (un)pack these buffers into the in-memory representation of the stored objects is part of the generated EDM code. Hence, new I/O backends effectively only need to be able to read and write contiguous arrays of memory.

3 The Frame concept and multithreading

The basic ideas of the *Frame* concept have already been introduced previously [5]. In short; the Frame is a thread-safe container that aggregates all relevant EDM data of a given category or interval of validity, e.g. an event or a run. Data stored in a Frame is accessible read-only and the hand-over of the ownership of data is clearly expressed in the API. In order to stay in line with the overall approach of offering *value semantics* in the interfaces the Frame is implemented using *type erasure*. This also allows the Frame to be constructed from almost arbitrary *Frame data*, which in turn decouples I/O operations entirely from Frame construction. As Frame data only have to provide access to the `CollectionBuffers` once queried for them by the Frame that owns them, they are free to defer actual work, e.g. decompressing data, as long as possible.

This potential deferral of work is in line with the general design philosophy of multithreading surrounding I/O of podio. The core components of podio, e.g. readers and writers for different I/O backends, can be used to implement more sophisticated functionality in event processing frameworks. Since these usually tackle their multithreading needs in different ways, we have deliberately kept the readers and writers free of any internal synchronization and we assume that they operate on their respective input and output files exclusively and in a single threaded fashion. All potentially necessary synchronization, e.g. if a writer receives multiple Frames from different threads or if a reader should supply Frame data to a multithreaded queue, has to be done externally by the framework. For more details on how this is done for Key4hep we refer to Section 5.4.

4 Schema evolution

Schema evolution, i.e. the possibility of an EDM schema to adapt to changing requirements, e.g. from new detector technologies or from novel reconstruction algorithms, is a crucial feature. It is crucial for data preservation efforts and long term usability of any EDM. In order to avoid having to solve schema evolution in all generality, we have chosen to implement the necessary evolutions as they arise in EDMs that are generated by podio. This still leaves a huge task to tackle, and was undoubtedly the most challenging development in podio so far. The main considerations for schema evolution for podio were

- being able to leverage existing capabilities of the backend, e.g. ROOT,
- having schema evolution work for all backends, and
- automating the generation of the necessary evolution code, while still allowing for user overrides if necessary.

As of the writing of these proceedings not all of these goals have yet been fully achieved. Nevertheless, many of the necessary building blocks are in place, and schema evolution is working for the default ROOT backend.

In our current design of schema evolution all the necessary evolutions are applied to the `CollectionBuffers` directly, before collections are even constructed from them. Hence, users will only ever see the latest version of the EDM in memory. The evolution of the buffers will also always happen in one evolution step, going directly from the schema version on file to the current schema version of the EDM.

4.1 Building blocks for schema evolution

The first step in schema evolution is to detect changes between two EDM schemas. To this end we have implemented a tool that reads two versions of the schema in the high level YAML format and compares them. It checks all detected evolutions against a pre-defined list of supported evolutions. Currently, this list very closely follows the schema evolution capabilities of ROOT. A major advantage of having this check very early in the process of generating code is that we can inform users about a potentially unsupported schema evolution before they are able to write data that is not backwards compatible with previously written data.

The second step is to read back the `CollectionBuffers` in the schema version they have been written. We achieve this via a central `CollectionBufferFactory` that is able to create empty `CollectionBuffers` for all known datatypes and schema versions. It is populated during dynamic library loading of a generated EDM and is effectively implemented as a map of a pair of datatype name and schema version to a buffer creation function. These creation functions as well as the necessary call to register them into the `CollectionBufferFactory` are all done via automatic code generation. Since the factory is immutable during the runtime of a program it is accessible concurrently from multiple readers that might live on several threads.

After reading the data buffers in an old schema version the final step is to evolve these buffers to the current schema version so that a collection can be constructed from them. We follow a very similar approach here as we did for buffer creation by employing another central `SchemaEvolution` instance that keeps track of evolution functions for all datatypes and schema versions. The major difference with respect to the `CollectionBufferFactory` is that here we allow the user to override automatically generated evolution functions if desired or necessary.

The hooks to execute the evolution function are placed inside the Frame directly after obtaining the `CollectionBuffers` from the Frame data. This is the earliest place where `CollectionBuffers` for a collection are actually guaranteed to exist, and also the latest place where we want to deal with older schema versions. It also makes it very easy for backends that have builtin schema evolution to simply provide already evolved buffers in which case the schema evolution will be a no-op. A schematic control flow with some details about the Frame internals are shown in Figure 2. As evident from there the `CollectionBuffers` also carry enough information about the collection type and the schema version to be able to identify the correct evolution function.

4.2 Current status

As of the writing of these proceedings, schema evolution is working for the ROOT backend, mainly because it's the ROOT backend who does most of the heavy lifting. The only feature which is not natively supported is the renaming of data members, for which we implemented the necessary evolution code. For technical reasons this evolution path actually foregoes most of the building blocks described above. It does not make use of the `SchemaEvolution` function registry for example, and always requests buffers in the current schema version from

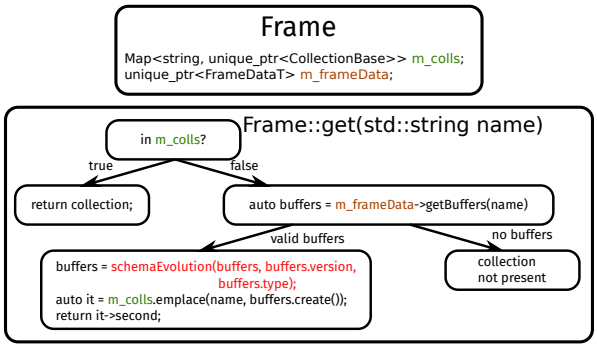


Figure 2: Minimal schematic Frame implementation and control flow inside `Frame::get` when requesting a collection by name.

the `CollectionBufferFactory`. This is necessary to actually trigger the schema evolution mechanism of ROOT.

For the support of other backends the necessary schema evolution building blocks are in place. This includes the code generation for populating the `CollectionBufferFactory` with the required buffer creation functions. The main missing piece is the code generation of the actual schema evolution functions. However, it would in principle already be possible now to provide user defined evolution functions to achieve schema evolution with backends without builtin schema evolution.

We have implemented easy to re-use and automated test cases for the currently supported schema evolution capabilities. These allow us to add tests for other backends with very little effort.

5 Other recent developments

There have also been several other smaller recent developments in podio. Most of them involved refactoring existing code to allow us to implement all the necessary functionality for schema evolution and the Frame concept. These were completely transparent for users. Additionally, there were some additional new features that were added to podio, as well as some developments that were not completely transparent. We will discuss these further in the following paragraphs.

5.1 Stable collection IDs

For somewhat historical reasons the initial assignment of collection IDs to collections was based on the insertion order into a Frame. This approach is untenable in a multithreaded context as insertion order can no longer be guaranteed. Additionally, collection IDs are a core part in the persistency of inter-object relations. Hence, a given collection name ideally always maps to the same collection ID, such that relations might even be reconstructed on files that have been split during processing.

To achieve this we have switched the calculation of the collection ID to use the *MurmurHash3* algorithm to compute a 32 bit hash value from the collection name. The choice of 32 bits was on the one hand based on the fact that the collection ID used 32 bits before, so that this change could be implemented effectively transparently for users and we would not

need to increase the size of the ObjectIDs by 50%[1] to go to 64 bits. On the other hand we also checked that there are no collisions in the collection names that are currently in use inside the Key4hep project. We are confident that at least for the foreseeable future there should be no collisions in the collection IDs. However, even if there are, a switch to 64 bits would be completely backwards compatible.

5.2 Persistency of the datamodel definition

Although podio now provides a schema evolution mechanism, sometimes it might be useful or necessary to read the data of a file back with the original schema version, or to simply compile an EDM library to read the data back in the first place. In order to make it possible to retrieve the original version of the datamodel definition in YAML format we have started to embed it into the shared EDM library itself, but also to store it as meta data into all data files that are written by podio. In both cases the datamodel definition is converted into the JSON format as that allows for a more compact string representation. This string is embedded as a raw string literal into the shared EDM library, which can be retrieved with various tools that allow one to read binary files, e.g. via *readelf*. For the retrieval of the datamodel definition from datafiles we have integrated the necessary functionality into the `podio-dump` utility.

5.3 Adding an RNTuple I/O backend

RNTuple is the designated successor of ROOTs TTree data format [8, 9] and will address future needs for I/O in HEP. Since podio is used for future collider studies it is natural to strive to also support this feature as an additional I/O backend. Given the nice separation of I/O concerns and operating on the data introduced with Frame based I/O, there were no real technical challanges to overcome from the podio side. The major issues that we faced had to do with trying to persist data types that were not yet supported by RNTuple, e.g. `uint16_t`, but these were promptly resolved by the ROOT developers, so that the RNTuple based backend in the end just requires a slightly more recent version of ROOT. Given that the RNTuple interface is still in experimental state, also the RNTuple backend of podio should be considered experimental and not used for production.

5.4 Frame integration into k4FWCore

One of the core motivations for developing the Frame concept was the support of multithreaded frameworks and to facilitate the integration of podio based EDMs, specifically EDM4hep, into the core, Gaudi [11] based, framework of Key4hep, *k4FWCore* [10]. Among the core services this packages offers are the following:

- `DataHandle`s that give access to podio based EDM collections inside an algorithm,
- a `PodioDataSvc` that manages the I/O of these collection as well as the creation of the necessary `DataHandle`s.

Both of these were previously implemented partially in terms of functionality from standalone podio and some additional custom implementions of readers and writers for podio data files. The integration of Frame based I/O into k4FWCore also allowed us to clean up some of the differences in implementation that have grown over time w.r.t. standalone podio. Here we have simply replaced the custom readers and writers for podio files in k4FWCore with the ones that standalone podio offers.

[1] The ObjectID consists of two 32 bit integers, a CollectionID and an ObjectIndex.

For users of the Gaudi based framework the Frame is not visible and is instead just used behind the scenes to implement the `DataHandle`s and the `PodioDataSvc`. For most existing algorithms this was a completely transparent change. However, if an algorithm made use of the `PodioDataSvc` directly some interface changes were unavoidable and required intervention. In order to allow for a gradual migration to the new functionality we have kept the pre-existing functionality in the `PodioLegacyDataSvc`. This will be deprecated and then removed in the midterm future.

To facilitate access to file level meta data we also introduced a `MetaDataHandle` that works similar to the `DataHandle` with the major difference being that it can only be accessed for writing during algorithm initialization or finalization, i.e. when Gaudi is running on a single thread. It is backed by another Frame and makes use of the standard podio I/O functionality. The switch to the `MetaDataHandle` had the biggest impact on algorithms that made use of *collection parameters*, e.g. encoding strings for interpreting bit field values in the data. Although these parameters never changed in practice, it would have still been possible to change them from a technical perspective. Applying the now more restrictive, but thread safe, scheme for meta data access required some work in those algorithms that need to write meta data. A particular challenge, for which we do not yet have a fully satisfactory solution, is algorithms for which these meta data only become available during event processing. Here we have resorted back to requiring execution on a single thread, which then also allows us to slightly relax the conditions for meta data write access.

6 Conclusion & Outlook

The podio EDM toolkit has been and is currently used by several communities to do physics studies. The file format that is written by podio has been stable since the introduction of Frame based I/O, and we try to keep backwards compatibility for these files even before a v1.0 release. Having finished a first version of schema evolution capabilities for the default ROOT backend it is now also possible to actually evolve datamodels that make use of podio. Nevertheless, there are still some developments that we want to finalize before we announce a first stable version of podio.

The most important of these developments are related to schema evolution. The most important part that is not yet implemented here is the handling of multiple older schema versions. We expect this to be solvable with reasonable effort since the main work will be related to handling multiple input YAML files, whereas all the complicated issues in code generation are already solved. We plan to defer the support for schema evolution for non-ROOT backends until after the release of v1.0, as this should be fairly independent and does not have an impact on the file format written by the ROOT backend.

Another important aspect is documentation. Here we have to integrate the latest developments and features and overall organize the existing documentation into a coherent set of pages and text. We have setup basic automation for the generation of documentation from the source code as well as from existing text documents and we plan to extend this to also cover the deployment of documentation onto a webpage.

A potential development that we plan to investigate after the release of v1.0 of podio is related to the in-memory layout of the data in podio. Currently the data in the Data Layer, see Figure 1a, are stored in AoS layout. However, there might be some performance gains to be made by switching to a different layout, e.g. *struct-of-array (SoA)*. This layout can be more cache friendly or allow for better optimiziations if operations are only necessary on a subset of members of a given data type. It would potentially also allow for a more easy hand-off to GPUs or other heterogeneous resources. Since podio is already generating code that features

the three implementation layers, described in Section 2, we think that we can implement this transparently for users. However, we have not yet started any developments in this direction.

To summarize and finish these proceedings, we want to again highlight the fact that podio is in use by several communities in production already. The introduction of schema evolution and the Frame concept should make podio future proof. After having resolved the few remaining issues mentioned in the text, we plan to release v1.0 of podio shortly and continue to incorporate the feedback and feature requests from the user communities afterwards.

Acknowledgements

This work beneted 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] F. Gaede, B. Hegner, G.A. Stewart, EPJ Web Conf. **245**, 05024 (2020)
[2] F. Gaede, B. Hegner, P. Mato, J. Phys. Conf. Ser. **898**, 072039 (2017)
[3] *podio github repository*, https://github.com/AIDASoft/podio (2021), accessed: 2021-02-12
[4] F. Gaede, G. Ganis, B. Hegner, C. Helsens, T. Madlener, A. Sailer, G.A. Stewart, V. Volkl, J. Wang, EPJ Web Conf. **251**, 03026 (2021)
[5] F. Gaede, T. Madlener, P. Declara Fernandez, G. Ganis, B. Hegner, C. Helsens, A. Sailer, G. A. Stewart, V. Voelkl, PoS **ICHEP2022**, 1237 (2022)
[6] P. Fernandez Declara et al., PoS **EPS-HEP2021**, 844 (2022)
[7] P. Fernandez Declara et al., EPJ Web Conf. **251**, 03025 (2021)
[8] J. Blomer, P. Canal, A. Naumann, D. Piparo, EPJ Web Conf. **245**, 02030 (2020), 2003.07669
[9] J. Lopez-Gomez, J. Blomer, J. Phys. Conf. Ser. **2438**, 012118 (2023), 2204.09043
[10] V. Volkl, J. Pöttgen, B. Hegner, A. Zaborowska, J. Cervantes, C. Helsens, B. Francois, Z. Jiaheng, A. Sailer, A. Stano et al., *key4hep/k4fwcore:* (2021), https://doi.org/10.5281/zenodo.5109722
[11] G. Barrand et al., Comput. Phys. Commun. **140**, 45 (2001)