EDM4hep and podio - The event data model of the Key4hep project and its implementation

Frank Gaede[1,*], *Gerardo* Ganis[2], *Benedikt* Hegner[2,3,**], *Clement* Helsens[2], *Thomas* Madlener[1,***], *Andre* Sailer[2], *Graeme A* Stewart[2], *Valentin* Volkl[2], and *Joseph* Wang[4]

[1]DESY
[2]CERN
[3]Institute for Advanced Computational Science, Stony Brook University, New York, USA
[4]Bitquant Digital Services

Abstract. The EDM4hep project aims to design the common event data model for the Key4hep project and is generated via the podio toolkit. We present the first version of EDM4hep and discuss some of its use cases in the Key4hep project. Additionally, we discuss recent developments in podio, like the updates of the automatic code generation and also the addition of a second I/O backend based on SIO. We compare the available backends using benchmarks based on physics use cases, before we conclude with a discussion of currently ongoing work and future developments.

1 Introduction

At the core of every HEP experiment's software framework lies the event data model (EDM). It is the EDM that defines the language that physicists use to express their ideas, and also the interface and communication channels between the different framework components. Hence, it is crucial that an EDM captures all the use cases that arise in the data processing and analysis and that it is implemented consistently and efficiently. Both of these aspects are addressed in the Key4hep [1–3] project; the former is addressed by the EDM4hep [4] library, while the latter is covered by the underlying podio [5] library.

The EDM4hep project aims to design and define a common EDM for the Key4hep project that should be easy to use for future lepton and hadron colliders and will be covered in Section 2. The podio EDM toolkit aims to facilitate the efficient implementation of EDMs in C++ by automatically generating all the necessary code from a high level description of the EDM. It is described in more detail in Section 3, where we also present the integration of a new I/O backend and related performance studies. Finally, in section 4, we present currently ongoing work, some known current limitations as well as future plans on how to fix them and other improvements.

[*]e-mail: frank.gaede@desy.de
[**]e-mail: benedikt.hegner@cern.ch
[***]e-mail: thomas.madlener@desy.de

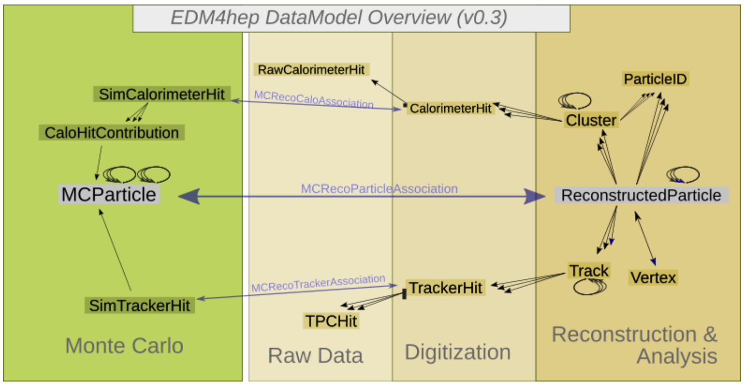


Figure 1. Overview of the contents of EDM4hep and the relations among the different data types that it defines. The labelled associations between Monte Carlo simulation and the reconstruction and analysis data types are distinct data types themselves in EDM4hep. Relations and their direction are depicted by arrows, where multiple arrows imply the possibility of multiple relations.

2 EDM4hep

The contents of the EDM4hep are very similar to the ones of LCIO [6], which has already been successfully used by the linear collider community. It features essential data types for representing measurement data characteristic for typical HEP experiments as well as data types that allow to condense these low level data into high level descriptions of a complete collision event. Additionally, and equally important, similar data types are present to describe simulated events. A strict separation between the data types used in simulation and the ones in event reconstruction is enforced by not having the possibility to access any simulated data directly from the reconstruction side. Instead dedicated association data types that connect the two worlds exist.

Figure 1 shows all the currently present data types and their relations among each other. As can be seen EDM4hep has similar capabilities as other EDMs and, e.g. allows one to build decay hierarchies for MC particles, as an *MCParticle* can have multiple relations to other mother or daughter *MCParticle*s. Similarly it is possible to reconstruct short lived particles from longer lived decay products by combining *ReconstructedParticle*s. All relations in EDM4hep are directed, i.e. a *ReconstructedParticle* can hold references to *Cluster*s or *Track*s, but not vice versa.

2.1 EDM4hep and LCIO

The LCIO EDM has been used successfully for more than 15 years in physics and detector studies for ILC and CLIC and CEPC and provides a complete and sufficient event data model for generic lepton collider physics analyses. LCIO therefore provided an ideal base for defining the EDM in EDM4hep. The closeness of the two EDMs will eventually facilitate the porting of the large linear collider software stack to Key4hep. The transition to EDM4hep also offers the possibility to introduce a more modern coding style, in particular with *value semantics* and *automatic reference counting*, freeing the user of any memory handling burden, in comparison to the *pointer semantics* used in LCIO. At the same time LCIO, developed and extended over many years, provides a significant set of utility classes and convenience functions that facilitate the correct and simple application of LCIO in user code, e.g. for meta data handling or relation navigation. In particular the convenience functions in some EDM classes can easily be implemented also in EDM4hep using the *extra code* mechanism described below. Work is ongoing to port existing LCIO utility code or develop new utilities in a dedicated package: EDMd4hep-utils [7].

3 podio

The previous sections have highlighted some of the things a generic EDM needs to be able to support, most importantly, the ability to handle unique and non-unique relations between objects of arbitrary data types. It should also offer an easy-to-use interface to its physicist users, as well as an implementation that leverages the available computing power as efficiently as possible. EDMs generated by podio offer a simple user API and use concrete types, favoring composition over inheritance as well as well defined object ownership. To free the users from the implementation details all code is automatically generated from a high level description in YAML format (see section 3.2). In this section we will only give a brief introduction to the general design principles [8, 9] and instead focus on the most recent developments.

3.1 The three layers of podio

One of the key ideas of podio is to use plain-old-data (POD) types wherever possible. This allows for a relatively simple memory model, performant I/O operations and also supports vectorization. To facilitate the handling of relations among objects in the EDM podio uses three layers to separate the data handling with the POD types from the rest. An illustration of this layered approach is shown in Figure 2.

The **POD Layer** holds arrays of the actual data structures, e.g. *HitData* with position and amplitude information.

The **Object Layer** consists of transient objects (*HitObject*), which handle the relations between the objects of the EDM and also manage the POD objects.

The top layer, the **User Layer** comprises lightweight handles to the objects (*Hit*) and collections of them, e.g. *HitCollection*. These handles offer *value semantics*, completely freeing the user of any resource management duties or worries of how to best pass these handles around. The design goal of podio is to allow for full functionality using only objects that are present in the User Layer.

3.2 Automatic code generation

Automatic code generation offers several advantages. It frees the user of the often cumbersome and error prone details of implementation, allowing her instead on focusing on designing the EDM by having the possibility of starting quickly and quick turn-around times for later improvements. Furthermore the generated code is consistent, homogeneous and implementation improvements can be easily deployed to the entire EDM. This code generation is handled by a python script that has been almost completely re-implemented recently to use the Jinja2 [10] template engine and now produces C++17 compliant code.

It starts from a description of the entire EDM in a single YAML file. Each data type in this file has a few mandatory and several optional fields which can be filled by

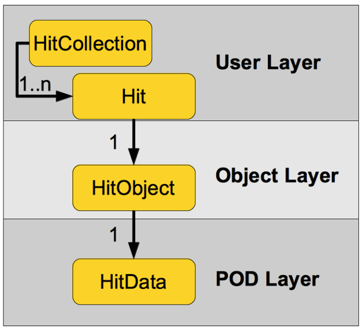


Figure 2. The three layers of data types in podio: The *POD Layer* holding the actual data, the *Object Layer* handling relations and managing the resources and the *User Layer* with the user facing handles and collections.

```
edm4hep::ReconstructedParticle:
  Description: "Reconstructed Particle"
  Author : "F.Gaede, DESY"
  Members:
    - float energy                // [GeV] energy of the reconstructed particle
    - edm4hep::Vector3f momentum  // [GeV] particle momentum
  OneToOneRelations:
    - edm4hep::Vertex startVertex // start vertex associated to this particle
  OneToManyRelations:
    - edm4hep::Cluster clusters  // clusters that have been used for this particle
    - edm4hep::Track tracks      // tracks that have been used for this particle
  ExtraCode:
    declaration: "
    bool isCompund() const { return particles_size() > 0; }\n"
```

Listing 1: Simplified YAML snippet describing the *ReconstructedParticle* class of EDM4hep showing parts of the present data members and relations.

the user, an example is shown in Listing 1. A limited validation is run on the input YAML file, e.g. to make sure that the data types really only define POD structures and to a lesser extent also to make sure that the EDM is internally consistent. It is also possible to add custom code to enhance the functionality of data types. However, this custom code is not validated to be valid C++ code before the automatic code generation and non-valid C++ will only be caught during compilation.

To further enhance the usability, dedicated CMake [11] functions are offered for downstream usage that nicely wrap the different steps from calling the code generation script to compiling the generated code into shared libraries that offer the functionality of the EDM, including I/O, see Listing 2.

3.3 Supporting different I/O backend libraries

Another key design goal of podio is to be agnostic to the library that is used for persistency. This is achieved by encapsulating the functionality to encode and resolve

```
find_package(PODIO)

# generate the c++ code from the yaml definition
PODIO_GENERATE_DATAMODEL(edm4hep edm4hep.yaml headers sources IO_BACKEND_HANDLERS "ROOT;SIO")
# compile the core data model shared library (no I/O)
PODIO_ADD_DATAMODEL_CORE_LIB(edm4hep "${headers}" "${sources}")
# generate and compile the ROOT I/O dictionary
PODIO_ADD_ROOT_IO_DICT(edm4hepDict edm4hep "${headers}" src/selection.xml)
# compile the SIOBlocks shared library for the SIO backend
PODIO_ADD_SIO_IO_BLOCKS(edm4hep "${headers}" "${sources}")
```

Listing 2: CMake configuration of EDM4hep for generating and compiling three shared libraries, one for the core data model and one each for the currently supported I/O backends: ROOT and SIO. The configuration is adapted here slightly for better readability, but is essentially the same as the one that is shipped with EDM4hep.

the relations between objects into the collection classes, where all the relation information is packed into arrays of POD types. The collections are also responsible for packing and unpacking the data of all contained objects into or from contiguous data buffers. Hence, essentially all that an I/O backend library has to be able to do is to read and write several continuous data buffers per collection.

The default I/O backend used by podio is using ROOT [12, 13] that offers automatic generation of the necessary streamer code from dictionaries. The current implementation stores the collection data and the relation data in different branches of a *TTree* in columnar data layout. An alternative I/O backend based on the SIO (*Simple Input Output*) [14] library has recently been fully integrated. SIO is the underlying I/O layer of LCIO and stores binary records holding complete events in the case of podio. It also allows to split the steps of compressing and decompressing from writing and reading to file in a thread safe way, allowing them to happen on separate threads. However, the current implementation does not yet exploit this feature. The necessary wrapper code for the SIO primitives is automatically generated by the code generator (see section 3.2) and is also compiled into a shared library. This library will be automatically loaded at runtime by the SIO reader and writer classes. Similar to the code generation, generating the different additional libraries necessary for the two backend backends is facilitated by CMake functions, see Listing 2.

3.4 Benchmarks comparing the ROOT and SIO backend

The two philosophies of laying out the data in the produced files between the SIO and ROOT I/O backends have different advantages and disadvantages for different use cases. Especially when only parts of the complete event record are needed for analysis, the columnar data layout of ROOT I/O should have favorable performance over SIO. On the other hand, for usage in HEP experiment frameworks, where the complete event record has to be read SIO might offer better performance. In the benchmarks presented in the following we have used ROOT version 6.22/08 and SIO version 00-01 on a laptop computer with an `Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz` processor, 16 GB of RAM and a SSD hard drive. All involved packages have been compiled with the default gcc-9.3.0 that comes with Ubuntu 20.04. Both backends are used with their default settings regarding compression algorithms and levels, as podio does not yet offer the possibility to easily change that for the user.

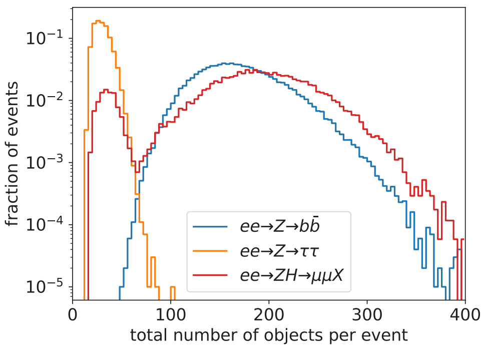


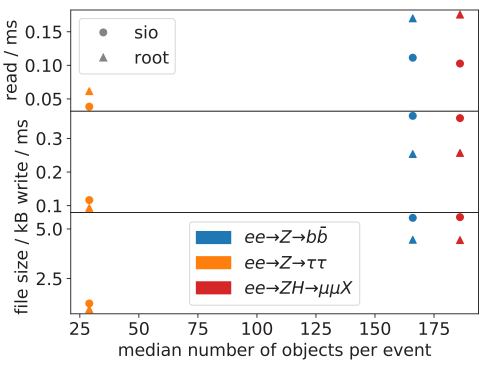
Figure 3. Distribution of the total number of objects per event for the different physics cases that are used for benchmarking.

To record benchmark data, podio offers two static decorators to instrument the reader and writer classes. These decorators simply wrap all calls to member functions of the original readers and writers into a pair of calls of `std::chrono::high_resolution_clock::now()` and record the elapsed time.

To assess the writing performance of both currently present backends we make use of the k4SimDelphes package [15] that is part of the Key4hep software stack. This package uses the Delphes fast simulation framework [16] to simulate the detector response for generated MC particles. It offers interfaces to a few different MC generators and stores the resulting event contents using EDM4hep. This allows us to explore different possible physics use cases, which differ in their event contents, i.e. the number and type of the different physics objects as well as their relations among each other. However, as Delphes only does a parametrized detector response simulation, mainly the high-level *MCParticle* and *ReconstructedParticle* classes will be used, whereas low-level objects, such as calorimeter or tracker hits are completely absent. For the reading benchmarks we use a small standalone executable that tries to mimic the usage pattern of a HEP experiment framework, by reading in all collections that are present in the event and furthermore also resolving all relations between the objects.

The physics cases that we use are $ee \rightarrow Z \rightarrow b\bar{b}$ and $ee \rightarrow Z \rightarrow \tau\tau$ at FCC-ee ($\sqrt{s} = 91$ GeV) as well as $ee \rightarrow ZH \rightarrow \mu\mu X$ (Higgs recoil) at ILD ($\sqrt{s} = 250$ GeV). Figure 3 shows the distribution of the total number of objects per event for these cases. While the $ee \rightarrow Z \rightarrow \tau\tau$ case leads to relatively small events, the $ee \rightarrow Z \rightarrow b\bar{b}$ case mainly produces medium sized events, whereas the Higgs recoil at ILD case produces events of mixed sizes.

Figure 4 (left) shows a comparison of the two backends and their per event read and write performance as a function of the median number of objects per event. As expected, smaller events can be written and read quicker than larger ones and can also be stored in smaller output files. In general the ROOT backend is able to produce smaller output files than the SIO backend, regardless of the size of the events, as shown in Figure 4 (right). As far as read times are concerned, the SIO backend is faster by roughly 30 to 40 %, depending on the size of the events. On the other hand the ROOT backend has better write performance for all event sizes. Overall, the correlation between the time spent in I/O operations per event and the number



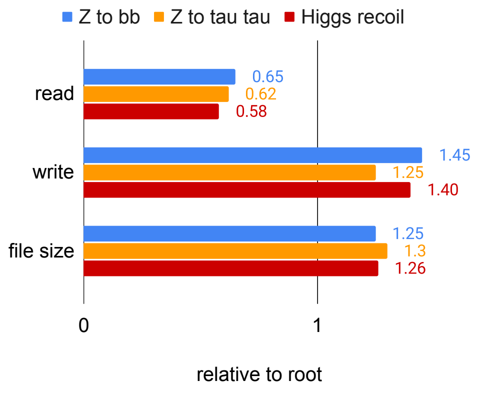


Figure 4. Left: Per event read and write times and resulting file size per event as a function of the median number of objects per event for the different physics cases and the two I/O backends. Right: Overview results of the total read and write times and resulting file size of the SIO backend compared to the ROOT one.

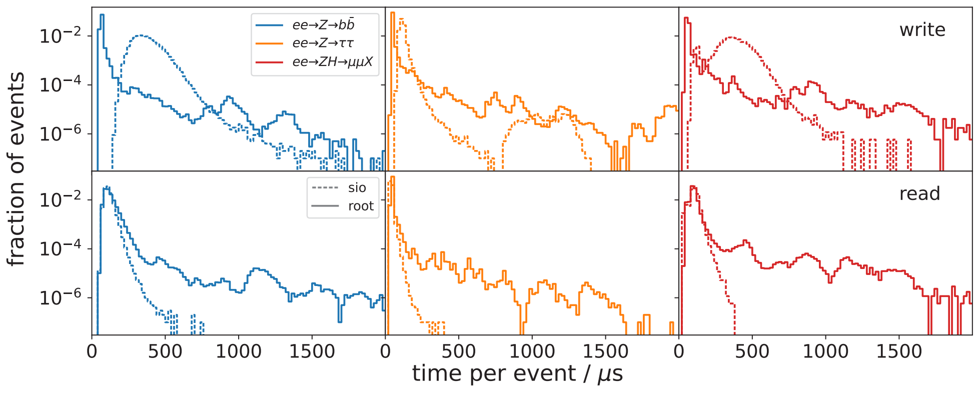


Figure 5. Time spent in I/O operations per event when writing (top row) and reading (bottom row) events of the different physics benchmark cases using podio for the ROOT and SIO backends.

of objects in the event is more pronounced for the SIO backend as for the ROOT backend. This is visible in Figure 5 that shows the distributions of the per event read and write times for all benchmark cases and the two backends. In case of the SIO backend the distributions follow a similar structure as the ones shown in Figure 3, whereas they exhibit long tails and in less differences between the physics cases for the ROOT backend. At least for the SIO case this can be readily understood by considering that the amount of data that is written or read per event is more or less proportional to the contents of the event, whereas for ROOT I/O there might be some overhead introduced by the columnar data layout.

For the Higgs recoil physics case we have also instrumented a small plotting macro that reflects a typical analysis use case, where only parts of the events are needed, in this case only the collection of all *ReconstructedParticle*s. We find that the total read time of the SIO backend is only affected very little, whereas the ROOT backend achieves a speed up of slightly more than a factor two, resulting in equal run times for both I/O backends. These results are readily understood, as SIO has to read in the full event block and only saves a bit of time in not having to resolve all the relations, whereas the ROOT backend can actively take advantage and only read the necessary collection data. Interestingly, the ROOT backend is able to read around 98 % of the events quicker than the SIO backend, but the long tail of read times for the last 2 % (also seen in Figure 5) of events leads to roughly equal performance for all events.

4 Currently ongoing work and future plans

Both EDM4hep and podio are actively used in the Key4hep project, not only to test the software stack, but also to do physics studies. Hence, they are already in a somewhat production ready state. Nevertheless, feedback from the involved communities has revealed a few issues that need to be addressed before usage in large scale productions becomes feasible. This section presents a loosely connected list of issues that have been reported as well as some plans to address them in the near and midterm future. Furthermore, we also present some ongoing work and open questions that need more effort before a final answer can be given.

4.1 Finalize EDM4hep definition

As mentioned in section 2, EDM4hep draws heavy inspiration from LCIO and FCC-edm. Both of these EDMs have been mainly used for lepton collider physics studies and also the current physics studies focus on physics at such colliders. However, Key4hep and EDM4hep should ideally be suitable to use for all kinds of future collider projects, including hadron colliders. The collision environment in a hadron collision is vastly different from the one in lepton collisions and so the EDM also has to support potentially significantly different use cases. Recently, studies in the context of FCC-hh have been started to investigate how well the current version of EDM4hep is able to support the workflows of hadron collider physics analysis.

4.2 Improvements for podio generated EDMs

There are still a few use cases that are not yet fully supported by EDMs that are generated via podio. A long standing issue is the missing schema evolution of EDMs in podio. This is a necessary feature that allows to read files that have been written with a different version of the EDM than the one that is currently used. We do not yet have any actual implementation, but we plan to also automate this schema evolution as far as possible, e.g. by generating necessary conversion code from two versions of the yaml definition of an EDM. We plan to also add handling of customized conversion code to allow for non-trivial schema evolution needs as well.

Another shortcoming is the one of reference collections, i.e. the possibility to have collections of objects that do not actually store objects themselves but rather reference objects from other collections. This use case is supported by LCIO and used in the linear collider community, where there is usually a large collection of *ReconstructedParticle*s that comprises all of the reconstructed particles in a collision event. Collections of, e.g., muons are implemented as reference collections that only store references to those elements of the reconstructed particle collection that are identified as muons. This allows to build several distinct collections of particles without having to actually copy their data around, in this way also avoiding errors of not updating all the necessary copies in the end. In order for EDM4hep to support this use case an additional data type, *RecoParticleRef*, has been introduced as a workaround, but has proven to be cumbersome in its usage. Implementing reference collections in podio is currently being worked on.

4.3 Integration of podio into Key4hep

There is an existing implementation of the I/O system in the Key4hep framework, that is based on the same ROOT I/O that is shipped with podio. However, over time the two implementations have diverged and are now no longer fully compatible. Thus, we aim to consolidate the two implementations again and in doing so try to incorporate some of the experience that we have gained. Also the example implementations for easily running an event loop and reading and writing files that come with podio have been used for beyond their original design goal of unit testing. Hence, we plan to implement several components that can be combined to achieve the desired functionality by suitably combining them. Given that they are targeted to be used at the very bottom of an experiment framework or analysis code, we have to avoid introducing limiting design choices and aim for rather general components. We are currently working on designing the scope of these components and their interfaces.

4.4 Support for flat data formats

In recent years many python analysis frameworks have found their way into HEP analysis. Many of these are built around the idea of "flat data" formats and so called columnar data analysis. Although podio and the EDMs are targeted more at the usage inside HEP software frameworks where the rich structure of objects and relations among them is a necessary feature, we would like to explore the possibilities of also supporting flat data formats where such information is much harder to represent. The ROOT backend writes *TTree*s that can be loaded into an RDataFrame [17], which can be used to gain some first experience and to collect some feedback on how the current EDM4hep can be used in such a context. We find that from a purely technical point of view it would already be possible to implement a usable analysis framework using RDataFrame on top of the usual file structure that is produced by podio. However, while access to the data of the objects is trivial, the handling of relations between different objects is very error prone as it requires to use and manipulate several integer indices.

We are currently investigating if and how it is possible to support such flat data formats, also in view of such usage in Key4hep. As of now, there is no clear best solution to this problem, but we are currently also starting to look into the usage of RNTuple [18] to see if it could offer benefits not only for flat data formats, but maybe also improve I/O performance for other use cases.

4.5 Usage with heterogeneous resources

The usage of heterogeneous resources such as GPUs will only become more important in the future, hence, podio generated EDMs should be efficiently usable on them. We think that our POD based memory layout of the data should be beneficial for usage on such resources. However, currently we have not yet done any tests into this direction and we are collecting use and benchmark cases to investigate how to best support heterogeneous resources.

4.6 Code robustness

While the design of podio from the start has tried to make it hard to make mistakes in memory handling or unnecessary copy operation, additional effort has been put to further improve on the code robustness. We have activated a yet more stringent warning level during compilation and thereby addressed potential code issues, such as removing unnecessary copy constructors or potential shadowing issues. We will continue to explore new language and compiler features in the future to ensure the robustness, ease of use and high quality of the generated code.

5 Conclusions

The EDM of the Key4hep project, EDM4hep, has been defined in a first version. It has been tested in first physics studies and further evaluation is ongoing. EDM4hep uses podio as its generator and it has sparked new developments there. Here we have presented the recent work on the automatic code generation, the integration of an additional I/O backend based on SIO as well as some benchmarks comparing this backend to the default ROOT based one. We have also identified future avenues that we want to explore to further improve the quality of EDM4hep and the podio toolkit.

Acknowledgements

This work benefited from support by the CERN Strategic R&D Programme on Technologies for Future Experiments (https://cds.cern.ch/record/2649646/, CERN-OPEN-2018-006). This project has also received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

References

[1] A. Sailer, G. Ganis, P. Mato, M. Petrič, G.A. Stewart, EPJ Web Conf. **245**, 10002 (2020)

[2] *Key4hep github organization*, https://github.com/key4hep (2021), accessed: 2021-02-12

[3] W. Fang, P. Fernandez Declara, F. Gaede, G. Ganis, B. Hegner, C. Helsens, X. Huang, T. Li, W. Li, T. Li et al., *Key4hep: Status and Plans* (2021), submitted to this conference

[4] *EDM4hep github repository*, https://github.com/key4hep/EDM4hep (2021), accessed: 2021-02-12

[5] *podio github repository*, https://github.com/AIDASoft/podio (2021), accessed: 2021-02-12

[6] F. Gaede, T. Behnke, N. Graf, T. Johnson, eConf **C0303241**, TUKT001 (2003), `physics/0306114`

[7] *EDMd4hep-utils github repository*, https://github.com/key4hep/EDM4hep-utils (2021), accessed: 2021-02-26

[8] F. Gaede, B. Hegner, P. Mato, J. Phys. Conf. Ser. **898**, 072039 (2017)
[9] F. Gaede, B. Hegner, G.A. Stewart, EPJ Web Conf. **245**, 05024 (2020)
[10] *Jinja2*, https://palletsprojects.com/p/jinja/ (2021), accessed: 2021-02-12
[11] *CMake*, https://cmake.org/ (2021), accessed: 2021-02-12
[12] R. Brun, F. Rademakers, Nucl. Instrum. Meth. A **389**, 81 (1997)
[13] 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`
[14] *SIO github repository*, `https://github.com/iLCSoft/SIO` (2021), accessed: 2021-02-12
[15] *k4SimDelphes github repository*, `https://github.com/key4hep/k4SimDelphes` (2021), accessed: 2021-02-12
[16] J. de Favereau, C. Delaere, P. Demin, A. Giammanco, V. Lemaître, A. Mertens, M. Selvaggi (DELPHES 3), JHEP **02**, 057 (2014), `1307.6346`
[17] V.E. Padulano, J.C. Villanueva, E. Guiraud, E. Tejedor Saavedra, EPJ Web Conf. **245**, 03009 (2020)
[18] J. Blomer, P. Canal, A. Naumann, D. Piparo, EPJ Web Conf. **245**, 02030 (2020), `2003.07669`