PODIO: An Event-Data-Model Toolkit for High Energy Physics Experiments

To cite this article: F Gaede et al 2017 J. Phys.: Conf. Ser. 898 072039

View the article online for updates and enhancements.

Related content
- Small cells for foams
- The Detector Safety System of the ATLAS experiment O Beltramello, H J Burckhart, S Franz et al.
- Crystal calorimeters in the next decade Ren-yuan Zhu
PODIO: An Event-Data-Model Toolkit for High Energy Physics Experiments

F Gaede¹, B Hegner² and P Mato²
¹ DESY, Notkestr. 85, 22607 Hamburg, Germany
² CERN, 1211 Geneva 23, Switzerland
E-mail: frank.gaede@desy.de

Abstract. PODIO is a C++ library that supports the automatic creation of event data models (EDMs) and efficient I/O code for HEP experiments. It is developed as a new EDM Toolkit for future particle physics experiments in the context of the AIDA2020 EU programme. Experience from LHC and the linear collider community shows that existing solutions partly suffer from overly complex data models with deep object-hierarchies or unfavorable I/O performance. The PODIO project was created in order to address these problems. PODIO is based on the idea of employing plain-old-data (POD) data structures wherever possible, while avoiding deep object-hierarchies and virtual inheritance. At the same time it provides the necessary high-level interface towards the developer physicist, such as the support for inter-object relations and automatic memory-management, as well as a Python interface. To simplify the creation of efficient data models PODIO employs code generation from a simple yaml-based markup language. In addition, it was developed with concurrency in mind in order to support the use of modern CPU features, for example giving basic support for vectorization techniques.

1. Introduction
The event data model (EDM) lies at the heart of every HEP experiments software framework. It is the EDM that defines the interfaces and communication channels between the different software modules and algorithms in the data processing chain. It is thus crucial that the EDM is well designed and implemented in a consistent and efficient way. While the former will clearly depend on the experience of the user, PODIO can help to achieve the latter. Experience from LHC and the linear collider community shows that existing solutions partly suffer from overly complex data models with deep object-hierarchies or unfavorable I/O performance. PODIO was created to address these issues, particularly aiming at future HEP projects like the Future Circular Collider (FCC). In the next section we introduce the main design choices for PODIO and their motivation. In section 3 we highlight the key aspects of the underlying implementation and provide some code examples. Before concluding we present the current usage of PODIO in HEP.

2. Design of PODIO
2.1. Basic considerations
Many of the design choices are inspired by previous experience of the LCIO[1] package used for the studies by the linear colliders and the Gaudi Object Description [2] applied in the LHCb collaboration at the LHC. One key idea of PODIO is to use as much as possible POD data
types (see 2.2) to keep the memory model simple, provide for fast I/O operations and efficiently support vectorization. PODIO offers a simple user API and class hierarchies by using concrete types, favoring composition over inheritance and a well defined object ownership. Automatic code generation has some advantages, as it:

- minimizes user mistakes
- provides quick turn-around times for improvements on the back-end as well as for extensions of the EDM
- ensures the consistency and homogeneity of the EDM
- allows for a quick start with using PODIO

PODIO provides a C++ and Python interface, is thread-safe and allows by design to implement different I/O layers. A first implementation of the I/O layer is based on ROOT[3].

2.2. POD data types

The C++11 standard defines a POD (Plain Old Data) as a type that is either:

- a scalar type or
- a class type (class or struct or union) that:
  - is a trivial type
  - is a standard layout type
  - has no non-static members that are non-POD
- an array of such types

The above definition implies that PODs (and arrays thereof) can be easily copied in memory or to and from disk using memcpy or fwrite respectively. This is the main reason for the expected performance advantage of using PODIO. For simplicity PODs can be thought of as being equivalent to fixed size C structs - even though this not strictly correct.

2.3. The three PODIO layers

In principle just using POD like data structures would be sufficient to define an EDM and indeed this has been the common practice for experiments in the time before C++ and object orientation have been introduced in HEP. Exposing PODs directly to the user is error prone and inconvenient. This is particularly true for handling the relations between objects. PODIO introduces two additional layers of lightweight classes on top of the actual PODs to address these issues. This is illustrated in Fig.1 for an example Hit class.

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

The object layer consists of transient, lightweight objects which handle the relations between the individual objects in the EDM (HitObject). These relations can be either of type one-to-one or one-to-many and are stored in the POD layer using suitable ObjectIDs (see 3.3). The objects in this layer also handle optional, intrinsic vector members. Such vector members break the PODness of the data structures but are sometimes needed or at least requested by users. The simplest such example are strings.

Finally the user layer introduces handles to the EDM objects (Hit) and collections of EDM object handles, e.g. HitCollection. In most cases only these classes will occur in user code. Due to the nature of the handle construct this results in value semantics (no pointers are needed).

3. Implementation

In this section we give a brief overview on the key aspects of the PODIO implementation.
3.1. Object Ownership
Unclear object ownership and memory leaks are a common problem in many C++ applications. PODIO makes it close to impossible for the user to make mistakes in this respect due to the 

value semantics described in the previous section. The actual ownership is hidden from the user and implemented in the object layer in two stages:

- before registering data objects with an event store they are reference counted and garbage collected
- after registering with the event store the ownership is transferred to the event store which handles the object lifetime

This introduces a small additional costs on object creation time but no costs later Registering of objects with the event store is done transparently through the collection as shown in Listing 1.

Listing 1. Example for object creation and ownership

```cpp
auto & hits = store.create<HitCollection>("hits");
auto h1 = hits.create(1.,2.,3.,42.); // init w/ values
auto h2 = hits.create(); // default construct
h2.energy(42.);

auto h3 = Hit();
auto h4 = Hit();
hits.push_back(h3);

// h1,h2,h3 are automatically deleted with collection
// h4 is garbage collected
```

3.2. Code generation
PODIO creates all C++ and Python code for the users based on a description of the EDM structures in a `yaml` file. Apart from user convenience and code robustness this also offers fast turn around times for potential improvements on the backend like the underlying I/O or for extensions of the data model on the user side. The individual EDM data structures are composed of the following elements:

- basic type data members

Figure 1. Schematic view of the three PODIO layers with an example Hit data object.
- components (structs of basic types)
- references to other objects

It is also possible to define additional user code for member functions in the yaml files. This feature allows to provide additional convenient functions which for example return data derived from the original member attributes. These extra functions will be included in the generated user handle classes. An example yaml declaration is shown in Listing 2.

### Listing 2. Example definition of EDM entity in yaml.

```yaml
# LCIO MCParticle
MCParticle:
Description: "LCIO MC Particle"
Author: "F.Gaede, B.Hegner"
Members:
- int pDG // PDG code of the particle
- int generatorStatus // status as defined by the generator
- int simulatorStatus // status defined by simulation
  ...
OneToManyRelations:
- MCParticle parents // The parents of this particle.
- MCParticle daughters // The daughters of this particle.
ExtraCode:
  const_declaration:
    "bool isCreatedInSimulation() const {
    return simulatorStatus() != 0 ;
    } \n"
```

Python is treated as first class citizen in PODIO, i.e. one can use pythonic code for iterators etc. This is possible by generating some additional usability code in Python on top of the binding that is created with PyROOT. Listing 3 demonstrates this with a small Python application that prints the energies of all hits for all events.

### Listing 3. Example Python program.

```python
store = EventStore(filenames)
for i, event in enumerate(store):
    hits = store.get('Hits')
    for h in hits:
        print h.energy()
```

3.3. Relations

Relations between objects are at the heart of every EDM. As already mentioned in section 2.3, relations need to be implemented differently in every layer. While the user clearly would like to use references or vectors of referenced objects, these can neither be easily made persistent nor are vectors\(^1\) allowed in PODs. Therefore pointers are converted into collection indices by introducing a dedicated `ObjectID = collectionID + collectionIndex`. Every object in PODIO is uniquely identified by its `ObjectID` and thus we can use these `ObjectIDs` in the POD layer and on disk for storing the relation. Note that this can be done independently of the specific I/O system. After reading data back from disk we convert the arrays of `ObjectIDs` back into the corresponding vectors of referenced objects.

As all of this is done in the object layer the users need not be concerned with the details of the underlying implementation. From a relation defined in the yaml file member functions are

\(^1\) with vectors we refer to C++ vectors of variable size
generated in the corresponding handle class. The use of these functions is straightforward as is demonstrated in the example Listings 4 and 5.

**Listing 4.** Example for creating cluster-hit relations.
```cpp
auto & hits = store.create<HitCollection>("hits");
auto & clusters = store.create<ClusterCollection>("clusters");
auto hit1 = hits.create(); auto hit2 = hits.create();
auto cluster = clusters.create();
cluster.addHit(hit1);
cluster.addHit(hit2);
```

**Listing 5.** Example for reading cluster-hit relations.
```cpp
for ( auto h = cluster.Hits_begin(), end = cluster.Hits_end(); h!=end; ++h){
    std::cout << h->energy() << std::endl;
}
auto hit = cluster.Hits(42);
```

When reading back the relations, the referenced objects can be either accessed via iterators or directly through their index.

### 3.4. Vectorization

Vectorization is a key technique for exploiting the capabilities of modern CPUs. Often a _struct-of-arrays_ (SoA) layout of the data in memory is preferred over an _arrays-of-structs_ (AoS) layout to make better use of vector instructions. In principle PODIO allows to choose either representation at the implementation of the POD layer. This choice has no effect on user code but it has to be made at compile time, where one has to keep in mind that the on-demand transformation between complete SoA and AoS representations is highly inefficient. Nevertheless we provide convenience methods for the on-demand transformation which could be used in dedicated vectorizable code:

```cpp
auto & x_array = hits.x<10>(); // return the value of the
auto & y_array = hits.y<10>(); // first 10 elements in std::array
```

The use of these functions mandates proper performance measurements on real use cases.

### 3.5. I/O

The current implementation of the I/O layer in PODIO is still rather rudimentary as the PODs and ObjectIDs are directly stored using ROOT I/O with auto generated streamer code from dictionaries. While this code is working very effectively on arbitrary C++ classes, it has not yet been optimized to take advantage of the properties of PODs that would allow for very fast I/O. A simple I/O library that will write out complete arrays of PODs with one _fwrite_ statement is currently under development. With this library it will be possible to measure the maximum performance gain that one can reach by using PODs for the I/O. One aspect that will need special consideration is of course dependence on the CPU architecture. While most processors in use today are _little endian_, it is not guaranteed that the memory layout of the array of structs is identical on all CPUs. An eventual implementation of an I/O component that is optimized for PODs will have to address this, for example by having a fall-back to the member wise XDR [5] based I/O used in ROOT and LCIO.
4. PODIO in use

PODIO is developed in the context of the FCC software [6] framework. It is actively used by this group in combination with Gaudi [7] as well as in stand alone C++ and Python applications. The linear collider community is investigating the use of PODIO for an evolution of LCIO. The idea is to improve the underlying I/O performance while keeping the well established API of LCIO largely unchanged. This should be possible by making good use of the extra code functionality described above. The LHCb collaboration is interested in using PODIO for their data model upgrade and has created the lhcbio demonstrator during a coding sprint. PODIO has been adopted by the Hep Software Foundation (HSF) [8] as an incubator project.

5. Conclusion

PODIO is an EDM toolkit that is developed in the context of FCC and the LC community with general HEP in mind. One key idea of PODIO is to use POD data types to keep the memory model simple and provide for fast I/O operations. Automatic code generation not only minimizes user mistakes and ensures quick turn-around times but also makes it very easy to get started with PODIO which is available on GitHub [9].

Acknowledgments

This project has received funding from the European Union’s Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168 (AIDA2020).

References

[1] Gaede F, Behnke T, Graf N and Johnson T 2003 eConf C0303241 TUKT001 sLAC-PUB-9992, CHEP-2003-TUKT001 (Preprint physics/0306114)
[2] Gaudi Object Description, available at: URL http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/DataDictionary/Documents/GaudiObjDesc_docu.pdf
[3] Brun R and Rademakers F 1997 Nucl. Instrum. Meth. A389 81–86
[4] Evans C et al. YAML Ain’t Markup Language (YAML) URL http://yaml.org/spec/1.2/spec.pdf
[5] XDR: External Data Representation Standard URL http://www.rfc-base.org/rfc-4506.html
[6] Bernet C et al. The FCC Software: how to keep software experiment independent these proceedings
[7] Barrand G et al. 2001 Comput. Phys. Commun. 140 45–55
[8] HSF webpage URL http://hepsoftwarefoundation.org
[9] PODIO GitHub page URL https://github.com/hegner/podio