Event Data Model Toolkit

Hegner, B (CERN)

24 September 2018

The AIDA-2020 Advanced European Infrastructures for Detectors at Accelerators project has received funding from the European Union’s Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

This work is part of AIDA-2020 Work Package 3: Advanced software.

The electronic version of this AIDA-2020 Publication is available via the AIDA-2020 web site <http://aida2020.web.cern.ch> or on the CERN Document Server at the following URL: <http://cds.cern.ch/search?p=AIDA-2020-D3.4>

Copyright © CERN for the benefit of the AIDA-2020 Consortium
**Abstract:**

PODIO is an event data model (EDM) toolkit for the efficient creation of Event Data Models in C++ with high performance I/O. It has been developed in the context of the studies for the Future Circular Collider (FCC) with applications to the linear collider community in mind. It overcomes issues observed with previous EDMs used in HEP by addressing thread safety, simplicity and memory management from the start. We present the design and implementation of PODIO, its current status as well as future plans.
AIDA-2020 Consortium, 2018
For more information on AIDA-2020, its partners and contributors please see www.cern.ch/AIDA2020

The Advanced European Infrastructures for Detectors at Accelerators (AIDA-2020) project has received funding from the European Union’s Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168. AIDA-2020 began in May 2015 and will run for 4 years.

Delivery Slip

<table>
<thead>
<tr>
<th>Name</th>
<th>Partner</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authored by</td>
<td>B. Hegner [Task coordinator]</td>
<td>CERN</td>
</tr>
<tr>
<td>Edited by</td>
<td>F.Gaede [WP coordinator]</td>
<td>DESY</td>
</tr>
<tr>
<td>Reviewed by</td>
<td>W.Pokorski [WP coordinator]</td>
<td>CERN</td>
</tr>
<tr>
<td></td>
<td>F.Gaede [WP coordinator]</td>
<td>DESY</td>
</tr>
<tr>
<td></td>
<td>D. Bortoletto [Deputy Scientific coordinator]</td>
<td>OXFD</td>
</tr>
<tr>
<td>Approved by</td>
<td>F.Sefkow [Scientific coordinator]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Steering Committee</td>
<td></td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

1. INTRODUCTION ........................................................................................................................................... 4

2. DESIGN .......................................................................................................................................................... 5
   2.1. POD TYPES .............................................................................................................................................. 5
   2.2. THE THREE LAYERS OF PODIO ........................................................................................................... 5

3. IMPLEMENTATION ......................................................................................................................................... 6
   3.1. OBJECT OWNERSHIP ............................................................................................................................ 6
   3.2. CODE GENERATION ............................................................................................................................... 6
   3.3. RELATIONS ............................................................................................................................................ 7
   3.4. VECTORIZATION ................................................................................................................................... 8
   3.5. I/O IMPLEMENTATIONS ......................................................................................................................... 9

4. PODIO IN USE ................................................................................................................................................. 9

5. FUTURE PLANS / CONCLUSIONS / RELATION TO OTHER AIDA-2020 WORK ......................... 10

6. REFERENCES ............................................................................................................................................... 11

ANNEX: GLOSSARY ......................................................................................................................................... 12
Executive summary

POADIO is an event data model toolkit for the efficient creation of Event Data Models in C++ with high performance I/O. It overcomes issues observed with previous EDMs used in HEP by addressing thread safety, simplicity and memory management from the start. PODIO has been developed in the context of the studies for the Future Circular Collider (FCC) with applications to the linear collider community and other HEP experiments in mind.

1. INTRODUCTION

The event data model (EDM) lies at the heart of every HEP experiment’s 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 unfavourable I/O performance.

POADIO, or plain-old-data I/O, is a Python and C++ library to support the creation and handling of data models in particle physics. It is based on the idea of employing plain-old-data (POD) data structures wherever possible, while avoiding deep-object hierarchies and virtual inheritance. This is to both improve runtime performance and simplify the implementation of persistency services.

At the same time, it provides the necessary high-level functionality to the physicist, such as support for inter-object relations and automatic memory-management. In addition, it provides a (ROOT assisted) Python interface. To simplify the creation of efficient data models, PODIO employs code generation from a simple YAML-based [4] mark-up syntax.

To support the usage of modern software technologies, PODIO was written with concurrency in mind and gives basic support for vectorization technologies.

In the next section we introduce the main design choices for PODIO and their motivation. The follows a section that highlights the key aspects of the underlying implementation and provides some end-user code examples.

This document is mainly based on the current PODIO reference publication [8].
2. DESIGN

Many of the design choices are inspired by previous experience of the LCIO [1] package used for the studies performed by the linear collider groups and the Gaudi Object Description [2] applied in the LHCb collaboration at the LHC. The following four principles are the base of the design:

1. One key idea of PODIO is to use as much as possible plain-old-data (POD) types to keep the memory model simple, provide for fast I/O operations and efficiently support vectorization.
2. Another fundamental design choice is favouring the use of concrete types, relying on composition instead of class inheritance.
3. Based on the problematic experience with object lifetimes in HEP code, in particular when supporting both C++ and Python, a special emphasis was put on a well-defined object ownership.
4. Finally, PODIO relies on code generation, which comes with various advantages, as it:
   1. minimizes user mistakes
   2. provides quick turn-around times for improvements on the back-end as well as for extensions of the EDM
   3. ensures the consistency and homogeneity of the EDM
   4. allows for a quick start in using PODIO

Other driving considerations where the support on equal footing for C++ and Python in the user code, thread-safety and a design that allows multiple I/O storage back-ends. In the following a few details on the resulting design are given. A more detailed summary can be found in the documentation on the project GitHub page [8].

2.1. POD 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 is not strictly correct.

2.2. THE THREE LAYERS OF PODIO

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 provide an ease-of-use to physicists:

1. The POD layer holds the arrays of the actual data structures.
2. The object layer consists of transient, lightweight objects which handle the relations between the individual objects in the EDM. These relations can be either of type one-to-one or one-to-many and are stored in the POD layer using suitable ObjectIDs. 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 of arbitrary size.

3. Finally, the user layer introduces handles to the EDM objects (Hit) and collections of EDM object handles. In most cases only these classes will occur in the user code. Due to the nature of the handle construct this results in a so called value semantics. This shields bare pointers from users in C++ and is the natural representation in Python.

3. IMPLEMENTATION

In this section we give a brief overview on the key aspects of the PODIO implementation. Firstly, the object-ownership is being addressed.

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:

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

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

```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 [4] file. Apart from user convenience and code robustness this also offers fast turnaround times for potential improvements on the back-end 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
- 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 below:

```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. The following listing demonstrates this with a small Python application that prints the energies of all hits for all events.

```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 before, relations need to be implemented differently in every layer. While the user clearly would like to use pointers or vectors of pointers, these cannot be made persistent nor are variable sized objects, such as vectors, 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 from disk we convert the arrays of ObjectIDs back into the corresponding vectors of pointers.
As all of this is done in the object layer the user need not be concerned with the details of the underlying implementation. From a relation defined in *yaml* member functions are generated in the corresponding handle class. The use of these functions is straightforward as is demonstrated in the following two example listings. The creation of cluster vs. hit relations happens as follows, using the reference semantics:

```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);
```

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

```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);
```

### 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 the user code but it has to be made at compile time, whereby one has to keep in mind that the on-demand transformation between complete SoA and AoS representations is highly inefficient. Nevertheless, PODIO provides convenience methods for the on-demand transformation which could be used in dedicated vectorizable code. The use of these functions mandates proper performance measurements on real use cases.
3.5. I/O IMPLEMENTATIONS

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 works 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 can be reached by using PODs for the I/O. One aspect that will need special consideration is the 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 [6] as well as in standalone 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 started to evaluate 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) [7] as an incubator project.
5. FUTURE PLANS / CONCLUSIONS / RELATION TO OTHER AIDA-2020 WORK

The PODIO project will continue to improve the actual I/O performance by implementing and comparing different formats as described in the previous section. We will follow up on the evaluations and needs of the linear collider and LHCb communities as well as other interested HEP groups. PODIO might also be a good candidate for the definition of event data formats for ongoing test beam activities, such as the common DAQ system developed in WP5.
6. REFERENCES


### ANNEX: GLOSSARY

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PODIO</td>
<td>Plain-Old-Data I/O Library</td>
</tr>
<tr>
<td>FCC</td>
<td>Future Circular Collider</td>
</tr>
<tr>
<td>ILC</td>
<td>International Linear Collider</td>
</tr>
<tr>
<td>YAML</td>
<td>Yet Another Markup Language</td>
</tr>
</tbody>
</table>