podio::Frame

A proposal for a generalized data container

Thomas Madlener
Mar 8, 2022
Introduction & Overview
podio::Frame - main purposes

- Act as a container that aggregates and owns all relevant data
- Offer an easy to use and thread safe interface to access those data
  - (Immutable) read access to collections and *meta data*
  - Insert collections and *meta data* (via ”destructive move”)
  - Once inserted into the Frame data is immutable by design and no mutable access is granted afterwards
- Define an *interval of validity* or category (e.g. Event, Run, LumiSection) for the contained data
  - The lifetime of the Frame defines the lifetime of the contained data (and meta data)
podio::Frame - why Frame and not Event

- The **Frame** is a more general concept
  - Functionality for reading, e.g. a **Run** is essentially the same as for reading an **Event**, the two differ mainly by their (data) content
- Having a clear concept of “lifetime” or “interval of validity” without immediately tying that to an interpretation (e.g. **Event**)
  - Making this lifetime explicit instead of having it implicitly defined as right now
- We can probably not exhaustively list experiment differences w.r.t. naming things, but we can offer the necessary I/O functionality
  - Related to the *meta data* discussion, since we cannot foresee all the levels of metadata that are necessary
- Also works for experiments that have no clear notion of an “event” but rather deal with, e.g. read-out frames
Meta data - a definition (attempt)

- *meta data* is all data that does not fit into either the EDM or the *podio::UserDataCollection*
  - Maybe *extra data* would be a better name, as it will probably be used to store additional data as well as “true” meta data
  - The distinction between *meta data* and *extra data* is a bit murky / academic
- Usually implemented as some sort of generic key - value store
- Ideally there aren’t too many use cases for this in the end with podio based EDMs
  - Adding new datatypes is easy in podio
- Basis in podio is *podio::GenericParameters*
  - Offers a key - value storage for *int*, *float* and *std::string* as well as *vectors* thereof
  - (Plan to) not directly expose via the *Frame* interface
General functionality of a Frame

- Access to data read from file / Construct without data
- Possibility to add new data
- Ownership of the contained data
- Support for different I/O libraries
- Potential support for different policies with a single interface
  - Lazy unpacking (`prepareAfterRead` and potential decompression) of collections, e.g.
  - How to handle missing collections
  - (Key / name) collision behavior on collection insertion
- Thread safe for “general use”
  - Inserting and reading from multiple threads will / should not lead to a race condition
  - Probably via mutexes + locking (or maybe via policy?)
The basic interface is rather simple.

The major points are:

- Collections have to be moved into the Frame and become invalid after a call to put.
- It is not possible to get mutable data access.

Some additional functionality is needed for giving a Writer access to the stored data.

*Method naming obviously up for discussion.*
General I/O philosophy and assumptions

- I/O is assumed to be one thread per file
  - Blocks until all requested data is written / read
- Readers provide the data for a “complete” frame in (almost) arbitrary format
  - Can also be a subset of all collections
  - Combination of many frames (e.g. pile up mixing) into one not part of core podio
  - No “lazy reading”, i.e. once the data has left the reader, the frame is detached from it

- Writers request buffers to be written from the frame
  - Does not take ownership of these buffers
  - There can be multiple writers operating on one frame

- **podio provides the necessary building blocks for more complex workflows**
  - E.g. asynchronous reading / writing

---

^1It is in fact not impossible to do this, but it will not be part of core podio
I/O in diagrams

- **single threaded**
  - ROOTReader provides ROOTRawData
  - SIOReader provides SIORawData

- **potentially multithreaded**
  - Reading raw data and constructing a frame from it is a two step process
  - Makes it possible to do unpacking on a different thread than the one that reads

- **potentially multithreaded**
  - Writing can happen with multiple threads, e.g. each writer on its own thread
  - Writers can write different contents, e.g. SIM & RECO into separate files
    - Need one writer “per content”
A `Frame` is constructible from (almost) arbitrary raw data
- Can be different for each I/O system
- Needs to provide access to the buffers of a desired collection

A writer can request (a subset) of collection buffers to write
- Frame takes care of preparing these buffers
- Frame needs to be kept alive until writing all buffers is done
  → writer interface not concerned with this
The user perspective (single threaded)

```cpp
TrackCollection doTracking(const TrackerHitCollection& hits);
VertexCollection doVertexing(const TrackCollection& tracks);

podio::ROOTReader reader("hits.root");
podio::SIOWriter writer("reco_tracks.sio");

for (size_t i = 0; i < reader.getNEntries("event"); ++i) {
    auto event = podio::Frame(reader.readNextEntry("event"));
    const auto& hits = event.get<TrackerHitCollection>("hits");
    auto tmpTracks = doTracking(hits);
    const auto& tracks = event.put(std::move(tmpTracks), "tracks");

    const auto& vertices = event.put(doVertexing(tracks), "vertices");
    const auto& recos = ReconstructedParticleCollection();
    event.put(std::move(recos), "recos");

    writer.writeEntry(event, "event");
}
```

// At this low level we need to know the category
// that we want to read

// Create an event with the contents from the file

// Get hits from event and
// do the tracking

// Store the tracks by moving them into the event
// Retain a const reference for later use
// tmpTracks now in "valid but undefined state"
// and is no longer usable

// Temporaries don't need the explicit move

// Not keeping the const ref is also fine

// Pass (a const ref to) the event to the writer
// Also the writer needs to know the category

// frame goes out of scope and all data is destroyed
General functionality overview summary

- **Frame** acts as owning container of data and defines an *interval of validity* (or category) for this data
  - Takes ownership of inserted data
  - Only gives immutable access to stored data
- Readers provide (almost) arbitrary raw data from which a Frame is constructed
  - The reader relinquishes ownership once the raw data leaves its control
  - No connection between a reader and a Frame
- The writers only get (references to) the buffers of data that should be written
  - Have to make sure that the write operation is done by the time the Frame is destroyed
- podio provides the main building blocks for constructing more complex workflows, but it will not offer “off-the-shelf” solutions for those
Collection
meta data
Collection meta data

- Collection meta data should be easily accessible from the collection directly
  - Currently have to go through the EventStore
- In LCIO each collection owns their own meta data container
  - Written separately for each event and for each collection
- In the Frame approach all meta data is foreseen to be owned by a Frame
- What is the correct lifetime for collection meta data?
  - Do we want the distinction between “true” meta data and extra data with potentially vastly different lifetimes?
- Simplest solution is probably to somewhat follow the LCIO approach
  - Collection meta data lifetime == lifetime of the frame containing collection
  - Pass requests from collection to frame and adapt keys under the hood
    → Still write separately for each frame, but at least only for collections that use it
- Need to touch collection interface in any case
Status & Open Questions
Current status

• Prototype of Frame and some policies without any I/O support yet
  • Mainly to figure out whether the overall implementation strategy could work
• Prototype interface for unpacking policies and two prototype implementations
• Prototype interface for collision policies and two prototype implementations
• Identified some parts in the current I/O handling that definitely need to change
  • Need to touch that in any case for schema evolution, develop together with that
Open questions / discussion points

- Do we want the frame as a concept in Podio?
- How do we handle collection meta data / How do we define its lifetime?
- Which customization points do we want to offer?
  - And what should their default behavior be?
- Which higher level functionality do we still want to offer as part of Podio and which parts do we definitely push to users / a framework?
  - Which (high level) functionality do we need at v1.0?
  - See what works upstream and pull in things that seem of general use?
Frame policies and customization
Frame policies

- We will not be able to satisfy everybody with one Frame implementation
- Offer a few (selected) customization points that allow podio users to alter some of the Frame runtime behavior via policies
- Possible customization points could be
  - The unpacking behavior - lazy (on demand) vs. eager (at Frame construction)
  - Collision handling on insert - throw an exception vs. overwrite existing vs ...
  - Handling of missing data - throw an exception vs. default vs ...
  - Locking policy - e.g. have no locking at all if only used on a single thread
- Policies should be as orthogonal to each other as possible
- Foresee that users want to define their own policies
Frame interface w/ policies

- Constructors get policies as additional argument
  - Rest of the Frame interface remains untouched
- Each policy is essentially a class with few (mostly one) static member function(s) that defines its behavior
- FramePolicies is just an (arbitrary) type with typedefs that specifies which policies to use
- We define (sane) default policies
  - (Average) user doesn’t need to know about them

```cpp
struct FrameDefaultPolicies {
    /// policy for handling the unpacking behavior
    using UnpackingPolicy = EagerUnpacking;
    /// policy for handling name collisions on insertion
    using CollisionPolicy = ThrowOnCollision;
};

struct Frame {
    /// Empty Frame
    template<typename FramePolicies = FrameDefaultPolicies>
    Frame(FramePolicies);

    /// Frame from some RawData read from file
    template<typename RawDataT
            typename FramePolicies = FrameDefaultPolicies>
    Frame(std::unique_ptr<RawDataT>,
            FramePolicies);
    // ...
};
```
Frame customization - usage example

```cpp
auto event = Frame(); // empty frame with default policies
// Empty event with custom policies
auto customEvent = Frame(SomeFramePolicies{});

// Define a custom policy using the defaults as basis
// overriding the unpacking policy
struct LazyUnpackingDefaults : FrameDefaultPolicies {
    // Override the unpacking policy of the defaults
    using UnpackingPolicy = LazyUnpacking;
};

// Create a frame with lazy unpacking and data
auto lazyEvent = Frame(reader.readNextEntry("event"),
                        LazyUnpackingDefaults{});

// Define your own unpacking policy, e.g.
struct HybridUnpacking {
    /// necessary interface defined by how it used inside
    /// Frame, will fail to compile if not fulfilled
};

// No need to inherit from the defaults
struct MyPolicies {
    using UnpackingPolicy = HybridUnpacking;
    using CollisionPolicy = ThrowOnCollision;
};
auto myEvent = Frame(reader.readNextEntry("event"),
                      MyPolicies{});
```

- Assuming that the policies to use are already defined, using them is simple
- Can override some of the defaults or define entirely separate policy type
- The required interface for the individual policies is dictated by the Frame and how they are used there
- Frames with different policies can be used interchangeably
  - Some policies might have the power to change runtime behavior (e.g. collision handling)
Frame implementation basics

• Combination of type erasure\(^2\) and policy based design
  • Allows to have value semantics but with variable runtime behavior
  • Possible to use templated interface (not possible with “classic” polymorphism)
  • Has some implications also for readers & writers

• A Frame is a move-only non-copyable type
• Policies are injected at construction and cannot be changed afterwards
  • With “sane defaults” provided by podio
• Foresee possible nesting of frames
  • Setting up / managing hierarchy not done by core podio

\(^2\)similar to what std::function or std::any are doing, essentially:
  • Define an internal abstract base class
  • Implement that base class with an internal template class
  • Wrap everything into the actual class the user sees and interacts with
RawDataT requirements / interface

```cpp
struct FrameRawData {
    /// Try to retrieve the buffers for the desired
    /// collection (by name)
    std::optional<podio::CollectionBuffers>
    getCollectionBuffers(std::string);

    /// Get the metadata of this frame
    podio::GenericParameters getMetaData();

    /// Get a list of all available collection (names)
    const std::vector<std::string>&
    getAvailableCollections() const;
};
```

- Any class that fulfills this interface can be used to construct a **Frame**
- Decouples the **Frame** from any details of the actual data reading

- Can be queried for the buffers of a collection or get metadata
  - Ownership of these buffers then goes to the **Frame**
- Information about all the available collections
  - Mainly for unpacking all collections
- No concurrent access foreseen
  - Ensured by containing frame

---

1 This can in principle be used to implement thread safe, on-demand reading a la CMSSW or other approaches as long as internal concurrency is handled appropriately
Reader / Writer interfaces
Minimal Reader interface

```cpp
struct Reader {
    /// Constructor directly opens the passed file
    Reader(std::string filename);

    /// Get the number of entries that are stored for a
    /// given category
    size_t getNEntries(std::string category) const;

    /// Get the raw data (in principle in arbitrary format)
    /// for the next entry of a given category
    std::unique_ptr<ReaderRawData> getNextEntry(std::string category);
};
```

- Low level interface / building block
- **No concurrent access foreseen**
  - Callers have to synchronize
- Potential for more “utility” functionality
  - Reading a specific entry for a category (or skipping entries)
  - Having a list of (equivalent) files to read from
  - Reading only partial entries (if supported by I/O)
  - “Notification” of category change (e.g. for SIO)
Minimal Writer interface

```cpp
struct Writer {
    /// Constructor automatically opening output file
    Writer(std::string filename);

    /// write the passed entry under the given category
    void writeEntry(const podio::Frame& frame, std::string category);
};
```

- Low level interface / building block
- Ensures proper writing of different categories
- No concurrent access foreseen
- Callers have to synchronize
- Potential for more “utility” functionality
  - Writing only parts of the frame (e.g. via `registerForWrite` or via dedicated argument to `writeEntry`)
  - Changing frame contents within a category (if supported by I/O)
Implications of type erased Frame for Readers & Writers

- In principle easily possible to have a type erased Frame and polymorphic IReader and IWriter interfaces
- Can also implement type erased Reader and Writer classes that wrap the actual implementations
- Functionality wise no real difference
- Some “syntactic” differences in usage
  - Pointer semantics (polymorphism) vs. value semantics (type erasure)
  - Can construct a consistency argument for podio out of that
- Some differences also for implementors of readers / writers
  - Polymorphic interfaces will always require inheritance
  - Type erasure makes it possible to wrap arbitrary implementations (as long as they offer the correct interface)
Supplementary Material
LCIO workflows / capabilities

- LCIO has the **LCEvent** that gives access to
  - The data stored in collections (defined by the EDM)
  - Some meta data (e.g. run & event number, weight, ...)
  - Meta/extra data in form of **LCParameters**

- **LCParameters** are essentially equivalent to podios **GenericParameters**

- Each collection has an instance of their own **LCParameters** attached
  → collection meta data has a lifetime of “event”

- Additionally there is the possibility to store collections of **LCGenericObjects**
  - Indexed based access to vectors of **int**, **float**, **double**

- LCIO also has an **LCRunHeader** with some meta data and **LCParameters**
Current podio vs. LCIO

- Overview table over the non-EDM possibilities of the two libraries

<table>
<thead>
<tr>
<th>Use case</th>
<th>LCIO</th>
<th>podio</th>
</tr>
</thead>
<tbody>
<tr>
<td>(arbitrary) user data</td>
<td>LCGenericObject</td>
<td>UserDataCollection</td>
</tr>
<tr>
<td>key-value @event</td>
<td>LCPARAMETERS of LCEVENT</td>
<td>GenericParameters (event meta data)</td>
</tr>
<tr>
<td>key-value @collection</td>
<td>LCPARAMETERS of collections</td>
<td>GenericParameters (collection meta data)</td>
</tr>
<tr>
<td>key-value @run</td>
<td>LCPARAMETERS of LCRUNHEADER</td>
<td>GenericParameters (run meta data)</td>
</tr>
</tbody>
</table>

- In podio all the different levels of metadata are currently exposed via the EventStore

- In both cases the users get direct access to the whole LCPARAMETERS / GenericParameters object
  - Enforcing immutability would be extremely restricting for the users
  - Without immutability constraints $\rightarrow$ sequence point in multithreaded contexts
Some object sizes for overhead considerations

- Size of some internals of the `GenericParameters` as well as the probably largest `edm4hep` data type

<table>
<thead>
<tr>
<th>object</th>
<th>size / bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>std::map&lt;K, V&gt;</code></td>
<td>46</td>
</tr>
<tr>
<td><code>std::unordered_map&lt;K, V&gt;</code></td>
<td>56</td>
</tr>
<tr>
<td><code>podio::GenericParameters</code></td>
<td>46 * 4 or 56 * 4</td>
</tr>
<tr>
<td><code>edm4hep::ReconstructedParticleData</code></td>
<td>116</td>
</tr>
<tr>
<td><code>edm4hep::ReconstructedParticleObj</code></td>
<td>116 + 68</td>
</tr>
<tr>
<td><code>edm4hep::ReconstructedParticleCollectionData</code></td>
<td>280</td>
</tr>
<tr>
<td><code>edm4hep::ReconstructedParticleCollection</code></td>
<td>280 + 16</td>
</tr>
</tbody>
</table>
"Classic" polymorphism

"Library side"

- Defines an abstract interface

```cpp
struct IReader {
    virtual std::string read() = 0;
};
```

"User side"

- Implementations have to inherit from IReader

```cpp
struct ROOTReader : public IReader {
    std::string read() override { return "root"; }
};

struct SIOReader : public IReader {
    std::string read() override { return "sio"; }
};
```

- Usage requires pointer semantics

```cpp
std::vector<std::unique_ptr<IReader>> readers;
readers.push_back(std::make_unique<ROOTReader>());
readers.push_back(std::make_unique<SIOReader>());

for (auto r : readers) std::cout << r->read();
```
Type erasure

“Library side”

- Essentially internalizes the abstract base interface

```cpp
class Reader {
    struct ReaderConcept {
        virtual std::string read() = 0;
    };

    template<typename R>
    struct ReaderModel final : public ReaderConcept {
        ReaderModel(R r) : m_reader(r) {}
        std::string read() final { return m_reader.doRead(); }
        R m_reader;
    };

    std::unique_ptr<ReaderConcept> m_self;

public:
    template<typename R>
    Reader(R r) :
        m_self(std::make_unique<ReaderModel<R>>(r)) {}

    std::string read() { return m_self->read(); }
};
```

“User side”

- Implementations are free standing classes that have to fulfill the interface required by the ReaderModel

```cpp
struct ROOTReader {
    std::string doRead() { return "root"; }
};

struct SIOReader {
    std::string doRead() { return "sio"; }
};

- Can be used with value semantics

```cpp
std::vector<Reader> readers;
readers.emplace_back(ROOTReader{});
readers.emplace_back(SIOReader{});

for (auto &r : readers) std::cout << r.read();
```