podio schema evolution

(or the gift that keeps on giving)

What makes schema evolution complicated in podio?

- podio is “just” the generator for datamodels
  - EDMs always start from just the YAML definition
  - Cannot simply implement the necessary operations during deserialization in C++
- The main ROOT based backend already has a schema evolution mechanism
  - Quite powerful, no need to re-invent the wheel here
  - podio now needs to be able to generate code that has schema evolution as opt-in
- Schema evolution is complicated in general
  - Parts of it can be done automatically, others require manual intervention
- The first implementation was rushed to get podio v1.0 out
  - We cut a few corners …

Goals for podio schema evolution

- Check schema changes during code generation
  - Flag everything that cannot be done automatically (by ROOT for now)
- Handle multiple old schema versions
  - Always evolve directly to the current version, no incremental evolution
- Allow for manual intervention if desired or necessary
  - E.g. non-trivial schema changes
- Users don’t see any of this

podio schema evolution (June 2025)

- Works for everything that ROOT can do automatically + a bit more
  - Adding / removing members
  - Implicit floating point conversions
  - Renaming members
- Partially tested
  - Rather convoluted test setup
  - Features that we claimed we support were no supported
    - Schema evolution mechanism does not support renaming datatype members
    - Schema evolution does not work for renamed datatypes
- Code generation should be able to handle several older schema versions not just one

Step 1: Refactor testing - Why?

- General approach: write data in old format and verify we can read it in new format
- Old test setup used an (almost) copy of the test datamodel
  - With a few changes here and there to test schema evolution
  - The two versions have started to diverge in some details unrelated to schema changes
  - Writes one file with several (old) collections and reads them back as new collections
  - Hard to verify what is actually tested (datatype names are not related to schema changes)
  - Adding new tests really cumbersome
- A better system is necessary
  - Isolated tests for individual schema changes
  - Easy to add / define new tests (+documentation)

Step 1: Refactor testing - How?

- AIDASoft/podio#817 for all details
- Remove all old tests
- Introduce some CMake machinery
  - Compile both versions of datamodel
  - Setup appropriate test environments for writing and reading (isolate dictionaries and libraries)
- Introduce some C++ machinery
  - Easily define checks in one source file
  - Reduce boilerplate
- Compile two binaries from source file
  - Some pre-processor trickery driven by CMake

```
ADD_SCHEMA_EVOLUTION_TEST(components_new_member)
ADD_SCHEMA_EVOLUTION_TEST(datatypes_new_member)
ADD_SCHEMA_EVOLUTION_TEST(implicit_floating_point_change)
ADD_SCHEMA_EVOLUTION_TEST(components_rename_member WITH_EVOLUTION)
ADD_SCHEMA_EVOLUTION_TEST(datatypes_rename_member WITH_EVOLUTION)
```

```
10:42:27 madlener@local:code_gen$ ls datatypes_new_member/
check.cpp  new.yaml  old.yaml
10:42:40 madlener@local:code_gen$ ls components_rename_member/
check.cpp  evolution.yaml  new.yaml  old.yaml
```

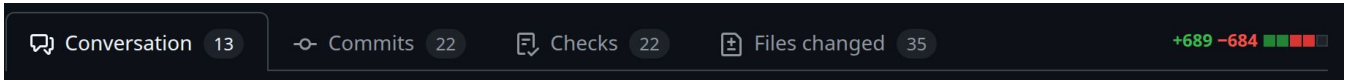
```
#include "datamodel/TestTypeCollection.h"

#include "check_base.h"

int main() {
  WRITE_AS(TestTypeCollection, {
    elem.comp().f = 3.14f;
    elem.energy(1.234);
  });

  READ_AS(TestTypeCollection, {
    ASSERT_EQUAL(elem.comp().f, 3.14f, "Implicit conversion
for component members doesn't work");
    ASSERT_EQUAL(elem.energy(), 1.234f, "Implicit conversion
for datatype members doesn't work");
  })
}
```

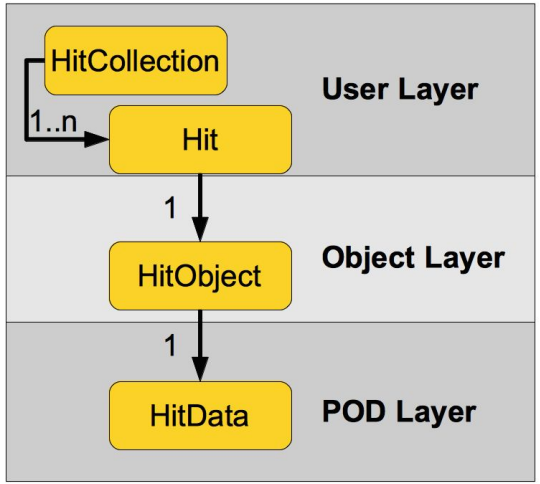
Step 1: Refactoring tests - Results



- Overhead in code for mini test framework compensated by removal of datamodel duplication
  - **Including ~100 lines of docstrings and documentation**
- Additional tests covering RNTuple
- More maintainable and extensible for future additions to schema evolution
  - LLMs can now do it almost first try ;)

Step 2: Version the Data

- Reminder: podio approach has three layers
  - Data stored as `vector<XYZData>`
- Currently we store the data “unversioned”
  - Only generate the current version
- Would like to be able to read back older versions
  - Need the definition of older versions
  - Initially triggered by ROOT bugs:
    - Original root-forum post
    - root#19973, root#19650, root#19644
- Caveat:
  - Changing type names breaks ROOTs automatic schema evolution



Step 2: Version the Data

AIDASoft/podio#803

- Several approaches possible
  - **Always introduce types (even if they have identical definitions)**
  - Detect changes and only introduce new types when necessary
- Put old versions into a version namespace
- Keep current version outside of version namespace
  - Keep ROOT schema evolution working
  - SIO is not sensitive to the exact type name
  - All but the POD layer can remain unchanged
  - **Keep existing datafiles readable without changes**

```
namespace datamodel {
struct XYZData { float e; };

// Just for future proofing, already
// declare this version as namespaced.
namespace v2 {
using XYZData = datamodel::XYZData;
}  // namespace v2

namespace v1 {
struct XYZData { float e; };
}  // namespace v1
}  // namespace datamodel
```

Step 3: Adapt code generation to handle multiple versions

- Sneak preview at AIDASoft/podio#828
- TL;DR: Works for schema changes that were already supported before
  - Adding / removing members, renaming members
  - **Requires change of grammar of the evolution yaml files** (breaking change!)
- More next week (if people are interested)

Next step: Extend evolution grammar for manual changes

- Needs to work for *ROOT IORules*
- Needs to be re-usable for other backends