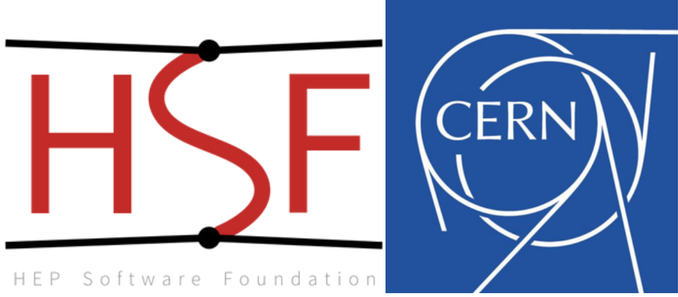


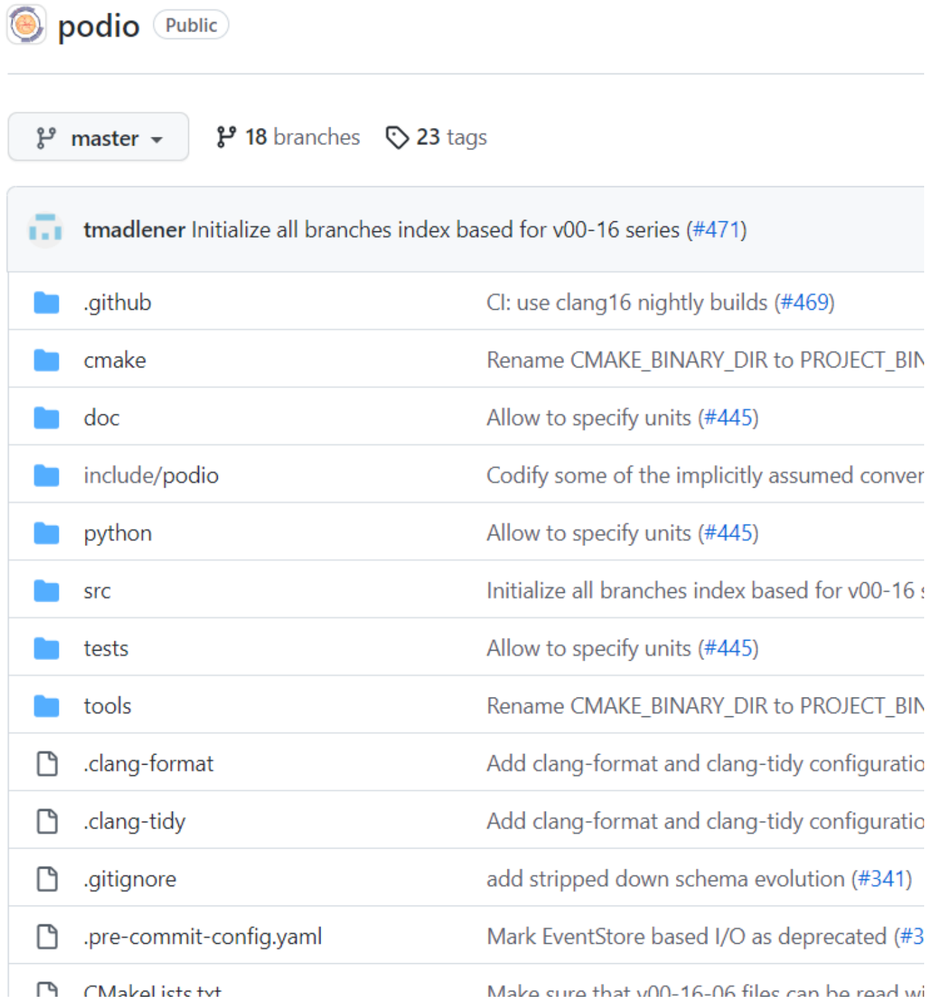
Enhancing PODIO: Enabling Julia Code Generation for HEP Data Models

Ananya Gupta
Indira Gandhi Delhi Technical University for Women

Mentors: Benedikt Hegner (CERN), Graeme A Stewart (CERN)

HSF-INDIA

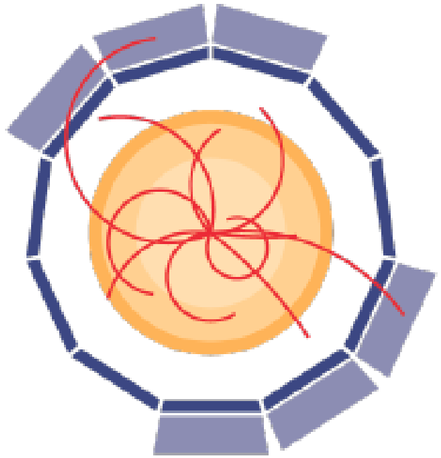
About PODIO



- PODIO is a Python library designed for particle physics data modeling.
- It focuses on plain-old-data (POD) structures for improved performance and simplicity.
- PODIO offers high-level functionality for inter-object relations and memory management.
- It provides a Python interface which supports ROOT and SIO persistency backends.
- A YAML file describing the data model is provided to the Python interface, which then generates C++ code, streamlining the data model creation process.

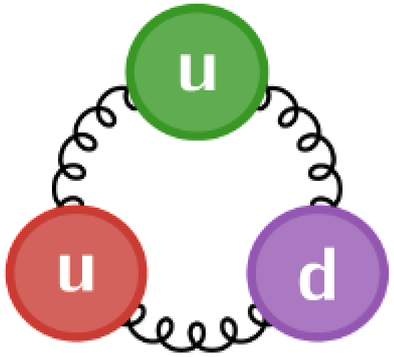
HSF-INDIA

About Project



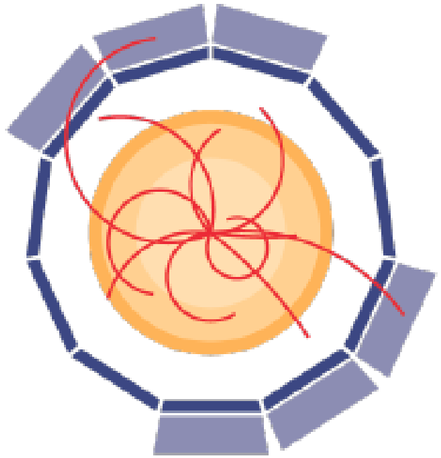
- **Primary Objective:** Add Julia language support in the PODIO library.
- **Project Goal:** Preserve PODIO's performance optimizations and expand its functionalities.
- **Project Focus:** Incorporate Julia code generation in the existing Python interface.
- **Project Outcome:** Provide HEP researchers with the option to generate Julia code for their data models and utilize its capabilities.

Why Julia?



- **High Performance:** Julia provides computational speed on par with C/C++, ensuring efficient data processing and analysis for HEP researchers.
- **User-Friendly Syntax:** Julia offers a Python-like syntax, making it accessible and easy to use for HEP researchers, streamlining their workflow.
- **Scientific Data Processing:** Julia is purpose-built for scientific data processing and offers a wealth of robust capabilities and libraries useful for HEP researchers.

Project Goals



- **Julia Language Support:** Add Julia language support in the PODIO library.
- **Preserve PODIO's Performance Optimizations:** Ensure the retention of PODIO's performance optimizations.
- **Leverage Feasibility from Prototype:** Build upon the feasibility demonstrated in the prototype developed during Google Summer of Code 2022.
- **Julia Code Generation:** Implement Julia code generation seamlessly within the existing Python interface.
- **Project Result:** Provide HEP researchers the option to generate Julia code for their data models and utilize its capabilities for enhanced data processing and analysis.

Shortcomings in the Previous Prototype

EventInfo.jl

datamodel > EventInfo.jl > ...

```
include("EventInfoStruct.jl")

module EventInfoModule
export EventInfo

function EventInfo()
    return Main.EventInfoStruct(
    Int32(0),
    )
end
end
```

CompWithInit.jl

datamodel > CompWithInit.jl > ...

```
include("CompWithInitStruct.jl")
using StaticArrays

module CompWithInitModule
export CompWithInit

function CompWithInit()
    return Main.CompWithInitStruct(
    Int32(0),
    Main.MVector{10, Float64}(undef),
    )
end
end
```

.jl

datamodel > .jl

```
include("SimpleStruct.jl")
include("NotSoSimpleStruct.jl")
include("StructWithFixedWithTypes.jl")
include("CompWithInit.jl")
include("EventInfo.jl")
include("ExampleHit.jl")
include("ExampleMC.jl")
include("ExampleCluster.jl")
include("ExampleReferencingType.jl")
include("ExampleWithVectorMember.jl")
include("ExampleWithOneRelation.jl")
include("ExampleWithArrayComponent.jl")
include("ExampleWithComponent.jl")
include("ExampleForCyclicDependency1.jl")
include("ExampleForCyclicDependency2.jl")
include("ExampleWithDifferentNamespaceRelations.jl")
include("ExampleWithArray.jl")
include("ExampleWithFixedWidthIntegers.jl")
include("ExampleWithUserInit.jl")

module

using ..SimpleStructModule: SimpleStruct
export SimpleStruct
using ..NotSoSimpleStructModule: NotSoSimpleStruct
```

- ".jl" file and anonymous module were generated for components and datatypes that did not have a namespace.
- There were no default parameters and thus user have to initialize empty constructor definitions first, before passing value to them.

Shortcomings in the Previous Prototype

ExampleReferencingType.jl

datamodel > ExampleReferencingType.jl > ...

```
1   include("ExampleReferencingTypeStruct.jl")
2   include("ExampleClusterStruct.jl")
3
4   module ExampleReferencingTypeModule
5   export ExampleReferencingType
6
7   function ExampleReferencingType()
8       return Main.ExampleReferencingTypeStruct{Main.
        ExampleClusterStruct,Main.
        ExampleReferencingTypeStruct}(
9       Vector{ Main.ExampleClusterStruct }(),
10      Vector{ Main.ExampleReferencingTypeStruct }(),
11      )
12  end
13  end
```

EventInfo.jl

datamodel > EventInfo.jl > ...

```
1   include("EventInfoStruct.jl")
2
3   module EventInfoModule
4   export EventInfo
5
6   function EventInfo()
7       return Main.EventInfoStruct(
8       Int32(0),
9       )
10  end
11  end
```

CompWithInit.jl

datamodel > CompWithInit.jl > ...

```
1   include("CompWithInitStruct.jl")
2   using StaticArrays
3
4   module CompWithInitModule
5   export CompWithInit
6
7   function CompWithInit()
8       return Main.CompWithInitStruct(
9       Int32(0),
10      Main.MVector{10, Float64}(undef),
11      )
12  end
13  end
```

- The structs, constructors as well as the collections all were returned in the Main scope thus polluting it. A good design choice in Julia would be to not pollute the Main Scope.
- The prototypic implementation had way too many separate modules, which unless is necessary should be avoided.

Workaround 1

Creating a single module for each data type and component such that the module for data type consists of struct, constructor and collection definitions and for component consists of struct and constructor definitions.

File: “<Component_name>.jl”

```
include(“<Other_Component_name>.jl”) #Members

module <component_name>
export <component_name>Struct  # Struct
export <component_name>  # Constructor

using ..<other_component_name> #sibling modules

mutable struct <component_name>Struct
 ...
end

function <component_name>(
 ... #default parameters
)
return <component_name>Struct(
 ...
)
end
end
```

File: “<Datatype_name>.jl”

```
include(“<Component_name>.jl”) # Members
include(“<Other_Datatype_name>.jl”) # One to One, One to Many Relations, Members

module <datatype_name>
export <datatype_name>Struct # Struct
export <datatype_name> # Constructor
export <datatype_name>Collection

using ..<component_name>
using ..<datatype_name> # sibling modules

mutable struct <datatype_name>Struct
 ...
end
function <datatype_name>(
 ... #default parameters
)
return <datatype_name>Struct(
 ...
)
end
<datatype_name>Collection = Vector{
<datatype_name>Struct }
end
```

File: “<package_name>.jl” # Parent Module

```
module <package_name>
include(“<Component_name>.jl”)
include(“<Datatype_name>.jl”)

using .<component_name>
export <component_name>

using .<datatype_name>
export <datatype_name>
export <datatype_name>Collection

end
```

Issues in Workaround 1

This approach was a great workaround but it failed Cyclic Dependency tests. During Cyclic Dependency tests the above design choice resulted in an infinite loop of file includes.

ExampleForCyclicDependency1.jl

datalayout > ExampleForCyclicDependency1.jl > ...

```
1  include("ExampleForCyclicDependency2.jl")
2
3  module ExampleForCyclicDependency1Module
4  export ExampleForCyclicDependency1
5  export ExampleForCyclicDependency1Struct
6  export ExampleForCyclicDependency1Collection
7
8  using ..ExampleForCyclicDependency2Module
```

ExampleForCyclicDependency2.jl

datalayout > ExampleForCyclicDependency2.jl > ...

```
1  include("ExampleForCyclicDependency1.jl")
2
3  module ExampleForCyclicDependency2Module
4  export ExampleForCyclicDependency2
5  export ExampleForCyclicDependency2Struct
6  export ExampleForCyclicDependency2Collection
7
8  using ..ExampleForCyclicDependency1Module
```

ExampleForCyclicDependency1.jl datalayout 1

Loop detected, this file has already been included. Julia(IncludeLoop) [Ln 1, Col 1]

ExampleForCyclicDependency2.jl datalayout 1

Loop detected, this file has already been included. Julia(IncludeLoop) [Ln 1, Col 1]

Workaround 2

In this workaround we reduced the number of modules to the number of namespaces the datamodel.yaml file has. Constructor definitions for components and both constructor and collection definitions for data types were placed in modules with names corresponding to their respective namespaces.

File: “<Component_name>Struct.jl”

```
include(“<Other_Component_name>Struct.jl”) # Members
mutable struct <component_name>Struct
 ...
end
```

File: “<Datatype_name>Struct.jl”

```
include(“<Component_name>Struct.jl”) # Members
include(“<Other_Datatype_name>Struct.jl”) # Members

mutable struct <datatype_name>Struct
 ... # use of parametric types
end
```

File: “<package_name>.jl” # Parent Module

```
module <namespace1>
export <component_name>
export <datatype_name>
export <datatype_name>Collection

using ..namespace2
include(“<Component_name>Struct.jl”)
include(“<Datatype_name>Struct.jl”)

function <component_name>(
 ... #default parameters
)
return <component_name>Struct(
 ...
)
end
function <datatype_name>(
 ... #default parameters
)
return <datatype_name>Struct(
 ...
)
end
<datatype_name>Collection = Vector{
<datatype_name>Struct }
end
```

```
module <namespace2> # Code contd.
export <component_name>
export <datatype_name>
export <datatype_name>Collection

using ..namespace1
include(“<Component_name>Struct.jl”)
include(“<Datatype_name>Struct.jl”)

function <component_name>(
 ... #default parameters
)
return <component_name>Struct(
 ...
)
end
function <datatype_name>(
 ... #default parameters
)
return <datatype_name>Struct(
 ...
)
end
<datatype_name>Collection = Vector{
<datatype_name>Struct }
end
```

HSF-INDIA

Issues in Workaround 2

The issue with the above design choice is that module ‘namespace2’ is being used within module ‘namespace1’, even though it's declared after module ‘namespace1’.

This results in an error in Julia, and it cannot be resolved because **Julia does not support forward declarations**. Therefore, there is no solution or method to achieve the above design choice without encountering errors.

Support for forward declarations is a long standing issue in the JuliaLang/julia.

Issue: https://github.com/JuliaLang/julia/issues/269

Thus we can’t use this workaround.

Workaround 3 (Approved)

In this workaround, we considered consolidating the modules into a single module identified by the package name. Our decision was to place constructor definitions for components and both constructor and collection definitions for data types within this single module.

File: “<Component_name>Struct.jl”

```
include(“<Other_Component_name>Struct.jl”) # Members
mutable struct <component_name>Struct
 ...
end
```

File: “<Datatype_name>Struct.jl”

```
include(“<Component_name>Struct.jl”) # Members
include(“<Other_Datatype_name>Struct.jl”) # Members

mutable struct <datatype_name>Struct
 ... # use of parametric types
end
```

File: “<package_name>.jl” # Parent Module

```
module <package_name>
export <component_name>
export <datatype_name>
export <datatype_name>Collection

include(“<Component_name>Struct.jl”)
include(“<Datatype_name>Struct.jl”)

function <component_name>(
 ... #default parameters
)
return <component_name>Struct(
 ...
)
end
function <datatype_name>(
 ... #default parameters
)
return <datatype_name>Struct(
 ...
)
end
<datatype_name>Collection = Vector{
<datatype_name>Struct }
end
```

Design Choice Analysis: Workaround 3

Advantages:

- **Simplified Module Structure:** This approach employs a single module, which eliminates complexities in the code structure, making it more straightforward and manageable.
- **Overcoming Prototype Shortcomings:** Workaround 3 effectively addresses and resolves all the limitations found in the prototype, enhancing the overall robustness of the solution.
- **Potential for Julia Package:** The generated code can be seamlessly transformed into a Julia package, offering reusability and scalability.

Limitations:

Namespace Constraint: All the constructors, and collection definitions for components and data types are consolidated within a single module, regardless of their respective namespaces in the datamodel.yaml file.

Consequently, users are **required to use unique names** for data types and components, even across different namespaces.

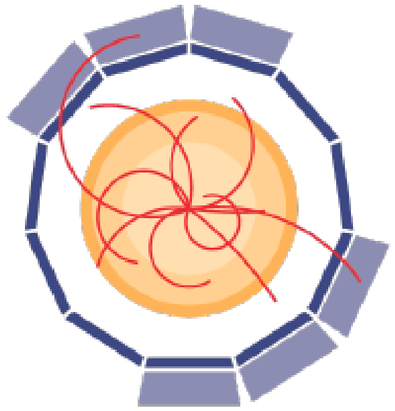
HSF-INDIA

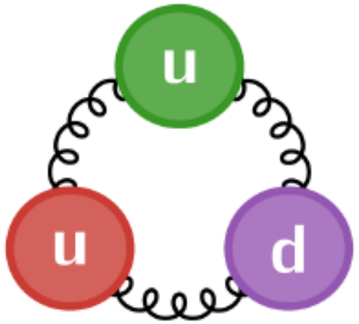
Project Milestones and Progress

- Resolved the empty namespace issue by assigning the ' .jl' file and anonymous module the ‘package_name’.
- Added default parameters in constructor definitions with support for Abstract types.
- Implemented a new design structure for Julia code based on workaround 3.
- Fixed tests in the unit test suite, covering the Julia code generation of the example data model.
- Organized file includes in lexicographical order in struct definition files.
- Created a pull request (PR) to merge the aforementioned changes into the PODIO library. PR#473
- Created a prototype Edm4hep Julia package based on workaround 3 for design and code robustness testing.

Key Learnings from the Project

- Julia
- Julia in High Energy Physics (Julia HEP)
- Unit Testing (Julia)
- Advanced Git Concepts
- CMake Build and Testing





HSF-INDIA

Thank You