Key4hep

EP R&D Software Working Group Meeting

Plácido Fernández Declara

June 30, 2021

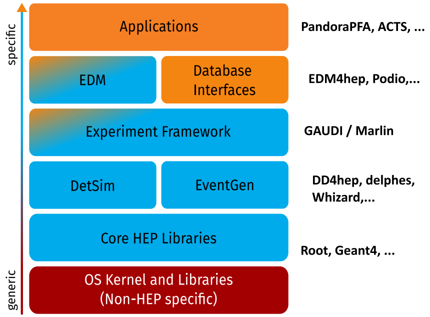
CERN

Table of contents

Key4hep

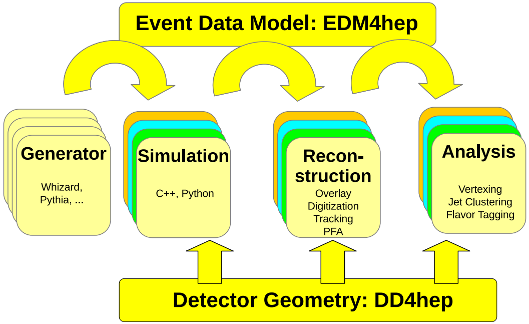
Key4hep

- It is a data processing software stack for future detector studies.
- Provide a complete software solution using the best software components.
- Experiments use a mix of common and very specific packages, were some of them actually are very similar in concept.
- Data processing framework: fast and full simulation, reconstruction, and analysis.
- Part of the Strategic R&D Programme on Technologies for Future Experiments



Key4hep components

- Key4hep has contributions from CLIC, CEPC, FCC, ILC communities
- Event Data Model: EDM4hep
  - PODIO used to create EDM4hep.
- Geometry information: DD4hep
- Packaging and development: Spack
- Simulation:
  - Fast simulation: k4SimDelphes (Delphes)
  - Full simulation: k4SimGeant4 (DDG4)
- Reconstruction & Analysis: k4MarlinWrapper



Key4hep common tooling

- Not only common software HEP tools, but also general software tools and practices.
- Templates provided for new packages to follow common structure with folders.
- Common use of build systems: CMake, gcc, clang.
- Encourage use of common software analysis tools: sanitizers.
- Encourage uniform consistent code style and practices: clang format, clang tidy.
- Common testing tools: Catch2, Pytest.
- Unified building and deployment with Spack and CVMFS.
- Moving towards common interfaces, frameworks, packages, versions, etc.
- Modern tooling: compilers, language standards,

Common Gaudi Framework Status

Common Gaudi Framework Status

- C++ application framework built at CERN.
- Well known and battle tested in HEP by LHCb and ATLAS
- Gaudi used as common framework to build several Key4hep components.
- It offers key components to develop Reconstruction, Analysis, Simulation, etc
    - Algorithms, Services, Tools, Transient Event Store, Converters, etc.
- Gaudi for parallel processing
    - Adaptation for efficient multi- and many-core processing.
    - Gaudi::Functional.
    - More modern, simple and more uniform code.

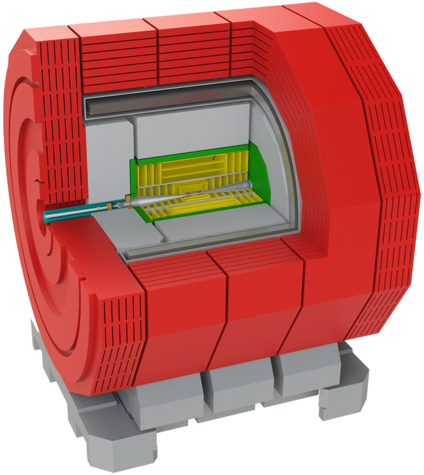
Common Gaudi Framework Status

- k4FWCOre: Data Service for PODIO collections.
- K4-project-template: template for upcoming Key4hep packages and components.
- k4SimDelphes: Delphes objects to EDM4hep
- k4SimGeant4: Geant4 based simulation (fast and full sim).
- k4Pandora: integration into Key4hep with Gaudi.
- k4MarlinWrapper: Marlin reconstruction algorithms to be used in Key4hep.
- Other packages, and new packages encourage use of Gaudi.
    - Latest Gaudi is preferred.

k4MarlinWrapper

k4MarlinWrapper I

- Bring existing, battle-tested algorithms and software from iLCSoft to future colliders.
- Integrate in a smooth non-disruptive way.
- Run algorithm chains that can contain previous, current and future algorithms.
- Added support for interfaces and converters that allow for the integration.



k4MarlinWrapper II

- Part of the Key4hep[1] project:
  - Brings analysis and reconstruction to the common software stack.
- *Marlin* Processors functionality made available in Key4hep through the *Gaudi* framework.
- It contains the necessary interfaces to deal with *Marlin* formats to be run from *Gaudi* algorithms.
  - Wrapper around Marlin Processors.
  - XML steering file to Python options file converter.
  - In-memory converters between event data models.
- Marlin source code is kept intact, and can be called on demand.

[1] `https://github.com/key4hep/`

Dependencies

k4MarlinWrapper can be built against the Key4hep CVMFS view. Main dependencies:

- Gaudi: to wrap Marlin processors and run the algorithms.
- Marlin: to run the underlying processors.
  - It will eventually disappear when only Gaudi Algorithms are used.
- LCIO: Event Data Model input/output used by Marlin.
- EDM4hep: Event Data Model input/output to be used across the framework.
  - Other event data models could be integrated.
- k4FWCore, k4LCIOReader, podio: leveraging synergies between other Key4hep packages and related.

Other general dependencies:

- ROOT, Boost

Configuration and running

- Config and running done via Python file as with the Gaudi Framework.
- Processor parameters defined for each instance, and list algorithms configured.
- On algorithm initialization of Marlin Processors, the `MARLIN_DLL` environment variable is used to load the necessary libraries.

```
MyTPCDigiProcessor = MarlinProcessorWrapper("MyTPCDigiProcessor"
MyTPCDigiProcessor.OutputLevel = INFO
MyTPCDigiProcessor.ProcessorType = "DDTPCDigiProcessor"
MyTPCDigiProcessor.Parameters = {
             "DiffusionCoeffRPhi": ["0.025"],
             "DiffusionCoeffZ": "0.08",
             "DoubleHitResolutionRPhi": ["2"],
             "DoubleHitResolutionZ": ["5"],
             "HitSortingBinningRPhi": ["2"],
             "HitSortingBinningZ": ["5"],
             "MaxClusterSizeForMerge": ["3"],
             "N_eff": ["22"],
             # ...
             }
algList.append(MyTPCDigiProcessor)
```

XML to Python converter

- A converter from XML steering file to Python options file is available as a Python script.
- It produces the list of Gaudi algorithms, including optional Processors.
  - These are left as commented algorithms that need to be manually uncommented by the user.
  - A comment is also included to indicate its configuration.
  - `# algList.append(MyFastJetProcessor) # Config.OverlayNotFalse`
- It now includes *Constants* parsing from the XML
  - It lists the `CONSTANTS = ` to be modified by the user
  - These are replaced in the processors with String substitution: `"%(DD4hepXMLFile_subPath)s" % CONSTANTS`
  - It now supports lists of arguments in the constants as well
- `Marlin -x` can create a steering file containing all the parameters for the known processors. This can be converted to python.

Listing algorithms

Incomplete options file example:

```
inp = PodioInput('InputReader')
InitDD4hep.ProcessorType = "InitializeDD4hep"
VertexFinder.ProcessorType = "LcfiplusProcessor"
JetClusteringAndRefiner.ProcessorType = "LcfiplusProcessor"
MakeNtuple.ProcessorType = "LcfiplusProcessor"
MyLCIOOutputProcessor.ProcessorType = "LCIOOutputProcessor"
output = PodioOutput("PodioOutput", filename = "my_output.root")

algList.append(input)
algList.append(InitDD4hep)
algList.append(VertexFinder)
algList.append(JetClusteringAndRefiner)
algList.append(MakeNtuple)
algList.append(MyLCIOOutputProcessor)
algList.append(output)
```

Reading input for k4MarlinWrapper

- **Reading LCIO** events can be managed by using the *Event Data Service*.
- A *LcioEvent* Algorithm wrapped by *k4MarlinWrapper* with path to input file will load the collections.
- To **read EDM4hep** events, *k4DataSvc* is to be used.
- A *PodioInput* Algorithm is used to indicate the collections to be read.

```python
from Configurables import EventDataSvc, LcioEvent

evtsvc = EventDataSvc()

read = LcioEvent()
read.OutputLevel = DEBUG
read.Files = ["path/to/file.slcio"]

algList.append(read)
```

```python
from Configurables import k4DataSvc, PodioInput

evtsvc = k4DataSvc('EventDataSvc')
evtsvc.input = 'path/to/file_EDM4hep.root'

inp = PodioInput('InputReader')
inp.collections = ['ReconstructedParticles',
    'EFlowTrack']
```

Writing output for k4MarlinWrapper

- To **write LCIO** events, a standard *MarlinProcessorWrapper* is used, selecting it to be a *LCIOOutputProcessor*.
- The relevant parameters of the collections are indicated to keep or drop collections, with other configuration options.
- To **write EDM4hep** events, *PodioOutput* is used.
- Output commands can be set to keep or drop collections.

```
from Configurables import MarlinProcessorWrapper

Output_DST = MarlinProcessorWrapper("Output_DST")
Output_DST.ProcessorType = "LCIOOutputProcessor"
Output_DST.Parameters = {"DropCollectionNames": [],
                         "DropCollectionTypes": ["MCPar
                         }
algList.append(Output_DST)
```

```
from Configurables import PodioOutput

out = PodioOutput("PodioOutput",
    filename = "my_output.root")
out.outputCommands = ["keep *"]

algList.append(out)
```

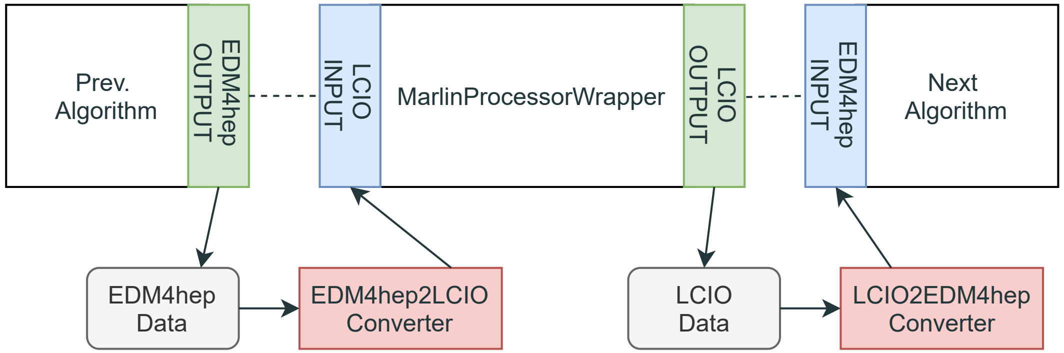
k4MarlinWrapper documentation

- Documentation for k4MarlinWrapper lives in two main places:
  - The Github repository
  - Key4hep documentation: https://key4hep.github.io/key4hep-doc/
- Developer focused documentation is found in the repository
  - It includes instructions on configuring, building and installing it.
  - Running instructions are included, directly calling the produced binary.
  - How read/write collections in different EDM supported formats.
  - How to use the EDM conversion Gaudi Tools.
- User focused documentation is found in Key4hep documentation:
  - Section ”Using the Key4hep-Stack for CLIC Simulation and Reconstruction”
- A set of simple, complex and unit test provide examples on how to run different uses cases.
- All documentation should (and will) live in the Key4hep documentation webpage.

EDM converters for tracking and analysis

EDM4hep <-> LCIO conversion

- In memory conversion between EDM4hep and LCIO needed to run Marlin Processors and Edm4hep based Gaudi Algorithms at the same time.



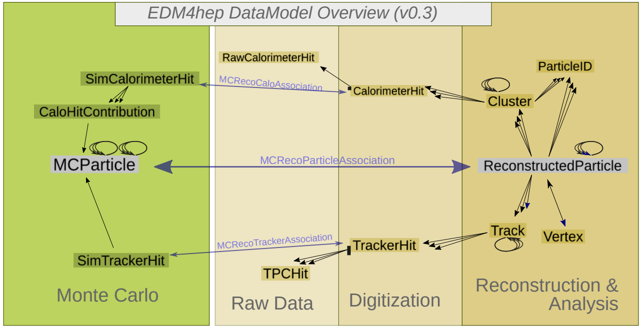
EDM4hep to LCIO conversion

- Converter implemented in *k4MarlinWrapper* as a Gaudi Tool.
- Configured in the options file indicating which processor needs to use the Tool to convert the EDM4hep event to LCIO format
- Events are read through the DataHandle from k4FWCore; these are converted and registered in the Transient Event Store (TES) to make them available to the rest of the framework.

```
MyFastJetProcessor = MarlinProcessorWrapper("MyFastJetProcessor")
# ...
edmConvTool = EDM4hep2LcioTool("EDM4hep2lcio")
edmConvTool.EDM2LCIOConversion = [
    "Track", "EFlowTrack", "EFlowTrackConv",
    "ReconstructedParticle", "ReconstructedParticles", "TightSelectedPandoraPFOs"]
# ...
MyFastJetProcessor.EDMConversionTool=edmConvTool
# ...
algList.append(MyFastJetProcessor)
```

Development of EDM4hep to LCIO conversion I

- During conversion some collections depend on each other.
  - Makes the conversion more convoluted.
  - Missing links between collections fixed after first conversion.
  - i.e. A `ReconstructedParticle` contains a `ConstVertex`, which then links to a `ReconstructedParticle` again.



Development of EDM4hep to LCIO conversion II

- Some inconsistencies were found between EDM4hep and LCIO.
    - i.e. EDM4hep `vertex.getAlgorithmType()` returns a `int`, whereas LCIO vertex expects a `std::string`.
- Some conversions will loose information.
    - Some collections have a 64 bit integer CellID in EDM4hep, that is saved as 32 bit integer type in LCIO
    - EDM4hep mass in some cases is a 64 bit double type, that is saved a a 32 bit float type in LCIO.
- Every collection to convert presents its particularities, no generic approach.
- Metadata associated to collections is organized differently (Event vs ”group of collections”)

LCIO to EDM4hep conversion

- Converter from LCIO format to EDM4hep needed:
  - Interoperability between different algorithms.
  - Write output in EDM4hep format.
- Actual conversion integrated from k4LCIOReader[2], part of Key4hep.
  - k4LCIOReader meant to be used to read input files.
  - Adapted to read in-memory collections and convert them.
- Integrated with k4FWCore to seamlessly integrate the converted types: Podio output used to write the collections back.
- Implemented as a Gaudi Tool: can be attached to any Gaudi algorithm from k4MarlinWrapper.

[2] `https://github.com/key4hep/k4LCIOReader`

Use cases for k4MarlinWrapper

CLIC reconstruction

It successfully computes the full CLIC reconstruction

- CLIC reconstruction computes a sequence with different overlays, digitisers, reconstruction, PFO selectors, trackers, vertex finding algorithms, and others.
  - Complete sequence instantiated from k4MarlinWrapper delivering same results.
  - Constants used in converted version.
- Input is converted with the updated XML to Python converter.
- Configurable optional processors by the users.

```
from Gaudi.Configuration import *
CONSTANTS = {'BCReco': "3TeV",}
parseConstants(CONSTANTS)
# ...
read = LcioEvent()
InitDD4hep = MarlinProcessorWrapper("InitDD4hep")
Config = MarlinProcessorWrapper("Config")
VXDBarrelDigitiser = MarlinProcessorWrapper("VXDBarrelD
VXDEndcapDigitiser = MarlinProcessorWrapper("VXDEndcapD
# ...
algList.append(InitDD4hep)
algList.append(Config)
# algList.append(OverlayFalse)
# algList.append(Overlay350GeV_CDR)
algList.append(VXDBarrelDigitiser)
algList.append(VXDEndcapDigitiser)
# ...
```

LCFI+ algorithm

- Group of algorithms to perform vertex finding, jet finding and flavour tagging for linear colliders.
- Different Event Data Models needed to use the group of algorithms in different experiments (i.e. FCC-ee)
  - LCFI+ implemented in Marlin using LCIO.
  - Generated input in EDM4hep, e.g., by Delphes
  - Output expected in EDM4hep.
- Successfully implemented with the in-memory converters.

```
inp = PodioInput('InputReader')
inp.collections = ['ReconstructedParticles', 'EFlowTrack']
# ...
edmConvTool = EDM4hep2LcioTool("EDM4hep2lcio")
edmConvTool.EDM2LCIOConversion = [...]
# ...
InitDD4hep.ProcessorType = "InitializeDD4hep"
InitDD4hep.EDMConversionTool=edmConvTool
# ...
lcioConvTool = k4LCIOReaderWrapper("LCIO2EDM4hep")
lcioConvTool.LCIO2EMD4hepConversion = [...]
# ..
JetClusteringAndRefiner.ProcessorType = "LcfiplusProcessor"
JetClusteringAndRefiner.LCIOConversionTool=lcioConvTool
# ...
out = PodioOutput("PodioOutput", filename = "my_output.root")
out.outputCommands = ["keep *"]
```

CLUE in Key4hep: CLUE4hep

- CLUster by Energy algorithm
- It can work in a standalone manner using common dependencies.
- Work in progress to integrate with Key4hep by Erica Brondolin.
- clicReconstruction output from k4MarlinWrapper used for CLUE.
- Fixing issues with dependencies between collections.

SCTau integration

- Integration with k4MarlinWrapper would provide all Marlin algorithms to be used in the SCTau (Super-Charm-Tau) software framework (Aurora).
    - Aurora is Gaudi based, so k4MarlinWrapper shares most of the components.
    - Different versions of core packages impede an easy integration.
    - k4MarlinWrapper can be included as an external.
- Geometry and sensitive detector TPC needs to be adapted from iLC to SCT.
- Event Data Model differs from EDM4hep (or LCIO)
    - To be rebased from EDM4hep to be compatible.

Conclusions and future directions

Ongoing work and future directions

- *k4MarlinWrapper* has been improved and extended to support more use cases.
    - Added converters and interfaces, extended functionality, bugs fixed.
    - Converters put into test, with improvements and fixes being developed after feedback is received.
- It has been successfully used for complete CLIC reconstruction with LCIO input.
    - EDM4hep input still presents some issues with metadata not being converted. Ongoing fix.
- Integration with SCTau is ongoing: EDM and framework integration. ongoing.[3]
- EDM converters: EDM4hep and LCIO types supported.
    - Changes to deal with reference collections to be consistent between LCIO->EDM4hep and EDM4hep->LCIO conversions.
    - Simplification of converters: remove need to indicate type.

[3] This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 871072

Conclusions

- Key4hep effort to provide complete software stack for future detectors keeps progressing, including contributions from major future colliders: CLIC, FCC, CEPC and ILC.
- Different experiments and use cases are now being tested using Key4hep or components of Key4hep.
- k4MarlinWrapper base functionality is set, but details, hidden assumptions and different conventions are highlighted when it is put into real test.
    - These are to be discovered, fixed and released.
- The whole Key4hep project presents lots of synergies and opportunities for common software good practices, removal of duplicities, simplification of processes... all while being generic enough for different future experiments.