Evolution of iLCSoft
Towards the Common Software Stack Key4HEP

André Sailer

CERN-EP-LCD

FCal Collaboration Meeting
September 19–20, 2019

Table of Contents

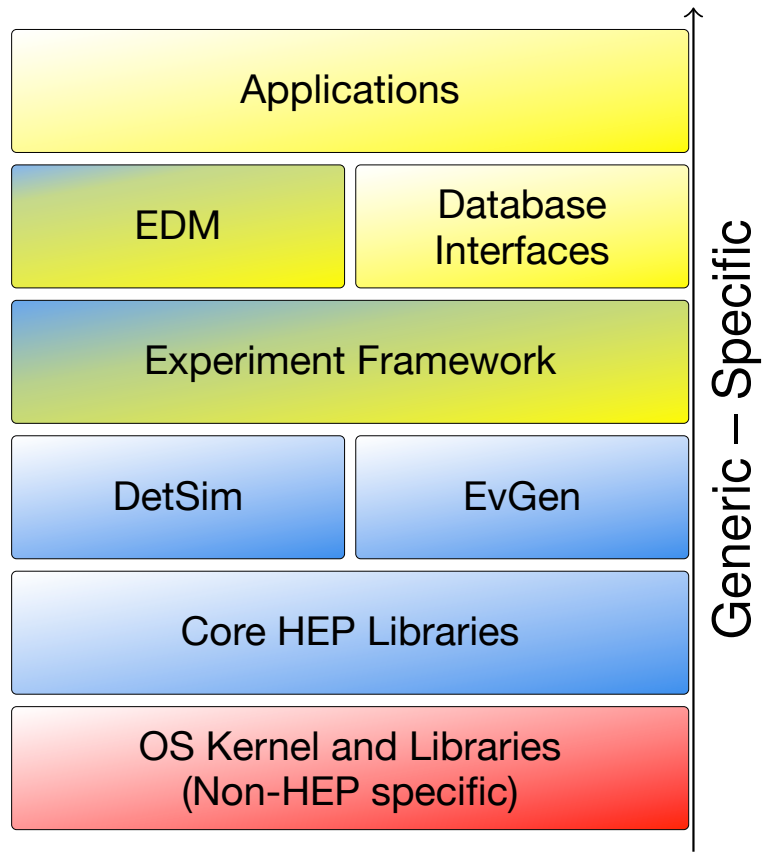
Section 1:

1 Key4HEP: The Turnkey Software Stack

A typical HEP Software Stack

Applications usually rely on large number of libraries, where some depend on others

- Interfaces to tracking and reconstruction libraries (PandoraPFA, FCALClusterer)
- (More or less) experiment specific event data model libraries
- Experiment core orchestration layer, which controls everything else: Marlin, Gaudi, CMSSW, AliRoot
- Packages used by many experiments: Geant4, Pythia, ...
- Usual core libraries (ROOT, Geant4, ...)
- Non-HEP libraries: boost, python, cmake ...



Motivation for Key4HEP

- Future detector studies critically rely on well-maintained software to model detector concepts and to understand a detector's potential and physics reach
- There is a scattered landscape of specific software tools on one hand, and integrated frameworks tailored to a specific experiment on the other hand
- Aim at a low-maintenance common stack for FCC, ILC/CLIC, CEPC, SCT with ready to use plug-ins to develop detector concepts
- Identified as an important project in the **CERN EP R&D initiative**, submitted EOI for AIDA++
- Reached consensus among all communities for future colliders to develop a common turnkey software stack at recent **Future Collider Software Workshop**

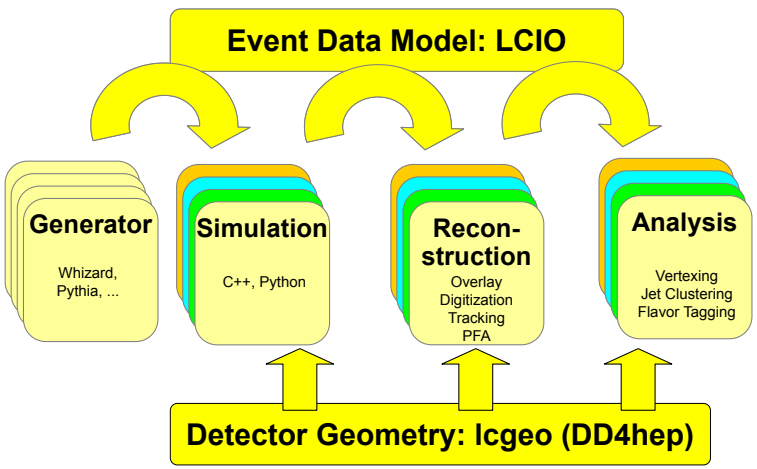
The Vision for the Turnkey Software Stack

The turnkey stack connects and extends the individual packages towards a complete data processing framework

- Convert a set of disconnected packages into a *turnkey* system
- Sharing as many common components reduces overhead for all users

- easy to use: for librarians, developers, users
  - easy to deploy
  - easy to extend
  - easy to set up
- full of functionality
- plenty of examples for simulation and reconstruction of detectors

Major ingredients: Event Data Model, Geometry Information, Framework



Data Processing Frameworks

- Data processing frameworks are the skeleton on which HEP applications are built
  - Some parts of the glue between execution of algorithms and frameworks is rather specific (usually the *services*; logging, histogram store, data store), so there is a huge advantage of using one framework
- `Marlin` very successfully used in LC community to run on different detector models or test beam data
  - Very nice features for automatic steering file configuration, ease of use
  - No support for concurrency at the moment
- `Gaudi` is used by LHCb, ATLAS
  - Supports concurrency

Key4HEP will be based on Gaudi. Additional developments for Gaudi when needed

Gaudi Modernisation Project

- Many *hard* improvements already finished
- Up-to-date documentation in the works
- Build system under rewrite

https://gaudi-framework.readthedocs.io/

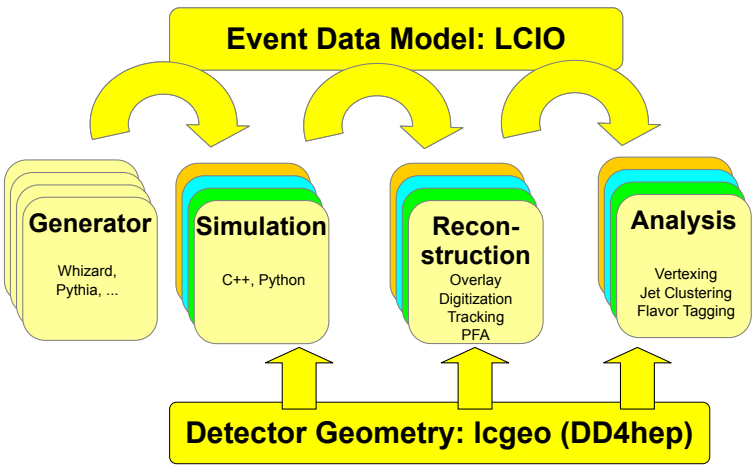
Section 2:

2 Adapting the CLIC Reconstruction to Gaudi

- Marlin & Gaudi
- Reading LCIO Events
- Re-Using Existing Processors
- E(DM)volution
- Implication for User Processors

Adiabatic Changes

- While transitioning to Key4HEP, need to be able to keep running the CLIC reconstruction
- Switch components one by one, validate changes
  - Geometry provided by DD4hep, no changes needed
  - Move framework from Marlin to Gaudi: wrap existing processors
  - Move from `LCIO` to EDM4HEP
  - Replace wrapped processors with native Gaudi algorithms



Marlin & Gaudi

Apart from some naming conventions, very similar ideas in the two frameworks* – glossing over technical details

	Marlin	Gaudi
language	`c++`	`c++`
working unit	Processor	Algorithm
configuration language	XML	Python
set up function	`init`	`initialize`
working function	`processEvent`	`execute`
wrap up function	`end`	`finalize`
Transient data format	`LCIO`	anything

- To start using Gaudi: use a generic wrapper around the processors
- Read LCIO files and pass the `LCIO::Event` to our processors

*Of course subtle differences emerge

LCIO Event “Algorithm”

- Reads LCIO files and places the `LCIO::Event` in the `gaudi::DataStore`
- `LCIO::Event` can be used for input and output as is the case in `Marlin`
- No need to change the way processors obtain `LCIO::Collections`

Marlin Processor Wrapper

Need one `gaudi::algorithm` to run any `Marlin::Processor`

- `initialize`
  - get processor instance of requested `ProcessorType` from `marlin::ProcessorMgr`
  - prepare `marlin::StringParameters` object
  - Call `Processor::setName` and `Processor::setParameters`
- `execute`:
  - get `LCIO::Event` from `gaudi::DataStore`, call `processor->processEvent`
- `finalize`:
  - call `processor->end`

Prototype: **https://github.com/andresailer/GMP**

Wrapper Configuration

- Translate the `XML` to `python`, using a stand alone python script
- Pass arbitrary number, types, and names of parameters to the processor

```
<processor name="VXDBarrelDigitiser" type="DDPlanarDigiProcessor">
  <parameter name="SubDetectorName" type="string">Vertex </parameter>
  <parameter name="IsStrip" type="bool">false </parameter>
  <parameter name="ResolutionU" type="float"> 0.003 0.003 0.003 0.003 0.003 0.003
  <parameter name="ResolutionV" type="float"> 0.003 0.003 0.003 0.003 0.003 0.003
  <parameter name="SimTrackHitCollectionName" type="string" lcioInType="SimTracke
  <parameter name="SimTrkHitRelCollection" type="string" lcioOutType="LCRelation">
  <parameter name="TrackerHitCollectionName" type="string" lcioOutType="TrackerHi
  <parameter name="Verbosity" type="string">WARNING </parameter>
</processor>
```

Wrapper Configuration

- Translate the `XML` to `python`, using a stand alone python script
- Pass arbitrary number, types, and names of parameters to the processor

```
VXDBarrelDigitiser = MarlinProcessorWrapper("VXDBarrelDigitiser")
VXDBarrelDigitiser.OutputLevel = WARNING
VXDBarrelDigitiser.ProcessorType = "DDPlanarDigiProcessor"
VXDBarrelDigitiser.Parameters = [
                                "IsStrip", "false", END_TAG,
                                "ResolutionU", "0.003", "0.003", "0.003", "0.003",
                                "ResolutionV", "0.003", "0.003", "0.003", "0.003",
                                "SimTrackHitCollectionName", "VertexBarrelCollectio
                                "SimTrkHitRelCollection", "VXDTrackerHitRelations"
                                "SubDetectorName", "Vertex", END_TAG,
                                "TrackerHitCollectionName", "VXDTrackerHits", END_T
                                ]
```

Wrapper Configuration

- Translate the `XML` to `python`, using a stand alone python script
- Pass arbitrary number, types, and names of parameters to the processor

```
<processor name="Refit" type="RefitFinal">
  <parameter name="EnergyLossOn" type="bool"> true </parameter>
  <parameter name="InputRelationCollectionName" type="string" lcioInType="LCRelation
  <parameter name="InputTrackCollectionName" type="string" lcioInType="Track"> SiTra
  <parameter name="Max_Chi2_Incr" type="double"> 1.79769e+30 </parameter>
  <parameter name="MultipleScatteringOn" type="bool"> true </parameter>
  <parameter name="OutputRelationCollectionName" type="string" lcioOutType="LCRelat
    SiTracks_Refitted_Relation
  </parameter>
  <parameter name="OutputTrackCollectionName" type="string" lcioOutType="Track">
    SiTracks_Refitted
  </parameter>
  <parameter name="ReferencePoint" type="int"> -1 </parameter>
  <parameter name="SmoothOn" type="bool"> false </parameter>
  <parameter name="extrapolateForward" type="bool"> true </parameter>
  <parameter name="MinClustersOnTrackAfterFit" type="int">3 </parameter>
</processor>
```

Wrapper Configuration

- Translate the `XML` to `python`, using a stand alone python script
- Pass arbitrary number, types, and names of parameters to the processor

```
Refit = MarlinProcessorWrapper("Refit")
Refit.OutputLevel = WARNING
Refit.ProcessorType = "RefitFinal"
Refit.Parameters = [
                    "EnergyLossOn", "true", END_TAG,
                    "InputRelationCollectionName", "SiTrackRelations", END_TAG,
                    "InputTrackCollectionName", "SiTracks", END_TAG,
                    "Max_Chi2_Incr", "1.79769e+30", END_TAG,
                    "MinClustersOnTrackAfterFit", "3", END_TAG,
                    "MultipleScatteringOn", "true", END_TAG,
                    "OutputRelationCollectionName", "SiTracks_Refitted_Relation", EN
                    "OutputTrackCollectionName", "SiTracks_Refitted", END_TAG,
                    "ReferencePoint", "-1", END_TAG,
                    "SmoothOn", "false", END_TAG,
                    "extrapolateForward", "true", END_TAG
                    ]
```

Configuration: Control flow

- XML `execute` section translated to a python list
- Can use python for string substitution, static `if/else`, loops, ...

```xml
<execute>
  <processor name="MyAIDAProcessor"/>
  <processor name="EventNumber" />
  <processor name="InitDD4hep"/>
  <processor name="Config" />
  <!-- ... -->
</execute>
```

```python
algList = []
algList.append(lcioReader)
algList.append(MyAIDAProcessor)
algList.append(EventNumber)
algList.append(InitDD4hep)
algList.append(OverlayFalse)
algList.append(VXDBarrelDigitiser)
#...
```

Need better understanding of Gaudi for dynamic control flow

Changes in iLCSoft

Very minimal changes needed in `Marlin`

Changes applied:

- Make `marlin::Processor::setParameters` and `marlin::Processor::setName` `public`
- Actually make the `Marlin` `EventSelector` part of the namespace to avoid clash with `EventSelector` from `Gaudi`

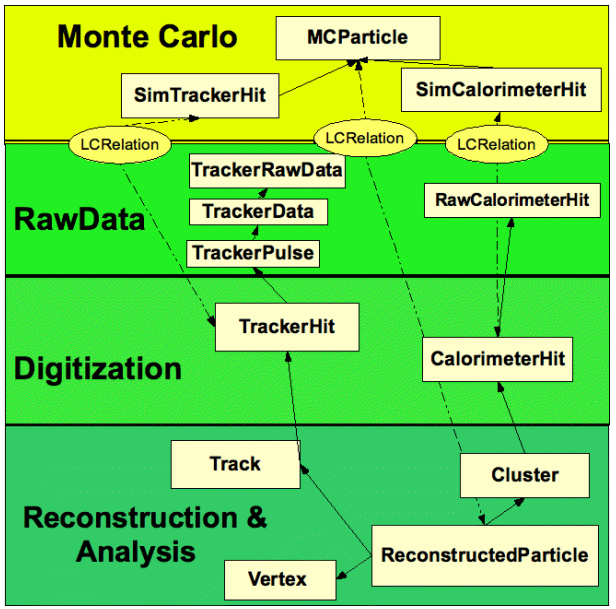
Changes to be done (recently discovered):

- Make it possible to call the `ProcessorEventSeeder` from the wrapper; move to functions from `private` to `public`

EventDataModel: EDM4HEP, podio, plcio

Use a common event data model to allow high degree of integration

- Using `podio` to manage the EDM and easily change the persistency layer
- Create *adapters* to migrate the existing `LCIO` data to the new EDM



- Develop an *alpha* version of EDM4HEP based on `plcio` prototype, to see how best to move forward
- EDM4HEP working group: https://indico.cern.ch/category/11461/

Implication for User Processors

- As long as we write out LCIO files: None
  - Users can still run Marlin and their own processors on the output files
- Can also run processors inside Gaudi using the wrapper and steering file conversion tool
- When we start transitioning to EDM4HEP
  - Bi-directional in memory adapters between EDM4HEP and LCIO?
  - Some work to adapt processors to `gaudi::DataStore` and EDM4HEP

Section 3:

3 Current Status

Current Status

Current Status

- Running full reconstruction chain for CLIC

To do:

- Deal with errors and exceptions from marlin processors
- Treat `Marlin` `processRunHeader`, `check` functions
- Make developments available beyond Marko's and my PC, with the help of the HSF packaging working group

Section 4:

4 Conclusion

Key4HEP Benefits for FCAL

- Benefit from framework developments of the LHC experiments by adopting the Gaudi Framework
  - Multi-threading schedulers
  - Access to heterogeneous resources
- Focus on modernisation and improvements of our algorithms
- Find new users for the wealth of functionality present in iLCSoft
- Make use of algorithms interfaced with Gaudi
- Use the same framework for studies in CLIC, FCC, CEPC, (eventually ILC)