EPJ Web of Conferences **245**, 10002 (2020) https://doi.org/10.1051/epjconf/202024510002
CHEP 2019

Towards a Turnkey Software Stack for HEP Experiments

André Sailer[1,*], *Gerardo* Ganis[1], *Pere* Mato[1], *Marko* Petrič[1], and *Graeme A* Stewart[1]

[1]CERN, Esplanade des Particules 1, 1211 Geneva 23, Switzerland

Abstract. Future HEP experiments require detailed simulation and advanced reconstruction algorithms to explore the physics reach of their proposed machines and to design, optimise, and study the detector geometry and performance. To synergize the development of the CLIC and FCC software efforts, the CERN EP R&D roadmap proposes the creation of a "Turnkey Software Stack", which is foreseen to provide all the necessary ingredients, from simulation to analysis, for future experiments; not only CLIC and FCC, but also for proposed Super-tau-charm factories, CEPC, and ILC. The software stack will facilitate writing specific software for experiments ensuring coherency and maximising the re-use of established packages to benefit from existing solutions and community developments, for example, ROOT, Geant4, DD4hep, Gaudi and podio. As a showcase for the software stack, the existing CLIC reconstruction software, written for iLCSoft, is being to be ported to Gaudi. In parallel, the back-end of the LCIO event data model can be replaced by an implementation in podio. These changes will enable the sharing of the algorithms with other users of the software stack.
We will present the current status and plans of the turnkey software stack, with a focus of the adaptation of the CLIC reconstruction chain to Gaudi and podio, and detail the plans for future developments to generalise their applicability to FCC and beyond.

1 Introduction

Experiments at future colliders require advanced software to simulate detector geometries and to reconstruct physics events in order to estimate and optimise the performance of the experiment and maximise the physics reach. The interplay between reconstruction algorithms and detector geometry, for example in particle flow clustering, means that the detector hardware cannot be developed and designed independently of the software. At the same time, developing and validating sophisticated algorithms, including accounting for a large number of edge cases, requires a significant amount of resources. Thus the different communities for future experiments – CLIC [1], FCC [2], ILC [3], SCT [4], STC [5], Muon Collider [6] – came to the agreement that the development of a common software solution would benefit everyone [7].

In these proceedings we describe the proposal of the turnkey software stack, outline the requirements and foreseen ingredients, and showcase the evolution of the reconstruction for CLIC towards this common software solution.

*e-mail: andre.philippe.sailer@cern.ch

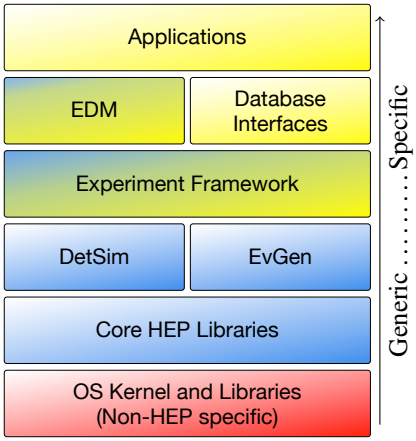

Figure 1. A stack of software libraries from generic to specific

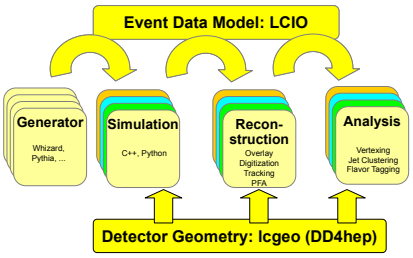

Figure 2. Processing chain from MC generators to simulation, reconstruction and analysis

1.1 Vision for the Turnkey Software Stack

The turnkey software stack, now dubbed Key4hep, should encompass all the libraries needed for simulation, reconstruction, and analysis. Figure 1 shows the different layers of a typical HEP software stack. The base is formed by standard libraries and the operating system, for example Boost, Python, CMake, and compilers. These products are typically developed outside of HEP. Building on top of these libraries are the HEP libraries that provide generic functionality – ROOT [8, 9], Geant4 [10–12], CLHEP [13]. Combining and extending these libraries are tools that address more specific needs but are still used by multiple experiments, for example detector geometry solutions like DD4hep [14–16], pattern recognition for particle flow clustering or neutrino experiments like PandoraPFA [17], or Monte Carlo event generators, like Pythia [18]. The *frameworks* – Marlin [19], Gaudi [20], CMSSW [21], AliRoot [22] – provide the orchestration layer which controls everything else. These frameworks usually require an event data model (EDM) for transient and persistent data, interfaces to databases, and many algorithms and tools that implement the simulation and reconstruction logic, or wrappers to other generic packages that provide the desired functionality, e.g., PandoraPFA, ACTS [23], or FastJet [24].

The *vision* of the turnkey software stack is to connect and extend these individual packages towards a complete data processing framework. Key4hep takes the packages and adds the *glue* to combine them into a turnkey system. The overhead for all users is reduced, through the sharing of many components. In addition, the turnkey software stack should be easy to set up and run for users, easy to extend for developers, and easy to deploy for librarians. To help users get started quickly, the stack has to be fully functional and come with plenty of examples that will allow new users to adapt it for their specific use case.

The major ingredients to reach this goal of a complete framework are an event data model, a source of geometry information, and a framework for configuration, control and other services. This is schematically shown in Figure 2.

To combine a wide range of libraries, considerations with respect to the interoperability have to be made. There is no one-size-fits-all rule, so that the proper level of interoperability has to be decided on a case by case basis. However, finding the right interoperability offers great *quality of life* for developers and users.

The lowest level of interoperability, that gives most freedom to developers to chose are common data formats that are passed between different programs, running on any hardware, as is done with HepMC [25], LCIO [26, 27], GDML [28], or via message passing. Callable interfaces demand a tighter coupling between libraries, but cross language calls are still possible. Care has to be taken that compatible compiler- or language-versions are used for the different pieces. Compiling libraries with different C++ standards can lead to very surprising and hard to understand errors. Beyond the interface definitions the details of error handling (exceptions or return codes), thread safety, library dependencies and run-time setup need to be defined. Languages with introspection capabilities, like Python, can facilitate this interoperability, which is used in the PyROOT environment to interact with the C++ objects in ROOT via the Python interpreter. Finally the component model – where each piece is part of a larger framework that defines how components are configured, how logging is done, who owns and decides on object lifetime and how to plug-in additional modules – offers maximum re-use of the components. This level of interoperability requires the adoption of a single framework.

1.2 Ingredients

Following the considerations laid out in the previous section, the ingredients can be chosen.

The first ingredient is the data processing framework, which is the skeleton on which everything else is built upon. The linear collider community has used Marlin for many years very successfully, but it suffers from a lack of development resources. Therefore, for the data processing framework the choice has fallen on Gaudi, because it has a large user and developer community, and offers support for access to heterogeneous resources, different architectures, and task-oriented concurrency. Features that are found to be missing in Gaudi will be contributed, for example the features that make Marlin easy to use like the automatic generation of steering file templates.

For the geometry information all concerned communities already use DD4hep, which offers a complete detector description for simulation, reconstruction and analysis.

A common event data model is needed for interoperability. This event data model will be managed by podio [29]. Based on an event data model described in a YAML file, podio creates the source code automatically. The automatic creation will allow one to easily change the persistency layer, as soon as they are implement in podio. The initial event data model will be based on the LCIO and FCC-EDM classes, and evolve from there as needed.

For the ease of use for librarians and developers, one needs to be able to build any and all pieces of the stack with minimum effort. This means going beyond sharing of the build results to sharing the build recipes. The investigations of the HEP Software Foundation packaging working group is pointing towards Spack [30] as the solution.

2 Evolution of the CLIC Reconstruction

The CLIC simulation and reconstruction workflow [31] is fully implemented in the iLCSoft environment, using Marlin, and DD4hep. Moving to Key4hep will bring benefit to the CLIC community through a more mature processing framework, which allows access to heterogeneous resources or to exploit concurrency. It will also allow a larger user base to use the tools developed in the linear collider community.

The ability to run the CLIC reconstruction during its transition to Key4hep is both necessary to continue studying detector performance at CLIC, as well as offering a unique validation opportunity for the new software stack itself. Thus one has to switch the components one after the other and validate the small steps only, instead of a complete re-validation after

Table 1. Comparison between Marlin and Gaudi

	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

years of work moving to a new framework. One of the ingredients for Key4hep, the DD4hep geometry description, is already used, so no changes are needed on this side.

The event data model used in Marlin, LCIO, is replaced by EDM4hep. As the event data model in EDM4hep is based on the LCIO event data model, it offers very similar objects, so that the replacement in the source code should be rather minimal with respect to the logic, and mostly be concerned with how objects are read or written to the data store.

The framework is replaced by Gaudi, but as will be shown below, the existing *processors* in Marlin can be run inside a wrapper as Gaudi *algorithms*. Table 1 shows a very basic view of the differences and similarities between the *workhorses* of the two frameworks. The largest difference between the two are the languages used for configuration. Besides this, the concepts are similar, and the differences are in the names of the set up, working and wrap up functions.

To show that moving from the Marlin to the Gaudi framework was feasible in practice a `MarlinProcessorWrapper` was developed. The `MarlinProcessorWrapper` is a Gaudi algorithm that can run any Marlin processor. It only takes three parameters: the logging `OutputLevel` for Gaudi logging, the `ProcessorType` which tells the `marlin::ProcessorMgr` which processor to load, and the `Parameters`, which is a list of `strings` that are parsed into a `marlin::StringParameters` object and given to the processor, which turns them into the expected types. Using the generic `Parameters` parameter allows one to call any processor regardless of the parameters it expects. A second algorithm was implemented, which reads an LCIO file and attaches the `lcio::LCEvent` to the Gaudi `EventService`, so it can be used in the `MarlinProcessorWrapper` and passed to the existing processors for input and output.

For the configuration, a stand alone script converts the Marlin XML steering files into Python files used by Gaudi. Listings 1 and 2 show the difference between the processor configuration inside Marlin and Gaudi. Converting the XML file to Python independently of running Gaudi, allows one to replace the wrapped processors by native algorithms one-by-one, which wouldn't be easily possible if the XML file where converted at run-time. The list of processors to execute is simply translated from an XML section to a Python list, as is shown in Listings 3 and 4.

To allow the `MarlinProcessorWrapper` to function only a small number of changes were needed in iLCSoft. The `private` functions, `setParameters` and `setName`, of the `marlin::Processor` class had to be made `public`, so that they could be called from the wrapper. Similarly the `marlin::ProcessorEventSeeder` requires some functions to be made `public`. The most vexing problem was caused by both Marlin and Gaudi containing an `EventSelector` class, which showed up as a run-time crash when the wrong destructor

```
<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>
  <parameter name="ResolutionV" type="float"> 0.003 0.003 0.003 0.003 0.003 0.003  </parameter>
  <parameter name="SimTrackHitCollectionName" type="string" lcioInType="SimTrackerHit">
    VertexBarrelCollection </parameter>
  <parameter name="SimTrkHitRelCollection" type="string" lcioOutType="LCRelation">
    VXDTrackerHitRelations </parameter>
  <parameter name="TrackerHitCollectionName" type="string" lcioOutType="TrackerHitPlane">
    VXDTrackerHits </parameter>
  <parameter name="Verbosity" type="string">WARNING </parameter>
</processor>
```

Listing 1. Extract from a Marlin steering file showing the configuration of a single 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", "0.003", "0.003", END_TAG,
  "ResolutionV", "0.003", "0.003", "0.003", "0.003", "0.003", "0.003", END_TAG,
  "SimTrackHitCollectionName", "VertexBarrelCollection", END_TAG,
  "SimTrkHitRelCollection", "VXDTrackerHitRelations", END_TAG,
  "SubDetectorName", "Vertex", END_TAG,
  "TrackerHitCollectionName", "VXDTrackerHits", END_TAG
]
```

Listing 2. The same configuration as listing 1 but in the Python expected by Gaudi. `END_TAG` is a string constant that signifies the processor wrapper that the next entry is a name of a parameter, this allows splitting of the list of strings into a map of parameter name and list of strings.

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

Listing 3. Extract from a Marlin steering file showing the list of processors to execute

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

Listing 4. The same configuration as listing 3 but in the Python expected by Gaudi

was called. This was solved by moving the Marlin `EventSelector` out of the global namespace.

With these changes and no modifications in any of the Marlin processors themselves, it is possible to run the complete CLIC reconstruction via Gaudi. Of course adapting to the EDM4hep and running the algorithms without the wrapper will require larger modifications. The co-execution of wrapped and native algorithms will also require conversions from LCIO to EDM4hep classes and back.

3 Summary

The turnkey software stack, Key4hep, aims to create a complete data processing framework for the benefit of future collider experiments. The stack will be built on established solu-

tions like ROOT, Geant4, DD4hep and Gaudi. Where it is found necessary or beneficial, new solutions are adopted, for example a new event data model EDM4hep based on podio, or the Spack packaging tool. This approach does not require one to completely abandon existing solutions. The development and application of the prototype processor wrapper for the CLIC reconstruction shows that the most valuable parts, the reconstruction algorithms, can be ported to the new framework with minimal effort. The existing processors can evolve into the Gaudi framework in parallel to continuous validation. Further developments of the event data model and adaptations to the new framework are currently in progress. The first milestone is evolving the processor wrapper beyond its prototype state and validate the results against the existing CLIC software. Then the individual processors will be adapted to Gaudi and made available for other users of the Key4hep stack. The software for the FCC experiments will also be adapted to Key4hep, but here only the event data model has to be adapted to EDM4hep [32]. The developments for the Key4hep software stack are not closed, and contributions and use by other experiments are welcome.

References

[1] CLICdp Collaboration, CLIC Collaboration, *The Compact Linear Collider (CLIC) - 2018 Summary Report*, https://doi.org/10.23731/CYRM-2018-002
[2] M. Mangano et al., *FCC Physics Opportunities: Future Circular Collider Conceptual Design Report Volume 1. Future Circular Collider* (2018), https://doi.org/10.1140/epjc/s10052-019-6904-3
[3] P. Bambade et al., *The international linear collider: A global project* (2019), 1903.01629
[4] *Super charm–tau factory*, https://ctd.inp.nsk.su/c-tau/
[5] Q. Luo, W. Gao, J. Lan, W. Li, D. Xu, *Progress of Conceptual Study for the Accelerators of a 2-7GeV Super Tau Charm Facility at China* (2019), https://doi.org/10.18429/JACoW-IPAC2019-MOPRB031
[6] J.P. Delahaye et al., *Muon colliders* (2019), 1901.06150, https://arxiv.org/abs/1903.01629
[7] *Future collider software workshop* (2019), https://agenda.infn.it/event/19047
[8] R. Brun, F. Rademakers, Nucl. Instrum. Meth. **A389**, 81 (1997)
[9] F. Rademakers et al., *root* (2018), https://doi.org/10.5281/zenodo.848818
[10] J. Allison et al., IEEE T. Nucl. Sci. **53**, 270 (2006)
[11] S. Agostinelli et al., Nucl. Instrum. Meth. **A506**, 250 (2003)
[12] J. Allison et al., Nucl. Instrum. Meth. **A835**, 186 (2016)
[13] https://gitlab.cern.ch/CLHEP/CLHEP
[14] M. Frank, F. Gaede, M. Petric, A. Sailer, *DD4hep* (2018), https://doi.org/10.5281/zenodo.592244
[15] M. Frank, F. Gaede, C. Grefe, P. Mato, J. Phys. Conf. Ser. **513**, 022010 (2013)
[16] A. Sailer, M. Frank, F. Gaede, D. Hynds, S. Lu, N. Nikiforou, M. Petric, R. Simoniello, G.G. Voutsinas, J. Phys. Conf. Ser. **898**, 042017 (2017)
[17] J. Marshall, M. Thomson, Eur. Phys. J. **C75**, 439 (2015)
[18] T. Sjostrand, S. Mrenna, P.Z. Skands, JHEP **05**, 026 (2006)
[19] F. Gaede, Nucl. Instrum. Meth. **A559**, 177 (2006)
[20] LHCb Collaboration, ATLAS Collaboration, *Gaudi v33r0* (2019), https://doi.org/10.5281/zenodo.3660964
[21] CMS Collaboration, *cmssw* (2017), https://doi.org/10.5281/zenodo.292299

[22] R. Brun, P. Buncic, F. Carminati, A. Morsch, F. Rademakers, K. Safarik, Nucl. Instrum. Meth. **A502**, 339 (2003)

[23] A. Salzburger, B. Schlag, C. Gumpert, F. Klimpel, H. Grasland, J. Hrdinka, M. Kiehn, N. Calace, P. Gessinger, R. Langenberg et al., *Acts project: v0.15.00* (2020), https://doi.org/10.5281/zenodo.3626878

[24] M. Cacciari, G.P. Salam, G. Soyez, Eur. Phys. J. **C72**, 1896 (2012)

[25] M. Dobbs, J.B. Hansen, Comput. Phys. Commun. **134**, 41 (2001)

[26] F. Gaede, T. Behnke, N. Graf, T. Johnson, *LCIO — A persistency framework for linear collider simulation studies*, in *CHEP 2003* (La Jolla, California, 2003)

[27] F. Gaede, T. Behnke, R. Cassell, N. Graf, T. Johnson, H. Vogt, *LCIO persistency and data model for LC simulation and reconstruction*, in *CHEP 2004* (Interlaken, Switzerland, 2004)

[28] R. Chytracek, J. Mccormick, W. Pokorski, G. Santin, IEEE T. Nucl. Sci. **53**, 2892 (2006)

[29] F. Gaede, B. Hegner, P. Mato, J. Phys. Conf. Ser. **898**, 072039 (2017)

[30] B. Morgan, G.A. Stewart, J.C. Villanueva, H.A. Willett, *Modern Software Stack Building for HEP* (2019), https://doi.org/10.5281/zenodo.3598985

[31] D. Arominski, J.J. Blaising, E. Brondolin, D. Dannheim, K. Elsener, F. Gaede, I. García-García, S. Green, D. Hynds, E. Leogrande et al., *A detector for CLIC: main parameters and performance* (2018), 1812.07337, https://cds.cern.ch/record/2649437

[32] C. Helsens, V. Volkl, C. Neubuser, G. Ganis, J.C. Villanueva, *A software framework for FCC studies: status and plans* (2019), https://doi.org/10.5281/zenodo.3599139