Turnkey Software Stack: Key4hep

André Sailer
CERN

Table of Contents

1 Vision for the Turnkey Software Stack
2 Framework
3 Geometry
4 Event Data Model
5 Packaging
6 Plans
7 Summary
A typical HEP Software Stack

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

- Interfaces to tracking and reconstruction libraries (PandoraPFA, ACTS)
- (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: DD4hep, Pythia, ...
- Usual core libraries (ROOT, Geant4, CLHEP, ...)
- Non-HEP libraries: boost, python, cmake ...
Towards a Turnkey Software Stack

- Members of the FCC, ILC, CEPC, SCT, CLIC, LHC communities met for a Future-Collider-Software Workshop in Bologna on June 12&13 https://agenda.infn.it/event/19047/
  - Reached an Agreement to create common event data model and common turnkey software stack

- The Vision: The turnkey stack connects and extends the individual packages towards a complete data processing framework
  - Major ingredients: Event Data Model, Geometry Information, Processing Framework
  - Sharing common components reduces overhead for all users
  - Should be easy to use for librarians, developers, users
    * easy to deploy, extend, set up
  - Full of functionality: plenty of examples for simulation and reconstruction of detectors

Turnkey Software Stack part of

- EP R&D Programme; chapter 10.7
- AIDA++ submission: software work package; partners: CERN, DESY, INFN
Interoperability of Packages

The right interoperability point between packages varies, but choosing it correctly provides great quality of life for developers and users

- Level 0 – Common Data Formats
  - E.g.: HepMC event records, LCIO, GDML, ALFA Messages

- Level 1 – Callable Interfaces
  - Details are important: error/exception handling, thread safety, library dependencies, runtime setup

- Level 2 – Introspection Capabilities
  - E.g.: PyROOT to interact with any ROOT (C++) class via the python interpreter

- Level 3 – Component Model
  - Software components of a common framework offer maximum re-use
  - standard way to configure components, logging, object lifetime and ownership, plug-in mechanism
  - Requires adoption of single framework

More on this in the presentation by G. Stewart at Bologna WS on Common Software Tools
Data Processing Framework

- 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
  - Concurrency support under development (See MarlinMT - parallelising the Marlin framework)

- Gaudi is used by LHCb, ATLAS, SCT, FCC (A software framework for FCC studies: status and plans)
  - Supports concurrency, access to heterogeneous resources, larger developer community

Based on considerations such as completeness, portability to various computing resources, architectures and accelerators, support for task-oriented concurrency and current adoption, it was agreed the best candidate to date is the Gaudi framework.
Geometry Information: DD4HEP

- **Complete Detector Description**
  - Providing geometry, materials, visualization, readout, alignment, calibration...

- **Single source of information → consistent description**
  - Use in simulation, reconstruction, analysis

- **Supports full experiment life cycle**
  - Detector concept development, detector optimization, construction, operation
  - Facile transition from one stage to the next

- **DD4HEP already in use by CLIC, FCC and many more, easy choice for geometry provider**

- **Latest developments:** [DD4HEP: a community driven detector description tool for HEP, CHEP’19](#)
PODIO

Adapted from PODIO: recent developments in the Plain Old Data EDM toolkit

- PODIO is an Event Data Model (EDM) toolkit for HEP
  - developed in Horizon2020 project AIDA2020
  - based on the use of PODs for the event data objects (Plain-Old-Data object)
- PODIO originally developed in context of the FCC study
  - addressing the problem of creating an EDM in a generic way
  - EDM described in yaml, c++ code auto-generated
  - allowing potential re-use by other HEP groups
- PODIO is used since its first release by the FCC studies (see FCC-edm)
- Adaptation to KEY4HEP ongoing
PODIO Core Features

- Three layers:
  - user layer (API): collections of EDM object handles, *HitCollection*
  - object layer: transient objects (*HitObject*)
  - POD layer: persistent information
- Clear ownership: objects owned by *EventStore* are persisted, other objects ref-counted
- Python as first class citizen
- Different I/O implementations, but currently only ROOT
The **KEY4HEP** EDM: EDM4hep

For a high degree of interoperability, **EDM4HEP** will provide a common event data model

- Using **PODIO** to manage the EDM (described by yaml) and easily change the persistency layer (ROOT, HDFS, ...)

- Develop an **alpha** version of **EDM4HEP** based on **LCIO** and **FCC-edm** in the next few weeks
  - See how best to move forward
  - Input from other interested parties highly welcome

- **EDM4HEP** working group:
  - https://indico.cern.ch/category/11461/
  - http://github.com/HSF/edm4hep
EDM4hep EDM Objects
Building, packaging and deploying software is a shared problem across HEP

- HSF Packaging Working Group is an active open forum for discussion and cooperation
- For ease of use for librarians and developers, need to be able to build any and all pieces of the stack with minimum effort
- Go beyond sharing of *build results* to sharing of *build recipes*
Spack

- Package manager and build orchestrator developed at Lawrence Livermore National Laboratory
- Originally developed for installing software to HPC systems
  - Strong emphasis on scientific software
- Supports multiple versions of software concurrently
  - Appends build hashes to install locations, RPATH used to resolve the correct dependencies
  - Common dependencies are shared
  - Support for different compiler toolchains as a core concept
- Dependencies are found and installed automatically
  - Full specification of all build options for dependencies supported
  - Will rebuild or install from existing binary build products
- Configuration on command line or from YAML files
  - Package descriptions written in Python
- Large community of contributors, supporting 3.5k packages
  - Active HEP sub-community (and Slack channel)
KEY4HEP Prototype Build

- Build a software stack that can be used for KEY4HEP workflows
  - Event generation
  - Simulation, with detector description
  - Reconstruction, with experiment software framework
  - Analysis

- Spack selected as package orchestrator to continue work
  - Version 0.1
  - Spack first builds its own compiler (currently gcc9.2.0), for full self-consistency
  - Key top level packages:
    - Pythia, Geant4, DD4hep, Gaudi, ROOT
    - Use Spack's packages.yaml to set reproducible build options
  - All building successfully
  - Binary packages uploaded to build cache
  - Installation from build cache to new path, checks RPATH for validity
  - Runtime environment set up by environment modules, including G4INSTALL, ROOTSYS variables
Gaining Experience with Spack

- Spack has been successfully tested as a build orchestrator for modern HEP software stacks
  - FCC
  - SuperNEMO
  - KEY4HEP
  - Neutrino experiments

- Production workflows now in development
  - Learning from FCC experience helps, switching to self-consistent Spack build actually makes this simpler
Plans and Goals

- Work on EDM4hep and podio, adapt as needed
- Investigate use of spack for developer and production use
- Adapt iLCSoft algorithms to Key4hep (Gaudi, EDM4hep)
  - prioritising those for CLIC reconstruction or beneficial for SCT
- Adapt FCCSW to EDM4hep
- Run CLIC reconstruction inside the Gaudi framework, starting with wrapper, then going to native algorithms
- Use adapted algorithms also in FCC reconstruction (e.g., PandoraPFA)
Studies for future experiments can benefit from having a complete software stack available to them.

Starting from CLIC and FCC studies, develop a turnkey software stack: KEY4HEP.

Using established solutions: ROOT, Geant4, DD4HEP, Gaudi, ...

Investigate, develop, or adopt where beneficial or necessary: EDM4HEP, Spack.

Interested parties are welcome to participate.
Backup Slides
Evolution of the CLIC Reconstruction
CLIC Reco Evolution: Adiabatic Changes

- Full CLIC reconstruction implemented in iLCSoft
- 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
- Incidentally will make iLCSoft functionality available to other users of the stack
Marlin & Gaudi

Apart from some naming conventions, very similar ideas in the two frameworks*

<table>
<thead>
<tr>
<th></th>
<th>Marlin</th>
<th>Gaudi</th>
</tr>
</thead>
<tbody>
<tr>
<td>language</td>
<td>c++</td>
<td>c++</td>
</tr>
<tr>
<td>working unit</td>
<td>Processor</td>
<td>Algorithm</td>
</tr>
<tr>
<td>configuration language</td>
<td>XML</td>
<td>Python</td>
</tr>
<tr>
<td>set up function</td>
<td>init</td>
<td>initialize</td>
</tr>
<tr>
<td>working function</td>
<td>processEvent</td>
<td>execute</td>
</tr>
<tr>
<td>wrap up function</td>
<td>end</td>
<td>finalize</td>
</tr>
<tr>
<td>Transient data format</td>
<td>LCIO</td>
<td>anything</td>
</tr>
</tbody>
</table>

- To start using Gaudi: use a generic wrapper around the processors.
- Prototype: [https://github.com/andresailer/GMP](https://github.com/andresailer/GMP)
- Read LCIO files and pass the LCIO::Event to our processors

*Of course subtle differences emerge
Wrapper Configuration

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

Marlin/XML

```xml
<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>
```
Wrapper Configuration

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

Gaudi/Python

```python
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
]```
Configuration: Control flow

- XML execute section translated to a python list

```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)
#...
```
Changes in iLCSoft

**Surprisingly** little changes needed in Marlin

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