The LArSoft architecture

The LArSoft project

December 10, 2015

Contents

1 Introduction

1.1 Purpose

The LArSoft toolkit is designed to enable simulation, reconstruction and physics analysis of data from any detection system based on Liquid Argon TPCs. Its common tools and algorithms render the development and analysis process more uniform across the Experiments. LArSoft is extensible to accommodate evolving Experiments' needs and adoption by new Experiments.

This document describes the current architecture of LArSoft toolkit. The architecture was developed according to, and therefore reflects, the consensus of LArSoft partners, including the adopting Experiments.

The document provides a reference for the reader interested in learning the general structure of LArSoft, its functional areas and interactions with the execution environment. It also offers guidelines for the contributor aiming to develop new algorithms within LArSoft or to use it together with external tools.

1.2 Scope

This document provides an overview of the architecture of LArSoft toolkit, including its relationship with the surrounding software environment. The internal flow of the different subsystems is also described. The document intends to capture and convey the significant architectural decisions which have been made on the system, that reflect into the current implementation or drive its development.

This document describes the architecture of LArSoft to date. It includes description of the communication protocols with libraries it relies on and with packages LArSoft cooperates with. The design and flow of different components is described.

Some commonly used LArSoft elements are employed to exemplify flows and connections, but this is no attempt to exhaustively describe each, or any, of the single elements.

2 Overview

The LArSoft toolkit aims to offer a solution for the typical data analysis scenarios of an experiment based on a Liquid Argon TPC detector:

- generation of physics pseudo-events
- simulation of physics processes in the detectors
- simulation of the readout response
- reconstruction of low and high level physics objects
- analysis and presentation of collected data
- graphical display of physics events

As an example, suppose a scientist may want to develop a new clustering algorithm optimized for a certain type of physics events. LArSoft offers interface to generators to produce either simplified physics events or more realistic ones that include, for example, cosmic radiation. It also provides the simulation of those events in the specific experiment detector. If the target processes are common enough, the experiment might have already executed these steps on large scale, also using LArSoft, and provided the necessary input. The scientist is then presented with standard interfaces to access geometry and detector information, and standard data structure to start the reconstruction with, including single wire hits suitable as starting point for a clustering algorithm, and to store the results into. She (or he) can use the standard framework environment to write an algorithm class and its framework module, compile it and test it immediately on simulated data. The result, in a standard LArSoft structure, can be immediately visualized in a event display, and improvement made to the code. Depending on the algorithm, the time between a code change and the visualization of its effect may take less than one minute. Finally, replacing the input with actual detector data, that uses the same format as the simulation, she will immediately see the performance in the real case.

As a different example, a scientist may want to compare two different algorithms analyzing reconstructed tracks. After the tracks are produced, either running the track reconstruction algorithm on simulated or real data, he (or she) will write one or more analysis algorithms and their framework modules to produce the necessary plots.

Since LArSoft has been designed to take advantage of the *art* framework[1], LArSoft users will extensively work with its concepts, including for example services and modules, and infrastructure, like *art* scripts to create skeleton modules, and the configuration based on FHiCL language[2].

3 Logical view: components

To provide the best solutions for LAr TPC simulation, reconstruction and analysis of data, LArSoft provides a collection of built in tools and algorithms, but also interfaces to other existing libraries. Figure 1 illustrates the relation between LArSoft and these libraries.

LArSoft is designed to rely on the *art* framework [1]. This framework provides an event data model, centralized configuration, data and configuration persistency, management of user algorithm through “modules”, exception handling, and more. LArSoft takes advantage of the libraries *art* framework depends on, by using them directly:

- FHiCL language [2] to propagate the configuration to its components
- message facility [3] to regulate text output to console
- CLHEP, ROOT, Boost libraries as needed

LArSoft also shares a common platform, *nutools* [4], with other neutrino experiments (NOνA). This library provides LArSoft with some basic event display facilities and simulation data structures.

LArSoft interactions include:

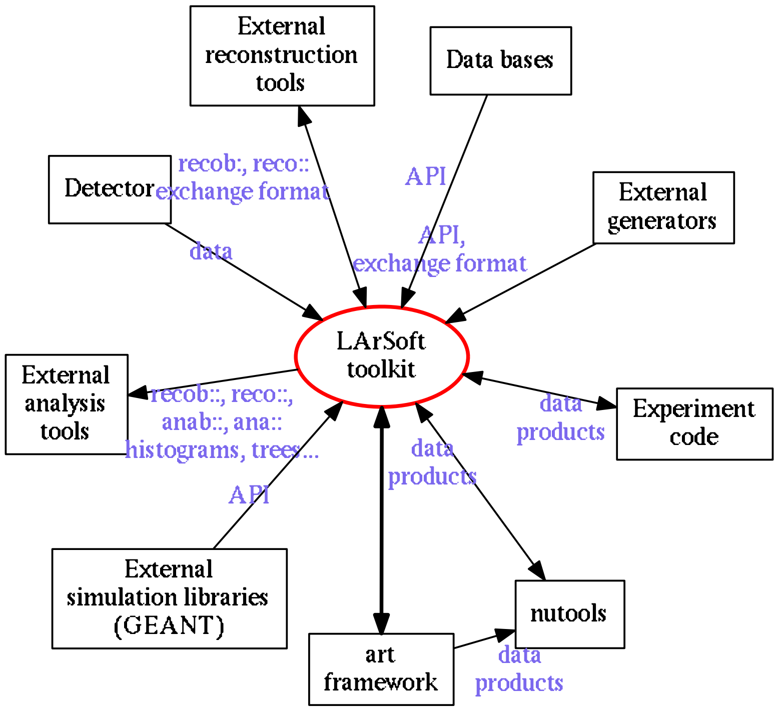

Figure 1: Relationship between LArSoft and other packages and libraries

- experiment detector data, through customized input modules converting data into LArSoft data classes
- external event generators (e.g., CRY [5], GENIE [6], via API or through HEPEVT exchange format; additional event generators are natively implemented in LArSoft
- GEANT [7], an external detector simulation library
- data bases, via direct connection or web proxy (*libwda* [8])
- external reconstruction tools (e.g., *pandora* [9]), through a LArSoft interface
- custom analysis tools, that can use LArSoft data classes directly or tailored data formats produced by custom LArSoft modules
- experiment code, written in the form of LArSoft algorithms, modules and services

4 Process view: workflows

5 Deployment view: development and extensibility

5.1 Testing

LArSoft development model allows multiple contributors to modify the code at the same time. This model can create conflicts and dysfunction in the code. Tests are instrumental to the early detection of such defects. LArSoft includes tests at two levels, called *unit tests* and *integration tests*.

Unit tests exercise a limited part of the system, typically a single algorithm. Ideally a unit test for an algorithm should test all the functions of that algorithm. In practice, tests for complex algorithms tend to set up and test a few known typical cases.

Integration tests involve the framework and one or more processing modules. These tests can reproduce real user scenarios, for example a part of the official processing chain of an experiment, and they can compare new and historical results. LArSoft tools allow these tests to be run at any time, and a standard suite of tests is meant to be automatically and periodically run.

6 Extensibility

The extensibility of LArSoft is largely based on the underlying framework, *art*. The *art* framework processes physics event independently, executing on each of them a sequence of modules. The framework also provides a list of global "services" that modules can rely on. Examples of services implemented by LArSoft include the description of detector geometry and channel mapping, the set of detector configuration parameters, and access to TPC channel quality information.

Our description focuses on extensibility in terms of new persistable data structures, of new algorithms implemented in LArSoft and of using external libraries.

6.1 Data products

LArSoft provides a basic set of persistable data classes. Each class is associated to a simple concept and a set of related quantities. For example, `raw::RawDigit` describes the raw data as read from a TPC channel; `recob::Cluster` describes a set of hits observed on a wire plane; `anab::Calorimetry` contains information about calibrated energy of a track.

A *data product* is a class that:

- is simple: contains just data and trivial logic to access it; more complex elaborations belong to algorithms
- contains only members from a small selected libraries: C++ standard library is highly recommended; ROOT classes are also accepted
- is not polymorphic

Limitations to ROOT I/O system impose restrictions on the types of allowed data members, e.g., on the set of supported C++11 containers. Relations between data products are expressed by *associations*. Associations are data products provided by *art* that can relate a data product, or an element of it, to another element from another data product. Examples of use in LArSoft include the association between a reconstructed hit and the calibrated signal it's reconstructed from, and between a cluster and all the hits that constitute it.

Data products have a fundamental structural role: they act as messages to be exchanged between algorithms. As such, they are also the format in which most of the results are saved. This allows to arbitrary split the processing chain in multiple sequences of jobs.

6.2 User code

Algorithms constitute, together with data products, the heart of LArSoft, and the ability for user to add their own algorithm is central to its design. In fact, LArSoft algorithms differ from users' algorithms only in the judgment that their purpose is considered of wider interest than just for the single user. Indeed, most of the algorithms in LArSoft were written by users to solve a specific problem, and then adopted into the common toolkit. LArSoft encourages users to produce

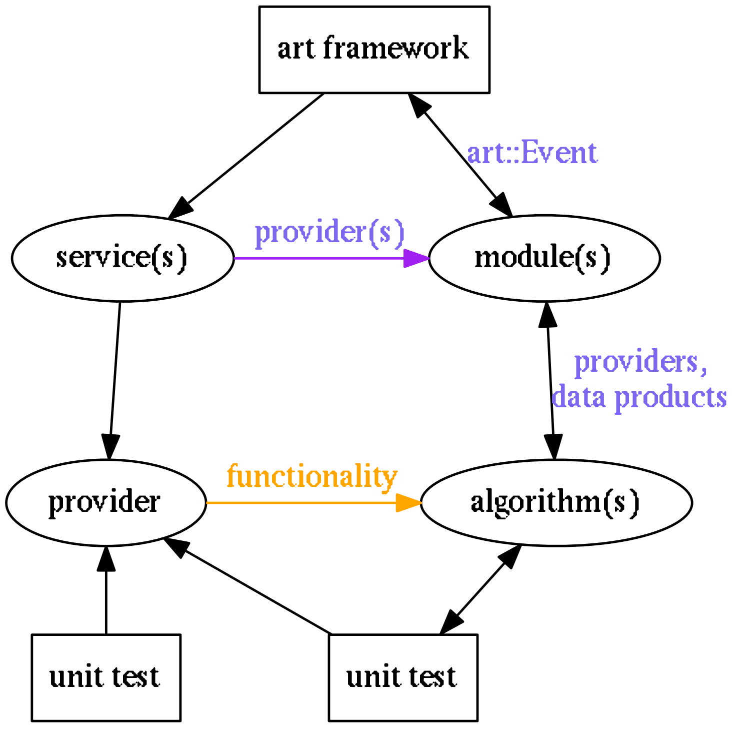

Figure 2: LArSoft algorithm and service model

algorithms that perform correctly on any liquid argon detector, and to integrate them into LArSoft itself.

The preferred model for algorithm structure is represented in fig. 2. We refer to it as *factorization* model. The underlying principle it is that the algorithm must be independently testable and portable, using the minimal set of necessary dependences. This also allows for the algorithms to be used in contexts where the *art* framework is not available, provided that some other system supplies equivalent functionalities as, and only when, needed. The model is made of two layers:

1. the algorithm, in the form of a class that

- is configurable with FHiCL parameter sets
- consumes LArSoft data products as input
- produces LArSoft data products as output
- has the minimal convenient set of dependencies
- elaborates a single event or part of an event at a time

2. a module for the *art* framework, that:

- owns and manages the lifetime of one or more algorithm classes
- provides the algorithm(s) with the configuration, the data products and the information it needs to operate
- delivers algorithm output to the *art* framework

Since algorithms often rely on services, the services also need to follow the same factorization model and be split in:

1. a *service provider*, in the form of a class that:

- is configurable with FHiCL parameter sets
- has the minimal convenient set of dependencies
- provides the actual functionalities

2. a service for the *art* framework, that:

- owns and manages the lifetime of its service provider
- provides modules with a pointer to the provider
- when relevant, propagates messages from the framework (e.g., the beginning of a new run) to the provider

The module is also responsible of communicating to its algorithms which service providers to use. Algorithms exclusively interact with service providers rather than with *art* services.

Other important guidelines for the development of algorithms are:

- interoperability: they should document their assumptions in detail, and correctly perform on any detector if possible
- modularity: each algorithm should perform a single task; complex tasks can be performed by hierarchies of algorithms
- maintainability: they should come with complete documentation and proper tests

Figure 2 shows that if algorithms are not framework-dependent, their unit test can also be framework-independent. Therefore, not only those algorithms can be developed in a simplified, framework-unaware environment, but they can also be tested in that same development environment. In other words, the full development cycle, of which testing is an integral part, can seamlessly happen in the same environment.

6.3 External libraries

We call "external" any library that does not depend on LArSoft, with the possible exception of its data products. Examples in this category are GENIE, GEANT4, and *pandora*.

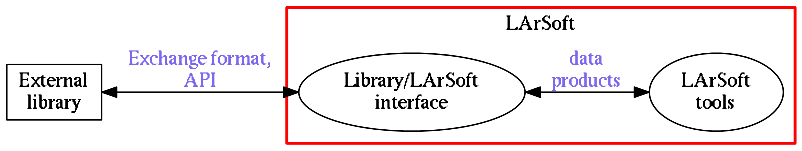

Figure 3: Interaction between LArSoft and an external library

LArSoft's modularity can accommodate contributions from external libraries into its workflow (fig. 3). The preferred way is to use directly the external library via its interface. This requires an additional interface module between LArSoft and the library, in charge of converting the LArSoft data products into a format digestible by the external library, configuring and driving it, and extracting and converting the results into a set of LArSoft data products.

This is exemplified in the interaction between LArSoft and *pandora* (fig. 4): *pandora* uses its own data classes for input hits, particle flow results and geometry specification. A base module exists that reads LArSoft hits, converts them

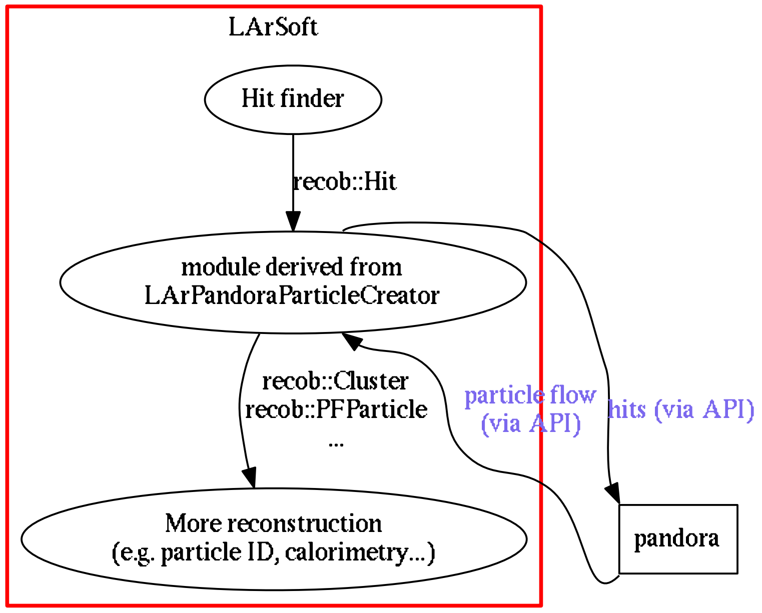

Figure 4: Interaction between LArSoft and *pandora*

into *pandora*'s, translates geometry information, and recreates out of *pandora* particle flow objects LArSoft clusters, tracks, vertices, and more.

This approach has relevant advantages: it can be fairly fast; it allows a precise translation of information; it provides the greatest control on the flow within the library; it defines and tracks the configuration of the external library. Its greatest drawback is the need for the LArSoft interface to depend on the external library. If this limitation is not acceptable, a more independent communication channel can be established via exchange files. In this case, LArSoft interface translates data products into a neutral format, possibly based solely on ROOT objects or on a textual representation, and back into data products. The external library is in charge of performing the equivalent operations with the library data format. This is for example the generic communication mechanism with event generators that support HEPEVT format. The strong decoupling comes at the price of a fragmented execution chain and the burden of additional configuration consistency control, for example to ensure that a consistent geometry was used for the information (re)entering LArSoft.

7 Physical view: repositories and packages

This stuff is going to be rearranged

A Architecture

A.1 Overview

LArSoft encompasses a collection of tools that can be roughly groups in the following categories:

1. simulation
2. reconstruction
3. analysis

- display

The typical full processing chain (fig. 5) includes a reconstruction and an analysis sequence. For simulated events, a preliminary simulation sequence can be run.
Processing chains are defined by the experiments according to their needs. LArSoft inherits the flexibility from the *art* framework, that provides users with the flexibility of choosing and arranging processing modules at will, with the only limitation that each module must be provided with all the information it needs to operate. The same module can also be executed multiple times, with different configurations.
The event display is capable of showing many of the data classes from the simulation and reconstruction steps, and it includes a limited ability of running modules with different configuration at run time.

LArSoft also provides an extensible set of data structures describing objects involved in many levels of the physics analysis, e.g., the time-dependent shape of signal from a photon detector, a simulated neutrino or a reconstructed electromagnetic cascade. The use of these common structures is key to flexibility, allowing to replace and directly compare algorithms that use the same data structures.

A.2 Simulation

The purpose of LArSoft simulation is to describe a realistic response of the detectors to a known physics event ("truth"). Since the result of the simulation should be equivalent to the output of the detectors, this result is represented by the same data classes. The truth information, not available from the detector, is produced and stored in additional structures.

The complete simulation chain is summarized in fig. 6. The process is typically described as three steps:

1. event generation

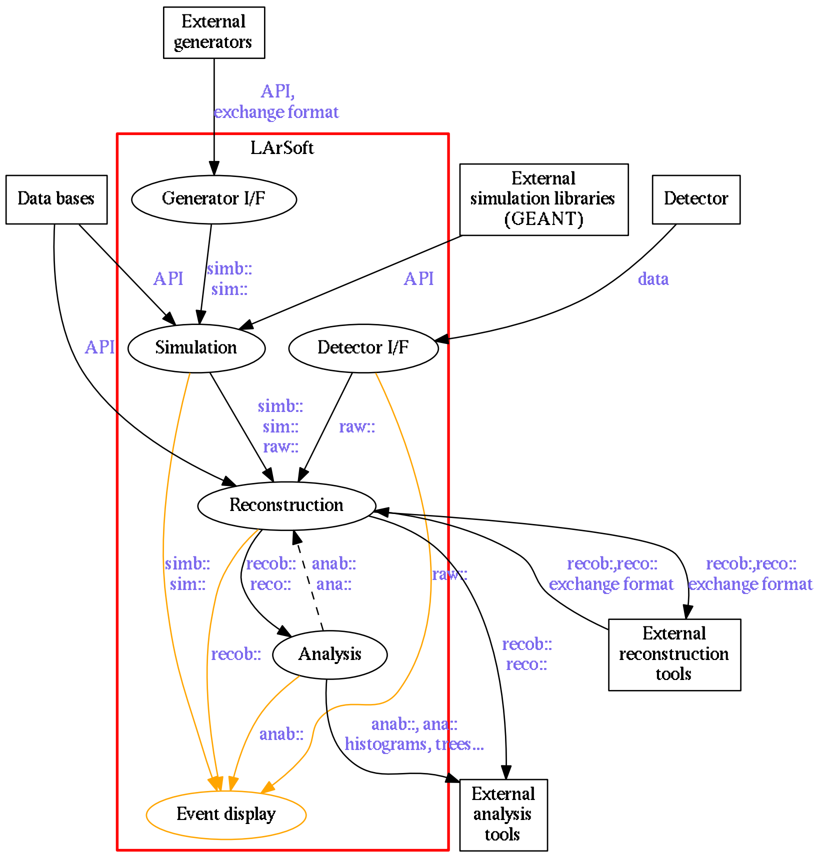

Figure 5: Components of LArSoft and their interaction with external libraries

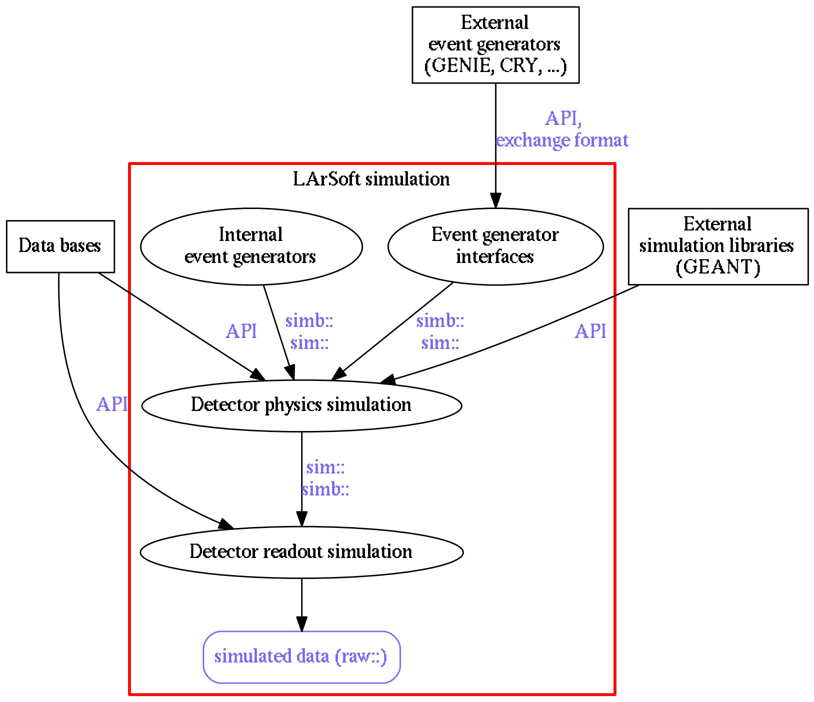

Figure 6: LArSoft simulation flow

2. detector physics simulation
3. detector readout simulation

The physics event can be generated by an external program or library. LArSoft interfaces directly to GENIE generator (neutrino interactions) and CRY (cosmic rays). It can also read a generic HEPEVT[ref] format. In addition, LArSoft provides built-in generators to simulate single particles, Argon nucleus decays, and more.

The detector physics simulation includes the interaction of the generated particles with the detector, and the propagation to the readout of produced photons and electrons. This part of the simulation relies on GEANT4 for the interaction of particles with matter. Photon and electron transportation to the readout are implemented in built-in code. Detector parameters (e.g., the intensity of the electric field) can be acquired from the job configuration or from a custom data base.

The last step transforms the physics information, electrons and photons, into digitized detector response, including the simulation of electronics noise and shaping. This is typically implemented with experiment-specific code.

A.3 Reconstruction

The reconstruction phase provides standard physics objects to describe the physics event. Reconstruction delivers objects with different level of sophistication and from different steps, as for example hits describing localized charge deposition as detected on a wire, down to a complete hierarchy of three-dimensional tracks. These objects are handed over for further analysis.

Starting from detector response, either real or simulated, there are many possible patterns of analysis. The more "traditional" one (fig. 7) starts with the calibration of the signals, attempting to suppress noise and revert electronics distortions, and then it proceeds with the reconstruction of charge deposition on a single TPC wire (*hits*), to *cluster* them in groups lying on the same wire plane, and finally with combining clusters from different planes in trajectories (*tracks*) and particle cascades (*showers*), connected by interaction points (*vertices*). The hierarchal connection between them is called a *particle flow.* Many options are implemented in LArSoft for each of these steps, that are interchangeable as they use the same input and output classes.

During any of these steps the detector and data acquisition parameters can be acquired from experiment data bases.

Any external library that utilizes LArSoft data classes to receive inputs and deliver results is also fully interchangeable with the algorithms implemented in LArSoft. A noticeable example is the *pandora* pattern recognition toolkit, that accepts LArSoft hits as input and can present its results in the form of LArSoft clusters, tracks and particle flow objects.

Further common analysis steps are the calibration of the energy deposited in liquid argon by the interacting particles and their identification as specific types (e.g., muons, protons, etc.).

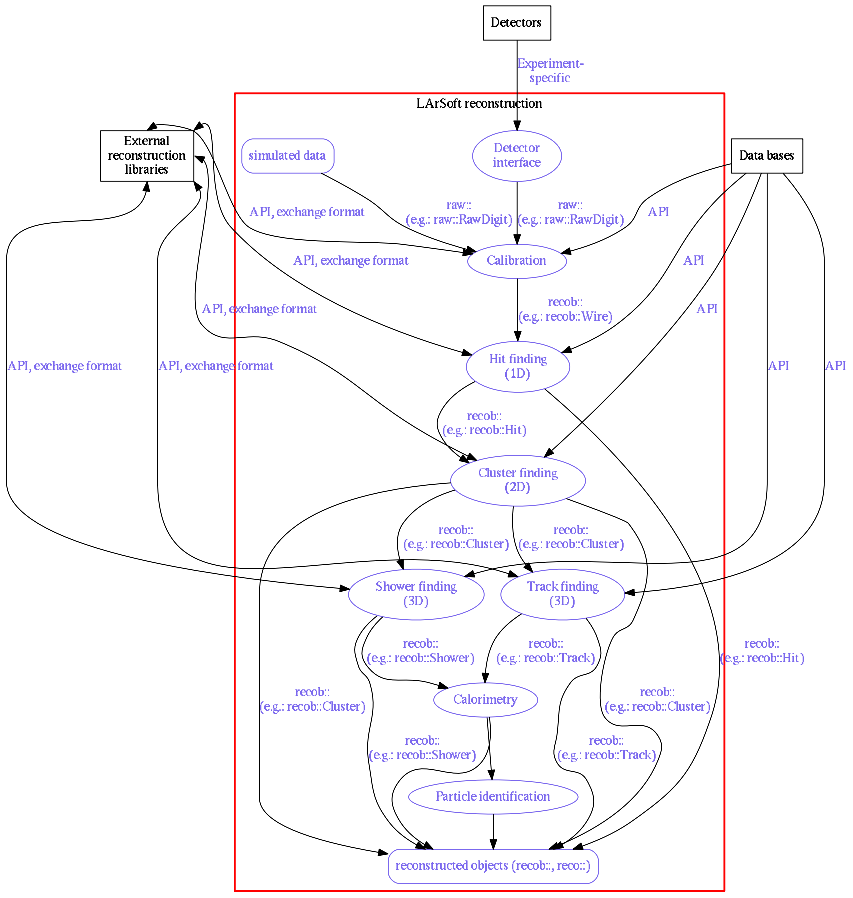
Detectors
Experiment-specific
LArSoft reconstruction
External reconstruction libraries
simulated data
Detector interface
Data bases
raw:: (e.g.: raw::RawDigit)
raw:: (e.g.: raw::RawDigit)
API
API, exchange format
Calibration
API, exchange format
API
recob:: (e.g.: recob::Wire)
API, exchange format
API
Hit finding (1D)
API, exchange format
API, exchange format
API
API
recob:: (e.g.: recob::Hit)
Cluster finding (2D)
recob:: (e.g.: recob::Cluster)
recob:: (e.g.: recob::Cluster)
Shower finding (3D)
Track finding (3D)
recob:: (e.g.: recob::Shower)
recob:: (e.g.: recob::Track)
recob:: (e.g.: recob::Hit)
recob:: (e.g.: recob::Cluster)
Calorimetry
recob:: (e.g.: recob::Cluster)
recob:: (e.g.: recob::Shower)
recob:: (e.g.: recob::Track)
Particle identification
reconstructed objects (recob::, reco::)

Figure 7: LArSoft “traditional” reconstruction flow

References

[1] Mike Wang Robert Kutschke, Marc Paterno. *The art workbook*. Fermilab, 2015. URL: `https://web.fnal.gov/project/ArtDoc/Pages/workbook.aspx`.

[2] The *art* team. Fhicl configuration language.

[3] Message facility.

[4] nutools. URL: `https://cdcvs.fnal.gov/redmine/projects/nutools`.

[5] Cry.

[6] Genie.

[7] Geant4.

[8] libwda.

[9] pandora.

Comments

Sunday 7:51:39, Ruth Pordes `ruth@fnal.gov` ("Re: First draft of the architecture description document")

[Q 001] does the scope include human interfaces as well as software?
[A 001.1] [GP] I thought mostly not, but I am not completely sure what human interface includes. To be clarified. *(TODO)*

[Q 002] nutools event display facility and simulation data structures – still does not make sense to me. Is Visualization one special kind of analysis or does Larsoft have specific interfaces to it?
[A 002.1] [GP] Visualization is a special kind of analysis. But our event display crosses the border with its (limited) ability to *interactively* reprocess the input.

[Q 003] page 4 – can components of the chain be re-executed during a single pass?-
[A 003.1] [GP] I have added a couple of sentences in the previous-to-last paragraph of Architecture > Overview section, that I hope give an answer. The answer is very much in the features of art, that I have not covered at all in this text. Should we? *(TODO)*

[Q 004] does event display have a specific meaning - I'll include it in the Requirements glossary – it is different from a generalized visualization and I presume the definition should explain this? Also, if the event display is in nutools it is not part of larsoft??? Can we share a glossary in some fashion?
[A 004.1] [GP]

[Q 005] Figure 2. You explicitly mean Detector not DAQ ? Does/shoud daq show up somewhere
[A 005.1] [GP] in practice DAQ products is what we communicate with. It doesn't have to be only that, but I guess that is it effectively what happens. *(TODO)*

[Q 006] a Fluka interface is in the works with integration hoped for before the end of Dec. Can you include a sentence on this interface?
[A 006.1] [GP] Erica, confirm? *(TODO)*

[Q 007] page 10.Unit test. These are important. These are not the only tests. I don't see them referred to and perhaps some more specifics might be useful?
[A 007.1] [GP] I added a section about testing. I have added a few words also at the point Ruth specified (at the end of "User code" section). I think it would be good to add a "test" block in one of the high-level diagrams, but I can't figure out where (probably in 5, but how?). Or maybe we have to add a *development model* section?