The LArSoft architecture (draft)

The LArSoft project

January 29, 2016

Contents

1 Introduction

1.1 Purpose

The LArSoft toolkit enables 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, and facilitate direct sharing of code and experience between 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 reflect into the current implementation or drive its development.
Some commonly used LArSoft elements are mentioned to exemplify flows and connections, but this is no attempt to exhaustively describe each, or any, of the single elements.

This document describes the architecture of LArSoft to date. At the time of writing, LArSoft `v05_00_00` is in Release Candidate 2.

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 detector 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

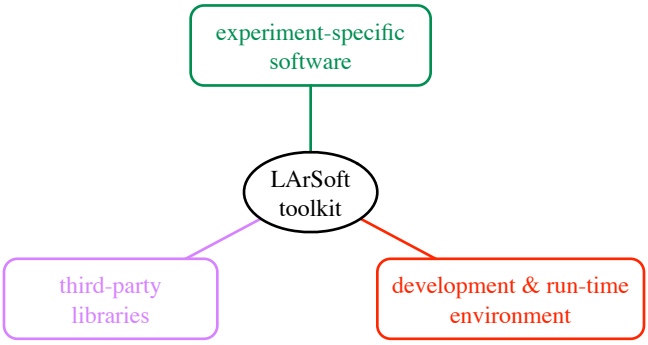

Figure 1: Relationship of LArSoft with other software categories.

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 structures to start from, including hits on single wires 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 data 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, running the track reconstruction algorithm on either simulated or real data, she will write one or more analysis algorithms and their framework modules to produce the necessary plots.

3 Logical view: components

To provide the best solutions for LAr TPC simulation, reconstruction and analysis of data, LArSoft interacts with other software aimed to provide developers with tools commonly in use by the broader physics community, standardize code development, and allow for experiment-specific needs (fig. 1).

Physics developers typically rely on copious libraries providing general or physics-specific services (fig. 2). LArSoft already offers:

- access to a framework, *art* [1], providing essential functionalities including an event data model, an event loop, workflow definition and control, plug-in of code, distribution and tracking of job configuration, serialization of the results, and more

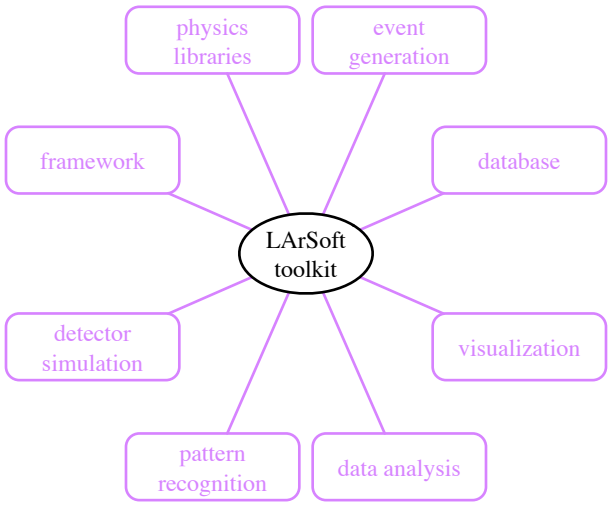

Figure 2: Relationship between LArSoft and third-party libraries.

- proxy-, web-based access to data bases via *libwda* [2], or direct access to PostgreSQL databases[1]
- physics libraries, as CERN CLHEP [3] and *nutools* [4]
- event generation packages: GENIE [5], CRY [6], HEPEVT [7] files
- detector simulation libraries (to date, only GEANT4 [8])
- pattern recognition libraries, like *pandora*
- data analysis tools, like CERN ROOT [9]
- visualization aids, also with CERN ROOT and *nutools*

Additional libraries are expected to be added in the future.

LArSoft is designed to accommodate specific needs from the experiments. Experiments directly contribute LArSoft content when it's suitable, i.e. when of general utility and experiment-agnostic. In the other cases, experiments interface to LArSoft though many channels (fig. 3):

- detector geometry is provided in GDML or ROOT format
- detector conditions can be learned via static configuration or from experiment databases

[1] Experience has shown that direct database does not scale well with the number of accessing jobs.

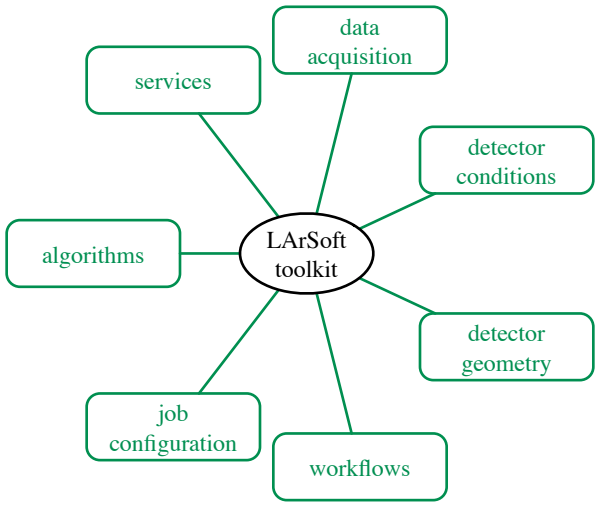

Figure 3: Relationship between LArSoft and experiment-specific software.

- detector data is acquired by special *art* modules or by standard *art* files (e.g., produced by *art*DAQ) containing standard LArSoft data products
- specialized services and algorithms can be plugged in using the *art* framework
- job configuration, controlling the data to be processed and the sequence of actions to perform, is specified in FHiCL language [10]
- workflows are defined by the experiment, typically by using custom scripts that include the execution of LArSoft main program, `lar`

LArSoft has a large number of interdependent components, and provides the users tools to facilitate code development (fig. 4). LArSoft code is organized in repositories that can be compiled when needed. The building system ensures consistent builds among all supported platforms. The UPS [11] distribution system ensures that the same consistency is preserved at run time. Infrastructure for automatic execution of user tests is also provided, together with a growing number of tests exercising parts of LArSoft tools.

3.1 Internal components

Most LArSoft components can be grouped into some broad functional categories. Some of them are well established, while others are being developed now or have been just designed. The following list touches the main ones, without being exhaustive:

detector information

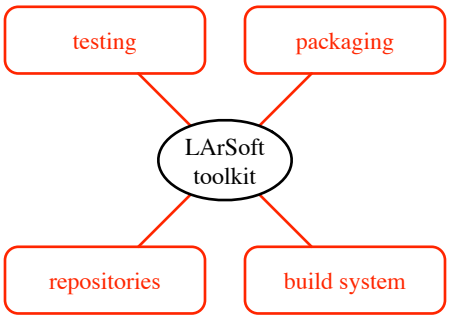

Figure 4: Relationship between LArSoft and development software categories.

- detector geometry description
- detector information services: liquid argon and detector properties, readout timings and settings, readout channel quality
- calibration services: readout channel pedestals
- map of residual electric charge in TPC volume

persistent data structures ("data products"), grouped in

- raw data, unprocessed from the detector
- simulation from event generator ("truth") and detector simulation
- reconstruction of detector and physics objects
- optical data, raw or processed, from the optical detectors
- analysis results of reconstructed objects

operations

- physics event generation
- detector simulation: TPC and optical detectors
- readout simulation: template modules for TPC and optical detectors
- simulation of optical triggers
- calibration template modules
- object reconstruction: 1D (TPC wire hits, optical hits), 2D (TPC hit clusters), 3D (tracks, showers, vertices) and time (optical flashes)
- TPC hit simulation and correlation between reconstructed objects and generated particles
- energy and momentum reconstruction ("calorimetry")
- particle identification
- global event reconstruction

programming utilities and framework interface

- physical constants
- helpers for common framework usage patterns (e.g., creation of associations between data products)

graphical display of generated and reconstructed objects ("event display")

example of analysis module

A more detailed illustration of the relations between some of the simulation and reconstruction components and the environment will be given respectively in sections 4.1 and 4.2.

4 Process view: workflows

LArSoft tools can be sequenced and combined to define complete workflows. The typical usage is aligned to three main types of "standard" workflows:

1. simulation
2. reconstruction
3. analysis

where the first step, simulation, is of course skipped when processing real detector data. LArSoft does not directly define these processing chains. Rather, it inherits the flexibility from the *art* framework, which provides users with the flexibility of choosing and arranging processing modules at will. Thus, the Experiments define the steps of each workflow according to their needs. Still, these needs are fairly shared, and it is possible to characterize a "typical" chain for each workflow.

4.1 Simulation workflow

The purpose of a simulation workflow 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 in both cases by the same data structures.

The main results of this workflow are:

- data products representing the detector response (e.g. `raw::RawDigit` and `raw::OpDetWaveform`)
- data products representing the simulated physics (e.g. `simb::MCTruth`, `simb::MCParticle`)

The complete simulation chain is illustrated in fig. 5. The process is typically divided in three steps:

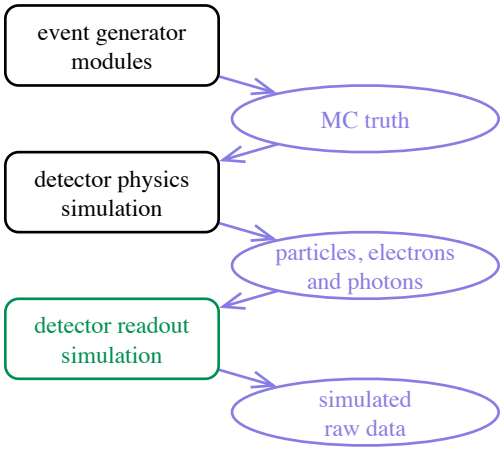

Figure 5: A typical LArSoft simulation workflow. Blocks represent operations and ellipse represents data. Block colour coding follows fig. 1.

1. event generation
2. detector physics simulation
3. detector readout simulation
   - TPC: signal on the wires
   - optical detector(s)
   - "auxiliary" detectors (e.g. scintillator pads)

The physics event can be generated by an external program or library (fig. 6). LArSoft can interface directly to generators that offer API, through a specific driver module. Currently driver modules are offered to work with GENIE generator (neutrino interactions) and CRY (cosmic rays). LArSoft can also read events stored in HEPEVT format. In addition, LArSoft provides built-in generators for single particles, Argon nucleus decays, and more.

The detector physics simulation includes interaction of generated particles with the detector, and transportation to the readout of produced photons and electrons. This part of the simulation currently relies on GEANT4 for the interaction of particles with matter. Photon and electron transportation is 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 Experiment database.

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, and separately for each sub-detector type.

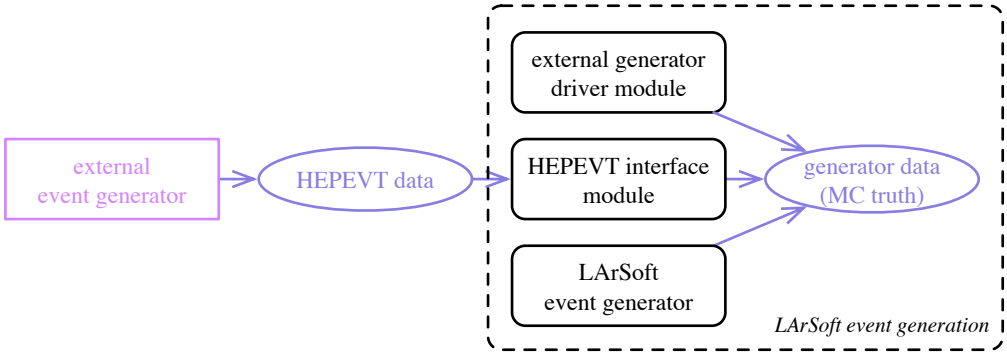

Figure 6: Event generation workflow. Blocks represent operations and ellipses represent data. Block colour coding follows fig. 1. The dashed line encompasses the part of the workflow driven via LArSoft configuration.

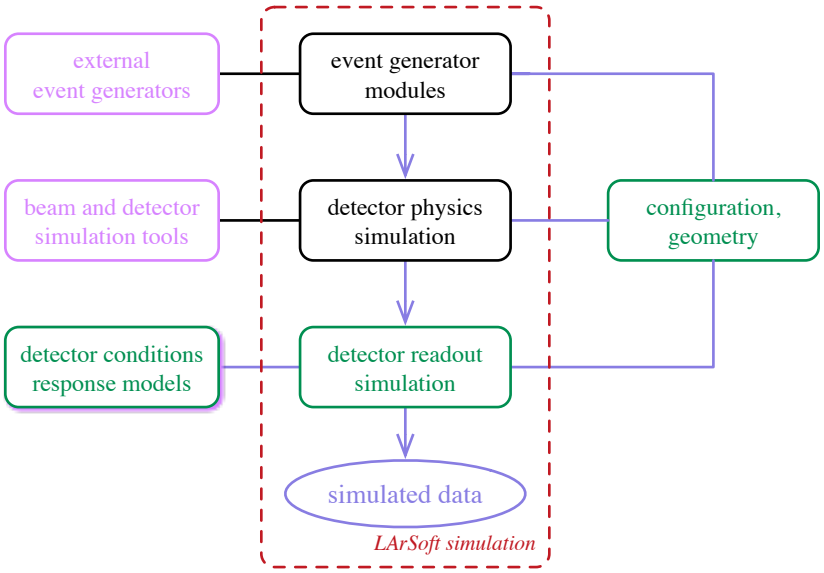

Figure 7: Components involved in simulation and their relation. Blocks represent operations and ellipse represents data. Lines represent communication between components: via data transfer (purple lines) and via API (black lines). Block colour coding follows fig. 1. The encircled area maps the simulation workflow illustrated in fig. 5.

4.2 Reconstruction workflow

The reconstruction phase produces standard physics objects describing the physics event. Reconstruction delivers objects with different level of sophistication, 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. Through the workflow, detector and data acquisition parameters can be acquired from Experiment databases.

Many possible reconstruction strategies are possible. LArSoft allows them to be applied indifferently to data produced by a real detector or simulated. Support is planned for mixing the two types of sources together. The more "traditional" one (fig. 8) proceeds through:

1. calibration of the signals, noise suppression and removal of electronics distortions;
2. independent reconstruction of charge deposition on each TPC wire (*hits*);
3. definition of *clusters* from hits lying on the same wire plane;
4. combination of clusters from different planes in trajectories (*tracks*) and particle cascades (*showers*);
5. identification of interaction points (*vertices*);
6. hierarchal connection of them into *particle flow* structures. Many options are implemented in LArSoft for each.

Different algorithms can be chosen to perform each of these steps. 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.

This workflow is extremely simple and factorizes the process in highly independent steps. Alternative workflows can and have been developed.
For example, a derivation of the workflow described above consists in a complete first pass tuned for reconstruction and subsequent identification of background objects (mostly cosmic rays); in their elimination at the level of hits; and in a second pass tuned for reconstruction of neutrino interactions.
Other approaches include direct track reconstruction without clustering; direct clustering from calibrated or uncalibrated channel signals by image processing algorithms, bypassing the construction of hits; a quick, preliminary track reconstruction followed by a complete reconstruction utilizing the first result to better direct the algorithms; inverting the order of track and vertex finding; and more.

LArSoft and *art* modularity supports any acyclic workflows, with any predetermined number of (potentially optional) steps. It does not accommodate cyclic workflows.

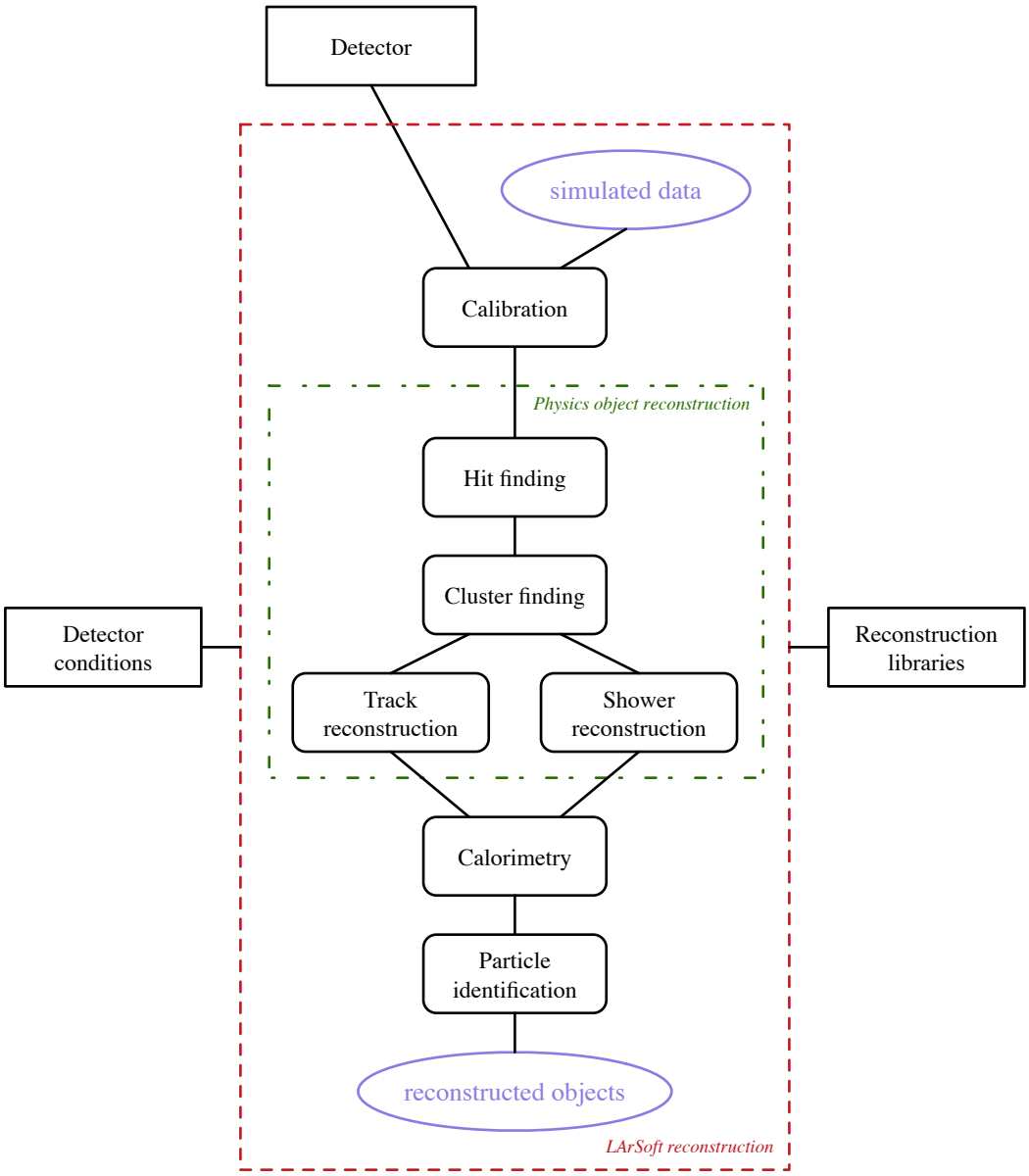

Figure 8: A typical LArSoft reconstruction workflow. The red contour delimits the components within LArSoft toolkit, and the green one encloses the reconstruction of basic physics ojects. Blocks represent software and ellipses represent data.

4.3 Analysis workflow

Analysis workflows are the most vaguely defined, due in part to the more diverse goals, and partly to the fact that in this relatively early stage the Experiments have devoted most of the time to simulation and reconstruction. No prototype analysis workflow has emerged yet.

The calibration of energy deposited in liquid argon by interacting particles and their identification as specific types (e.g., muons, protons, etc.) have been classified sometimes as "analysis", sometimes as "reconstruction". Another common analysis task is evaluation of reconstruction performances and comparison between different algorithms and strategies.

Calibration activities, for example pedestal analysis, characterization of argon purity, mapping of the electric field, also fall in this category and they are ideal candidates for the standardization of workflows.

5 Deployment view: development and 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*. An event is defined by an input module. In most Experiments it is bound to a single pulsed beam interaction with the detector, but test beam Experiments, non-beam Experiments and non-beam analyses (e.g. proton decay) may need to define different event boundaries. 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.

In this section we describe the development environment and then focus on the main handles LArSoft offers developers for the sake of extensibility, including new serializable data structures, new algorithms and the use of external libraries.

5.1 Development environment

LArSoft is designed for and supports the use of a development environment based on:

- UNIX Product Support (UPS) for access to dependent packages
- *cetbuildtools* [12] as build system
- Multi-Repository Build[13] (MRB) to coordinate build and execute software from different repositories
- git (recommended) or SVN for version control

LArSoft is fully supported on the following platforms:

- Scientific Linux Fermi: version 6
- Darwin: version 13 (OS X 10.9 "Maverick") and 14 (OS X 10.10 "Yosemite")

LArSoft typically supports the two most recent versions of these operating systems[2]. Support is also planned for the long term support release of Ubuntu Linux (16.04 LTS).

A typical workflow starts with the set up of a working area. After the area is created, subsequent utilization of it requires just a simple set up. LArSoft provides a script for this set up, and it is common practice for the Experiments to provide customized ones.

The development, whether it is creation of new code or modification of existing one, follows the following workflow:

1. development-specific set up of the existing working area
2. importing the source code to be modified, if any; this code will persist in the area
3. modifications as needed
4. building
5. optional (and recommended) run of a standard test suite
6. installation for running

The execution of LArSoft code including user development, as described above, follows this workflow:

1. run-time specific set up of the existing working area
2. execution of the software

The execution of LArSoft code as distributed, without modification, has a simpler set up that does not require a development working area.

LArSoft currently provides no facility to execute code remotely, including job submission to remote clusters. The Experiments supply workflows and scripts for this type of execution.

5.2 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.

[2] The actual supported versions depend also on the underlying support of the O.S. by Fermilab.

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.

5.3 Data products

LArSoft provides a basic set of serializable 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 correlated 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* which can relate a data product, or an element of it, to another element from another data product. Examples of use in LArSoft include associations 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.

5.4 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 their own specific problems, and then adopted into the common toolkit. LArSoft encourages users to produce algorithms that perform correctly on *any* liquid argon detector, and to integrate them into LArSoft itself.

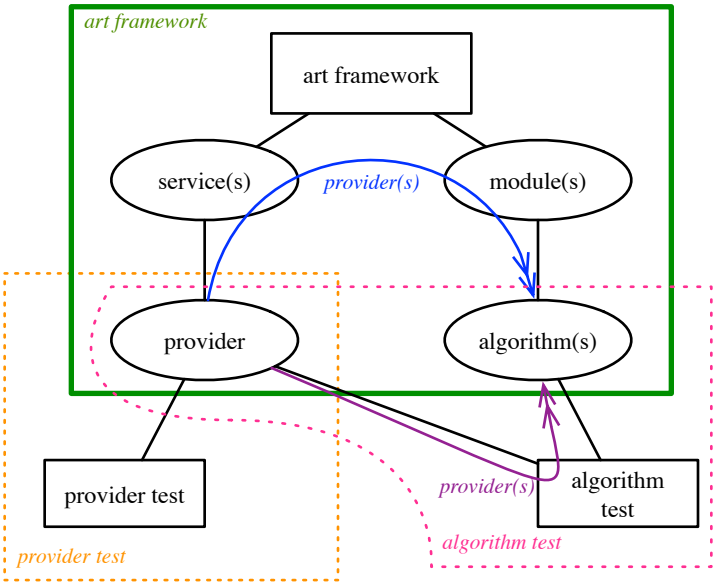

Figure 9: LArSoft algorithm and service model. Black lines represent ownership. The coloured arrows show the path the algorithm obtains the provider through. The green line contours the standard execution environment. Dotted lines describe testing environments: both service providers and algorithms can be tested without involving the full framework.

The preferred model for algorithm structure is represented in fig. 9. We refer to this 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 actual functionality
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, reacts to messages from the framework (e.g., the beginning of a new run) and propagates them to the provider as needed

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

Figure 10: Interaction between LArSoft and an external library.

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 9 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.

5.5 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*.

LArSoft's modularity can accommodate contributions from external libraries into its workflow (fig. 10). 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. 11): *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 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

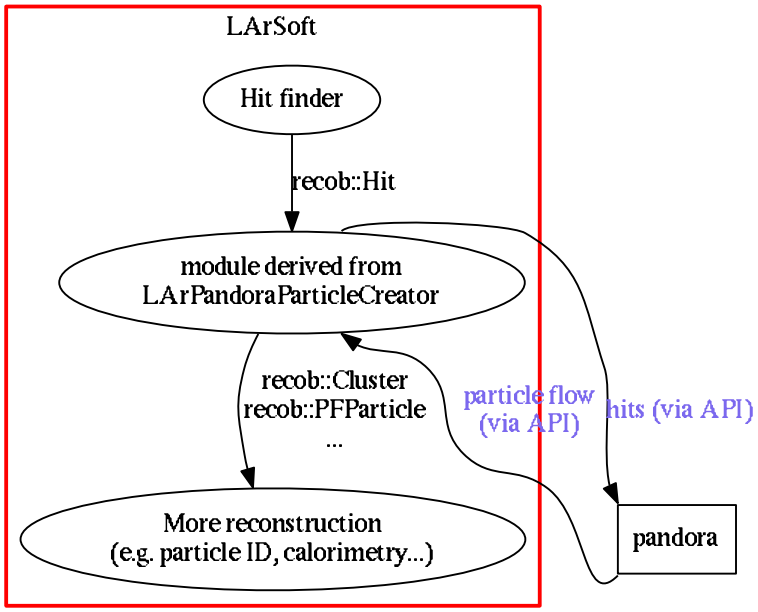

Figure 11: Interaction between LArSoft and *pandora*

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.

6 Physical view: repositories and packages

LArSoft supports the use of the UNIX Product Support (UPS) system for deployment of LArSoft itself and of the additional software it depends from. This system is organized in *products* containing executable code for a specific platform and auxiliary data as needed. LArSoft set up demands from UPS a specific version of almost every library LArSoft depends on, including for example the GNU compiler, Boost libraries and CERN ROOT.

LArSoft code base is organized in *repositories* grouping different functionalities. The current list of repositories is:

larcore independent of data products (e.g. geometry description)

lardata defining the shared data products

larevt code independent of simulation and reconstruction algorithms (e.g. calibration, database access)

larsim detector simulation

larreco physics object reconstruction

larana depending on simulation or reconstruction algorithms (e.g. particle identification, calorimetry)

larpandora interface with pattern recognition package *pandora*

lareventdisplay ROOT-based visualization tool

larexample examples of LArSoft modules

larsoft "umbrella" product

Additional LArSoft repositories do not contain source code:

larsoft_data containing small-size, slowly-changing data files

lar_ci providing a Continuous Integration test system that allows instant, thorough test of the code

LArSoft repositories are maintained in Fermilab Redmine as git repositories.

6.1 Local LArSoft installation

LArSoft can be installed in any supported platform, either with:

binary installation copying prebuilt UPS products from Fermilab server into a local UPS directory

source installation copying, building and installing into a local UPS directory the source code of each and every dependent package

Both installation patterns are supported via a single script. In this way, LArSoft can be installed in virtual machines, personal computers as well as in computing clusters.

References

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

[2] libwda.

[3] Cern clhep.

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

[5] Genie.

[6] Cry.

[7] Hepevt format.

[8] Geant4.

[9] Cern root.

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

[11] Ups.

[12] *cetbuildtools*.

[13] Multi-repository build.