LArSoft/LArLite Interoperability Proposal

Chris Jones, Marc Paterno

DRAFT FOR REVIEW

Overview

The purpose of this document is to advance the discussion regarding the interoperability of LArSoft and LArLite. The first sections of the document describe the current status of LArSoft and LArLite. The following sections describe our proposals for allowing the interoperability of the two systems, and which we believe will improve ease of use and maintainability. **This is not a final proposal**; this document is intended to elicit feedback to allow the production of a final proposal. The final proposal will define the tasks to be undertaken by members of the SCD, developers of LArSoft, and developers of LArLite.[1]

Our description of the use of LArLite is the result of a series of interviews of users (and the initial author) of LArLite, conducted by the authors. We have synthesized the data obtained from these interviews. These interviews drove our proposals. The people we interviewed were Kazu Terao (the initial author of LArLite), Wes Ketchum, David Adams, Ryan Grosso, and Corey Adams.

What is LArLite?

LArLite is two related systems: (1) a project layout generator and build mechanism and (2) a modular “event loop” driver and ancillary software, built by (1). Part of the challenge of describing its current uses and possible future development is being clear about which system is being discussed.

Build mechanism

The core of LArLite is a generator for producing a specific directory structure, GNU Makefiles and skeleton source files, including the necessary mechanisms to generate ROOT dictionaries. The design is such that there is a recommended (by the author) way of combining the generated code with other code generated in the same manner. The structure and use of the Makefiles is not enforced in the system, which allows users to tailor it to their own personal needs.

Modular event loop driver

When many people speak of LArLite they are not referring to the build mechanism but instead to a modular event loop driver that is built using that mechanism. Algorithms are packaged into

[1] We have begun preliminary tasks already; David Dagenhart has started familiarizing himself with the development model for *art* and LArSoft.

modules. The event loop driver reads events in sequence and passes the event data from module to module.

As part of the infrastructure for the event loop driver there are C++ classes which represent frequently-used data products. Some of these data products are designed to mirror LArSoft data products while others are experiment-specific.

Why do people use LArLite?

For the build mechanism, and development of algorithms

1. Faster build than when developing in LArSoft.
2. Smaller installation than needed for developing in LArSoft.
3. Faster installation with fewer steps than required for LArSoft.
4. Better stability; LArSoft's head is updated frequently, leading to breakage of user code.
5. Availability on Ubuntu.
6. Freedom to commit code to one's own repository, no need to synchronize with the LArSoft community.

For the event loop driver, doing analysis and developing (testing) algorithms

1. Event loop is simpler, thus faster, than the one in *art*. The speed difference depends on the tasks being done, and so is hard to meaningfully quantify. Very large factors are reported in a case when the *art* framework was configured to do time-consuming but needless work on every event.
2. Definition of data products can be experiment-specific.
3. Existing frequently-used data products in LArLite have a preferred interface to those in LArSoft, especially for iteration.
4. Modification of data products is easier; modified code can be committed to one's own repository without need for coordination between experiments or groups.
5. Integration with PyRoot for analysis or for writing tests is simple.
6. LArLite writes bare ROOT data files containing a TTree.

Recommendations: How to use LArSoft from LArLite

Using LArSoft from LArLite means:

1. Reading art/ROOT data files.
2. Using LArSoft data products (read from art/ROOT files, or created directly in LArLite); this means using the LArSoft classes directly, not copying them.
3. Using existing LArSoft algorithms in the LArLite event loop driver is not included. None of the users we interviewed are currently doing this, nor when asked did they wish to do so. The common pattern of use is to use LArSoft to generate LArLite data files, and then to work in the context of LArLite. Given that it would take considerable extra effort to include the ability to use LArSoft algorithms in LArLite, and lacking a clear statement of its importance, we do not think the effort should be expended. This limitation should be

reviewed after there is experience in using the result of the integration, if there is sufficient community interest.

Using LArSoft data products

LArSoft should be used through a binary distribution. Only the part of LArSoft necessary to provide usable data products needs to be accessed. This is analogous to how LArLite is already using ROOT.

1. LArLite should have Makefile fragments that give access to LArSoft headers and libraries.
2. The LArLite setup should establish the user's environment in order to access the necessary parts of LArSoft, *e.g.*, properly setup the path to find the shared libraries.

Reading art/ROOT data files

It should be possible to read art/ROOT data files, containing LArSoft products, from outside of the *art* framework. This requires:

1. Access to the ROOT dictionaries for the required classes.
2. Access to the LArSoft and *art* header files and libraries for the required classes.
3. The ability to resolve *art::Ptr* objects (which makes available the use of *art::Assns*).
4. The *art::FindOne* and *art::FindMany* classes must be made to work.
5. An *art::Event*-like class is needed for type-safe access to data products in an art/ROOT file.
6. Both sequential and random access into *art*/ROOT data files must be supported.

The user should be able to open multiple art/ROOT data files simultaneously in their own code outside of *art*, (e.g. their event loop driver). It would be the user's responsibility to ensure whatever consistency requirements are relevant are met.

Recommendations: How to develop, in LArLite, code that is interoperable with LArSoft

In this context, *interoperable* means two things: (1) a dynamic library built using LArLite's build mechanism should be linkable directly to a program utilizing LArSoft, and (2) the source code for such a library can be moved to LArSoft without modification.

1. The LArLite build mechanism should be able to build a dynamic library containing an algorithm, so that that one can build and link an *art* module that uses that algorithm.
2. LArLite directory naming conventions may need to be modified to support this.
3. There must be a development workflow devised that must be followed for the development of the subset of LArLite code that is to be available in LArSoft. It must be possible to migrate LArLite code from using some other set of development procedures to using these required development procedures. This allows a user to start development however they are most comfortable, and to adopt the required procedures for compatibility with LArSoft only when that is desired. Options for this workflow will be

presented in a follow-up document. Requirements that must be satisfied by this procedure are described in the appendix.

Suggested changes to LArSoft and *art*

1. LArSoft should better document and advertise the binary distributions.
2. It should be possible to do a binary installation of the LArSoft data products without needing to install all the rest of LArSoft.
    a. LArSoft repositories may need some reorganization to support this.
    b. It will be necessary to identify the parts of *art* needed by the LArSoft data products, and to allow the distribution of that part of *art*.
    c. The external dependencies of these products should be minimized.
3. ROOT 6 is capable of understanding much more C++ than was ROOT 5. LArSoft and *art* should move to using ROOT 6, and the LArSoft/LArLite integration effort should proceed using the versions of *art* and LArSoft that have moved to ROOT 6.
4. The file paths used for #include statements in LArSoft may need to be modified, in order to make the migration of code meet the above requirements.
5. *art::Ptr* must be made to work outside of the framework. This may involve some reorganization of *art* itself, which is related to (2b) above.
6. The LArLite data products should be evaluated to see what changes in LArSoft data products should be made to improve LArSoft's usability.

In addition to the changes above, it may be necessary to introduce an *artlite* package (name to be determined), to contain the "light framework support" components (*e.g.* a "lite Event" class). This new package should depend only upon those parts of *art* that are necessary, which means that *art* itself may need to be split. This split should be done in a fashion such that existing users of *art* do not need to make modifications of their code.

Suggested changes to LArLite

The LArLite build mechanism should allow (but not require) the user to setup *art* and LArSoft. This might be made available as an add-on to the LArLite build mechanism. This will allow users to be able to develop, under the LArLite build mechanism, a complete *art* module, and to immediately use that module in *art*.

We are concerned that the current build mechanism makes it too easy to accidentally produce an inconsistent build (*e.g.* part of a system built with C++03, and part with C++14). More investigation is needed to determine how to help assure consistency while retaining ease of use. Shared code (*e.g.* code that uses the LArLite event loop driver) may need stricter rules than an individual user's code.

Better version control over the LArLite code may be needed; we are not aware of an existing policy for control. Tagged releases, if they do not already exist, are needed.

Code that goes into the LArLite-dependent repositories that are shared and supported should follow a set of best practices agreed upon for the project. We suggest LArSoft and LArLite should embrace the same policies.

Appendix

Requirements for the development procedure for LArLite code to be used in LArSoft

Note that no special preparation must be needed while developing code using LArLite. The required development procedure is only needed for code in LArLite that is to be used from LArSoft, when it is determined that that code is to be used from LArSoft.

LArSoft needs full control over the code it will release. Possible solutions include the follow; others are possible. This might be accomplished by LArSoft controlling a repository forked from one originally created by a LArLite user. LArSoft and *art* also already have the facility to make use of UPS products built from source code not under version control by LArSoft or *art* (e.g. ROOT and GEANT4); the same techniques could be used to create controlled versions of UPS products for code managed in LArLite repositories.

There must be a prescribed procedure for migrating code changes between LArSoft and LArLite that prevents unintentional divergence of any source code being maintained in both. Note that one way of meeting this requirement is to require the source to reside in exactly one master repository. There are also other possible solutions, such as a workflow required for interacting with forked repositories as described above.

As mentioned in the main body of the proposal, there shall be a follow-up document proposing several solutions, describing their strengths and weaknesses, including the authors' recommendations. That document should be presented to the LArSoft and LArLite stakeholders for a final selection of a solution. That document is to be used as an additional input for the production of a final work proposal.