

Architecture and Design for the LArSoft Continuous Integration Build System

*Marc Mengel, Erica Snider, Patrick Gartung, Mark Dykstra,
Lynn Garren, Gianluca Petrillo
Fermilab*

Version 0.4
August 14, 2014

Abstract

The LArSoft Integration System (LIS) will integrate the build and test of LArSoft and related experiment software with the Central Build and Integration Service (CBIS) operated by the Fermilab Scientific Computing Division. This document describes the architecture of the One of the primary functions of this system is to allow continuous integration (CI) testing of the LArSoft and experiment software. This document describes the overall architecture of the LIS, the specific details that pertain to the CI functionality, the relationship of the various components to each other, and other design elements needed to meet the system requirements as defined in the LArSoft CI System Requirements document, v0.6.

1. Overall architecture

The LArSoft continuous integration (CI) system has the following major components:

- A hardware CI system comprised of a server and a set of distributed CI slave nodes on which workflows are executed.
 - The CI server and some fraction of the CI slave nodes are provided as part of the Central Build / Integration Service¹ (CBIS) operated by the Scientific Computing Division.
- A CI application and workflow automation engine based on the Jenkins CI system² that runs on the CBIS.
- A Jenkins CI configuration that defines the elementary CI workflow(s) to be run by Jenkins, as well as the triggers, build parameters, and other configuration parameters required to operate the system.
- A set of “build-step” scripts run within the workflows that drive test operations or execute elements of workflows. The latter may be called “workflow scripts”.
- A set of utilities and functions in the CI system software infrastructure that are independent of Jenkins, and that provide required or useful test and administrative functionality.

¹The requirements for the central build / integration service are documented in CD-DocDB-5319.

²The Jenkins CI system is documented in detail at <http://jenkins-ci.org/>.

- A test-result database in which specific results from particular runs of the system are stored, and from which reports are generated.
- A web-based test result reporting system that provides access to all test result information via an intuitive and simple interface.
- LArSoft and experiment-specific test scripts and commands that perform tests of that software.
- The LArSoft and experiment-specific software that is to be tested.

1.1. Software management and maintenance

The Jenkins configuration, build-step, workflow, infrastructure utilities and functions are maintained in the `larci` software repository within the Fermilab Redmine instance. The structure of the software and the repository is such that it can be turned into a ups product, although that will not be a requirement for using the package.

Unit test software lives in the same repository as the code being tested. All tests are defined using `cet_test()` macros from the `cetbuildtools` package in the appropriate `CMakeLists.txt` files within the repository. By convention, all such definitions will be in `CMakeLists.txt` files under a top-level sub-directory named `test` within the repository.

Stand-alone integration test scripts live in the highest-level repository available for the test. Tests of the reconstruction or simulation chain, for instance, should live in the experiment repositories. Integration tests of the data products, on the other hand, could live in the `lardata` repository. In all cases, the scripts that drive the test should live under the top-level `test` sub-directory in the repository.

2. Jenkins CI configuration

The Jenkins configuration consists of a set of pre-defined parameters that are used to pass run-time information into Jenkins, and a CI workflow.

2.1. Run-time build / trigger parameters

A build / integration workflow will be triggered in a single Jenkins Project on the CBIS by an HTTPS Post request, with the following minimum set of parameters:

- Revision specification, which can be any of:
 - None -- defaults to the “develop” branch
 - OR
 - Global branch / release tag -- modules will be checked out with that tag if present, falling back to the “develop” branch if not.
 - OR
 - Specific list of (module ; branch, tag, revision, or built-release) pairs
- Forwarded grid proxy -- this proxy will be used by the workflow to fetch input files for testing, transfer output files, etc. Various repository accounts on the repository server `cdcvs.fnal.gov` will need to maintain a current grid proxy file for build triggers sent from repository hooks
- Test Suite(s) -- a specific list of integration test suite names to run. Details of the integration test setup are described below.
- The list of products on which integration tests are to be performed.

These and other parameters are passed to underlying build-step and workflow scripts as command line arguments.

An additional set of build parameters can be used to override parameters defined in LIS configuration files. The parameters and allowed values is tabulated below.

- The list of products on which integration tests are to be performed in the order in which they are to be run. (Unit tests are performed as part of the build process on all repositories that are built by the system.)
- The command or commands to use to check-out, build and install the software to be tested.
- Whether to log results to the DB
- ***** Anything else?? *****

A certain number of parameters defined in test configuration files can also be overridden using build/trigger parameters. See section [Appendix A](#) for details of those parameters.

2.1.1. Trigger identifier tag

From each trigger, the system will construct an identifier to be associated with the triggered workflow that can be used to simplify the task of associating particular results with specific trigger events. The identifier will include sufficient information extracted from trigger parameters to make this possible. For example, the type of trigger (push to develop or manual), the code revision, the test suite, and a time stamp are possible values to include in the tag name.

***** update this when the tag format is defined *****

2.2. The Jenkins CI workflow

We have attempted to minimize the detail of the CI system that is exposed directly to Jenkins. To that end, the Jenkins CI workflow contains only a single step that calls a basic CI workflow script that is responsible for driving the bulk of the CI process.

3. Build-step / workflow scripts

There are three build-step / workflow scripts in the CI system:

- A “build” script that is responsible for pulling and building the appropriate code (which includes running the pre-installation unit tests via `mr b test`), then (locally) installing the result.
- A “test-runner” script that manages all the stand-alone post-installation tests.
- A single “CI workflow” script that drives the process given the specification for the software to be tested and the test suite to execute.

3.1. The Basic Workflow

Each repository uses the same basic workflow shown in Fig. 1 to obtain the correct code revision and run tests. The steps and commands used to check-out, build, unit test and install the specified code revision for a particular product is configurable. The build workflow specification is defined in a configuration file that is read at run-time. Non-default workflows can be selected via trigger parameters.

The two branches in Fig. 1 depict a case where two distinct build workflows have been defined. The one on the left is provided by default. The system, however, makes no logical distinction between configured workflows.

3.1.1. Workflow assumptions

Trusted software sources

The configuration is agnostic with respect to the repository from which code is checked out, except that it be from a trusted source. Similarly, installation of source or binaries from tarballs is allowed, provided that the source is trusted. All trusted sources must be registered in the global configuration file.

Uniform code revision specification

Any set of commands used to implement the build workflow must respond appropriately to the code revision specification passed into the build script.

Build system independence

The system can use any available build system.

cvmfs availability

The system assumes that the LArSoft cvmfs instance hosted by the OSG is mounted and available. This is considered a trusted source for code and tarballs.

Working area directory structure

It is assumed that all activities take place in a working area structured like one provided by mrb. In particular, the top-level working area directory will have the following sub-directories:

- srcs, where all source code should be located
- build_*, where all build activities take place
- localProducts_*, where all executables and libraries are installed

There will be additional sub-directories created during the testing process.

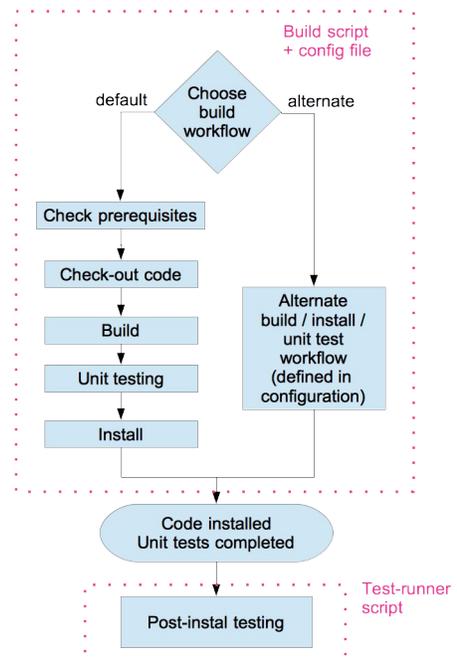


Figure 1: The basic workflow and the larci scripts used to implement it. The commands in the workflow are specified in a configuration file.

Set-up command

A setup command must be provided in the configuration that will provide all the user set-ups required to run tests against the locally installed software.

Configuration files

A global run-time configuration file is located in the `larci` repository. Future versions may allow alternate locations for this file.

Local run-time configuration files are located ***** under the user account used by the system ***??**

3.2. The build script and the default build workflow

These are the steps that occur within the build script in the general case, with the actions for the default build workflow provided as an example. They are implemented via a combination of configuration and build-step scripts. At build-script start-up, the current directory is assumed to be the top-level directory of the work area.

The build script directs standard output and standard error from any build workflow to a log file.

1. Check pre-conditions.

The intent of this step is to ensure that any assumptions made by subsequent steps are met prior to invoking them. In principle, attempting to fix any problem detected is acceptable at this phase. Alternatively, the build can be aborted at this point.

- a) Check that required resources are available.
 - i. Fix or abort
- b) Check that required externals are available.
 - i. If not, pull externals.

2. Check-out specified code.

This step must take command arguments that specify the repository, product and code revision (either develop head (default), branch head, tag name, commit hash), which are taken from the Jenkins trigger parameters, and perform the steps required to check-out the specified code.

- a) The default build workflow assumes that the code is in Fermilab Redmine git repositories, and uses `mrbc gitCheckout` to retrieve the code.

3. Invoke the build system to build the code

Creates libraries and binaries, but does not install or perform unit tests.

- a) The default build workflow uses `mrbc build` for this step.

4. Run unit tests

These tests are run out of the build area prior to installation.

- a) The default build workflow uses `mrbs test` for this step.
5. Install the code into the local work area.

Any ups products should go into the `localProducts` sub-directory.

- a) The default workflow uses `mrbs install` for this step.
6. Perform post-installation steps.

3.3. The test-runner script and post-installation tests

The test-runner script operates by reading a test configuration file located in the top-level `test` sub-directory of each product. This configuration file specifies test names, test commands and arguments, files to be pulled, dependencies on other tests, standard test utilities to apply, and so on for each test, and a list of test suites and the tests to run in each. More details on the format of the test configuration file can be found in [Appendix A](#). It is assumed that code developers will modify this configuration file as needed in order to run the desired tests at the desired times. The conventions used for test suite definitions will be discussed later.

The following steps are performed by the test-runner script.

1. Loop through products in an order taken from the active configuration.
2. Read the test configuration from the top-level `test` sub-directory for each product.
3. Identify the specified test suite and the tests within that suite
 - a) Evaluate the declared dependencies, and identify the order in which the tests must run, and those that can be run in parallel (ie, create the equivalent of a directed acyclic graphical representation of the dependencies).
4. Pull all unique files requested to a common area.
5. Perform required setup procedure appropriate to the version of the products being tested.
6. Execute the tests within the suite in the appropriate order, allowing for parallel execution where possible.
 - a) The current directory at test start-up is a sub-directory of the top-level working area with a name that identifies the test
 - b) The test-runner directs standard out and standard error for each test to a log file in the test sub-directory. The name of this file is independent of the test being run.
 - c) A non-zero return status by any test script will cause the test to fail. Tests that depend on a failed test will not be run. The system will interpret the absence of a log file for a test as an indication that the test was skipped.
 - d) After test completion, any configured test utilities are run.
 - i. A non-zero return status from any of these utilities will cause the test to fail.
 - ii. Unless the test is configured otherwise, a failure of one of these utilities will not prevent execution of tests that depend upon the failed test. ***** ie, do we need a stop_on_failure flag for utility functions managed by the framework *****

- iii. All configured utilities will be run, regardless of the return status or configuration of any other utility.
 - iv. The return status of each utility will be written to the test log file.
7. Perform post-test functions. These may include:
- a) Report consolidation activities
 - b) Log file transport back to Jenkins server

4. Utilities and functions

The `larci` software includes a set of utilities that can be used to augment tests, as well as a number of functions to assist CI system administrators and end-users / developers. The utilities are focused on providing features that are likely to be useful in many tests. Providing this optional functionality as part of the testing infrastructure relieves developers of the need to code such commonly used features into each test, thereby increasing the probability that such features will be included, and simplifying test interpretation by contributing to more uniform test reports.

Most of the utilities are designed to function within the test-runner environment and extract information about tests from the system or test output. This information can then be displayed as part of a test result, or used in a decision that contributes to the pass-fail status of the test. (A test failure from these utilities need not prevent processing of any other tests in the suite.) Note that for a given test, all utilities will be run, regardless of the success or failure of any individual utility.

The functions are primarily focused at performing commonly used actions required to utilize and configure the system. They are not typically used within the test environment itself.

4.1. CPU time limit checks

This utility measures the CPU time consumed by a test script and compares it to a set of limits defined in the test configuration. A test that exceeds the allowed CPU range will cause the test to fail, but will not interrupt execution of the test, or affect the running of that depend on the test, unless specifically requested in the test configuration.

These checks are not to be confused with the per test-suite timeout. Exceeding the timeout will stop all test processing at that point.

4.2. Peak memory limit checks

This utility measures the memory usage of a test script and compares it to a limit defined in the test configuration. The memory limits are evaluated after the test exits. A test that exceeds the allowed memory limit will cause the test to fail, but will not interrupt execution of the test, or affect the running of subsequent tests.

4.3. art log file scanning

Since most integration tests are likely to involve art, we provide a utility for scanning art log files. The scan will perform a number of functions:

1. Search for lines expected in any successful art job. Failure to find any expected line will cause the test to fail.
2. Search for lines that suggest a severe error occurred (e.g., contain the words “abort”, and

“terminate”). Finding any error lines will cause the test to fail.

3. Extract CPU information printed by the art TimeService, and collate into a report (or use the TimeService report).
4. Extract memory usage as a function of time, and collate into a report.
5. Write a separate log file with a condensed summary of results., and print a highly condensed summary to standard output.
6. Exit with the appropriate pass/fail exit status

Other scanning utilities can be used by modifying the build workflow configuration, or via trigger parameters.

4.4. art CPU and memory data logging

As noted elsewhere, some data will be written to a database so that the information can be collated into reports of various sorts, particularly those that benefit from comparing historical values, such as CPU times for a given test or module.

When enabled in the configuration or via trigger parameters, this utility will gather information made available via other scanning utilities and write the data into the results database.

4.5. Output file sanity checks

In some cases, the success or failure of a test can be deduced from very simple checks on an output file. When enabled in the configuration, this utility performs the following checks on the specified output files:

1. Check that the file exists. A missing file will cause the test to fail.
2. Check the size of the file against minimum and maximum size limits. A file that exceeds the limits will cause the test to fail.
3. Check that the file is readable by the intended target program. Root should be able to open all *.root files, for instance.

Wildcarding is allowed in the filename specification.

A test that fails will prevent processing subsequent dependent tests.

5. Test result database

The utility of some tests is enhanced by comparing a result to historical results. The test result database will collect selected data values from appropriately configured tests. Direct uploads to the database may also be possible via a web interface.

Writing data to the database requires that the appropriate utility function be configured for the test and that the writing to the database is enabled via the appropriate trigger parameter.

The list of data collected includes the following:

1. The trigger identifier tag
2. The date and time at which the test was run
3. The software product and version for which the trigger was issued

4. The software product and version for all products built and tested as a result of the trigger. For LArSoft, this will include information about each constituent product.
5. The `larci` version being used
6. The type of machine on which the test was run
7. The CPU scale factor for this machine
8. The test suite name
9. All test names
10. The pass / fail / did not run status of each test
11. In the case of failure, the type of failure
12. The CPU and wall clock time required to run the test script
13. For art jobs, the CPU time spent in each module
14. For art jobs, The peak memory consumption for the job
15. For art jobs, the memory consumption as a function of time during execution of the test
16. Any other information that may be needed to describe the tests performed and their results.

6. Web-based test result reporting and display

Jenkins provides a native web-based test result reporting system that provides simple monitoring and display of Jenkins projects. Test results themselves are formatted and displayed via web pages constructed by utilities that run on the build slave after testing is completed. Additional utilities that run on a web server that is independent of Jenkins further organize results and provide access to historical data for comparing results across time. All utilities live in the `larci` repository.

This part of the system is still under design. It will involve parts that are completely independent of Jenkins. Some parts of this system, however, may access test results or test result logs stored on the Jenkins server.

7. End-user test requirements and configuration

The basic requirements for end-user tests is that:

1. They be executable scripts using a commonly available scripting language or program that can be invoked after the appropriate end-user environment setup commands,
2. That the exit status be zero in the case the test succeeds, and non-zero in all other cases.
3. All scripts should live under the top-level `test` sub-directory in the product source code. The directory structure under the top-level `test` sub-directory is arbitrary.
4. Each test must either have a definition in the local test configuration file for that product and belong to at least one test suite, or have an entry in the appropriate `CMakeLists.txt` file under the top-level `test` sub-directory.

7.1. Test script arguments

Test scripts may take arguments. The values of the arguments must be fully specified in the test configuration file. Argument substitution from trigger parameters is also possible.

7.2. Input files

The system provides a mechanism for automatically pulling input files from specified sources to the job. File are transferred via tools in the ifdh client library, which provides grid compliance and site independence. Input files are staged to a common data directory for the test suite, then linked to standard input file directory under the test-specific sub-directory. The procedure ensures that there are no redundant file copies.

7.3. Output files

Output files are placed in a dedicated output file directory under the test-specific sub-directory. No files are moved by the CI system. Any files copied from the build slave node must be done within the test script or copied afterward.

7.3.1. Log files

The test-runner opens log files for each major step in the build workflow, and for each post-installation test. The former are placed in the top-level test sub-directory on the slave node, while the latter are in the relevant test-specific sub-directories. All log files are copied to the CI server where they are accessible via the Jenkins web interface. The CI system display has links to these files.

Appendix A: The test configuration file and test examples

The test configuration is defined by an INI formatted file named `ci_tests.cfg` located in the top-level `test` sub-directory of the product or repository being tested. The file consists of a set of test and test suite definition blocks comprised of either a test or suite identifier followed by a series of parameter-value pairs that describe the configurable characteristics of the test or suite.

The test block has the following structure:

- a line with “[test <test_name>]” where <test_name> is an alias for the test being configured.
 - The name consists must contain only alpha-numeric characters, as well as dashes “-” and underscores “_”.
 - No white-space characters are allowed.
 - The name of must be unique within the configuration file.
- a set of lines with “<parameter> = <value>”, where <parameter> is the parameter name, and <value> is a string literal representing the value of the parameter.

The allowed parameters and value types are listed in Table A.1:

Parameter	Value	Description
script		Path to an executable file that will run the test.
args		The arguments to be passed to the test script
requires	space separated list	The list of test names that provide direct input to this test. The names must be among the list of other tests in the current suite.
cpu_score	hh:mm	Expected normalized CPU time required to run the test.
min_cpu	hh:mm	Minimum allowable normalized CPU time
max_cpu	hh:mm	Maximum allowable normalized CPU time
mem_score	MB	Expected peak memory size
max_mem	MB	Maximum allowable peak memory size
scan_log	<file> [<type>]	Request a standard scan of type <type> for log file <file> . The default type is “art”
infile	<file>	URI to required input file <file>. For inputs from previous tests, specify the relative path to the file from the current test directory, so “../<reqd test name>/output_files/<filename>”.
outfile	<file> [<min size> <max size> [[<app>]]	Declaration that output file <file> exist when the test is completed. Optionally, require that the file be at least <min size> MB, but less than <max size> MB in size. Optionally specify that the file should be readable by the application <app>, where <app> is either “root” or “art”

Table A.1. Allowed test configuration parameters.

The suite block has the following structure:

- a line with “[suite <suite_name>]” where <suite_name> is an alias for suite
 - The name consists must contain only alpha-numeric characters, as well as dashes “-” and underscores “_”.
 - No white-space characters are allowed.
 - The name of must be unique within the configuration file.
- a set of lines with “<parameter> = <value>”, where <parameter> is the parameter name, and <value> is a string literal representing the value of the parameter.
- a set of test definition blocks, where each of the specified tests become part of the suite.

The set of allowed suite parameters is shown in Table A.2:

Parameter	Value	Description
testlist	<t1> [<t2> [<t3>...]]	The list of test alias names to be included in the test.
A test definition block		Any test definition that occurs within a suite block will be included in the suite.

A trigger event specifies a single test suite to run. Test suites may generate other triggers, thereby creating a chain of dependent suites that are executed in response to a single initiating trigger. The order of tests within a particular suite is determined entirely within by the declared dependencies between tests. The system assumes that all tests may be run in parallel provided that all required tests have completed. The CPU and memory scores may also be used in test scheduling decisions. Actions that should occur after all tests are completed should be included in the post-test step of the test workflow.

The configuration file is read and parsed by the test-runner script immediate prior to test execution.

The following is an example configuration file.

```
# Example configuration file. Any line starting with "#" is
# interpreted as a comment

# Test definitions
#####
[test lar_ci_echo]
  script = /bin/echo
  args = this is a test

[test lar_ci_echo2]
  script = /bin/echo
  args = this is a another test

# This test will execute only if lar_ci_echo and
# lar_ci_echo2 succeeds
#
[test lar_ci_depends]
  script = /bin/echo
  args = this depends on other tests
  requires = lar_ci_echo lar_ci_echo2
```

#This test must have two files in the output area when completed.

#

[test lar_ci_prodsingle]

script = \$LAR_CI_DIR/test/prodsingletest.sh

args =

outfile= single_gen.root

outfile = single_hist.root