# Continuous Integration for Concurrent Computational Framework and Application Development

Derek R. Gaston,* John W. Peterson, Cody J. Permann, David Andrs, Andrew E. Slaughter, and Jason M. Miller
*Fuel Modeling and Simulation, Idaho National Laboratory, P.O. Box 1625, Idaho Falls, ID 83415*

September 6, 2013

**Abstract**

Development of scientific software relies on specialized knowledge from a broad range of diverse disciplines including computer science, mathematics, engineering, and the natural sciences. Since it is rare for a given practitioner to simultaneously be an expert in each of the aforementioned fields, teamwork and collaboration are becoming the norm for scientific software development. This short paper discusses specific software development conventions that have lead to the success of the MOOSE multiphysics framework at Idaho National Laboratory (INL).

## 1. Introduction

Continuous integration [1] is a software development practice that espouses frequent re-integration of developer changes into the main development trunk. This idea is essential to rapid development of scientific software that is closely tied to an underlying framework [2]. As both the framework and applications are developed, the changes to each need to be tested against each other to ensure compatibility. Longer periods between integration are detrimental, and incompatibilities with complex interdpendencies are more likely to arise the longer these periods become. The MOOSE [3] project utilizes direct continuous integration between the framework and all applications; this practice has precipitated the rapid development of high quality software.

More than thirty different MOOSE-based applications are currently under development at various national laboratories and universities, all of which are following the continuous integration approach. Both the software development methodologies discussed here, and the streamlined object-oriented interface of MOOSE itself have contributed to the success of the project and the ease with which MOOSE-based applications can be developed. Example applications include: BISON [4] (nuclear fuel modeling), MARMOT [5] (microstructural evolution of nuclear fuel), RAT [6] (chemical reactive transport in porous media), MAMBA (microstructural effects of deposition on nuclear fuel rods), and HYRAX (ZrH precipitation in nuclear fuel cladding).

*Corresponding author, derek.gaston@inl.gov. 

## 2. Shared Repository

A source code control repository is essential to the success of any collaborative software development effort. Although there are many source code control packages available, only a few are appropriate for the "shared" (by framework and application developers) style of repository used by the MOOSE project. Subversion [7] provides a compelling platform for source code control in this scenario: it is stable, robust, and provides a simplified workflow suitable for use by non-experts.

While Subversion is utilized for the repository housing both MOOSE and MOOSE-based applications, "power users" working with MOOSE need access to some of the more modern features of distributed version control systems. Because of the excellent Subversion integration provided by "git," a fundamentally different revision control software package [8], these power users can maintain a "git svn clone" of the MOOSE repository. This allows them to take full advantage of git's ability to make local commits and rapidly switch between different branches of development as the task at hand requires.

A unique aspect of the shared MOOSE software development strategy, one that particularly differentiates it from similar projects, is the manner in which both applications and the framework coexist within the same source code control repository. This adds the ability to commit "across" both the framework and applications simultaneously, which aids in rapid development. Although this configuration arose organically from the early days of the project (when nearly all users were colocated at INL) it has persisted even now that development has become more geographically dispersed.

A major benefit of this arrangement is the flexibility it provides developers in the context of making application programming interface (API)-altering changes. Most other scientific software frameworks go to great lengths to preserve (or gracefully deprecate) APIs because they cannot possibly know the extent to which other developers depend on those APIs, and frequently changing APIs can, in severe cases, lead to a mass exodus of users. API preservation of this type naturally leads to more complicated code bases with more branch statements and compile-time directives, and exacerbates the problem of backwards compatibility support.

In the MOOSE ecosystem, making API changes imposes an up-front cost on the primary developer: it is his or her responsibility to make certain all the affected applications are simultaneously updated to use the new

API, and hence pass all the tests, before committing the change. The upside of this approach is that the development team is no longer faced with the backwards compatibility support issue, which can persist for long periods of time, even up to the lifetime of the project. Obviously, such an approach can have some logistical difficulties (user accounts, permissions), issues "scaling" to large numbers of dependent applications, and is inconvenient when new applications require large numbers of specialized support libraries. Nevertheless, it has served the MOOSE development team effectively for several years, and there are no plans to alter it in the near future.

Another unique aspect of the MOOSE project's shared source code control approach is the simultaneous existence of "development" and "stable" areas within the same repository. Note that "areas" here means separate directories within the same repository and not, for example, separate development branches within a single revision control system. In practice, anyone who clones the MOOSE repository gets two copies of MOOSE and, at their discretion, can build applications against either the stable or development version.

Building applications against the stable version of MOOSE allows a user to be more insulated from hourly changes that have not yet passed the full regression test suite (see Section 4) on all supported compiler/architecture combinations. On the other hand, building applications against the development version of MOOSE means always using the most recently committed MOOSE revision, which may be temporarily (typically only a few minutes to an hour) slightly ahead of the stable version.

Since there is usually a minimal delay between the time when the development version is updated and the time when it merges into the stable version, there is typically not a great deal of practical difference in simply using one version or the other. The most important difference is that no human developers are allowed to directly commit anything in the stable area—only a special user, controlled by the continuous integration system, is allowed to commit there, and this only occurs after the tests have successfully passed. The complete development cycle is depicted in Fig. 1. The dual devel/stable areas in the repository also have another important, partially psychological, effect: the low barrier to entry required to begin "hacking" on MOOSE (just change directories, no need to download something new or "check out" a different branch) can encourage new developers to take partial ownership in the joint framework development process.

## 3. Cascading Build System

The build commands used by developers and users rely on a sophisticated and hierarchical framework based on GNU Make. Executing the "make" command from any point in the repository (i.e., within an application) will automatically cause all of its required components to be compiled and linked. Additionally, nested dependencies

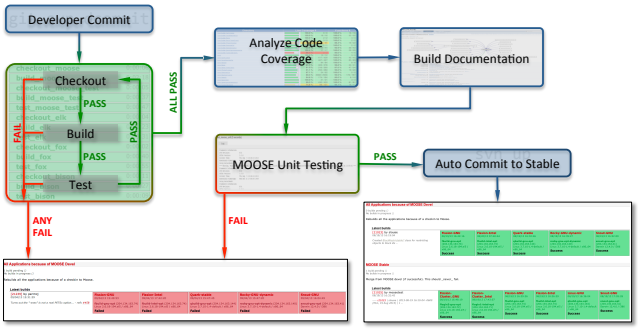


Figure 1: Flowchart depicting the MOOSE project's automatic build and testing system. After a developer commits a change, MOOSE and all the applications are built and tested. If any test fails, the system exits and reports failure. Otherwise, once all tests have succeeded, "stable" MOOSE is automatically updated, making it available to users.

are automatically resolved for applications that depend on each other. For example, as shown in Fig. 2, the Mammoth application relies on four sub applications (which have their own, additional dependencies) and everything depends on MOOSE. The Mammoth application need only prescribe the top-level of dependencies; all subsequent applications are compiled and linked automatically.

To ease development, reverse building is also automatically handled. Executing the command "make up" on any application or library will build the application itself and all applications that depend upon it. Referring to Fig. 2, executing the "make up" command in the FOX application will build FOX as well as BISON and Mammoth. This ability is further expanded to include running the tests for each application: the command "make test_up" builds the applications and executes each application's test suites (see Section 4).

This automatically cascading build system has a significant effect on the way MOOSE-based applications are developed. The simplified nature of making one application depend on other MOOSE-based applications and libraries removes psychological barriers associated with reusing the work of others. This capability, together with MOOSE's highly object-oriented and extensible architecture, leads to an extensive amount of code reuse among the applications, significantly decreasing the amount of time necessary to create a new application.

## 4. Automated Testing

With hundreds of developers working on dozens of applications, and a dedicated team modifying the framework daily, the software in the repository is in a constant state of flux. A comprehensive testing system prevents the framework and applications from getting out of sync due to bugs and software incompatibilities. The testing system or "test harness" is a custom Python application that provides an object-oriented, plugin-based architecture, similar to that of MOOSE itself, for designing test

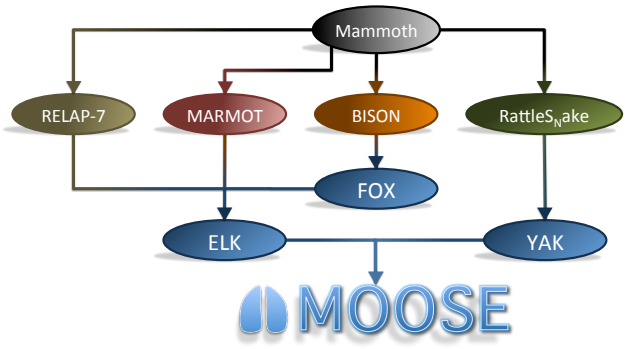


Figure 2: Flowchart of automated dependency build system for MOOSE and applications.

types. Individual tests are defined using an input file syntax, similar to MOOSE-based applications, that specifies what the test should do, the inputs, and the postconditions for determining test success or failure. Each application or library within the MOOSE system thus defines its own set of tests, ensuring reproducibility of trusted results even as potentially large amounts of code are changed in the repository. The test suite of each MOOSE-based application is run automatically each time a commit is made to the framework or to another application upon which it depends.

Three main categories of tests are supported: regression tests, "expected error" tests and unit tests. Regression tests are, by far, the most common form of test used in the MOOSE project. A regression test specifies both a simulation to perform, and a verified correct ("gold") solution to compare against. The gold solution might be a field variable or a post-processed quantity such as the total heat flux through a boundary. All future executions of the test will compare their output with the gold solution, and deviations (to within some test-defined tolerance) will be reported as failures. Regression tests, while not a silver bullet, are essential for ensuring reproducibility of results while the framework and applications are under constant development.

Many errors can arise in scientific simulations: input parameters can be out of bounds, material models may be evaluated outside their region of validity, solvers can fail to converge, the system can run out of memory, there may be erratic filesystem behavior, etc. Because of this, scientific applications are riddled with error checking routines. Typically, these error conditions go untested (the code runs successfully) and therefore the error checks themselves are prone to failure, i.e. they may no longer faithfully report the error they were meant to. Within the MOOSE testing system, this is handled with so-called "expect error" tests. Simulations designed to produce specific errors are executed, and the testing system verifies that the correct error message is reported. If the code exits successfully or terminates for any reason other than the "expected" reason, it is considered to have failed.

Unit tests focus on a single C++ class, and verify that specific aspects of the API for said class perform their functions properly. MOOSE unit tests are implemented within the CPPUnit [9] testing framework, which automates much of the process of setting up, testing, and "tearing down" the class being unit tested. Comprehensive unit testing in finite element software is difficult to achieve due to the many interrelated pieces of data that comprise each calculation. Due to this limitation, only sufficiently simple classes which can be effectively separated from most of their external dependencies are currently unit tested within MOOSE.

The test harness contains several other features that further encourage test-driven development. During development, it is possible to quickly run specific subsets of tests which match a regular expression provided by the user. Larger tests can even be launched and monitored on a PBS-managed cluster; the results are automatically gathered and summarized by the testing system. The testing system can also be invoked in parallel (by specifying number of MPI processes or threads), with specific command line options, or through external memory checking tools such as valgrind [10]. Finally, testing is integrated with the build system through the "make test_up" command, allowing developers to check the build and tests for all dependent applications with a single command.

Obviously, testing is only beneficial when the tests themselves are well-designed, comprehensive, reliable, and are consistently run when any new code is committed to the repository. While it is the practice of some software development projects to disallow all commits which fail to pass the test suite, this type of policy is often needlessly obstructionist and detrimental in today's environment of rapid development cycles. As mentioned previously, the MOOSE repository is set up with separate "devel" and "stable" directories, the latter being updated automatically when all tests pass in the former (see Fig. 1). The Trac site (described in Section 5) is integral to allowing developers to monitor the progress of their commits to the development area, diagnose the causes of failed tests, and make new commits to address the issue. During this process, application developers can continue to use (and update) their stable copy of MOOSE without interruption.

## 5. Documentation and Wiki

To facilitate the collaborative development process, the MOOSE project utilizes a community driven "wiki" website powered by Trac [11]. The primary function of the Trac site is to catalog almost any type of information relevant to the framework and applications. Developers are free to edit or add to parts of the wiki they have permission to access, these permissions are based on the applications they have source code control access to. Some examples of information that can be found on the wiki

are:

- Setup and installation instructions
- Descriptions of each MOOSE-based code housed in the repository
- Information about tools that can be used while developing MOOSE-based applications (debugging, revision control, text editors, etc.)
- Partial differential equations describing the physics solved by an application
- Links to automatically generated documentation and code coverage statistics

In addition to these community-developed and curated sections, the Trac website also provides several other critical functions for the project. The "ticketing" system provided by Trac is integral to the MOOSE development process. Each issue (an issue may be either a defect or task) requiring the attention of a developer has a "ticket" (submitted through the Trac site) associated with it. Every change to MOOSE is required to reference one or more ticket numbers; the revision control numbers associated with the changes are also automatically cross-linked back to the relevant ticket, thereby providing a "paper trail" of the work performed on a particular issue.

The Trac website also contains links to documentation about MOOSE and MOOSE-based applications. This documentation includes automatically generated Doxygen [12] API documentation, input file syntax, test code coverage, test timing, code standards adherence, and the MOOSE training manual. Each of these categories of documentation is automatically regenerated each time code is committed to the repository, providing an up to date resource for developers and MOOSE-based application users.

Finally, as previously mentioned, the Trac website is also where the status of the testing system is reported. The "Build Status" page displays green (pass), yellow (in-progress), or red (fail) boxes for each set of tests on each platform. This provides a graphical representation of the combined status of the overall project, giving developers the capability to make an initial diagnosis of failures with a single glance. Test failures also generate an email message with a detailed description of the problem, and send it to the developer who made the offending commit. A complete history of every change made to the framework and applications, along with the corresponding pass/fail status of each test, is maintained by Trac, and is instrumental in quickly tracking down the root cause of an issue.

## 6. Closing Remarks

The MOOSE project incorporates many software engineering techniques, such as shared source code control repositories, continuous integration, issue tracking, and automated/community-driven documentation, that are considered essential by many in today's fast-paced world of scientific software development. A number of practices unique to the MOOSE project, in particular the combined stable/development directories and their tight integration with the testing system, as well as the shared nature of the source code control repository between framework and applications, were discussed in detail. The scientific software community is a vibrant, fast-growing collection of extremely talented individuals. One of its greatest strengths has always been in the rapid dissemination and assimilation of useful information. The authors hope the unique aspects of the MOOSE project discussed in this paper will be found in that category.

## References


[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[2] R. A. Bartlett, "Integration strategies for computational science & engineering software," in *Software Engineering for Computational Science and Engineering, 2009. SECSE'09. ICSE Workshop on*, pp. 35–42, IEEE, 2009.

[3] D. Gaston, C. Newman, G. Hansen, and D. Lebrun-Grandié, "MOOSE: A parallel computational framework for coupled systems of nonlinear equations," *Nucl. Eng. Design*, vol. 239, pp. 1768–1778, 2009.

[4] R. L. Williamson, J. D. Hales, S. R. Novascone, M. R. Tonks, D. R. Gaston, C. J. Permann, D. Andrs, and R. C. Martineau, "Multidimensional multiphysics simulation of nuclear fuel behavior," *J. Nucl. Mater.*, vol. 423, pp. 149–163, 2012.

[5] M. Tonks, D. Gaston, P. Millett, D. Andrs, and P. Talbot, "An object-oriented finite element framework for multiphysics phase field simulations," *Comp. Mat. Sci.*, vol. 51, no. 1, pp. 20–29, 2012.

[6] L. Guo, H. Huang, D. Gaston, C. Permann, D. Andrs, G. Redden, C. Lu, D. Fox, and Y. Fujita, "A parallel fully coupled fully implicit solution to reactive transport in porous media using preconditioned Jacobian-Free Newton-Krylov method," *Advances in Water Resources*, vol. 53, pp. 101–108, Mar. 2013.

[7] B. Collins-Sussman, "The subversion project: buiding a better CVS," *Linux Journal*, vol. 2002, no. 94, p. 3, 2002.

[8] `http://en.wikipedia.org/wiki/Git_(software)`.

[9] `http://sourceforge.net/projects/cppunit`.

[10] `http://valgrind.org`.

[11] `http://trac.edgewall.org`.

[12] `http://www.doxygen.org`.