Journal of
open research software

Slaughter, A E et al 2015 Continuous Integration for Concurrent MOOSE Framework and Application Development on GitHub. *Journal of Open Research Software,* 3: e14, DOI: http://dx.doi.org/10.5334/jors.bx

ISSUES IN RESEARCH SOFTWARE

Continuous Integration for Concurrent MOOSE Framework and Application Development on GitHub

Andrew E. Slaughter[1], John W. Peterson[1], Derek R. Gaston[1], Cody J. Permann[1], David Andrš[1], and Jason M. Miller[1]

[1] Modeling and Simulation Department, Idaho National Laboratory, P.O. Box 1625, Idaho Falls, ID 83415
andrew.slaughter@inl.gov, jw.peterson@inl.gov, derek.gaston@inl.gov, cody.permann@inl.gov, david.andrs@inl.gov, jason.miller@inl.gov

Corresponding author: Andrew E. Slaughter

For the past several years, Idaho National Laboratory's MOOSE framework team has employed modern software engineering techniques (continuous integration, joint application/framework source code repositories, automated regression testing, etc.) in developing closed-source multiphysics simulation software (Gaston et al., *Journal of Open Research Software* vol. 2, article e10, 2014). In March 2014, the MOOSE framework was released under an open source license on GitHub, significantly expanding and diversifying the pool of current active and potential future contributors on the project. Despite this recent growth, the same philosophy of concurrent framework and application development continues to guide the project's development roadmap. Several specific practices, including techniques for managing multiple repositories, conducting automated regression testing, and implementing a cascading build process are discussed in this short paper. Special attention is given to describing the manner in which these practices naturally synergize with the GitHub API and GitHub-specific features such as issue tracking, Pull Requests, and project forks.

Keywords: continuous integration; github; multiphysics

(1) Introduction

The MOOSE (Multiphysics Object Oriented Simulation Environment) framework is an open-source computational platform for developing scientific applications. MOOSE development relies on direct continuous integration [1, 2] between the framework and all derived applications. As described in [3], this integration was originally implemented using a shared repository strategy. Since that publication, MOOSE has been released under the GNU LGPL 2.1 license and is currently available via GitHub [4].

To support individual application development, and because many of the MOOSE-based applications contain export-controlled content, the single repository design is no longer possible. This paper is a follow-on to our previous work [3], and details the integration of the open-source version of MOOSE in both public and private repositories, while still maintaining continuous integration and software engineering best practices. We also discuss the impact that open sourcing has had on user engagement levels with the MOOSE development process.

Originally, the MOOSE framework and its applications were developed concurrently within a single SVN repository hosted at the Idaho National Laboratory (INL). Now that the framework has been moved to GitHub, the continuous integration process requires that MOOSE-based applications hosted in GitHub repositories (both public and private) and the INL-hosted Git repositories (based on the GitLab platform [5])—which naturally have different owners, permissions, and levels of accessibility—interoperate seamlessly with the MOOSE framework repository on GitHub. While this paper focuses specifically on the interaction between GitHub-based repositories, related strategies have also been implemented to facilitate GitHub-GitLab interactions for all of the techniques discussed in Section 2.

Separating the framework and applications into different repositories eliminates some of the benefits of the shared repository model detailed in [3]. In particular, the ability to commit "across" the framework and applications simultaneously is lost. Developers now face the additional burden of ensuring that changes to the code which affect both MOOSE and its dependent applications are synchronized. Section 2 describes, in detail, how repository forks, Git submodules, Pull Requests, and automated testing are combined in order to alleviate this burden.

(2) Development Strategy

As discussed in [3], a key benefit of the shared repository model was that the framework developers' need to maintain backwards compatibility with previous APIs was

largely absent—the developers could update the applications *along with* changes to the framework in a single, atomic commit. This basic approach continues in the new multi-repository model, but of course it is no longer possible to achieve with a single commit into a single repository. Instead, the following technologies are critical for making this approach viable in the multi-repository model:

1. GitHub forks.
2. The cascading build system.
3. Git submodules and automated regression testing.

The roles of each of these technologies are described in the following subsections.

2.1 Forking Stork

MOOSE-based applications are created by Stork [6] through a repository "fork," a standard practice in GitHub-based development. The fork provides a simple method for tracking MOOSE-based applications, and allows MOOSE developers to pull code, build and test the applications, and submit Pull Requests to the fork owners, providing them with the necessary updates when changes are made to the MOOSE API. Note that submission of a Pull Request does not require write access for the submitter: it merely makes code available which an application owner may then choose to incorporate into the project, or reject.

Fig. 1 shows this development strategy in action: Pull Requests are issued to MOOSE-based applications which were originally created by forking Stork. As changes are made to the MOOSE framework, these applications are monitored in order to determine if the changes have altered their behavior. The monitoring is implemented through the cascading build and automated testing systems, as discussed in the following sections. The techniques used to monitor the various MOOSE-based applications are themselves undergoing rapid development, and will be discussed in greater detail in subsequent publications.

The process of *manually* submitting Pull Requests to all Stork forks is clearly not sustainable: at the time of this writing, 125 forks are in existence, and even when the required patch is relatively small, it may be non-trivial to automate the process in an application-independent manner. One exception is the case where the required changes can be applied using a script, for example when the MOOSE input file syntax changes in a relatively simple way. In this case, the GitHub web API [7] can be used to automatically create a Pull Request for each dependent application, an example of which is shown in **Fig. 2**.

Considering the limitations involved in manually updating applications, a related approach that uses Git submodules [8] has been successfully employed. In this alternate approach, the MOOSE-based application

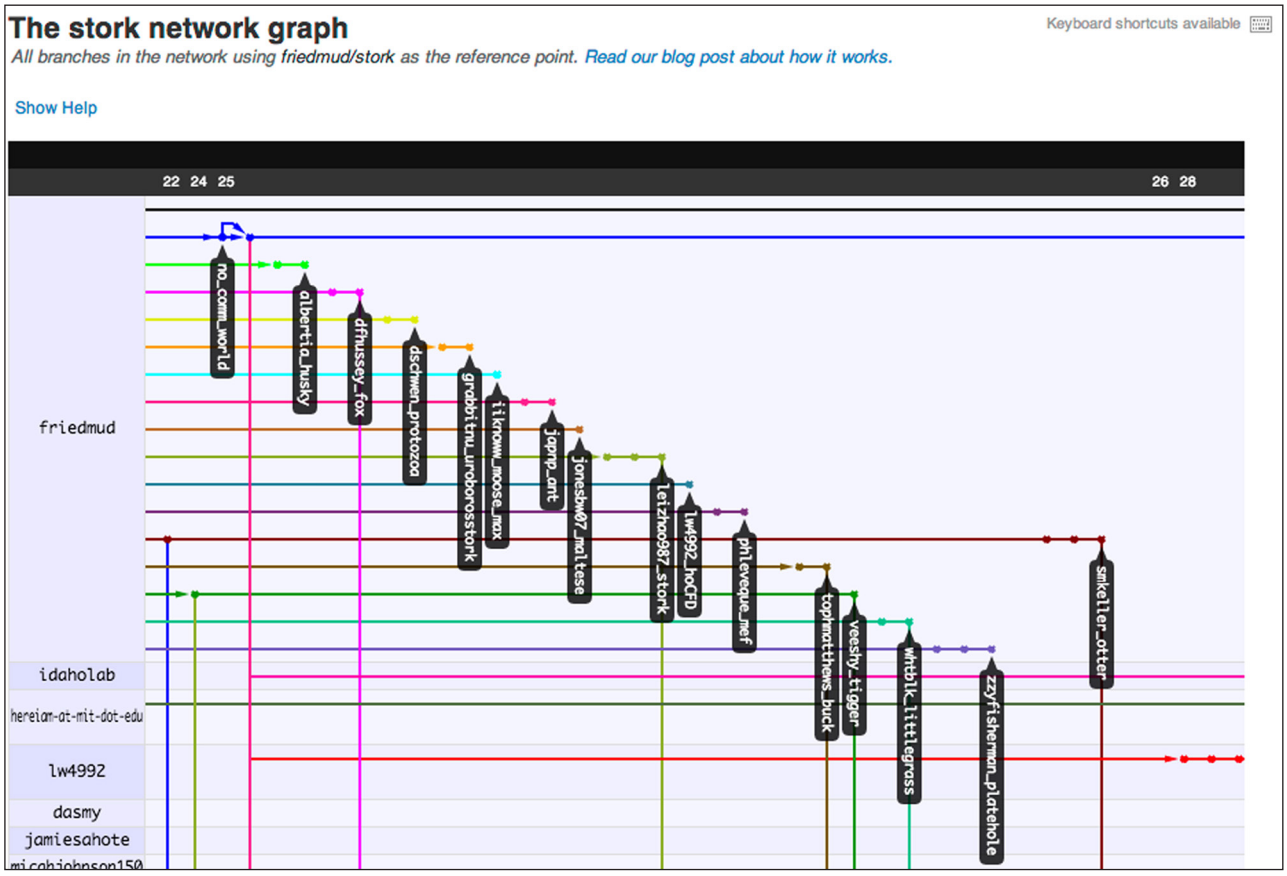

Figure 1: Screenshot of GitHub network graph for MOOSE developer (friedmud; Derek Gaston) fork of Stork that contains branches for each of the derivative applications that required an update due to a change in the MOOSE framework (`https://github.com/friedmud/stork/network`).

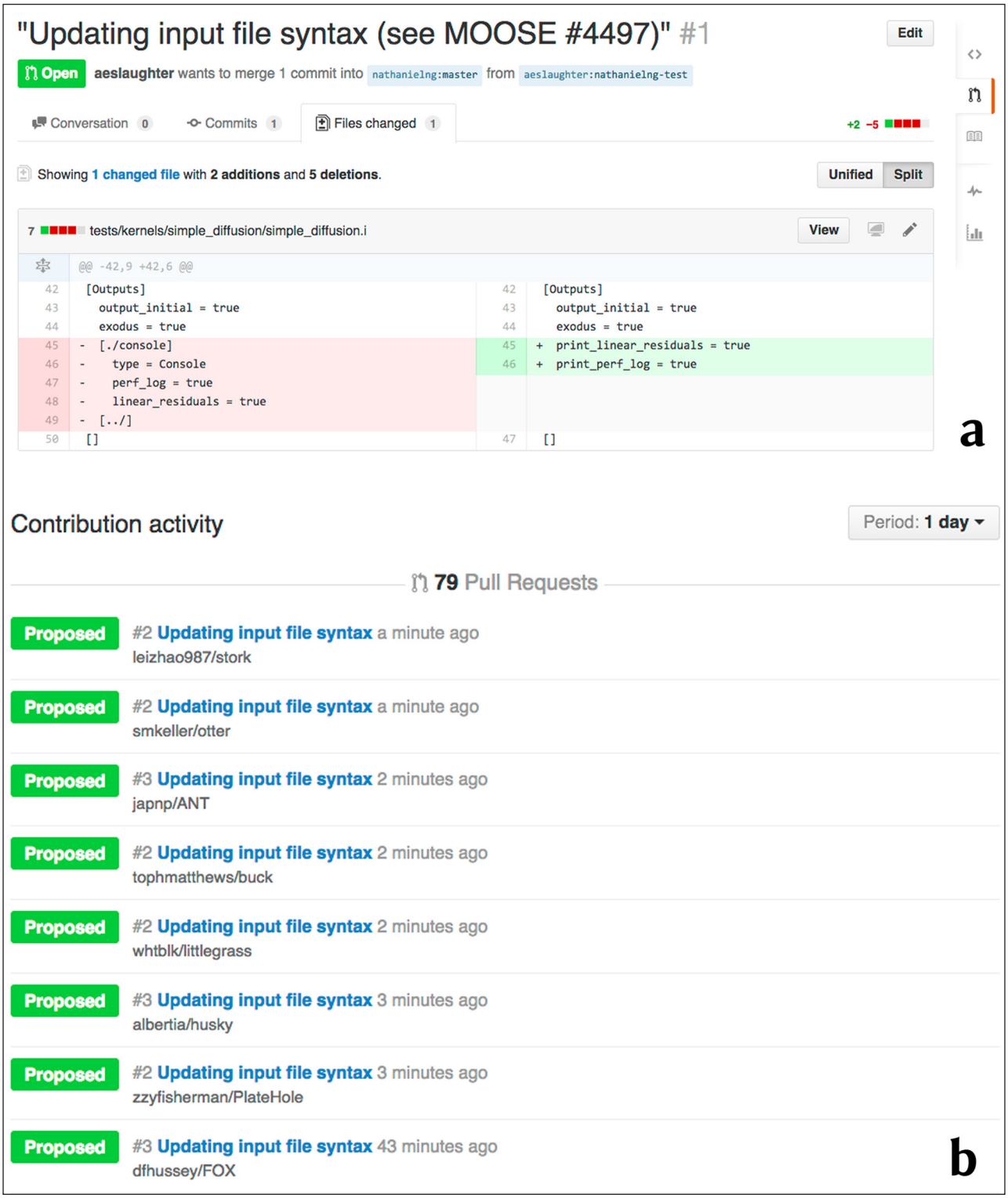

Figure 2: GitHub screenshots showing: (a) the patch for a Pull Request created via the GitHub API, and (b) the list of automatically generated Pull Requests for various forks.

repository contains a Git submodule for MOOSE which can be updated, either automatically or manually, based on the results of the regression testing system. The MOOSE-based Pika [9] application, for example, is updated in this way. This submodule update can also be restricted to a development branch within the application repository, allowing some users to continue running (and even developing) the application with a prior version of MOOSE, while other developers move forward with the latest API, with the goal of merging at some future date. Additional details of this strategy are discussed in Section 2.3.

2.2 Cascading Build System

The cascading build system, as discussed in [3], still operates in essentially the same manner as it did in the earlier shared repository model, with various enhancements to

support the multi-repository configuration. The build system now automatically detects MOOSE-based applications that are stored in directories alongside the moose directory (e.g., a MOOSE installation in `~/projects/moose` would find `~/projects/app0` and `~/projects/app1`). Additionally, we employ a set of environment variables (e.g. `$MOOSE_DIR`) to support a user-defined directory structure. As before, executing the '`make`' command from any MOOSE-based application directory will automatically rebuild not only the application, but all of its dependencies as well. Additionally, running "`make test_up`" from the `$MOOSE_DIR/framework` directory will automatically build and test all applications. This single command allows a MOOSE developer to ensure that the changes are compatible with the other MOOSE-based applications installed before proposing a GitHub Pull Request.

2.3 Automated Testing

The continuous integration strategy employed by MOOSE requires every commit to undergo a series of tests to ensure that all MOOSE-based applications continue to build and pass their regression test suites (details of the Python-based "test harness" used by MOOSE and the applications are discussed in [3]). Toward this end, a multi-level build and testing system called MooseBuild was developed (Gaston et al., in preparation) that integrates with GitHub, and facilitates the testing of MOOSE and its applications. The MooseBuild process is depicted graphically in **Fig. 3**, and the steps of the entire process from GitHub Pull Request to commit in the INL-hosted repositories is described in the following list.

1. **Pull Request Created:** All changes to MOOSE require GitHub Pull Requests [10]. Creating a Pull Request triggers the MooseBuild system, as shown in **Fig. 3**.
2. **Pull Request Testing:** Proposed changes are checked for adherence to coding standards, compiled, and tested—before being merged into the repository—on various compilers and in various configurations (valgrind, debug, MPI, threaded). Test results are reported as comments and continuous integration status updates [11] on the Pull Request. Both framework- and application-level tests exist. If a framework-level test fails, the Pull Request will most likely not be merged as-is, and the author will need to address the failures. The failure of an application-level test may or may not prevent the merge, depending on the way MooseBuild is configured.
3. **Review:** All code changes go through a peer-review process prior to being merged, both to ensure correctness and to determine the appropriateness of the changes for the referenced issue number. In the

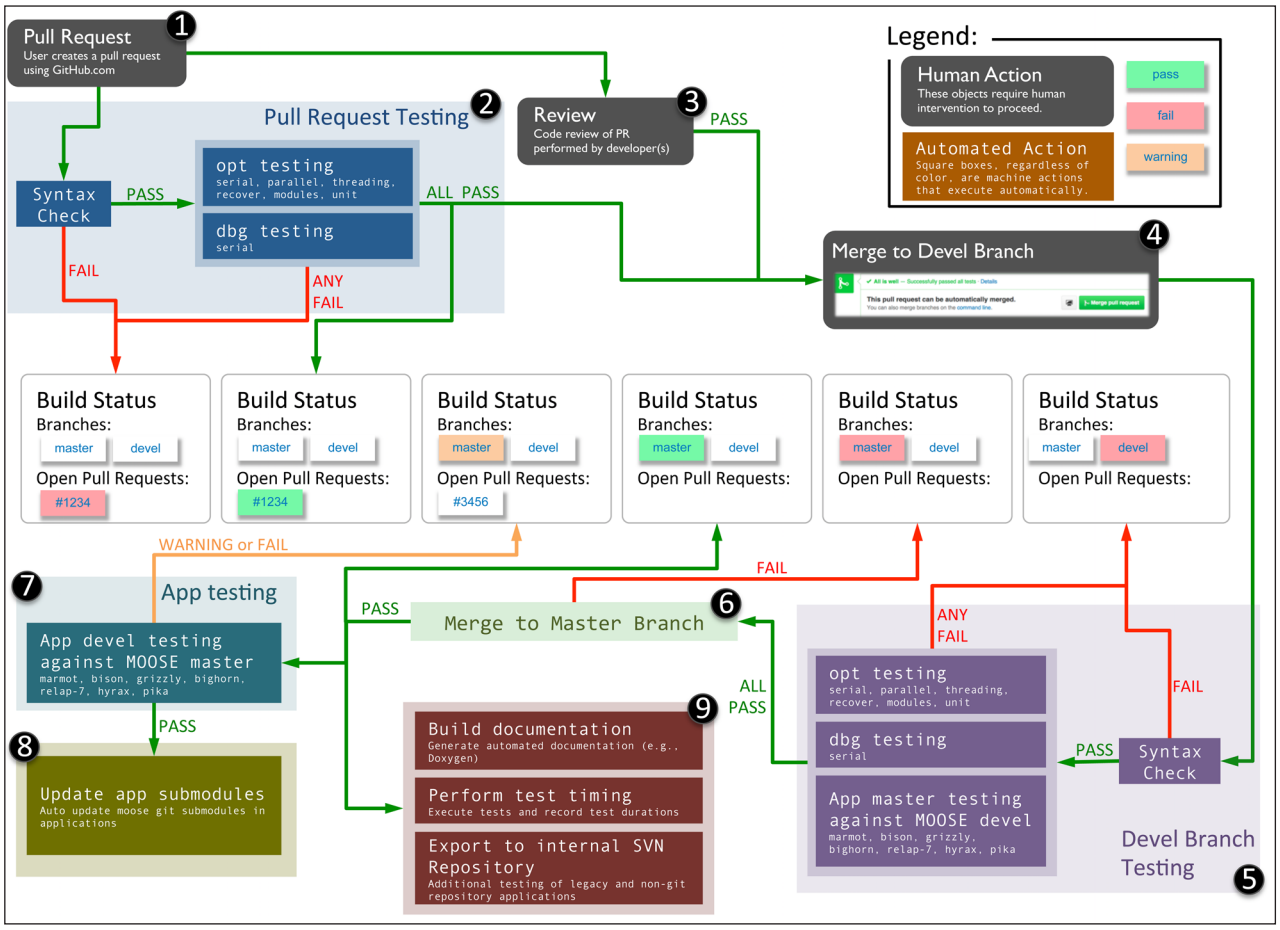

Figure 3: Flowchart depicting the development process and associated MooseBuild testing system. For a more detailed description, see Section 2.3.

MOOSE project, code proposed by any given developer must be reviewed and merged by someone else—you cannot merge your own Pull Request.

4. **Pull Request Merge:** After the Pull Request has passed the tests and one or more members of the MOOSE developer community have signed off on it, the Pull Request is merged (by clicking the "green button" on GitHub) into the `devel` branch of the repository.
5. **Devel Branch Testing:** A merge to the `devel` branch triggers a second round of testing that includes a suite of tests similar to Step 2. These are included to allow changes that are pushed directly to the development branch to be tested completely, even though this is a rare occurrence. Additionally, the `devel` branch of MOOSE is tested against the `master` branches of various other open- and closed-source MOOSE-based applications. Depending on the application, test failures will either cause the automated system to report a failure (and therefore prevent the MOOSE `devel` branch from merging into `master`) or simply be noted for future reference.
6. `devel` **to** `master` **Merge:** After the application tests are completed, the `devel` branch is automatically merged into `master`. This merge triggers Steps 7 and 9 to occur simultaneously.
7. **App Testing:** A second round of application testing occurs when the `master` branch is updated. This round tests the applications' `devel` branch against the MOOSE `master` branch.
8. **Update App Submodules:** If the application tests pass in the second round of application (Step 7), the MOOSE submodule within the application is updated. Further details of this process are discussed below.
9. **Documentation Updated:** After the `master` branch is updated, the various documentation-related tasks are executed. These include updates to the Doxygen-based source code documentation, input file syntax listings, and test timing data.

The computational resources required to run the continuous integration system are fairly unexceptional: we currently employ ten rack-mounted Ubuntu-based build machines, each with dual 8-core Intel Xeon E5-2450 2.1GHz CPUs, 96GB of system memory, and a single 256GB solid state hard drive. The testing (Steps 2, 5, and 7) typically requires about 15 minutes to complete, and the entire process, from Pull Request to merging in the `master` branch, is usually finished in about one hour.

Further elaboration on the two rounds of application testing and automatic submodule updates is warranted at this stage of the discussion. The system is designed to allow for framework- and application-spanning changes to occur without the `master` branches of either ever being in an invalid state. In general terms, this is accomplished by pushing application-breaking changes to the MOOSE GitHub repository (either directly to `devel` or to a special "integration branch" created expressly for the purpose) and then updating a dependent application's MOOSE submodule to this commit on the same branch where the application itself is updated. The key point is that the Git submodule allows applications to be based on a version of MOOSE that is *not yet* in the `master` branch, but will be at some time in the future. The basic steps of this approach are given in further detail as follows:

1. The developer makes the necessary changes to the framework and application locally.
2. The framework changes are pushed to an integration branch in the MOOSE GitHub repository.
3. The application's MOOSE submodule is updated to a commit on the integration branch.
4. A Pull Request is submitted to the application with the proposed changes as well as a submodule update of the MOOSE framework that points to the integration branch.
5. After the application Pull Request is merged, a Pull Request is submitted from the integration branch to the `devel` branch in the MOOSE GitHub repository.

We emphasize that the `master` branches of both the framework and the application are in a valid state during the entire process. This is important because it ensures that a user who tracks the `master` branch of their application can stay up-to-date without worrying about updating to an incompatible version of MOOSE, provided that the application is a part of the automated testing process. Furthermore, even if the integration branch is not merged into MOOSE `devel` in a timely manner (say, within a day or two) the dependent applications can continue to use revisions from the integration branch as long as necessary.

2.4 Discussion

It is important to understand that MOOSE, its derivative applications, and the continuous integration strategy discussed in this paper are ongoing research projects. The MOOSE framework and the applications are required to adapt and improve, in a synchronized manner, on a daily basis. To date, we have not employed traditional versioning or major-minor-point releases in the development of the framework. We envision the automated and manual submodule updates discussed here as a more fine-grained application of this concept, but recognize that a shift in both user and developer practices will be required for this approach to truly flourish.

We furthermore understand that maintaining, updating, and testing in the manner described in this paper does not scale particularly well in the number of applications. Therefore, in the future, we envision the need for stricter requirements on the application testing process, including e.g. minimum test coverage requirements, maximum test suite execution times, and willingness to merge compatibility pull requests from upstream in a timely manner. MooseBuild is being developed with extensibility in mind, and with the ability to allow external applications to test on their own hardware and report the results back to a

centralized server or servers, thereby reducing the overall computational requirements for testing.

(3) Documentation and Wiki

The content from the INL-hosted Trac [12] "wiki" described in [3] has since migrated to a public website [13]. This page is a "one-stop shop" for all MOOSE-related support topics and documentation resources, including: installation instructions, details of MOOSE plugin system APIs, descriptions of MOOSE physics modules, and links to automatically-generated documentation (code coverage, test timing and input file syntax). All of the automatic documentation mentioned in [3] is still available; documentation of the MOOSE physics modules is an ongoing effort. The modules documentation is a key addition to the existing MOOSE framework documentation, and is essential for accommodating and enabling new users who can now obtain MOOSE without first receiving formal, in-person training on its use.

(4) Impact of Open Source

Releasing MOOSE with an open source license was a strategic decision made with the goal of increasing the number of users and developers who can actively work on the project. It is instructive to look at the periods both immediately before and immediately following the public release of MOOSE on GitHub when trying to gauge the impact which opening the code has had on the project. **Table 1** shows the change in both the number of commits and contributors in the four month periods immediately preceding and immediately following March 2014, the date when MOOSE officially went open source. The increased frequency and size of commits over the approximately one year period following March 2014 is also evident in **Fig. 4**. **Fig. 5** highlights the increase in the rate of commits from outside developers as well as the number of unique contributors to the MOOSE framework.

In addition to new framework developers, the overall number of MOOSE-based applications and application developers has also increased. In the prior six years of closed-source MOOSE development, approximately thirty applications were created. Since open-sourcing MOOSE, over 125 public Stork forks and approximately 10 new closed-source applications [14] have been created, demonstrating the important role that ease of access plays in the growth and improvement of scientific software.

	Lines Added/ Removed	Number of Commits	Number of Contributers
Before	+24373/−12618	377	16
After	+49964/−33164	1069	24

Table 1: Number of lines added/removed, commits, and unique contributors both before and after the open sourcing of MOOSE. "Before" and "After" refer to the four month periods immediately before and after the open sourcing of MOOSE, respectively (see the Appendix for how these value were determined).

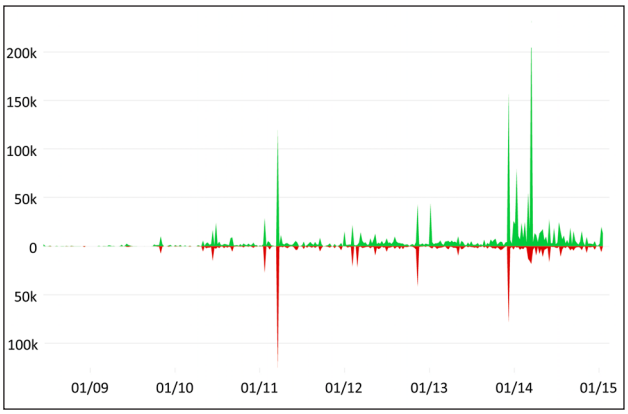

Figure 4: Number of source code lines added (green) or deleted (red) per week for the MOOSE repository (as reported by GitHub). Note the increased activity after MOOSE was open-sourced in March, 2014.

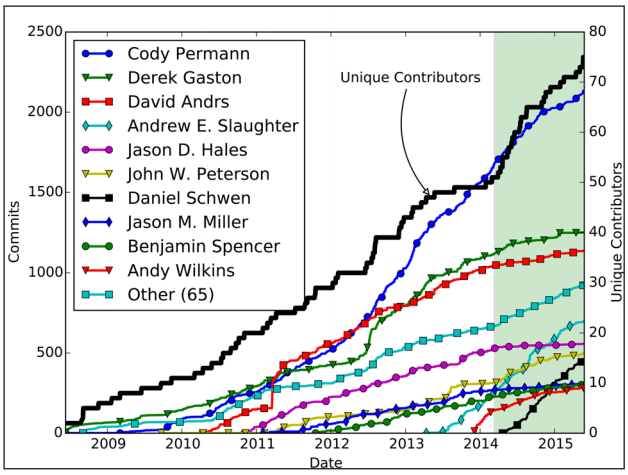

Figure 5: Left axis: commit history for the top ten MOOSE developers compared to all "other" contributors. Right axis: cumulative number of unique contributors over time (thick black line). The shaded region highlights the time frame during which MOOSE has been available as an open source library on GitHub.

(5) Future Work and Closing Remarks

The system and methodology used by the MOOSE project for continuous integration and concurrent framework and application development on GitHub is rapidly evolving. The development aspects discussed here represent the MOOSE team's current strategy for handling the many challenging multi-repository integration issues which have arisen thus far, but it is inevitable that the supporting infrastructure will continue to evolve and improve over time. As the concepts discussed here are further streamlined, the burdens of managing multiple repositories will likewise be reduced. Finally, the initial community response to the open sourcing of MOOSE and the introduction of the related software engineering practices has been very positive. We are confident that further improvements will continue to attract new users and developers to the project.

Competing Interests

The authors declare that they have no competing interests.

Author Information

Appendix

The results provided in **Table 1** were obtained by analyzing the four months prior to and following the open-source release of MOOSE. The Git revisions corresponding to these dates are:

- `10-Oct-2013: 961fcf56`
- `10-Mar-2014: b4739561`
- `10-Jul-2014: 793e4ce6`

Only changes to source and header files were recorded, across both modules and frameworks. The following commands were used for producing the total lines removed/added, the number of commits, and the number of contributors, respectively:

```
git log --numstat -- pretty="%H" <hash1>..<
    hash2> -- '*.[Ch]' | awk 'NF ==3 {plus +=$1;
    minus +=$2} END {printf ("+%d, -%d\n",
    plus, minus)}'

git rev-list --count <hash1>..<hash2>

git shortlog -s <hash1>..<hash2>
```

where `<hash1>..<hash2>` = `961fcf56..b4739561` and `b4739561..793e4ce6` for the four months prior to and after open-sourcing, respectively.

References

1. **Duvall, P M, Matyas, S** and **Glover, A** 2007 Continuous integration: improving software quality and reducing risk. Addison-Wesley. Available at: http://books.google.com/books?vid=ISBN9780321336385.
2. **Deshpande, A** and **Riehle, D** 2008 Continuous integration in open source software development. In: Russo, B et al. (Eds.) *Open Source Development, Communities and Quality, ser. The International Federation for Information Processing (IFIP)*, Springer, vol. 275, pp. 273–280. DOI: http://dx.doi.org/10.1007/978-0-387-09684-123
3. **Gaston, D R, Peterson, J W, Permann, C J, Andrš, D, Slaughter, A E** and **Miller, J M** Jul. 2014 Continuous integration for concurrent computational framework and application development. Journal of Open Research Software, 2(1): pp. 1–6, Article e10. DOI: http://dx.doi.org/10.5334/jors.as
4. https://github.com/idaholab/moose.
5. https://about.gitlab.com.
6. https://www.github.com/idaholab/stork.
7. https://developer.github.com/v3.
8. http://git-scm.com/book/en/v2/Git-Tools-Submodules.
9. https://www.github.com/idaholab/pika.
10. **Dabbish, L, Stuart, C, Tsay, J** and **Herbsleb, J** 2012 Social coding in GitHub: Transparency and collaboration in an open software repository. In Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work, ser. CSCW '12. New York, NY, USA: ACM, 2012, pp. 1277–1286. DOI: http://dx.doi.org/10.1145/2145204.2145396
11. https://developer.github.com/v3/repos/statuses.
12. http://trac.edgewall.org.
13. https://www.mooseframework.org/wiki.
14. http://mooseframework.org/wiki/TrackedApps.

How to cite this article: Slaughter, A E, Peterson, J W, Gaston, D R, Permann, C J, Andrš, D and Miller, J M 2015 Continuous Integration for Concurrent MOOSE Framework and Application Development on GitHub. *Journal of Open Research Software*, 3: e14, DOI: http://dx.doi.org/10.5334/jors.bx

Published: 20 November 2015

]u[*Journal of Open Research Software* is a peer-reviewed open access journal published by Ubiquity Press

OPEN ACCESS