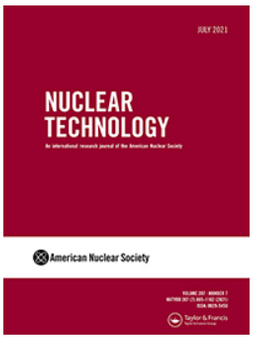

Nuclear Technology

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/unct20

Continuous Integration, In-Code Documentation, and Automation for Nuclear Quality Assurance Conformance

Andrew E. Slaughter, Cody J. Permann, Jason M. Miller, Brian K. Alger & Stephen R. Novascone

To cite this article: Andrew E. Slaughter, Cody J. Permann, Jason M. Miller, Brian K. Alger & Stephen R. Novascone (2021) Continuous Integration, In-Code Documentation, and Automation for Nuclear Quality Assurance Conformance, Nuclear Technology, 207:7, 923-930, DOI: 10.1080/00295450.2020.1826804

To link to this article: https://doi.org/10.1080/00295450.2020.1826804

Published online: 28 Jan 2021.

NUCLEAR TECHNOLOGY · VOLUME 207 · 923–930 · JULY 2021
DOI: https://doi.org/10.1080/00295450.2020.1826804

Continuous Integration, In-Code Documentation, and Automation for Nuclear Quality Assurance Conformance

Andrew E. Slaughter,* Cody J. Permann, Jason M. Miller, Brian K. Alger, and Stephen R. Novascone

Idaho National Laboratory, Advanced Scientific Computing, Computational Frameworks, 1955 N. Fremont Avenue, Idaho Falls, Idaho 83415

Received May 13, 2020
Accepted for Publication September 16, 2020

Abstract — *The Multiphysics Object Oriented Simulation Environment (MOOSE) is an open-source, finite element framework for solving highly coupled sets of nonlinear equations. The development of the framework and applications occurs concurrently using an agile, continuous-integration software package. Included in the framework is an in-code, extensible documentation system. Using these two tools in union with the repository management tools GitHub and GitLab, a software quality plan was created and followed such that MOOSE and a MOOSE-based application (BISON) have been shown to meet the American Society of Mechanical Engineers' Nuclear Quality Assurance-1 standard. The approach relies heavily on automation for both testing and documentation. The resulting effort demonstrates that a rigorous software quality plan may be implemented that incurs a minimal impact on day-to-day development of the software, satisfying the stringent guidelines necessary to operate the software in a safety function within a nuclear facility.*

Keywords — *Nuclear Quality Assurance-1 standard, MOOSE, documentation, software quality, testing.*

Note — *Some figures may be in color only in the electronic version.*

I. BACKGROUND AND MOTIVATION

The Multiphysics Object Oriented Simulation Environment (MOOSE) is an open-source, finite element framework for creating massively parallel simulation tools.[1] The framework is designed for solving highly coupled sets of nonlinear equations, with applications ranging from nuclear fuel performance[2] to neutronics[3] to thermal, hydraulic, and mechanical simulations for rock fractures.[4] The development process of the framework and derivative applications, such as BISON (Ref. 5), use an agile, continuous integration development process[6] that meets the software quality standards as defined by the American Society of Mechanical Engineers' Nuclear Quality Assurance-1 (NQA-1) standard.[7]

This paper aims to summarize the various components of the development process employed by MOOSE and BISON to meet the NQA-1 standard for software development. To meet the standard, three aspects are compulsory for the process:

1. It must fit within an agile, continuous-integration software design strategy.

2. It must add value to the development process with minimal burden on the average software developer.

3. It must be easily extended to any MOOSE-based application.

*E-mail: andrew.slaughter@inl.gov

The first aspect is necessary since MOOSE uses an established, automated, continuous-integration development process that allows for concurrent framework (MOOSE) and application development (e.g., BISON) (Refs. 6 and 8). This strategy has proven successful and is well liked by the developers, thus it has not been altered significantly. The second aspect requires any changes to the existing methodology to be useful, in general, and require minimal effort. Finally, to further reduce the burden of qualifying software, the process used is extendable to any MOOSE-based code (e.g., BISON) to allow for new codes to be deployed in a safety function in a rapid, sustainable fashion.

The second and third aspects are the primary result of the work presented. Implementing a quality program is costly and time consuming. The process developed for MOOSE and BISON provides a clear development strategy that adds value to the projects and meets a rigorous software quality standard. Moreover, the strategy can be implemented by any MOOSE-based application without difficulty.

This paper is organized into six sections. Section II describes the overarching software quality assurance plan (SQAP). Section III describes the continuous integration, verification, enhancement, and testing (CIVET) software: the automated, continuous-integration system. Section IV highlights unique features of the underlying documentation system (MooseDocs) used for creating in-code software documentation, including the NQA-1 content. Section V details the in-code documentation necessary to automate the documents needed to meet the NQA-1 standard, and Sec. VI offers closing remarks.

II. SOFTWARE QUALITY ASSURANCE PLAN

The development process for MOOSE and applications is governed by a change control board (CCB) and follows a well-designed management plan that is compliant with the NQA-1 standard with respect to software. This plan is used by both the framework and applications. Figure 1 is a flowchart of the management plan.

An important aspect of the plan is that it allows for both solicited and unsolicited change requests to occur ("A" in Fig. 1). As an open-source project, MOOSE regularly receives change requests from people outside of the CCB, and the development process allows for these changes while adhering to the NQA-1 standard. Another key aspect, as shown in "B" of Fig. 1, is the iterative development process that is enabled by automated testing and human review. This is pivotal for agile development.

The entire process, from testing to the human reviews, is automated and requires little interaction from

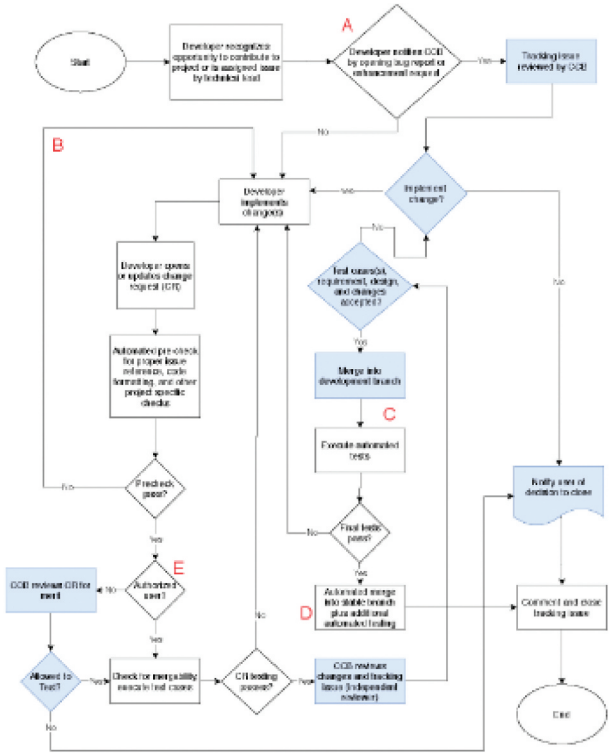

Fig. 1. CCB workflow for MOOSE and application development. Shaded regions require human interactions from CCB, and the letters in red near boxes are used within the main text to detail important aspects of the diagram.

maintainers. As such, the process is followed naturally for all proposed changes to the code, and developers are not required to understand the process to successfully navigate it. That is, if you are developing MOOSE, you will be guided through the SQAP. By default, new contributors are marked as unauthorized ("E" in Fig. 1) to limit the potential for malicious code being executed automatically.

MOOSE and applications depend on an operating system and other software libraries that do not follow an NQA-1 software quality program. These dependencies are listed as "support software" and are managed by the CCB by ensuring that combinations of operating systems and support software are maintained and validated at all times. All required support software is version locked and validated before being distributed to end users and developers.

The minimal impact for developers and the flexibility within the process is enabled by the automated testing that occurs using the CIVET and existing GitHub/GitLab interfaces. Only fully tested and reviewed contributions can be merged into the code base. Section III details the automated testing the CIVET used to implement the workflow presented here.

III. CONTINUOUS-INTEGRATION SYSTEM

The CIVET was designed for concurrent, continuous integration of a framework (i.e., MOOSE) and derivative applications. This concept is detailed in Refs. 6 and 8 and hinges on ensuring that changes to MOOSE are compatible with several supported applications (e.g., BISON). This compatibility check occurs in "D" of Fig. 1 and is integrated into the overall SQAP of the framework and applications. This section will provide an overview of the CIVET software, highlight the methods used for concurrent framework and application development, and elucidate the key features for following the SQAP.

The CIVET is a Django-based[9] client/server continuous-integration and testing system that operates with common hosted source code repository systems, such as GitHub[a] and GitLab.[b] The CIVET is written in Python and contains about 20 000 lines of code. The tool includes extensive capabilities, including

1. a comprehensive, unified dashboard with the status of all tested repositories and recent events
2. a common recipe and script repository to allow sharing of scripts
3. test target dependencies and priorities
4. scheduled, optional jobs, and dynamic jobs activated by the type of file altered
5. the ability to pin, cancel, and re-execute jobs
6. auto canceling of outdated tests due to new content being added
7. test clients that can run anywhere with access to the server
8. multiple client configurations (e.g., varying compilers or operating system)
9. required approval for unknown users
10. a work-in-progress mode to disable automatic testing for rapid collaboration
11. the ability to hide output for nonauthorized users.

The concept of concurrent framework and application development was the primary motivation for creating the CIVET, in particular the ability to interleave testing across many applications and to automatically update application submodules. For more information regarding the motivation behind the creation of the CIVET, please refer to Refs. 6 and 8. The project is developed and maintained following a similar process as MOOSE using a Git branching model[10] with continuous integration via change requests.

Extensive testing is performed for each change merged into the MOOSE development branches. This testing can take several hours to complete, and merges occur many times per day. Performing the complete set of tests for each merge quickly becomes a bottleneck. The CIVET solves this problem by intelligently pinning, canceling, and if necessary, restarting jobs as merges occur so that two merges are being tested at all times with the goal of minimizing the time to merge. Figure 2 gives an example of four development merges and how the CIVET attempts to maintain the progress of the first merge testing and the latest, nonfailing merge testing. Notice that testing #3 continues after failing. In this case, testing will continue to be performed to the extent possible for the failing merge to provide the most information about failures as possible. But knowing that #3 will not merge because of the failure, the CIVET restarts #2 to continue to attempt to keep open two passing paths for testing. This logic, as stated, aims to minimize the simultaneous testing and maximize the time to merge on the development branch and is entirely automated. GitLab introduced a similar feature, named merge trains,[11] to tackle this problem directly within the merge requests rather than within the development branch as done in the CIVET.

MOOSE and applications follow a traditional Git branching model as defined by Ref. 10. Support for concurrent

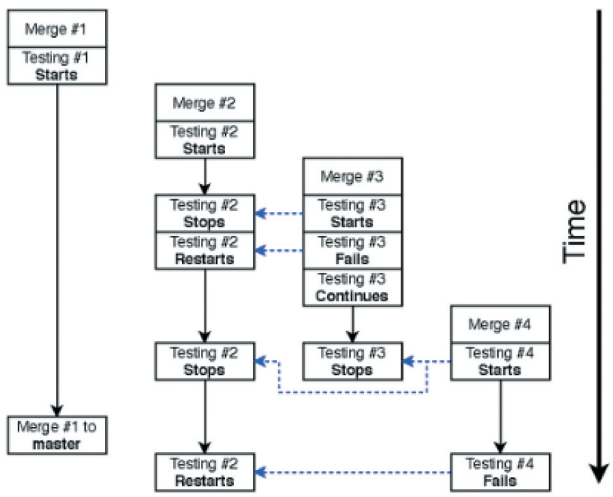

Fig. 2. Illustration of the CIVET auto-cancel feature designed to minimize testing on the development branch of MOOSE and maximize the merge rate by keeping the oldest and latest passing merges running. The vertical direction shows events occurring over time, and the horizontal arrows indicate the trigger of event cancellation or restart. This feature assumes that resources are limited and additional testing jobs are available.

[a] https://github.com.
[b] https://gitlab.com.

framework/application development is accomplished using the CIVET via Git submodules and an "integration branch" named "next" that serves as a staging area for changes to the framework that result in application failures. Merges occur into the next branch and then from next into "devel," with the latter being automatic. The integration branch is described in detail in Ref. 6. Originally, an integration branch was created temporarily, but now this branch is permanent and named the next branch. In MOOSE, the development branch described in the SQAP and highlighted as "C" in Fig. 1 is comprised of the next and devel branches together, where the next branch serves as a permanent integration branch. The next branch in this context differs from the traditional definition[12] in that the history is preserved; mistakes or failed ideas are removed using a revert. Figure 3 is an illustration showing a subset of the complete SQAP process. The first block comprises the iterative development process that occurs within a change request, the second two blocks are the development branch, and the final block is the stable branch.

The final step, "Master Branch Testing" in Fig. 3, includes application submodule updates that are critical to the concurrent framework/application development strategy. Figure 4, at the time of writing, is the interapplication dependency of applications tested at Idaho National Laboratory (INL). For example, as shown in Fig. 4, the application Grizzly depends on Blackbear, which depends on MOOSE. When a merge is made into the master branch, Blackbear and Grizzly are tested against this new version of MOOSE. If either application compiles and passes testing, the MOOSE submodule is updated within these applications. Similarly, a merge to the master branch of Blackbear will trigger Grizzly testing. If it passes, a submodule update of Blackbear within Grizzly automatically occurs.

The updating of dependent applications is a key feature of the system. The integration branch is central to the implementation; it allows for changes to the framework that impact an application to be staged. Using the integration branch, the applications are updated to operate with the proposed framework changes, and the associated submodule is updated to the integration branch. This guarantees that the stable branch of all applications operate, at all times, with the submodule(s) contained within the application.

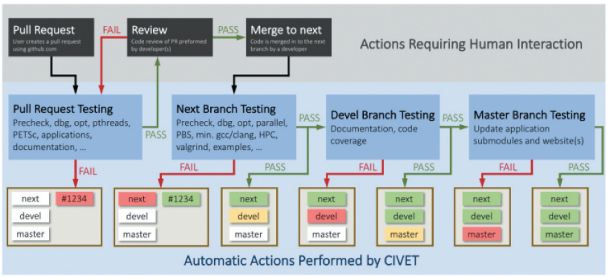

Fig. 3. Development process for MOOSE implemented using the CIVET.

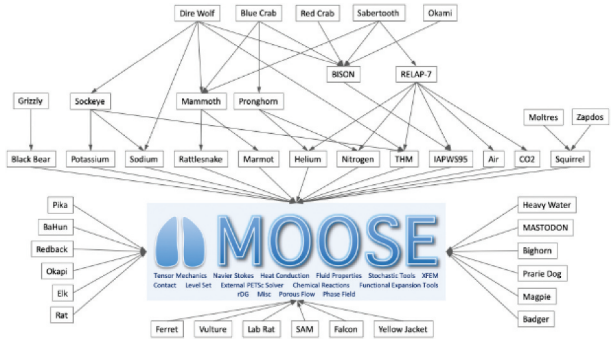

Fig. 4. Application dependency graph of private- and public-tested applications at the time of writing.

An important aspect of this process is that the integration and testing go beyond standard unit and regression testing and include documentation testing as well. Even documentation for an application can depend on the framework documentation, ensuring both the application and the application documentation are current and accurate at all times. This is an important ability when considering qualifying codes, which are based on each other, to meet the NQA-1 standard. The system that handles documentation (MooseDocs) is discussed next in Sec. IV.

IV. MOOSEDOCS: IN-CODE DOCUMENTATION SYSTEM

Writing and maintaining comprehensive documentation is a challenging task, in particular open-source software often lacks proper documentation. A 2017 survey performed by GitHub found that "documentation is highly valued, frequently overlooked, and a means for establishing inclusive and accessible communities" all the while 93% of those surveyed stated that project documentation was incomplete or confusing.[c]

Exacerbating the document dilemma is software required to meet NQA-1 (Ref. 7) software development standards. This adds an extensive set of standards that must be followed for development, as well as a long list of associated content. Acquiring an NQA-1 certification can take years of effort and can be a significant financial burden.

MOOSE is not immune to documentation shortcomings. While the reasons for the lack of documentation for any code project vary widely, the problems within MOOSE were caused by two primary problems: lack of

[c] https://opensourcesurvey.org/2017.

```
!listing src/kernels/Diffusion.h

#include "Kernel.h"

class Diffusion;

template <>
InputParameters validParams<Diffusion>();

/**
 * This kernel implements the Laplacian operator:
 * $\nabla u \cdot \nabla \phi_i$
 */
class Diffusion : public Kernel
{
public:
  static InputParameters validParams();

  Diffusion(const InputParameters & parameters);

protected:
  virtual Real computeQpResidual() override;

  virtual Real computeQpJacobian() override;

(framework/include/kernels/Diffusion.h)
```

Fig. 5. Example use of the "listing" command within (upper) markdown file that results in (lower) a rendered copy of source code.

a developer-focused system for writing content and no means to enforce documentation creation.

A modern solution to these problems is to treat documentation as code.[13] There is no standard means for implementing such a concept, especially within the context of the NQA-1 agile development process detailed in Sec. II. To meet the needs outlined in Sec. I, namely that documentation must add value and be a minimal burden to the developer, an in-code documentation system called MooseDocs was created that conforms to an agile, continuous-integration development process.

MooseDocs is a standalone Python tool within MOOSE that includes the ability to convert markdown to HTML, LaTeX, and presentations. The entire system is extension based and includes a parallel, multipage parsing system. By design, the ability to create new input and/or output formats exists, and applications can create custom extensions to meet specific needs. MooseDocs differs from other source code documentation systems, such as Doxygen[14] and Sphinx,[15] in that it is not intended for documenting source code. In fact, Doxygen is used for the C++ within MOOSE. The core of MooseDocs is simply a means for rendering markdown and was originally based on MkDocs (Ref. 16); this dependency was removed to meet the performance and extensibility needs for the MOOSE project.

At the time of writing, MooseDocs included 22 extensions (all contained within the main repository) that range from implementing traditional markdown syntax to creating dynamic plots to building complete traceability documents for NQA-1. The main purpose of an extension is to provide a convenient and simple way to build verbose and consistent documentation. For example, the "listing" extension enables code snippets to be embedded by pulling directly from the repository, as shown in Fig. 5.

The source code for MooseDocs is a part of the MOOSE repository[d] and is composed of about 12 000 lines of Python. Since the code is a part of the MOOSE repository, all changes to MooseDocs follow the development process outlined in this paper. MooseDocs relies on the standard Python library as much as possible, but does require two external libraries. The live, local rendering uses the "livereload" package,[17] and the configuration files that dictate the operation of the system use the "PyYAML" package.[18] Two extensions, which are optional, rely on additional outside packages. The "graph" extension requires Plotly[19] and pandas,[20] and the "bibtex" extension requires Pybtex.[21]

Another commonly used extension, the "autolink" extension, enables other pages to be automatically linked across markdown files. For example, using `[Kernels/index.md]` will automatically link to the associated page for the MOOSE Kernels system and use the linked page heading, as shown in Fig. 6. The supplied name to the markdown file does not need to be complete: The extension will automatically locate the page that ends with the supplied path and error if multiple or no pages are located. This fuzzy look-up feature is ubiquitous within the various MooseDocs extensions and allows documentation writers to focus on writing.

For MOOSE, all content is created using markdown, and these files are a part of the repository so developers can create content using their preferred editor and add it to the repository using standard git commands. MooseDocs includes a live, local serve option that automatically updates with file changes. This allows developers to write documentation while viewing a live rendering. The system has configurable content management that allows for an arbitrary number of rendered outputs to be created, each with differing content. This feature is used to allow for applications to include documentation from the framework and each other, thus avoiding rewriting documentation for shared features. The entire system is testable itself. For example, the "syntax" extension enforces that each registered object within MOOSE (i.e., any code that has input file syntax) has documentation. If the content is not provided in a change request, the continuous integration testing fails.

```
The [Kernels/index.md] is designed for implementing residual calculations.

The <a href="../syntax/Kernels/index.html">Kernel System</a> is designed
for implementing residual calculations.
```

Fig. 6. Example use of the "autolink" extension that enables markdown syntax that includes links to (upper) other pages that is converted to link with (lower) the correct page heading in the rendered HTML.

[d] https://github.com/idaholab/moose/tree/next/python/MooseDocs.

A flexible, extendable documentation system is critical for meeting the needs of a rigorous software quality program that satisfies the NQA-1 standard. As detailed in Sec. V, MooseDocs was extended to yield a low-impact methodology for meeting the NQA-1 standard and follows the SQAP detailed in previous sections.

```
[Tests]
  [test]
    type = 'Exodiff'
    input = 'recompute_markers_during_cycles.i'
    exodiff = 'recompute_markers_during_cycles_out.e-s002'

    issues = '#6663'
    design = "Adaptivity/index.md Adaptivity/Markers/index.md"
    requirement = "The Adaptivity system shall allow for 'Marker' objects "
                  "to be recomputed with each adaptivity cycle."
  []
[]
```

Fig. 7. Example test specification for MOOSE.

V. IMPLEMENTATION

It is important to understand that the NQA-1 standard does not provide a prescribed methodology that must be followed. Rather, it is a standard that when satisfied allows an organization to supply an item (in this case software) to operate in a safety function within a nuclear facility.[7] The standard, with respect to software development procedures, is centered around traceability: the connection between a requirement, a design, and a test.[22–24]

The development strategy for MOOSE that satisfies the standard relies on a rigid definition of a requirement. As detailed in Ref. 22, a software requirement is comprised of specific features. Three of these features state that a requirement must be unambiguous, verifiable, and traceable. As such, within MOOSE every requirement must be comprised of at least one test and must be linked to one or many design documents, as well as an issue. The test(s) associated with a requirement must all be contained within a single test specification file. By definition, each test is verifiable regardless of the type: unit, regression, etc. The requirement is also naturally unambiguous; the test is exercising something specific that no other test is doing (if this is not true then the test is not needed). To explain traceability, an example will be utilized.

MOOSE uses a custom testing system with test specification for each test to be executed. For example, Fig. 7 is an example specification with a single test (it is also possible to provide multiple tests within a single specification).

The example specification in Fig. 7 has two sets of parameters. The first set of parameters, "type," "input," and "exodiff," define how to perform the test. In this case, it defines a regression test. The second set of parameters, "issues," "design," and "requirement," provide all the necessary information to define the necessary traceability:

1. *Issues*: This parameter includes a list of issue numbers within the repository management tool (i.e., GitHub). These issues are what is referred to as a change request. This can be a bug fix or an enhancement.

2. *Design*: This parameter lists all the markdown pages within the repository documentation that discuss the design of the feature being tested. This can include any number of files and often links to the required object documentation mentioned in Sec. IV.

3. *Requirement*: This is the NQA-1 requirement that the test is verifying to be satisfied.

These three test specification parameters, along with the associated design documentation, are all that is necessary for complete traceability between a proposed change, requirements, and design. Most importantly, these items—a test specification that includes the motivation behind the test (issues), a description of the test (requirement), and a description of the code being tested (design)—add value to the project regardless of the purpose. For MOOSE, these items are mandated for all changes, and if not included result in test failures that prevent merging a proposed change. From a developer perspective, they have a minimal impact on production. However, due to the mandate, traceability is satisfied and the NQA-1 standard can be met when they are integrated into the entire toolchain detailed in the SQAP (see Sec. II). From a documentation maintenance standpoint, placing these three items within the test specification places all requirement-related information in a single location, which removes the need to manage cross referencing between requirement text, design documents, and the associated testing.

Using these three items, which exist within all the test specifications of MOOSE, a software quality extension "sqa" was created within MooseDocs to automatically generate a complete list of the system requirements, requirements traceability matrix, and the requirements cross reference.[7] A portion of the traceability matrix for MOOSE is shown in Fig. 8. Most importantly, these aspects of the NQA-1 documentation are always up to date for every change in the repository. The reader is encouraged to visit the MOOSE website[e] to explore

[e] https://mooseframework.org.

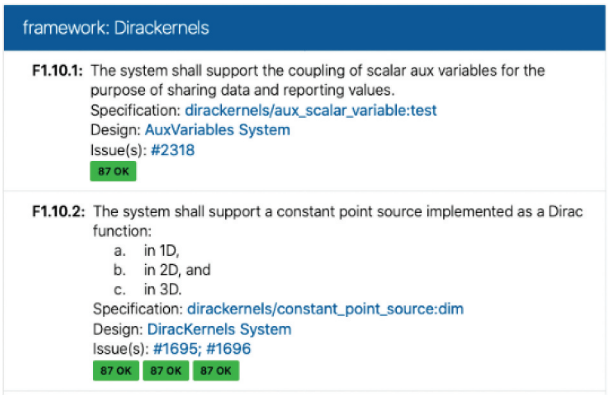

Fig. 8. Portion of the complete requirements traceability matrix automatically generated using the "sqa" extension of MooseDocs.

these complete documents in more detail. The traceability generated by MooseDocs includes testing information from all combinations of operating system and support software that were executed; most tests are executed 60 or more times before integrated into the master branch.

It is important to highlight that MooseDocs is not a substitute for quality exposition, attention to consistency, and reviews. In fact, all documentation is treated as code, and it is ultimately the code reviewer's responsibility to ensure that the complete documentation is in place for new functionality and that the documentation is updated in the case of functionality changes. However, MooseDocs was created to aid in this process through integrated error checks that are ever expanding. For example, the "appsyntax" extension allows for parameter names to be specified in the markdown as `[!param/Adaptivity/Markers/BoxMarker/inside]`. The system will render this as inside if the parameter exists within the application syntax or error if it is not found.

Of course, these three automatically generated lists are just a portion the necessary documents for the NQA-1 standard. Recall that in Sec. IV, a key feature of MooseDocs was that documentation, just like physics with MOOSE, are inheritable. In general, this is accomplished by the aforementioned content management system. The concept is extended further for NQA-1 documents with a custom template extension for MooseDocs. This allows the framework documentation of MOOSE to include a complete set of NQA-1 documents, such as the software test plan, the system design description, etc. This inheritance allows for applications, such as BISON, to inherit all the NQA-1 documents, including the requirements from the framework. Therefore, if an application follows the same development process described in Sec. II, it can automatically create design documents and annotate test specifications for a complete set of NQA-1–compliant documentation. The generated documents are comprised of the application's specific traceability tables and the correct links to the framework-level documentation. This concept allows INL to rapidly deploy any MOOSE-based application to serve a safety function with a minimal impact on the software developers.

VI. CLOSING REMARKS

MOOSE and MOOSE-based applications such as BISON have a well-designed SQAP plan that, when implemented, allows the codes to continuously maintain a complete set of NQA-1 documentation, meeting the standard for software serving a safety function with nuclear energy systems. The process relies heavily on a robust automated testing tool, the CIVET, and an in-code documentation system, MooseDocs, to implement the SQAP in such a fashion that all developers automatically follow the correct processes. The associated documentation is always current and accurate for every change to the framework and/or applications.

Acknowledgments

This work was funded under multiple programs and organizations: The INL Laboratory Directed Research and Development Program, and the U.S. Department of Energy's (DOE's) Nuclear Energy Advanced Modeling and Simulation and Consortium, United States of America for Advanced Simulation of Light Water Reactors programs. This research made use of the resources of the High Performance Computing Center at INL, which is supported by the DOE's Office of Nuclear Energy and the Nuclear Science User Facilities. This manuscript has been authored by Battelle Energy Alliance, LLC under contract number DE-AC07 -05ID14517 with the DOE. The U.S. government retains and the publisher, by accepting this paper for publication, acknowledges that the U.S. government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. government purposes.

ORCID

Andrew E. Slaughter http://orcid.org/0000-0002-4935-6812

References

1. C. PERMANN et al., "MOOSE: Enabling Massively Parallel Multiphysics Simulation," *SoftwareX*, **11**, 100430 (2020); https://doi.org/10.1016/j.softx.2020.100430.
2. J. D. HALES et al., "Asymptotic Expansion Homogenization for Multiscale Nuclear Fuel Analysis," *Comput. Mater. Sci.*, **99**, 290 (2015); https://doi.org/10.1016/j.commatsci.2014.12.039.
3. Y. WANG et al., "Rattlesnake User Manual," INL/ext-15-37337, Idaho National Laboratory (2016).
4. M. CACACE and A. B. JACQUEY, "Flexible Parallel Implicit Modelling of Coupled Thermal-Hydraulic-Mechanical Processes in Fractured Rocks," *Solid Earth*, **8**, *5*, 921 (2017); https://doi.org/10.5194/se-8-921-2017.
5. R. WILLIAMSON et al., "Multidimensional Multiphysics Simulation of Nuclear Fuel Behavior," *J. Nucl. Mater.*, **423**, *1*, 149 (2012); https://doi.org/10.1016/j.jnucmat.2012.01.012.
6. A. SLAUGHTER et al., "Continuous Integration for Concurrent MOOSE Framework and Application Development on GitHub," *J. Open Res. Software*, **3**, *1*, Article e14 (2015); https://doi.org/10.5334/jors.bx.
7. *ASME NQA-1-2008 with the NQA-1a-2009 Addenda: Quality Assurance Requirements for Nuclear Facility Applications*, 1st ed., American Society of Mechanical Engineers (2009).
8. D. R. GASTON et al., "Continuous Integration for Concurrent Computational Framework and Application Development," *J. Open Res. Software*, **2**, *1*, Article e10 (2014); https://doi.org/10.5334/jors.as.
9. "Django," Django Software Foundation; https://djangoproject.com (current as of May 13, 2020).
10. "A Successful Git Branching Model," Thoughts and Writings by Vincent Driessen (2010); https://nvie.com/posts/a-successful-git-branching-model (current as of May 13, 2020).
11. O. GOLOWINSKI, "How Merge Trains Keep Your Master Green," GitLab Docs; https://docs.gitlab.com/ee/ci/merge_request_pipelines/pipelines_for_merged_results/merge_trains (current as of May 13, 2020).
12. "gitworkflows—An Overview of Recommended Workflows with Git," git; https://git-scm.com/docs/gitworkflows (current as of May 13, 2020).
13. E. HOLSCHER et al., "Docs as Code" https://www.writethedocs.org/guide/docs-as-code (current as of May 13, 2020).
14. D. VAN HEESCH, "Doxygen" (2018); https://www.doxygen.nl (current as of May 13, 2020).
15. G. BRANDL and SPHINX TEAM, "Sphinx: Python Documentation Generator" (2020); https://www.sphinx-doc.org (current as of May 13, 2020).
16. T. CHRISTIE, "MkDocs: Project Documentation with Markdown" (2014); https://www.mkdocs.org (current as of May 13, 2020).
17. "LiveReload" GitHub (2020); https://github.com/lepture/python-livereload (current as of May 13, 2020).
18. "PyYAML"; https://pyyaml.org/wiki/PyYAMLDocumentation (current as of May 13, 2020).
19. "Collaborative Data Science," P. T. Inc. (2015); https://plot.ly (current as of May 13, 2020).
20. PANDAS DEVELOPMENT TEAM, "Pandas-dev/pandas: Pandas" (2020); https://doi.org/10.5281/zenodo.3509134 (current as of May 13, 2020).
21. "Pybtex"; https://pybtex.org (current as of May 13, 2020).
22. "IEEE Guide for Software Requirements Specifications," IEEE Std 830-1998, pp. 1–26, Institute of Electrical and Electronics Engineers (1998).
23. "IEEE Recommended Practice for Software Design Descriptions," IEEE Std 1016-1998, pp. 1–23, Institute of Electrical and Electronics Engineers (1998).
24. "IEEE Standard for Software Test Documentation," IEEE Std 829-1998, pp. 1–64, Institute of Electrical and Electronics Engineers (1998).