INL/EXT-15-36656

Light Water Reactor Sustainability Program

RAVEN Quality Assurance Activities

September 2015

DOE Office of Nuclear Energy

INL/EXT-15-36656

Light Water Reactor Sustainability Program

RAVEN Quality Assurance Activities

Joshua J. Cogliati

September 2015

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov/lwrs

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

EXECUTIVE SUMMARY

This report discusses the quality assurance activities needed to raise the Quality Level of Risk Analysis in a Virtual Environment (RAVEN) from Quality Level 3 to Quality Level 2. This report also describes the general RAVEN quality assurance activities. For improving the quality, reviews of code changes have been instituted, more parts of testing have been automated, and improved packaging has been created. For upgrading the quality level, requirements have been created and the workflow has been improved.

CONTENTS

FIGURES

TABLES

ACRONYMS

CSV	Comma Separated Variable file.
MOOSE	Multiphysics Object-Oriented Simulation Environment
QA	quality assurance
RAVEN	Risk Analysis Virtual Environment

RAVEN Quality Assurance Activities

1. Introduction

This report discusses the quality assurance (QA) activities needed to raise the Quality Level of Risk Analysis in a Virtual Environment (RAVEN) from Quality Level 3 to Quality Level 2. This report also describes the general RAVEN quality assurance activities. For improving the quality, reviews of code changes have been instituted, more parts of testing have been automated and improved packaging has been created. For upgrading the quality level, requirements have been created (see Appendix A) and the workflow has been improved.

The future and final RAVEN quality level determination has not yet being done and therefore it has not yet been determined if RAVEN [1, 2, 3, 4] will require a quality level 2 or 1. As already highlighted, the work described here would allow complying with quality level 2.

Most of the improvements implemented under the activities in support of this milestone are in accordance with the documents:

- Maintenance and Operations Plan for the MOOSE Project PLN-4003
- Configuration Management Plan for Modeling and Simulation Software PLN-4004
- Software Quality Assurance Plan for Modeling and Simulation Software PLN-4005
- Project Management Plan for Modeling and Simulation Software PLN-4213

Those documents were agreed to by the MOOSE and MOOSE-based software teams, to provide a common approach to fulfilling of QA requirements.

2. Quality Level

RAVEN is transitioning from being a purely developmental code to becoming an externally-used code. As part of that process, the Quality Level is being increased from Level 3 to Level 2. This level reflects that RAVEN will be beginning the maintenance and operations portion of its lifecycle. RAVEN development will continue with new features and other improvements, but the greater use requires more care to maintain RAVEN's stability and reliability. As part of a larger effort for MOOSE modeling and simulation, software management plans have been created and are being reviewed. These include the five documents: Maintenance and Operations Plan for the MOOSE Project PLN-4003, Configuration Management Plan for Modeling and Simulation Software PLN-4004, Software Quality Assurance Plan for Modeling and Simulation Software PLN-4005, Verification & Validation Plan for Modeling and Simulation Software PLN-4006 and Project Management Plan for Modeling and Simulation Software PLN-4213.

The documents commonly edited by the RAVEN team and by the other team developing software based on the MOOSE [5] platform, were reviewed and the necessary modification and documentation were implemented. One of the needs is the creation of requirements for RAVEN and these have been written. Another need is that when changes in the code occur, the code needs to be reviewed. This practice has been formalized and documented with the creation of a checklist that a second developer uses before any code is added to the main version. The checklist includes both a code review and running the regression tests.

3. Automated Quality Assurance

3.1 Cluster Testing

As part of improving the quality of RAVEN, automatic cluster testing was added. The regular MOOSEBUILD system that provides regression testing for RAVEN is not integrated with INL's HPC clusters. This was an issue because RAVEN has code that is only used when running on clusters. If changes occurred in other parts of RAVEN, then sometimes the cluster code would break, and RAVEN would no-longer be able to run on the cluster. The related problems might not get noticed until a user submits a job and RAVEN fails. The eventuality of not detecting problems with running on a cluster has been fixed by creating a set of tests that are automatically run every night on the INL HPC cluster. The scripts test three different methods of executing the code:

1. Running from interactive mode, where the user runs **`qsub`** themselves. In such a situation the user creates the proper parallel environment and executes RAVEN inside this environment.

2. Having RAVEN create the **`qsub`** command and run it. In this case the user asks RAVEN to create the parallel environment where the code is then executed.

3. Using custom mode where the user specifies the scripts and RAVEN follows those instructions. This also tests using **`pbsdsh`** which is an alternate way of running remotely compared to using MPI that is the more common approach to generating a parallel environment.

The script then emails the test results to the RAVEN development team so they can be checked and failures noticed.

3.2 RAVEN Packages

The several different types of RAVEN packages generated are listed in Table 1. The packages are groups of files that either provide RAVEN or provide things that RAVEN needs to be installed and run. These packages are generated using scripts as part of creating a release. The RAVEN team periodically produces releases for use by people who do not have access to the RAVEN git repository, and to help with installing RAVEN. There are several different types of RAVEN packages because different users have different needs. None of the current packages contain RAVEN C++ because that requires RELAP-7 as a dependency. The RAVEN source package contains the needed RAVEN, Crow and MOOSE source code and this package is mainly used for people who do not have access to INL's HPC systems or even if they want to perform the installation on a different machine. The libraries package contains the libraries needed to run RAVEN on OSX and is for people who do not install the MOOSE environment package. This package needs either the source package or use with git to get the RAVEN source code. The complete package contains both the libraries and the RAVEN code for a simple install on OSX.

Table 1 Types of RAVEN Packages

	Description	Operating System
Source Package	Contains the Source Code	Any
Libraries Package	Contains the Libraries used by RAVEN	OSX
Complete Package	Contains Libraries and RAVEN	OSX

3.3 Miniconda

The RAVEN project provides packages that can be used to simplify installing RAVEN. The original RAVEN packages for OSX generate only the Python modules and libraries that are not distributed with OSX. This works well when installing on a clean OSX install. Unfortunately, if the user installed or upgraded Python or the Python modules that RAVEN uses, that version might get used by RAVEN and sometimes this caused the installation of RAVEN to fail. Alternative OSX packages have been created that use Miniconda, a tool to install versions of Python. Miniconda is used to installs a separate installation of Python and all the Python modules and libraries that RAVEN needs. This isolates RAVEN from Python modules that a user might have installed for other Python software, which makes installation more reliable. In addition, Miniconda works with both Yosemite and Mavericks, which allows a unified libraries package to be created instead of providing two OSX packages. The use of Miniconda does have the disadvantage of increased package size, since Miniconda includes its own Python. This increases sizes from about 75 MB to 300 MB.

3.4 RAVEN Whole

RAVEN uses MOOSE, but the RAVEN framework only uses a portion of MOOSE. The portions of MOOSE used include the regression testing system and some of the Python modules. Since RAVEN only uses part of Moose, users only need to get that part to run the code. For releases, this is already done since the release scripts extract the necessary parts of MOOSE and put them in the packages. Releases however are only done periodically. In order to provide frequent updates, this needs to be done automatically. In order to make this easy for users to use, it makes sense to create a repository for this task.

For these reasons a whole RAVEN git repository has been created. This repository combines the needed parts of MOOSE and Crow with RAVEN in a single repository. A script automatically checks out RAVEN and the needed parts of Crow and MOOSE into a directory tree. Then these are tested, and if the RAVEN tests pass, they are pushed into the whole RAVEN git repository. Users of RAVEN can then just get the whole RAVEN, instead of using three different repositories. This simplifies installation and running. It makes it easy to keep updated with RAVEN and Crow and MOOSE. It does have the disadvantage of creating an effectively read-only repository, so developers will still need to use the existing separate repositories for their work but it could be used by beta testers that need to receive a prompt feedback from the developers.

3.5 Virtual Machine Testing

It is useful for RAVEN to run on a variety of different operating systems. However, testing on multiple operating systems is time consuming. In order to speed up this testing, a method for testing on multiple Linux distribution with virtual machines has been created. The virtual machines are an entire simulated computer and operating system that resides on the host machine. The virtual machines are created manually, but after that they can be automatically started. The main computer will simulate virtual machines that then test RAVEN. Currently this system is used to test the RAVEN whole repository and the RAVEN source code package. For both type of sources a script compiles RAVEN's modules, and then tests RAVEN. This works by putting a `rc.local` script in each computer. That script then calls a script in a user directory that does the compiling and testing. This has been tested with the Linux distributions Fedora 21, Ubuntu 14.4 and Ubuntu 15.4.

4. RAVEN Development Processes

The RAVEN development processes are used to ensure that RAVEN maintains its quality, manages to simplify software code development, and constructs the proper records for QA purposes. For each step of the development process, a set of guidelines and needed actions are defined that guide the developers from submitting the request for a new feature or bug fix to the merge of the new code into the production version of the code.

4.1 Merge Requests

One of the key parts of managing the RAVEN code configuration is merge requests. The git source code repository stores different code branches. Each branch has a version of the source code and also the history of that version. This allows developers to work on new features or bug fixes on a separate branch. There are also other branches that are permanent. There is a development branch where all new code is merged to and a master branch that development automatically merges to when the regression tests are passed. There also can be stable release branches. Those stable branches contain the version of the source code that has been released. If the version of the code is the one currently supported bug fixes are implemented to this version so that, while a new version is being developed the users can still enjoy improvements in the current stable version. So the developers do most of their work on specifically created branches. These changes on the branch then need to be merged with the main development branch.

When the developer is ready for others to use the newly developed code, they submit a merge request to merge it to the main development branch, or to a stable branch. A stable branch is a released branch that only has bug fixes added. The merge request has to be submitted to one of the team developer that has not taken part in the development to provide independent review. The gitlab software provides a GUI for processing merge requests as shown in Figure 1.

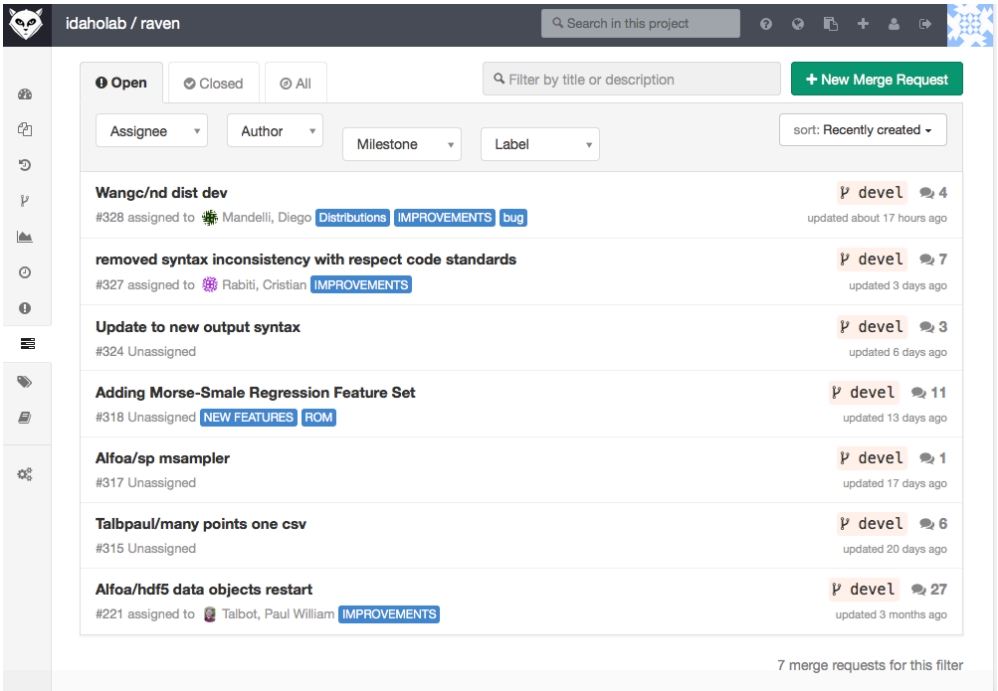


Figure 1 GitLab GUI for managing merge requests

4.2 Checklist

Recently, the RAVEN developers implemented a merge request checklist. This checklist is used for all code changes before they are merged into a main branch (the devel or a stable branch). The developer reviewing the merge request goes through the checklist and determines if the code passes or fails the requirements described in the checklist. If any of the items fail, then the merge request is not accepted and sent back to the original developer who needs to fix the problems identified. As part of the software quality plan, the code is reviewed and the tests are run. In addition the checklist ensures that all input changes are properly documented, that the code is properly documented with comments, new regression test are added if needed and other things. The XML input files for RAVEN have an **`xsd`** schema and this is used to validate inputs as part of the merge request review. If valid inputs would no-longer work after a merger request, a conversion script that can automatically convert the old input files needs to be provided as part of the merge request. The current checklist used is:

1. Review all computer code.
2. If any changes occur to the input syntax, there must be an accompanying change to the user manual and **`xsd`** schema. If the input syntax change deprecates existing input files, a conversion script needs to be added.
3. Tests should validate against **`xsd`** schema and pass
4. If significant functionality is added, there should be tests added to check this.
5. If it is a bug fix, tests should be added to prevent the bug from recurring, or the merge request should explain why tests are not added (such as an existing test would have caught it, but had an incorrect output).
6. All new functions and new classes should have comments. For C++ they should be in doxygen format, and for python they should be docstrings.
7. If the **`xsd`** schema is changed, it should be rebuilt.
8. If the manual is changed, it should be rebuilt.

4.3 Regression Tests

RAVEN uses regression tests to ensure that new code changes do not cause regressions (failure of already existing capabilities). Regressions are where the code originally worked, but a change causes different behavior. The regression tests are used as part of the checklist. They are also automatically run on the moosebuild website for the master and development branches and when merge requests are submitted. Moosebuild shows the results of the tests as shown in Figure 2. The numbers of regression tests have been increased. In November of 2014, there were 60 regression tests and in September of 2015 there are now 122 tests. New tests are added as new features are added, or as gaps in coverage are identified.

The regression testing system has been extended for RAVEN's unique needs. Because RAVEN is mostly Python, and unlike most other MOOSE applications, the primary output is not an Exodus file, RAVEN requires some customization of the MOOSE regression testing system. The MOOSE regression testing system is designed to allow applications to create new types of tests and extend how tests work. In previous work, the ability to test against a CSV file was added. This year's work added the ability to test if an XML file matched a reference (or "gold") XML file. In addition, for parallel running of the code, sometimes the outputs in the CSV file are order-dependent on which computer code finished first. With the original CSV comparison, this would be considered a different result. The current regression

tests allow these unordered results to be allowed when the test might return them. These changes in the regression system make more testing possible in RAVEN.

The code coverage is checked for regression tests. This checks what percentage of the lines or functions in the code are run when the regression tests are run. Because there is both Python code and C++ code in RAVEN, two different coverage tools are run. For the C++ code, the lines of code covered by the tests is 81%. For Python, a script has been created that uses Ned Batchelder's `coverage` tool. For the Python code the lines of code covered is 84%. Moosebuild generates reports automatically for both these as shown in Figure 3 and Figure 4.

4.4 Workflow for Change Requests

The workflow for committing code requires both review and tests. Figure 5 shows a graphical version of the workflow. The initial start is usually with an issue that either a RAVEN user or a RAVEN developer submits using the ticketing system. This issue is either a bug or a feature request. Then a developer creates a branch and works on resolving the issue. When a developer is finished, they submit a merge request, and a second developer reviews this merge request. The second developer either requests changes, accepts the merge request, or rejects it. If it is accepted, then the developer merges the request into the development branch. Then the regression tests are automatically run again, and if they pass, the development branch is merged into the master branch.

4.5 Development Issues

When bugs are found or new features are wanted, the users and developers can create an issue. Other developers can then find these issues. During the fiscal year 2015, 292 issues were created, and 70 are still open. Of the 70 open issues (e.g., feature requests), 6 are labeled as bugs. Figure 6 shows an example issue.

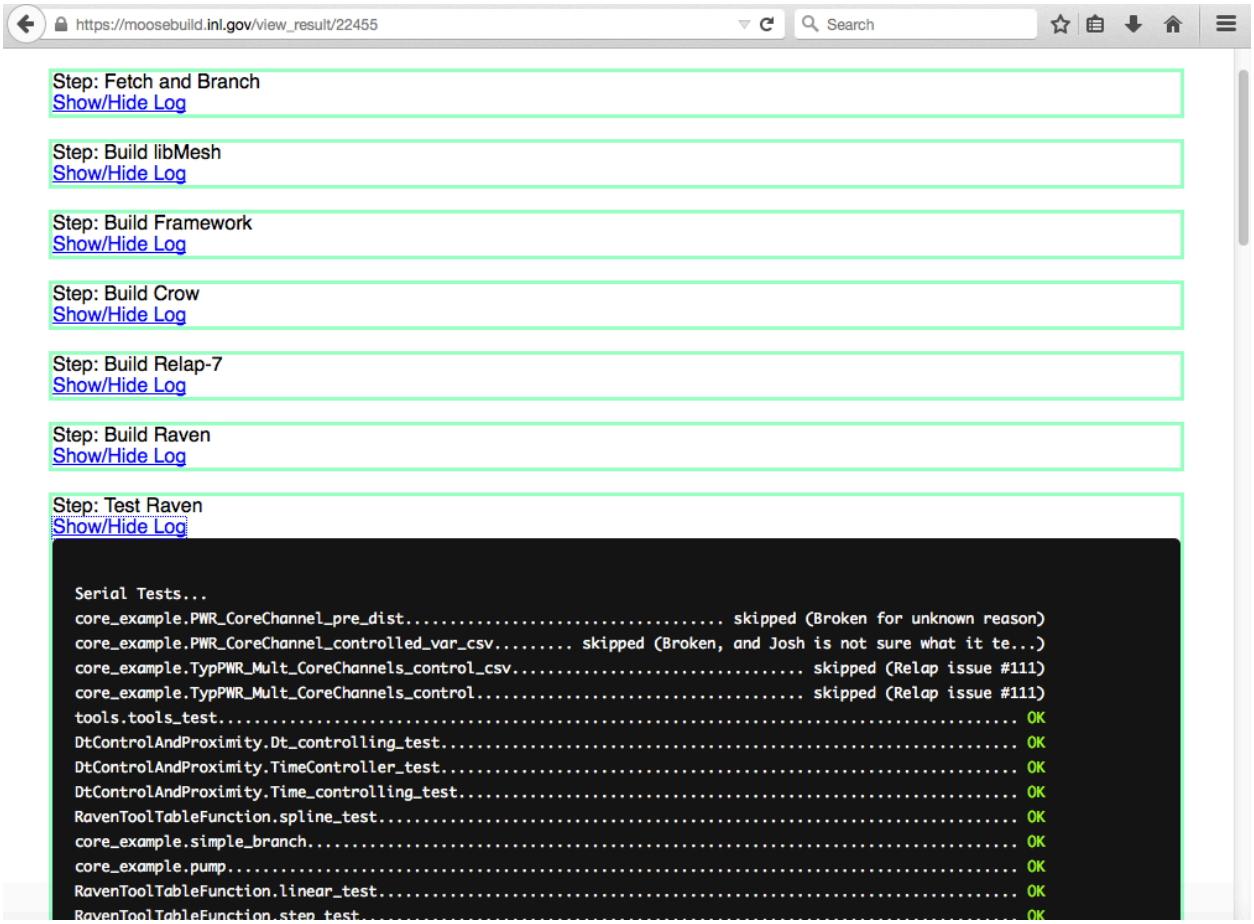


Figure 2 Regression tests on moosebuild

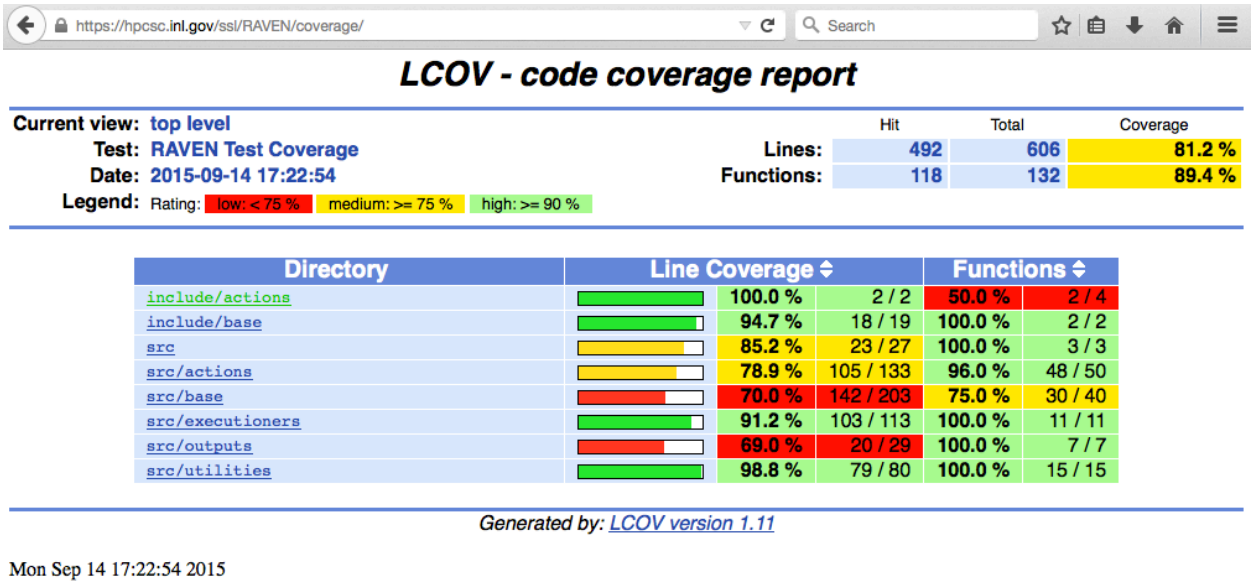
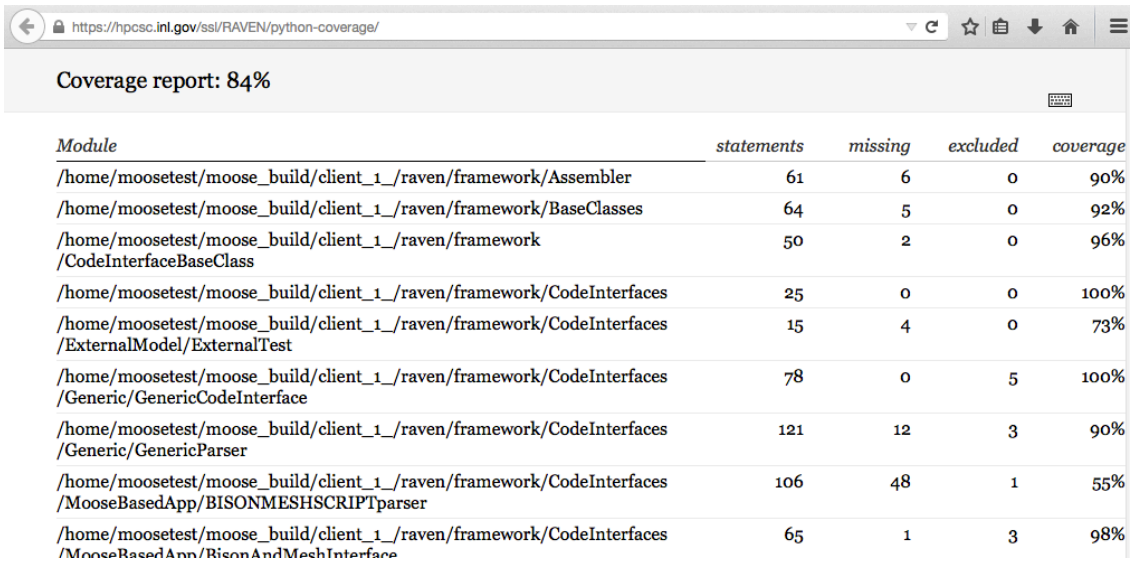


Figure 3 C++ code coverage



Module	*statements*	*missing*	*excluded*	*coverage*
/home/moosetest/moose_build/client_1_/raven/framework/Assembler	61	6	0	90%
/home/moosetest/moose_build/client_1_/raven/framework/BaseClasses	64	5	0	92%
/home/moosetest/moose_build/client_1_/raven/framework /CodeInterfaceBaseClass	50	2	0	96%
/home/moosetest/moose_build/client_1_/raven/framework/CodeInterfaces	25	0	0	100%
/home/moosetest/moose_build/client_1_/raven/framework/CodeInterfaces /ExternalModel/ExternalTest	15	4	0	73%
/home/moosetest/moose_build/client_1_/raven/framework/CodeInterfaces /Generic/GenericCodeInterface	78	0	5	100%
/home/moosetest/moose_build/client_1_/raven/framework/CodeInterfaces /Generic/GenericParser	121	12	3	90%
/home/moosetest/moose_build/client_1_/raven/framework/CodeInterfaces /MooseBasedApp/BISONMESHSCRIPTparser	106	48	1	55%
/home/moosetest/moose_build/client_1_/raven/framework/CodeInterfaces /MooseBasedApp/BisonAndMeshInterface	65	1	3	98%

Figure 4 Python Code coverage

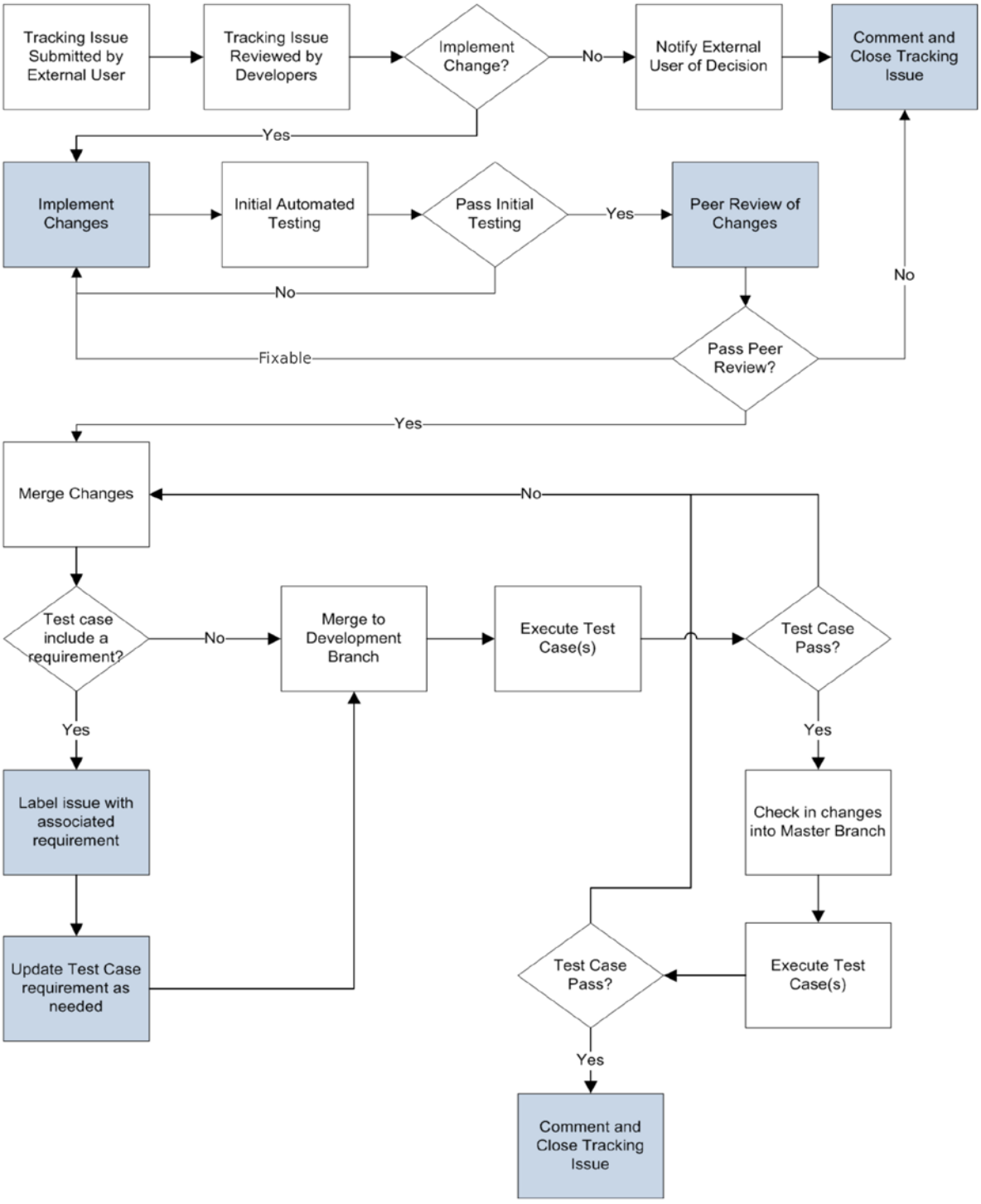


Figure 5 Code Change Process

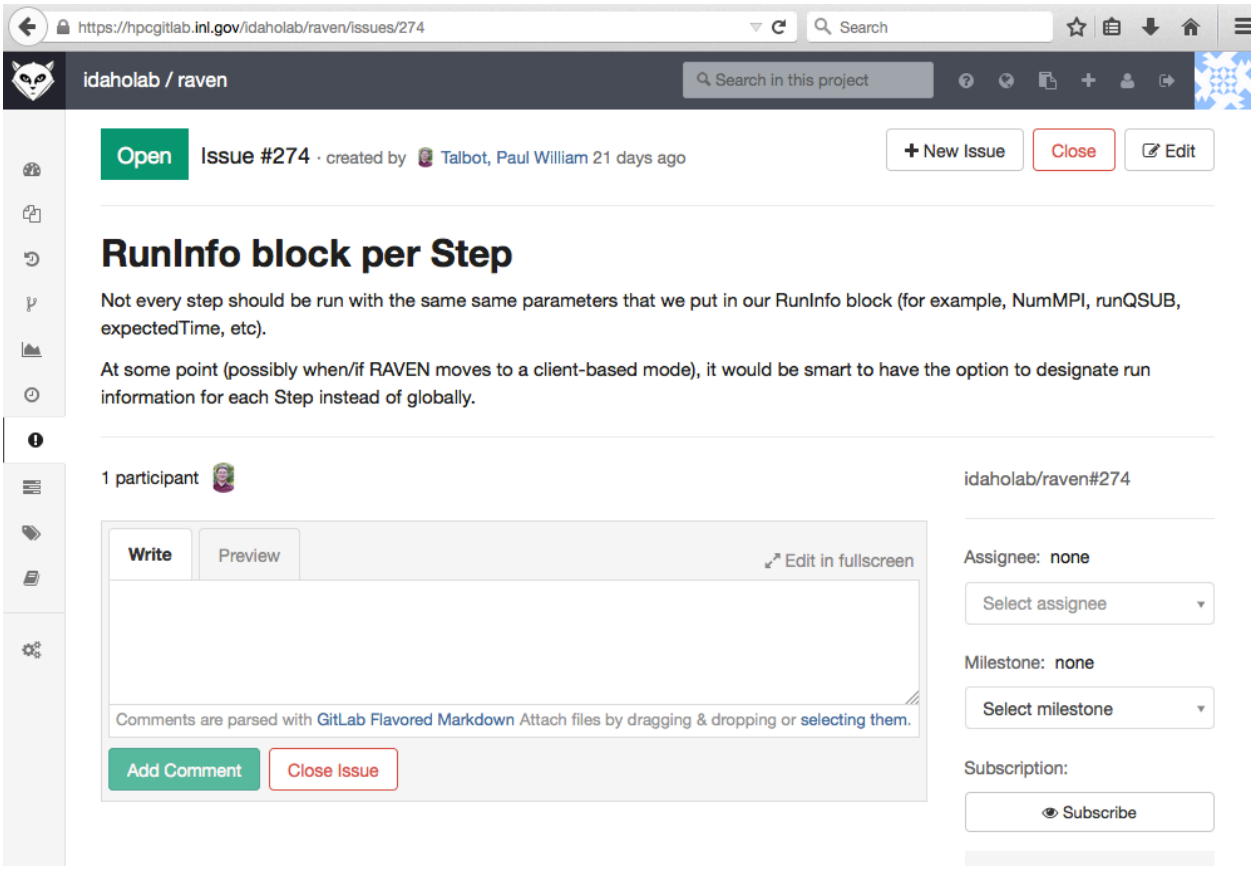


Figure 6 Example of an Issue

5. References

1. A. Alfonsi, C. Rabiti, D. Mandelli, J. Cogliati, and R. Kinoshita, "Raven as a tool for dynamic probabilistic risk assessment: Software overview," *in Proceeding of M&C2013 International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering*, on CD-ROM, May 5-9, Sun Valley (2013).
2. C. Rabiti, A. Alfonsi, D. Mandelli, J. Cogliati, R. Martinueau, C. Smith, "Deployment and Overview of RAVEN Capabilities for a Probabilistic Risk Assessment Demo for a PWR Station Blackout," Idaho National Laboratory report: INL/EXT-13-29510 (2013).
3. A. Alfonsi, C. Rabiti, D. Mandelli, J. Cogliati, R. Kinoshita, and A. Naviglio, "RAVEN and dynamic probabilistic risk assessment: Software overview," *in Proceedings of ESREL European Safety and Reliability Conference,* Wroclaw, Poland (2014).
4. A. Alfonsi, C. Rabiti, D. Mandelli, J. Cogliati, R. Kinoshita, "Performing Probabilistic Risk Assessment Through RAVEN*", in Proceedings of American Nuclear Society 2013 Annual Meeting "Next Generation Nuclear Energy: Prospects and Challenges*", Atlanta, GA (2013).
5. D. Gaston, C. Newman, G. Hansen and D. Lebrun-Grandi, "MOOSE: A parallel computational framework for coupled systems of nonlinear equations," Nuclear Engineering Design, 239, pp. 1768-1778, (2009).

Appendix A: Initial Requirements

Requirement: R1

RAVEN must be able to parallelize running external codes.
Explanation: RAVEN runs external codes, and sometimes they are not parallelized. RAVEN will run faster if it can run multiple codes at the same time when multiple cores are available. Even for parallelized codes it usually will be more efficient to run multiple instances in parallel than run one code parallelized.

Regression Test: testLHSBisonParallel

Requirement: R2

RAVEN must be able to provide external codes the files that are needed for their running.
Explanation: RAVEN runs external codes, and each instance may need a different input file that needs to be generated from the sampler choices. RAVEN also may need to read the output files in. (possibly with application specific code that is user provided.)

Regression Test: simple_framework

Requirement: R3

RAVEN must support 1-Dimensional probability distributions including generating random numbers from them.
Explanation: RAVEN needs to create different parameters for the simulations that it runs. For the non-adaptive sampling, probability distributions are used for this (including flat distributions). In order to do this, the distributions need to be able to calculate things like PDFs and CDFs and inverse CDFs.

Regression Test: test_distributions

Requirement: R4

RAVEN must support N-Dimensional probability distributions. It must support multivariate normal distributions and distributions defined by tabular data.
Explanation: The N-Dimensional probability distributions allow the user to model stochastic dependencies between parameters.

Regression Test: ND_external_MC

Requirement: R5

RAVEN must support a variety of samplers that use probability distributions to sample the input space.
Explanation: Once through samplers allow sampling strategies such as Grid sampling, Monte Carlo and Latin Hypercube sampling. These samplers allow the analyses to be performed.

Regression Test: testGridRaven

Requirement: R6

RAVEN must support adaptive sampling that use already gathered samples to determine where to do new samples.
Explanation: The adaptive samplers support sampling the input space, but in a more efficient manner. One example of these samplers is a limit surface search.

Regression Test: test_Adaptive_DynamicEventTreeRAVEN

Requirement: R7

RAVEN must support storing and retrieving data in a HDF5 database.
Explanation: RAVEN uses HDF5 databases to store inputs and results for simulations, as well as other auxiliary information.

Regression Test: test_merge_2_databases

Requirement: R8

RAVEN must support outputting data in CSV format.

Explanation: The user needs to be able to get the data and examine it and sometimes process it in other programs. Outputting the data in CSV files allows this use to be done.

Regression Test: test_iostep_load

Requirement: R9

RAVEN must support generating plots from the data it generates.
Explanation: The user needs to be able to see the progress of the algorithms, and what the results are graphically. As well, plots to be used in documentation and reports need to be outputted. The plotting capability of RAVEN is used for this.

Regression Test: test_output

Requirement: R10

RAVEN must be able to provide data to MOOSE based applications, and retrieve data if the application successfully completes.
Explanation: RAVEN uses external simulation software to calculate physical models. RAVEN creates input files, calls the external code, and then reads in the results.

Regression Test: testGridBison

Requirement: R11

RAVEN must be able to generate Reduced Order Models from its data and use them to predict responses from a system.
Explanation: Often the physical model is computationally expensive. For some models the relevant output parameters can be captured by a much simpler model that can be quickly calculated. This is the purpose for the Reduced order model.

Regression Test: test_rom_trainer

Requirement: R12

RAVEN must be able to provide data to a user provided python function, and retrieve the data from that.
Explanation: Sometimes all that is needed for the simulation is a function that can be calculated in Python. The external model allows this. This executes a python function to determine the result.

Regression Test: testExternalModel

Requirement: R13

RAVEN must be able to perform various calculation tasks, and transfer data to the next task.
Explanation: Sequences of calculation are one of the main uses of RAVEN. For example, a initial calculation can be used to generate data to train a ROM, and then later calculations

can use the ROM for faster calculation. As well, steps allow various post processing to be done.

Regression Test: testLimitSurfacePostProcessor