ORNL/TM-2016/572

M3MS-16OR0401086 – REPORT ON NEAMS WORKBENCH SUPPORT FOR MOOSE APPLICATIONS

Robert A. Lefebvre
Brandon R. Langley
Adam B. Thompson

September 23, 2016

Approved for public release. Distribution is unlimited.

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2016/572

Reactor and Nuclear Systems Division

M3MS-16OR0401086 – Report on NEAMS Workbench Support for MOOSE Applications

Robert A. Lefebvre
Brandon R. Langley
Adam B. Thompson

Date Published: September 2016

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-BATTELLE, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

ABSTRACT

This report summarizes the status of the Nuclear Energy Advanced Modeling and Simulation (NEAMS) Workbench from Oak Ridge National Laboratory (ORNL) and the integration of the MOOSE framework. This report marks the completion of NEAMS milestone M3MS-16OR0401086. This report documents the developed infrastructure to support the MOOSE framework applications, the applications' results, visualization status, the collaboration that facilitated this progress, and future considerations.

LIST OF FIGURES

1. INTRODUCTION

The Nuclear Energy Advanced Modeling and Simulation (NEAMS) Workbench is a new initiative for the 2016 fiscal year. It will facilitate the transition from conventional tools to high-fidelity tools by providing a common user interface for model creation, review, execution, and visualization for integrated codes. The Workbench can use common user input by templating engineering scale specifications to application-specific input requirements. This will enable multi-fidelity analysis of a system from a common input. The expansion of codes integrated under the Workbench will broaden the NEAMS tools user community and facilitate more science and engineering. This task was officially started in June of 2016 with a focus to support MOOSE applications, with BISON being the pilot application.

2. MOOSE FRAMEWORK APPLICATION INPUT

The MOOSE framework uses the GetPot [http://getpot.sourceforge.net/] input file and command line parser. The GetPot project is an anagram of the Linux 'getopt' utility which is used to process program command line arguments. GetPot extends command line argument processing with a hierarchical input format that describes blocks, sub-blocks, and named parameter values of an input. The MOOSE application instantiates an extension of the GetPot utility included in the LibMesh code repository [https://libmesh.github.io/doxygen/getpot_8h_source.html]. GetPot provides constructs that make it easy for users to enter their input into their MOOSE application. The extraction of this user input by the MOOSE application is concise and easy to use.

The MOOSE framework provides a capable input infrastructure that allows MOOSE applications to communicate an input definition, the available blocks, sub-blocks, and named parameters, to the user via a --dump or --yaml command line option. The --dump option produces the input definition in the GetPot format with certain input information (e.g., named parameter legal value enumerations) in comment fields. The --yaml command line option produces a YAML [YAML Ain't Markup Language: http://yaml.org/] formatted input definition. The YAML-formatted input definition provides a more complete input definition for input validation, but it still lacks some important reference information which will be discussed in a later section.

2.1 GETPOT INPUT PROCESSOR

GetPot is licensed under GNU Lesser General Public License 2.1, which allows extensions and redistributions. As such, LibMesh has incorporated the GetPot code and has made minor improvements. The GetPot block can contain comments, sub-blocks, and named parameters. The GetPot sub-block can also contain named parameters, comments, and nested sub blocks.

```
[block_name]
    # block content
    [./sub_block_name]
        # sub block content
        parameter_name = parameter_value
    [./]
[ ]
```

Figure 1. GetPot Block, Sub-Block, and Named Parameter Syntax.

The GetPot API allows for extraction of typed parameter values (integer, real, etc) for easy program integration. The input hierarchy is interpreted as sections, `block_name/sub_block_name`, which translates reasonably well into familiar Unix file hierarchies.

The GetPot input format is easy for users to learn, and the GetPot application programming interface (API) is easy to incorporate into C++ programs. However, GetPot input errors are not easily interpreted. Figure 2 depicts a typical syntax error message produced by a MOOSE application using GetPot.

```
Error: the following unidentified entries were found in your input
file:
word


*** ERROR ***
Your input file may have a syntax error, or you may have forgotten to
put quotes around a vector, ie. v='1 2'.
```

Figure 2. MOOSE application input syntax error message example.

The error message does include the word at fault, but it fails to provide the textual location, line, and column, of the word in the input file. This is not an issue if the fault is unique enough that a simple text search reveals the textual location. However, this does limit programmatic access to the textual location, preventing a user interface from helping facilitate resolution of the issue.

2.2 NEAMS WORKBENCH INPUT PROCESSOR

The NEAMS Workbench input processor uses lexicographical analysis—the Lexer—to present words. The Workbench uses the words' textual location to the input parser—the Parser—for semantic analysis. The result is a parse-tree data structure that represents all of the user-specified data, including hierarchy, regardless of correctness. Once the Lexer and Parser have produced a complete understanding of the user input, the validation engine can be invoked for a comprehensive validation check.

It is more complex to implement the Lexer and Parser than the GetPot implementation, but they provide a more robust capability that better facilitates users' input introspection and validation. The increase in program memory as a result of tracking textual location information is deemed worth the computational resources needed to allow effective error communication to the user.

The NEAMS Workbench extracts the textual location of blocks, sub-blocks, and named parameters from the parse-tree, and it presents quick-navigation items which use the textual locations to allow the user to jump to a section of interest. This is made possible by using the section's line and column. This is important regardless of the validity of the user's input, which could be under construction. The input is reprocessed as soon as the user has changed input, allowing quick-navigation items to remain current. Figure 3 depicts the NEAMS Workbench with two BISON input files open. Two input files are shown in the navigation panel, and their sections are listed. Selecting one of these sections will place the Workbench text editor's cursor at the beginning of that section. The section's textual location information is used to highlight where the section begins and ends.

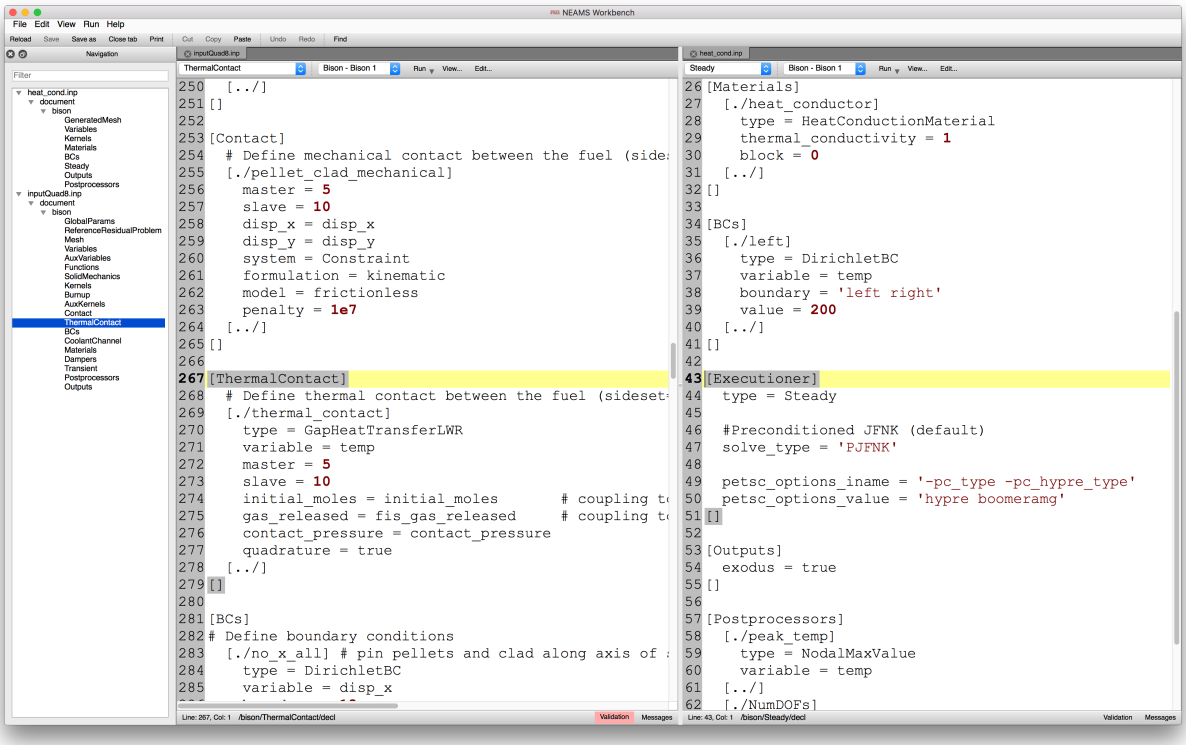

Figure 3. The NEAMS workbench MOOSE BISON input quick navigation.

The section information and textual location information are also used for input validation.

2.3 NEAMS WORKBENCH INPUT DEFINITION AND VALIDATION

The NEAMS Workbench Input Validation Engine supports 21 types of input definition rules for validation, but the MOOSE framework input only requires 8 validation rules. These 21 rules have been observed in previously supported input.

1. **MinOccurs** indicates the minimum occurrence of a component allowed in a context.
2. **MaxOccurs** indicates the maximum occurrence of a component allowed in a context.
3. **ValType** indicates the allowed type of a component (e.g., real, integer).
4. **ValEnums** indicates the allowed enumerated values of a component.
5. **MinValInc** indicates the minimum inclusive value of a component allowed.
6. **MaxValInc** indicates the maximum inclusive value of a component allowed.
7. **MinValExc** indicates the minimum exclusive value of a component allowed.
8. **MaxValExc** indicates the maximum exclusive value of a component allowed.
9. **ExistsIn** indicates the context set in which a component must exist.
10. **NotExistsIn** indicates the context set in which a component must not exist.
11. **SumOver** indicates the context in which a component must sum to a specified value.
12. **SumOverGroup** indicates the context and component group in which a component must sum to a specified value.
13. **ProdOver** indicates the context in which a component must multiply to a specified value.
14. **ProdOverGroup** indicates the context and component group in which a component must multiply to a specified value.

15. **IncreaseOver** indicates the content in which the components values must monotonically be increasing (weakly or strictly).
16. **DecreaseOver** indicates the content in which the components values must monotonically (weak or strict) be decreasing (weakly or strictly).
17. **ChildAtMostOne** indicates the set of sub section components from which at most one may exist.
18. **ChildExactlyOne** indicates the set of sub section components from which exactly one must exist.
19. **ChildAtLeastOne** indicates the set of sub section components from which at least one must exist.
20. **ChildCountEqual** indicates the set of sub section components from which the total number of occurrences must be equal to a specified number.
21. **ChildUniqueness i**ndicates the set of sub section components from which each component must be unique.

The MOOSE framework input currently only requires MinOccurs, MaxOccurs, ValType, ValEnums, MinValInc, MaxValExc, ExistsIn, ChildAtLeastOne, and ChildUniqueness. The minimum occurrence and maximum occurrence, value type, and value enumeration are rules that deal immediately with the component they are validating and are subsequently easy to understand and implement. The ExistIn, ChildAtLeastOne, and ChildUniqueness rules are more complex, requiring input context facilitated by an XPath [http://www.w3schools.com/xsl/xpath_syntax.asp]-like query mechanism which heavily relies on paths to components in the input as represented by the parse-tree.

The complete set of rules describing all input is stored in an input definition file commonly referred to as a *schema*. An ORNL-local MOOSE repository was extended to include a new command line option, *--definition*, that dynamically produces the schema file containing most information needed for input validation. The schema represents the same hierarchies that are available in the input with the addition of validation rules and metadata such as *component description*, *default*, *input type* and *template categorization*. The input type and template categorization are used to facilitate input auto-completion of the component from within the Workbench during file content creation. Figure 4 depicts a snippet of MOOSE BISON input definition illustrating the ValType, and ChildAtLeastOne rules and Description metadata field. The type of the `variable value` within the `DirechletBC` component must be a `String`. The ChildAtLeastOne rule indicates that the `variable` component could be specified in either the `GlobalParams`, relatively located at `"../../GlobalParams/variable/value,"` or the `DirechletBC` component as a subcomponent located at `"variable/value."`

```
GlobalParams{
…
    variable{ … }
…
} % end of GlobalParams
…
BCs{
…
    DirichletBC{
    …
        ChildAtLeastOne=[ "../../GlobalParams/variable/value"
                          "variable/value" ]
        variable{
           Description = "The name of the variable this boundary
condition applies to"
            value{
               ValType = "String"
            }
        } % end of variable
        …
    } % end of DirichletBC
…
} % end of BCs
```

Figure 4. MOOSE BISON input definition illustration.

The input validation engine uses these rules to provide immediate feedback to the user. Figure 5 depicts user feedback driven by the rules illustrated in Figure 4. Because the `variable` does not exist in the DirichletBC or GlobalParams block, a validation error is presented to the user which reads '`line:36 column:3 - Validation Error: DirichletBC has zero of: ["../../GlobalParams/variable/value" "variable/value"] - at least one must occur.`' Clicking this message in Workbench navigates the user's cursor to the location of the validation error, facilitating quick review and resolution. By defining the `variable` named parameter, the input definition is satisfied, and the validation error is no longer presented to the user. The validation error references line 36, column 3, which is the textual location of the sub-block containing the validation error.

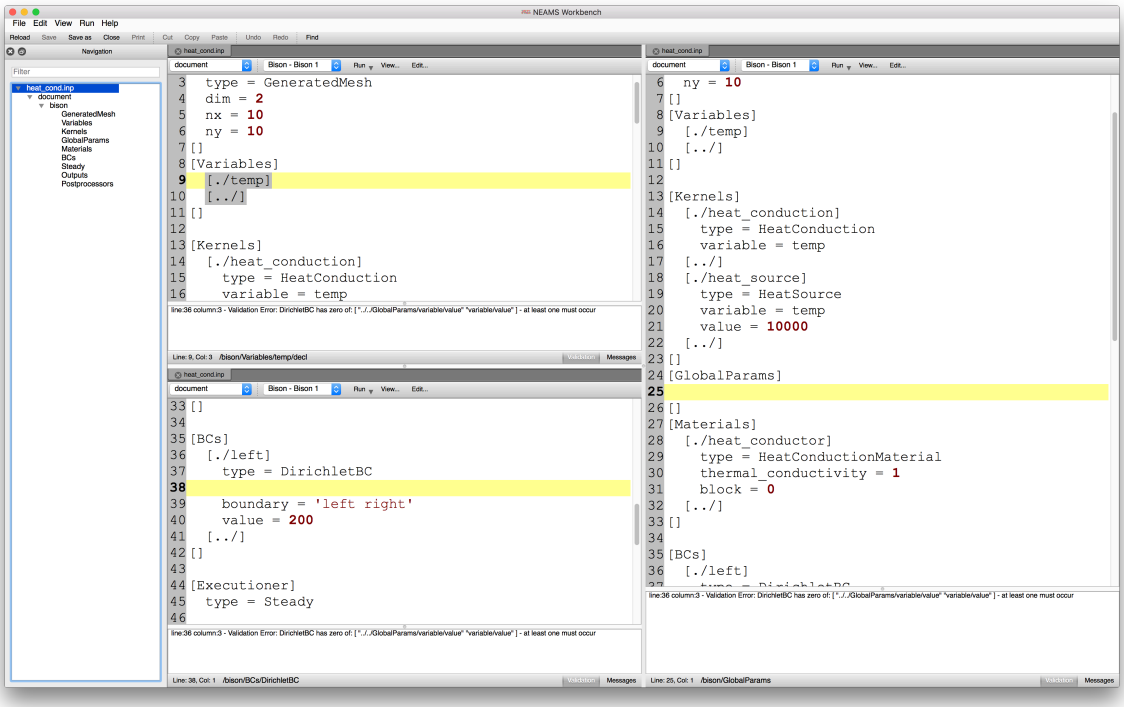

Figure 5. Workbench validation panel showing missing required input.

Figure 6 depicts resolution of the validation error by specifying the `variable` name parameter in the `GlobalParams` block.

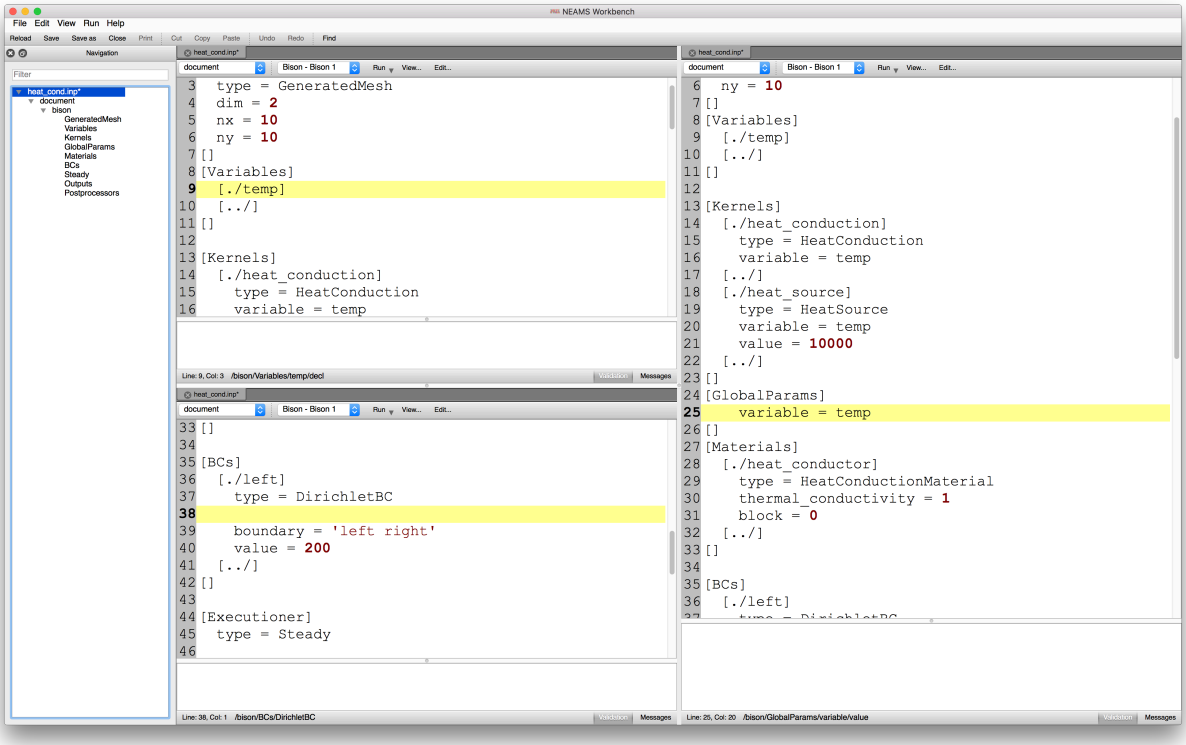

Figure 6. Workbench valid input with satisfied input definition.

2.4 NEAMS WORKBENCH MOOSE INPUT CREATION

At its most basic level, the Workbench is a text editor that makes input creation natural. However, the Workbench can jump start the user with added benefits such as auto-completion of input and input introspection. Input auto-completion uses a template construct in which the template contains placeholders for data of interest. For MOOSE input there is a minimum of three template constructs: the block, the sub-block, and the named parameter. These three templates could be presented as examples with text to be replaced by the user, minimizing the user's required input formatting or required syntax specification. Figure 7 depicts the simplest MOOSE block input template, which requires the user to edit *block* to be the appropriate block name. Figure 8 depicts the simplest sub-block template, again requiring the user to replace text, *sub_block* with the appropriate sub-block name. Figure 9 depicts the named parameter MOOSE input template, which requires the user replace both *parameter* and *value* with the appropriate name and value of the component.

```
[block]
[]
```

Figure 7. Simple MOOSE input block template.

```
[./sub_block]
[../]
```

Figure 8. Simple MOOSE input sub-block template.

```
parameter = value
```

Figure 9. Simple MOOSE input named parameter template.

The templates shown above aid the uninitiated user in legal syntax of the input, but they do little more. In Figure 9, the effort to replace all but one character slows the user. Preferred templates would contain the respective block, sub-block, and parameter, along with default values listing options available to the user for a given context. To do this, the input definition, in conjunction with the parse-tree and user's input cursor, are required. The user's input cursor provides the textual location, line, and column to Workbench, which can perform an input component lookup into the parse-tree. This lookup provides the input component of focus, subsequently providing the component's path to Workbench. Workbench uses the component's path to conduct the needed lookup into the input definition. The component's input definition communicates all possible input to the Workbench that is available to insert into the file at the user's cursor location. Figure 10, Figure 11, and Figure 12 illustrate the preferred, appropriately parameterized, block, sub-block, and named parameter templates. Workbench provides the `<InputName>` and `<InputValue>` attributes to satisfy the template.

```
[<InputName>]
[]
```

Figure 10. Parameterized MOOSE input block template.

```
[./<InputName>]
[../]
```

Figure 11. Parameterized MOOSE input sub block template.

```
<InputName>  =  <InputValue>
```

Figure 12. Parameterized MOOSE input named parameter template.

Figure 13 depicts a MOOSE BISON input with an auto-complete list presented to the user illustrating all available components at line 35. The auto-complete list incorporates the component's name and description, and Workbench uses the input definition for available components in conjunction with the

parse-tree to filter components as appropriate. For example, components that have already been specified, fulfilling the components maximum occurrence rule, are not available for additional insertion, as it violates the maximum occurrence rule.

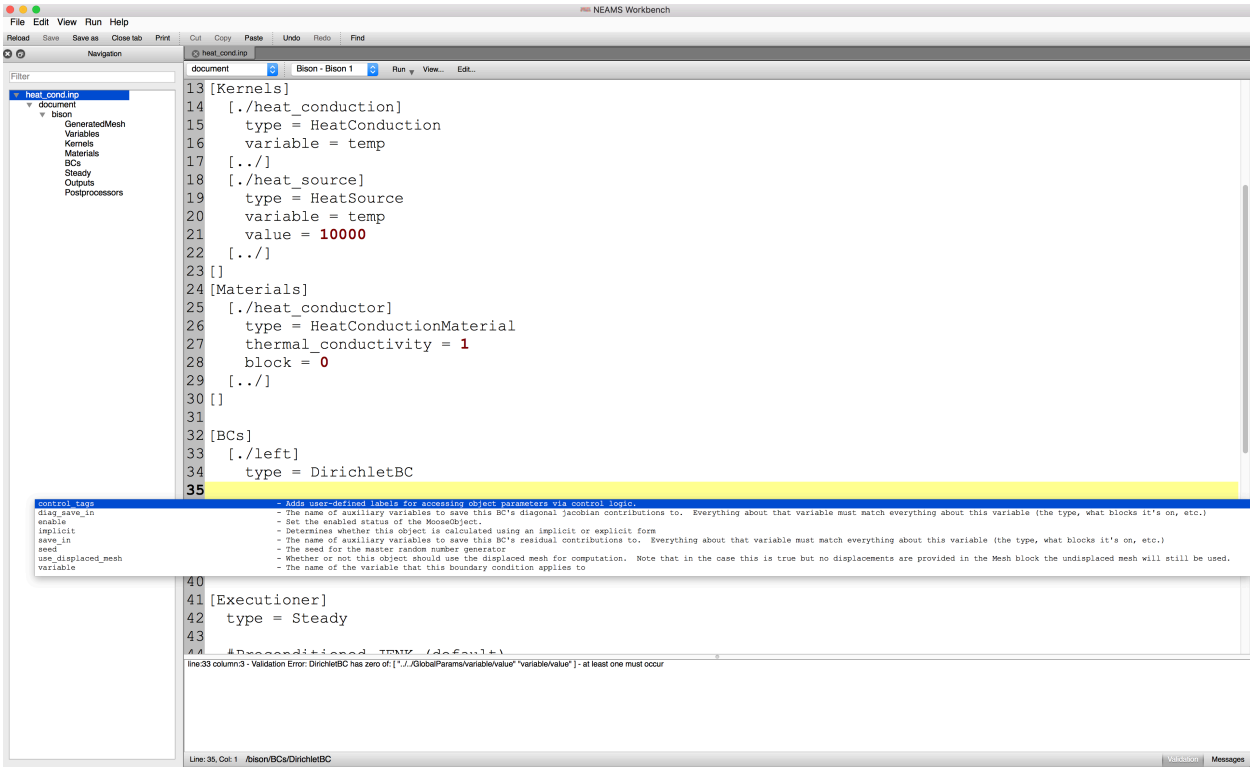

Figure 13. Workbench BISON input auto-completion.

Significant capabilities have been initiated to support MOOSE application input, and there are still areas to be improved, as well as user feedback and significant input features to be captured.

2.5 NEAMS WORKBENCH MOOSE INPUT CONSIDERATIONS

MOOSE currently uses the LibMesh extension of GetPot input processor. This input processor lacks the textual location information needed by Workbench to facilitate complex on-demand input validation. The Workbench input processor fulfills these needs, but it duplicates some functional capabilities. In the future, consolidation of these capabilities will facilitate MOOSE and Workbench development teams and will ensure that the user experience remains as consistent as possible.

During an August 2016 collaboration meeting between the MOOSE and Workbench development teams, some deficiencies were recognized in the MOOSE input infrastructure. Subsequently, the appropriate MOOSE development issues were opened and added to a NEAMS Workbench development milestone [https://github.com/idaholab/moose/milestone/4]. Resolution of these issues will facilitate generating a complete schema.

Generation of the input schema from the MOOSE application is an essential capability in facilitating Workbench's understanding of the MOOSE application. The contribution of the ORNL-local MOOSE repository's --definition feature back to the Idaho National Laboratory (INL) master MOOSE repository is needed to facilitate this capability.

The MOOSE framework greatly facilitates reuse and extension of input components, which allows accelerated application development. However, this has produced a focus on single MOOSE applications. This single application focus causes ambiguity as to which application can run the input. To facilitate the Workbench input generation, validation, and execution, a Unit of Execution construct was added, as depicted in Figure 14.

```
=application
<Application Input>
end
```

Figure 14. Workbench unit of execution.

Figure 15 illustrates the of the MOOSE BISON application's unit of execution within the Workbench.

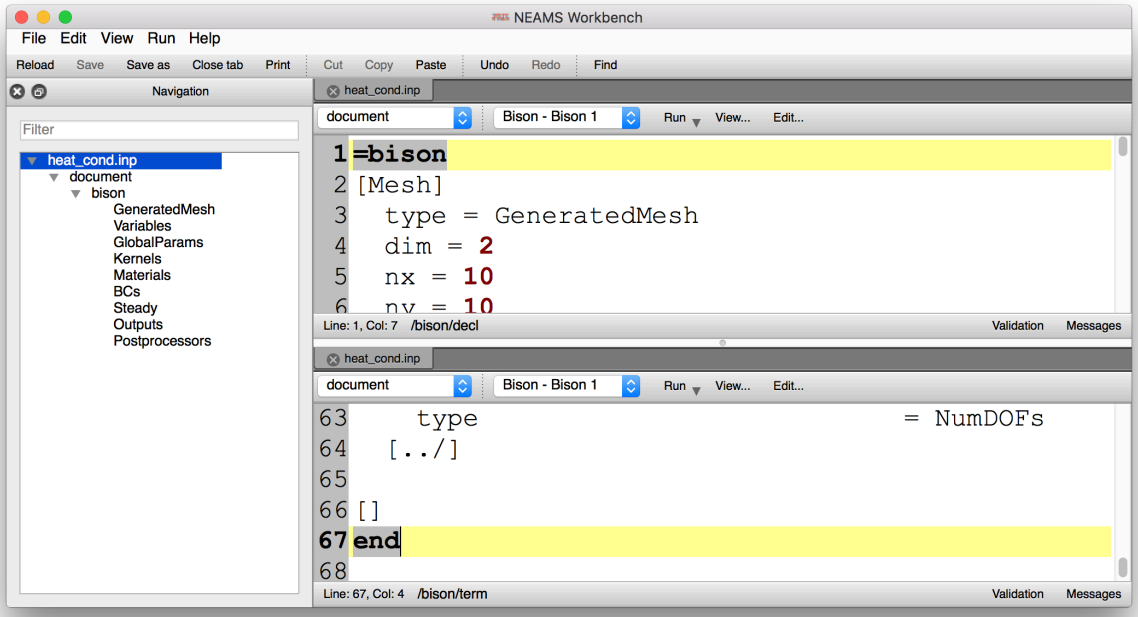

Figure 15. Example Workbench unit of execution for MOOSE BISON.

The unit of execution is accounted for by Workbench at the time of execution by the Workbench runtime environment.

3. WORKBENCH APPLICATION EXECUTION

A mission of the Workbench is to facilitate transition from conventional tools to high-fidelity tools. Many of these codes involve different means of invocation, and some have multiple means of invocation. To facilitate execution of these applications, a generic runtime environment interface was created. The intent of the runtime is to provide a consistent interface by which the Workbench can interact with each application in all necessary modes of operation (e.g., serial, parallel, and scheduled execution). The runtime is designed to provide a clean, convenient, consistent means for users to invoke NEAMS toolkit applications via the command line, separate from graphical user interface that is Workbench.

3.1 RUNTIME ENVIRONMENT

After careful consideration, the Python [www.python.org] scripting language was chosen for its combination of cross-platform support, object-oriented design, code readability, and power. The runtime is an object-oriented polymorphic design allowing reuse and extension. A generic base class provides the complete interface to communicate command line options and usage and to facilitate job setup, execution, and finalization of an application's input execution.

Some applications have no runtime environment, requiring the user to manually conduct all steps associated with running the applications. For example, the user might be required to copy the problem input file into a specific location with a specific file name, `input`,, and then invoke the application, thus producing temporary files and output file(s). This is error prone and should not be important to the user or to Workbench as this logic is application specific. The solution is to contribute a new runtime script—`new_app.py`—to the runtime repository, where the `new_app.py` overrides the setup, execution, or finalization logic as needed to fulfill the runtime interface. The setup logic might create a working directory, `TMPDIR`, and then copy the `problem.inp` into the `TMPDIR` as `TMPDIR/input`. The execution might invoke the application executable, passing application messages back to the calling application (command console, Workbench, etc.). The finalize logic might (1) combine the output files located in `TMPDIR` into logical order, (2) copy the output back into `problem.out`, residing next to `problem.inp`, and (3) delete the `TMPDIR` to clean up after itself.

Whatever the specific application logic is, there an application.py script will be available to allow consistent invocation and potentially providing great convenience for the application's typical command line user. As the runtime environment matures and additional features are added (queuing system, etc.) for the base class, all incorporated runtimes will benefit.

3.2 MOOSE BISON RUNTIME ENVIRONMENT

As of August 2016, the MOOSE BISON application, `bison-opt`, was understood to require its invocation to occur in the directory in which the input resides. Figure 16 depicts a typical invocation of the MOOSE BISON application.

```
/software/neams/moose/bison/bison-opt -i problem.inp
```

Figure 16. MOOSE BISON-OPT example invocation.

The `bison.py` runtime environment encapsulates the `bison-opt` command line options, the job setup, execution, and finalize logic with ~200 lines of Python. Most of these lines are propagating the dozens of command line options through the BISON runtime environment. Figure 17 depicts an example invocation of the BISON runtime environment. Note the complete path to the `problem.inp` input file. This is not only convenient, but it is also often considered standard.

```
/software/neams/rte/bison.py -i  /path/to/input/dir/problem.inp
```

Figure 17. MOOSE BISON runtime environment example invocation.

The Workbench can use this interface to allow users' invocation of multiple MOOSE application inputs in the same interface, as depicted in Figure 18.

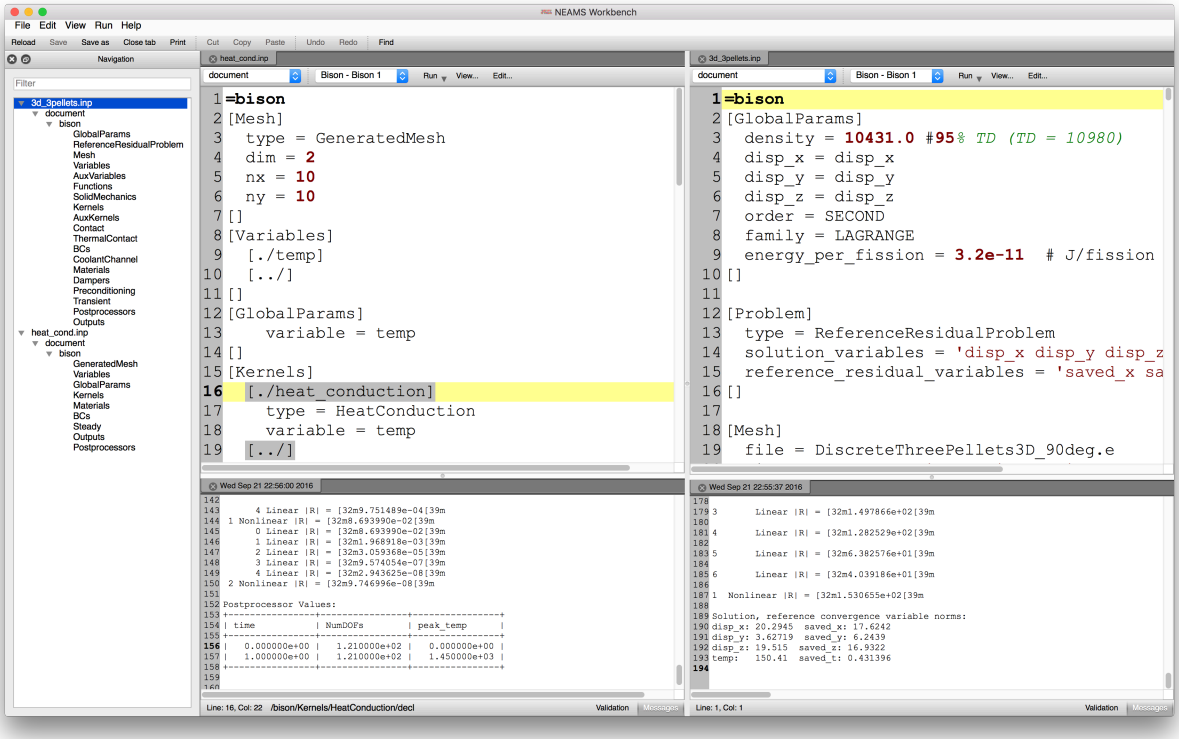

Figure 18. Workbench MOOSE BISON runtime invocation example.

3.3 FUTURE RUNTIME ENVIRONMENT CONSIDERATIONS

The runtime environment greatly facilitates the Workbench in its interactions with applications. Current interface has been shown to facilitate local execution of MOOSE BISON applications. With little additional work, more MOOSE applications could easily be added. More work is expected to integrate applications with little or no existing runtime environment. Finally, remote high performance clusters with potential scheduler/queuing interfaces must be designed and integrated to best facilitate large or long-running jobs.

4. VISUALIZATION INTEGRATION STATUS

The Workbench development team conducted a review of the LLNL VisIt visualization tool and Kitware Inc., ParaView visualization application APIs. Integration of both into the Workbench is desired. A complicating factor is the tools' common dependency on different versions of Kitware Inc.'s Visualization Tool Kit (VTK). A Workbench and VisIt collaboration meeting was held in August 2016 with Dr. Harinarayan Krishnan of Lawrence Berkeley National Laboratory (LBNL), and it was concluded with a prototype integration of the VisIt Viewer component in Workbench. A collaboration meeting with Bob O'Bara from Kitware, Inc., was held in late September 2016 to discuss the Kitware software solutions, including the ParaView visualization application.

In addition to the review of the VisIt and ParaView APIs, the MOOSE BISON data and visualization methods were reviewed with staff in CASL and the MOOSE development team. Integration of VisIt and ParaView and streamlining results acquisition and visualization are areas for continued work in FY17.

5. SUMMARY

The NEAMS Workbench is a new initiative for the 2016 fiscal year. It is intended to facilitate the transition from conventional tools to high-fidelity tools by providing a common user interface for model creation, review, execution, and visualization for integrated codes. In FY16, the Workbench development team collaborated with Cody Permann and the MOOSE development team at INL to integrate the initial pilot application, MOOSE BISON, into the Workbench. MOOSE BISON input can be created, reviewed, and executed on a local machine. Opportunities for input infrastructure collaboration, improvement in input validation, and results visualization of MOOSE applications still remain and will be addressed in FY17. Due to the design of the MOOSE framework, with little additional effort, additional MOOSE applications can easily be integrated into the Workbench, leveraging the features accomplished in FY16.

To facilitate the broad set of applications to be integrated into the Workbench, an object-oriented, polymorphic Python runtime environment was initiated. This runtime environment provides a consistent interface by which the Workbench or command line user can invoke integrated applications. Additionally, the runtime environment provides a central location by which application invocation conveniences (queuing support) will be centralized.

Visualization integration is still being investigated. In FY16, collaboration meetings were held involving the VisIt visualization tool attended by Dr. Harinarayan Krishnan of LBNL, and ParaView attended by Bob O'Bara of Kitware, Inc. The MOOSE BISON data and visualization methods were reviewed with staff members from CASL and the MOOSE development team. There are opportunities to incorporate existing VisIt and ParaView visualization capabilities into Workbench. Custom VTK capabilities can also be implemented to streamline data analysis workflows that are frequently observed in the application space.