Preliminary Integration of MCNP6 and PROTEUS into the NEAMS Workbench

Kurt A. Dominesey, Matthew D. Eklund, Peter J. Kowal, Wei Ji

Department of Mechanical, Aerospace, and Nuclear Engineering
Rensselaer Polytechnic Institute: 110 8th Street, Troy, NY, 12180
domink@rpi.edu; jiw2@rpi.edu

I. INTRODUCTION

The DOE Nuclear Energy Advanced Modeling and Simulation (NEAMS) Program has in recent years produced many advanced tools for the simulation of nuclear energy systems. To coordinate the use of these tools, the NEAMS Integration Product Line (IPL) has created the Workbench\(^1\), which seeks to present a uniform user interface for these nuclear reactor simulation codes, with convenient features for the creation and validation of modeling input and powerful visualization capabilities to process simulation results. To date, several tools have been incorporated into the Workbench, including the Argonne Reactor Computation (ARC) codes, MOOSE-BISON and the Design Analysis Kit for Optimization and Terascale Applications (DAKOTA). To augment the utility provided by these tools, (and in collaboration with Terascale Applications (DAKOTA). To augment the utility provided by these tools, (and in collaboration with the NEAMS Workbench developers) two high-fidelity neutron transport solvers, namely, MCNP6\(^2\) and PROTEUS\(^3\), are currently being integrated into the Workbench, an effort supported by the DOE NEAMS Program. This summary presents the preliminary integration design for both neutronics solvers. Crucially, a common “Workbench model” has been established to enable the convenient translation of models across multiple codes (and their respective domain-specific modeling languages, DSMLs). Thus, users can easily solve the same problem in different analysis codes without the need to repeatedly re-develop the model input for each individual code.

II. MCNP6 INTEGRATION DESIGN

MCNP6 is a Monte Carlo radiation transport code developed by Los Alamos National Laboratory. MCNP6 geometry is primarily created and represented using the constructive solid geometry (CSG) approach, though instances of unstructured mesh geometry (specified in the Abaqus input file format), can be embedded within CSG “pseudo-cells”, a feature introduced in MCNP6.1.1.

II.A Input Integration

Like MCNP6, the NEAMS Workbench employs a DSML to specify the problem geometry and material definitions. This NEAMS Workbench DSML is a subset of the Standard Object Notation (SON\(^1\)) language and will hereafter be referred to as the “Workbench DSML”.

For the purpose of integrating MCNP6 into the NEAMS Workbench, a “black-box” strategy is adopted (similar to the strategy adopted for ARC integration). That is to say, rather than modifying the source code of MCNP6 to parse and interpret the Workbench DSML, we instead seek to translate Workbench DSML files into a corresponding MCNP6 input deck, which can be consumed by MCNP6 without modification.

However, it is clear that a direct conversion (by textual manipulation only) from the Workbench DSML to the MCNP6 DSML would be onerous due to the notable syntactic differences, and inextensible to future DSMLs. A preferable approach is to deserializer the input DSML into a “Workbench model” (which is to say, a common model represented in an object-oriented programming language) which can then be reserialized to the user’s desired DSML. Beyond offering a more organized paradigm for DSML translation, this approach confers many appreciable benefits. Firstly, an intermediate model can serve as a “compatibility layer”, as it divides the DSML translation (i.e. Workbench DSML to MCNP6) into two independent processes: deserialization (Workbench DSML to common model) and reserialization (common model to MCNP6 DSML). Considering the case where Workbench has \(m\) output DSMLs, and \(n\) input DSMLs (\(n_m\) of which are also output DSMLs), this implies that Workbench developers would be required to implement \(nm+m\) translations in the absence of a common model, while only \(n+m\) translations (\(n\) deserializations and \(m\) serializations) would be required if a common model is employed, greatly reducing the developmental burden as the number of Workbench-supported DSMLs increases. Moreover, this development effort also becomes considerably simpler, as a well-designed object-oriented model promotes the re-use of existing classes to model similar concepts encountered across multiple DSMLs (e.g. materials, universes, CSG). Furthermore, the availability of an application program interface (API) to interact with a model programmatically may be of great utility to Workbench users already familiar with object-oriented programming languages.

To summarize the desired input workflow, a schematic view is provided in Figure 1 (with Workbench and MCNP6 as the desired input DSMLs, and MCNP6 and PROTEUS as the targeted analysis codes). It should be noted that while the user is only required to provide one form of input (written in purple), the user will be enabled to run any supported analysis code (written in
To function as this crucial common model, this work extends the open-source (permissively licensed) OpenMC\textsuperscript{1} Python module (which is used to create models for the OpenMC Fortran solver in the form of XML documents) to interpret the Workbench DSML and subsequently to prepare properly formatted MCNP6 decks. Where appropriate, this extension also includes the addition of relevant MCNP6 options not included in the OpenMC API, such as cell-wise “importance” values for variance reduction. However, this development does not impair compatibility with the OpenMC solver, as these options are not propagated to the XML files which it consumes. For the purpose of clarity, and to highlight their commonality across all Workbench-supported codes (not specifically OpenMC), these extended OpenMC Python models will hereafter be referred to as “common models” or “Workbench models”. Despite this, it should be noted that these models are, in actuality, valid OpenMC models, and can be used and manipulated in the same fashion that OpenMC users would expect.

\textbf{II.A.1. Workbench DSML to MCNP6}

In translating the Workbench DSML to the MCNP6 DSML, the initial step is to parse the Workbench DSML file, which is achieved using a parser from the Workbench Analysis Sequence Processor (WASP) project. This WASP parser is used to convert the Workbench DSML file into an XML document, which is then parsed and converted into JavaScript Object Notation (JSON). The JSON model is finally converted into a Workbench model by converting each Workbench JSON object into the appropriate Workbench object (such as an instance of the cell, material, or surface classes defined in the OpenMC Python module). Following this step, all cross-references are resolved (for example, associating cells with the materials that fill them, or lattices with the universes that fill each lattice element).

Once a valid Workbench model has been created, serialization to an MCNP6 deck is straightforward; each Workbench object is simply written out as a string (or series of strings) which represents the analogous construct as a card in the MCNP6 DSML. These cards are then organized to form the MCNP6 input deck.

To demonstrate this Workbench DSML to MCNP6 translation, a simple model has been created in the Workbench DSML consisting of a 5x5 lattice of fuel pins, immersed in water and surrounded by a cylindrical tank. This problem geometry (having been converted to an MCNP6 input deck and rendered by MCNP6’s plotting utility, PLOT) is displayed below in Figure 2.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2.png}
\caption{Simple 5x5 lattice model, as translated from the Workbench DSML.}
\end{figure}

While simple, this model is sufficient to demonstrate that the Workbench DSML can be translated to the MCNP6 DSML, and that the resulting input deck can be consumed by MCNP6 without modification by the end-user (or changes to the MCNP6 source code).

\textbf{II.A.2. OpenMC (Python) to MCNP6}

As an OpenMC Python model is employed as the Workbench model (the common model created as an intermediate step in translating from the Workbench DSML to the MCNP6 DSML), it follows that models created directly through the OpenMC Python API can also be converted to the MCNP6 DSML. In fact, this conversion is actually simpler, as the model is constructed directly in Python, and does not need to be deserialized from a textual DSML.

To demonstrate the capability to serialize OpenMC models to MCNP6 input decks, two models prototypical of light water reactor (LWR) geometry are presented. Both models are examples provided in the 0.9.0 release of OpenMC. The first is a square 17x17 fuel pin assembly with reflecting boundary conditions, taken from the Benchmark for Evaluation and Validation of Reactor Simulations (BEAVRS\textsuperscript{2}), shown (at left) in Figure 3. The second model is the Organisation for Economic Co-operation and Development (OECD) Nuclear Energy Agency (NEA) Monte Carlo Performance Benchmark. This benchmark is, in short, a simplified pressurized water reactor (PWR) containing 241 identical fuel
assemblies. Top-down and side views of the reactor geometry can be seen in Figure 3 (at center and at right, respectively).

Fig. 3. BEAVRS 17x17 fuel assembly (left), top-down (center) and side view (right) of the OECD NEA Monte Carlo Performance Benchmark (from axial and radial midplanes, respectively).

II.A.3. MCNP6 to MCNP6 (via Workbench Model)

While the ability to construct a Workbench model from the Workbench DSML (or to use OpenMC models directly) may be sufficient for certain Workbench users, it is expected that many potential users are already adept in the MCNP6 DSML and may wish to continue using the MCNP6 models they have already created. To this end, the Workbench model (i.e. an extended OpenMC Python model) has been further extended to also create Workbench models directly from the MCNP6 DSML. As opposed to the SON to XML to JSON pathway implemented for the Workbench DSML to common model conversion, the MCNP6 to common model conversion is achieved by matching MCNP6 cards with regular expressions, where the groups of the regular expressions are named in accordance with the keyword parameters passed to the appropriate constructor in the model API. The obvious implication of this feature (beyond facilitating the migration of established MCNP6 users) is that all Workbench-integrated codes which employ this Workbench model can, using this capability, access the extensive MCNP6 verification and validation suite for code-to-code validation. While progress in reading the MCNP6 DSML is preliminary, limited success has been achieved; to demonstrate this progress, Figures 4 displays two models (Tinkertoy-2 and Zeus-2, defined in the International Criticality Safety Benchmark Evaluation Project, or ICSBEP, Handbook) which have been successfully imported from the MCNP6 criticality validation suite.

Fig. 4. Tinkertoy-2 model (highly enriched uranium cylinders in a paraffin box, left) and Zeus-2 model (highly enriched uranium platters moderated by graphite and reflected by copper, right).

II.B Output Integration

Upon execution of an MCNP6 model, several output files are created, many of which are updated continuously during code execution. For post-processing of complete output files, the Workbench will employ the MCNPTools Python module (created by the MCNP6 development team), which allows for the deserialization of MCNP6 output files into Python data objects. Where appropriate, these deserialized outputs will also be used to construct the analogous OpenMC outputs (for example, instances of the OpenMC “statepoint” and “tally” classes).

However, as MCNPTools is only intended to operate on complete output files, an alternative approach of “fuzzy parsing” (parsing only selected portions of output) using Another Tool for Language Recognition (ANTLR) will be adopted for the real-time post-processing of textual file-streams. It should also be noted that while the current development effort considers only MCNP6 and PROTEUS specific output files, this “fuzzy parsing” strategy is expected to be appropriate for any analysis code with lengthy textual output files. Following the extraction of this data, ANTLR “listener” functions (functions which are called each time an ANTLR parser encounters a given parser rule) will be used to provide further post-processing, such as creating interactive plots, or reserializing the data to HDF5 datasets. As it is expected that end-users would like to customize the content and organization of these plots and tables, a domain-specific language (DSL) for Workbench post-processing will be developed to specify the format of these products. Additionally, while plots and HDF5 files are currently considered the primary end-products of Workbench post-processing, future efforts could extend this DSL to support other products, such as HTML webpages or LaTeX documents.

III. PROTEUS INTEGRATION DESIGN

PROTEUS is a massively parallel high-fidelity neutron transport code developed at Argonne National Laboratory (ANL) as part of the NEAMS Reactor Product Line (RPL).

III.A Input Integration

Given that PROTEUS employs a finite-elements solution to the neutron transport equation, code execution requires a meshed geometry. However, both the Workbench and MCNP6 DSMLs use the CSG approach to define problem geometry. In order to execute models defined in these Workbench-supported DSMLs using PROTEUS, a guided (or, preferably, automatic) mesh...
 generation utility for CSG models will be developed. Given the widespread usage of Cubit and Trelis to create meshes used in PROTEUS, it is probable that this utility also utilize Cubit and Trelis in this capacity. Automation (or near-automation) of this process will be achieved by effective use of the Cubit/Trelis Python API, as well as the programmatic conversion of Workbench CSG primitives to their Cubit/Trelis counterparts.

Beyond the problem geometry, users must also provide PROTEUS with a multi-group cross-section file (in the ISOTXS or ANLXS format), a material assignment file (which defines and assigns materials for each region of the mesh), and an options file. Therefore, to achieve PROTEUS integration, the ANL-developed GenISOTXS utility (which generates ISOTXS-formatted cross-sections from Serpent and OpenMC outputs) will be extended to process MCNP6 outputs, and utilities will be established to generate material assignment files from models annotated in Kitware’s Simulation Modeling Tool Kit (SMTK) and options files from appropriate Workbench DSML files. Users will also be provided the option to use the Workbench-integrated MC²-3 code in lieu of a GenISOTXS-supported solver.

III.B Output Integration

Like MCNP6, PROTEUS also generates a textual log file, and (optionally) a textual “edits” file, which specifies the rate of certain nuclear interactions within each mesh region. For these outputs, the “fuzzy parsing” approach used for MCNP6 post-processing will again be employed.

Additionally, PROTEUS also generates HDF5 files (of the same format produced by the UNIC code) which specify the flux distribution over the mesh. While this UNIC format is natively supported by VisIt, it is expected the user may still wish to generate custom plots and (possibly re-organized) HDF5 datasets from these UNIC-formatted outputs. To this end, utilities will be established such that users can again use the Workbench post-processing DSL to create plots and tables from UNIC-formatted HDF5 files.

IV. CONCLUSIONS

In conclusion, the preliminary design for the integration of MCNP6 and PROTEUS into Workbench has been demonstrated. Notably, a common “Workbench model” has been established extending the functionality of the OpenMC Python model to deserialize the Workbench and MCNP6 DSMLs, and to serialize to the MCNP6 DSML. Translation of Workbench DSML files into MCNP6 input decks has been demonstrated using a simple problem comprising a 5x5 fuel lattice in a cylindrical tank, and the translation of OpenMC models into MCNP6 input decks has been shown for the BEAVRS 17x17 assembly and the OCED/NEA Monte Carlo Performance Benchmark. Furthermore, the ability to translate MCNP6 input decks to Workbench models (and subsequently back to MCNP6 input decks) has been demonstrated for two models taken from the MCNP6 criticality validation suite (namely the Tinkertoy-2 and Zeus-2 models defined in the ICSBEP Handbook). Future efforts will seek to further integrate MCNP6 and PROTEUS into the NEAMS Workbench by developing a guided meshing utility to prepare PROTEUS geometry and by establishing MCNP6 and PROTEUS specific grammars for the “fuzzy parsing” of textual outputs.

ACKNOWLEDGMENTS

The authors of this paper would like to thank Robert Lefebvre and Nicolas Stauff for granting pre-release access to the NEAMS Workbench and the ARC integration module (PyARC), as well as their counsel throughout the development process.

This material is based upon work supported by the Department of Energy Office of Nuclear Energy under Award Number DE-NE0008707.

REFERENCES