ORNL/TM-2015/532

Warthog: A MOOSE-Based Application for the Direct Code Coupling of BISON and PROTEUS (MS-15OR04010310)

Alexander J. McCaskey
Stuart Slattery
Jay Jay Billings

September 2015

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

ORNL/TM-2015/532

Computer Science and Mathematics Division

Warthog: A MOOSE-Based Application for the Direct Code Coupling of BISON and PROTEUS (MS-15OR04010310)

Alexander J. McCaskey
Stuart Slattery
Jay Jay Billings

Date Published: September 2015

Prepared by
OAK RIDGE NATIONAL LABORATORY
P.O. Box 2008
Oak Ridge, Tennessee 37831-6285
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

EXECUTIVE SUMMARY

The Nuclear Energy Advanced Modeling and Simulation (NEAMS) program from the Department of Energy's Office of Nuclear Energy provides a robust toolkit for the modeling and simulation of current and future advanced nuclear reactor designs. This toolkit provides these technologies organized across product lines: two divisions targeted at fuels and end-to-end reactor modeling, and a third for integration, coupling, and high-level workflow management. The Fuels Product Line and the Reactor Product line provide advanced computational technologies that serve each respective field well, however, their current lack of integration presents a major impediment to future improvements of simulation solution fidelity. There is a desire for the capability to mix and match tools across Product Lines in an effort to utilize the best from both to improve NEAMS modeling and simulation technologies.

This report will detail a new effort to provide this Product Line interoperability through the development of a new application called *Warthog*. This application couples the BISON Fuel Performance application from the Fuels Product Line and the PROTEUS Core Neutronics application from the Reactors Product Line in an effort to utilize the best from all parts of the NEAMS toolkit and improve overall solution fidelity of nuclear fuel simulations. To acheive this, Warthog leverages as much prior work from the NEAMS program as possible, and in doing so, enables interoperability between the disparate MOOSE and SHARP frameworks, and the libMesh and MOAB mesh data formats.

The remainder of this report will describe this work in full. We will begin with a detailed look at the individual NEAMS framework technologies used and developed in the various Product Lines, and the current status of their interoperability. We will then introduce the Warthog application: its overall architecture and the ways it leverages the best existing tools from accross the NEAMS toolkit to enable BISON-PROTEUS integration. Furthermore, we will show how Warthog leverages a tool known as DataTransferKit to seamlessly enable the transfer for solution data between disparate frameworks and mesh formats. To end, we will demonstrate tests for the direct software coupling of BISON and PROTEUS using Warthog, and discuss current impediments and solutions to the construction of physically realistic input models for this coupled BISON-PROTEUS system.

INTRODUCTION

The Nuclear Energy Advanced Modeling and Simulation (NEAMS) program from the Department of Energy's Office of Nuclear Energy has successfully endeavored to provide a usable and advanced toolkit for the modeling and simulation of nuclear fuels and reactors. This effort has resulted in two comprehensive frameworks that developers can build upon to provide advanced simulation technologies for various aspects of a running nuclear reactor. Over the years of this development, the program has evolved into an organizational structure that is based on *Product Lines* for Fuels (Fuels Product Line, FPL), Reactors (Reactors Product Line, RPL), and Integration (Integration Product Line, IPL), with each providing, or using, a developed framework that best fits its needs. In a sense, the development of these frameworks has approached the same problem from two different perspectives: the Multiphysics Object Oriented Simulation Environment (MOOSE) from Idaho National Laboratory (INL) provides a multiphysics framework with a 'top-down' development approach, while the SHARP Nuclear Reactor Framework from Argonne National Laboratory (ANL) and the RPL provides a multiphysics framework with a focus on 'bottom-up' development. Both of these are equally valid approaches to the problem of multi-physics, coupled simulations of nuclear technology, but their disparate development has resulted in framework components that are difficult to mix and match. Components built on top of MOOSE are difficult to integrate and use in SHARP, and vice versa.

There is a strong desire in the NEAMS community to see this communication and coupling barrier between disparate framework components broken, and in doing so, enable the coupled use of the best components from one framework with those from the other. A specific example of this desire is for improved neutronics modeling in multi-physics fuels performance modeling. NEAMS has invested heavily in both a new fuels performance application called BISON (an application built on top of the MOOSE framework), and in a core neutronics application called PROTEUS (a stand-alone application utilized in the SHARP framework). Both of these applications represent the state-of-the-art in their representative fields and are major investments for the program. Currently, BISON lacks high-fidelity fission source feedback, relying primarily on experimentally-validated power profile piece-wise functions to stand in for a high-fidelity neutronics power calculation. It would be hugely beneficial to couple BISON to the best-from-SHARP core neutronics application, PROTEUS, but the mechanism for doing so is difficult and complex because both are based on incompatible frameworks. The purpose of this work is to develop the software tools necessary to efficiently and directly couple BISON and PROTEUS. This work will enable the interoperability of components from MOOSE/libMesh with components from SHARP/MOAB.

This report will demonstrate a viable, and efficient mechanism for this cross-framework coupling necessary to further increase overall solution fidelity from NEAMS toolkit components. We will detail a new MOOSE-based application called Warthog that wraps a PROTEUS neutronics calculation and maps solution data from PROTEUS to MOOSE/BISON through a solution-transfer library from Oak Ridge National Laboratory called DataTransferKit (DTK). To start, this report will discuss the individual frameworks in NEAMS, and the BISON and PROTEUS components. Then we will detail what DTK is and how it can be used in concert with a given application for solution transfer between differing mesh discretizations, as well as incompatible mesh libraries and data structures. We will then detail Warthog, and discuss how it leverages DTK and a wide array of existing NEAMS technologies to successfully couple BISON and PROTEUS into a cohesive whole. We will end with preliminary software coupling results and a look at future work geared at utilizing Warthog for physically realistic reactor models.

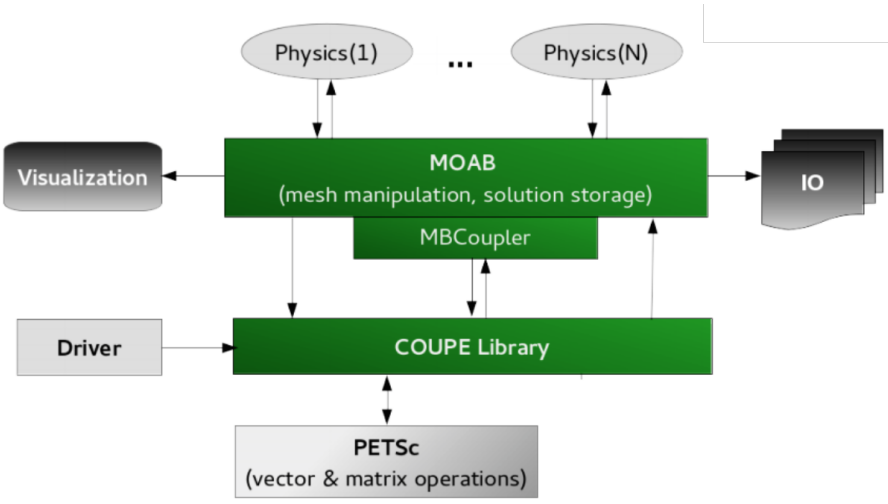


Fig. 1. Overall SHARP design, showing a strong dependency on the MOAB mesh database to promote coupling between disparate physics modules.

NEAMS FRAMEWORK TECHNOLOGIES

SHARP

The SHARP Nuclear Reactor Multiphysics framework is a tool designed and developed primarily at Argonne National Laboratory (ANL) that provides a 'bottom-up' multi-physics framework coupling strategy: it focuses on coupling *existing* verified and validated nuclear reactor physics codes. It provides a framework for total reactor core simulations by coupling existing thermal hydraulics, core neutronics, and structural mechanics physics codes. In its current state, SHARP couples NEK5000 [7], PROTEUS [17], and DIABLO [6] for thermal hydraulics, neutronics, and structural mechanics, respectively.

The difficulty inherent to coupling existing physics codes is two-fold: (1) each physics code relies on internal mesh data structures that do not easily map to other physics codes, and (2) each physics potentially operates on different spatial discretizations, which makes it difficult to exchange solution data in an efficient manner. To overcome these difficulties, SHARP relies on a coupling library called CouPE (Coupled Physics Environment) [15], developed as part of the Integration Product Line, to control and drive the overall multiphysics solve and handle solution transfers between physics applications. CouPE provides an intermediate mesh representation built on top of MOAB (Mesh-Oriented datABase) [19] to efficiently map each physics code's internal mesh representation to the other physics code in a given coupled solve. In a given iteration, CouPE executes a physics code and transfers its solution data to the intermediate MOAB instance, which is then pushed to the next physics code in the coupled solve. To overcome potential differing spatial representations, CouPE uses MBCoupler, a tool built on top of MOAB that interpolates solution data between different mesh discretizations. See Figure 1 for a high-level view of this architecture.

In this manner, SHARP is able to successfully couple existing physics codes and execute large multiphysics, coupled solves for advanced nuclear reactors. The burden of the work is simply the development of a MOAB adapter code that takes each physics code's mesh representation to MOAB and vice versa.

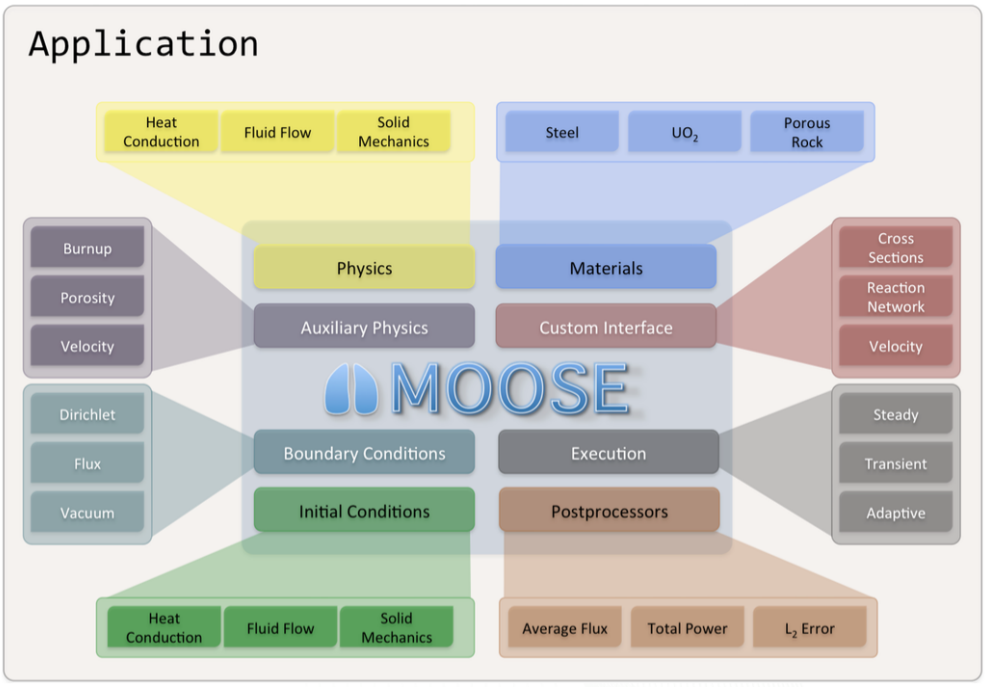


Fig. 2. High-level view of the MOOSE architecture [13]. MOOSE provides extensible sub-systems for physics components, boundary conditions, materials, and analysis tools, just to name a few.

MOOSE

The Multiphysics Object Oriented Simulation Environment (MOOSE) is primarily developed and maintained at Idaho National Laboratory (INL) under the NEAMS Integration Product Line. MOOSE provides a 'top-down' framework for the development of coupled multiphysics applications based on the finite element and Jacobian Free Newton Krylov (JFNK) methods in a quick and efficient manner [8]. MOOSE is written in C++ and relies heavily on advanced object-orientation to provide a large collection of extensible systems that perform tasks such as executing an overall solve, reading and writing meshes, declaring pieces of physics, and defining materials, just to name a few. See Figure 2 for a high-level view of MOOSE's extensible systems. The overall goal of MOOSE is to provide the tools necessary that allow the research to plug-and-play different physics, materials, solve types, etc. for a given system, and in doing so, facilitate overall research in a quick and efficient manner. To properly understand how Warthog leverages as much as possible from the MOOSE framework, we will now highlight a few objects in the framework's object heirarchy that Warthog builds on top of to enable BISON and PROTEUS coupling.

At the root of MOOSE's object inheritance tree is the *MooseObject*. All extensible systems in MOOSE inherit from this object, and in doing so, pick up a very useful property: the ability to be dynamically created from a string declaration in an input file using MOOSE's custom factory pattern. This enables the overall extensibility of MOOSE as it allows users to declare new sub-systems through a simple ASCII text declaration in an input file.

Primary amongst MOOSE's extensible systems is the *MooseApp* class. Users of MOOSE who wish to

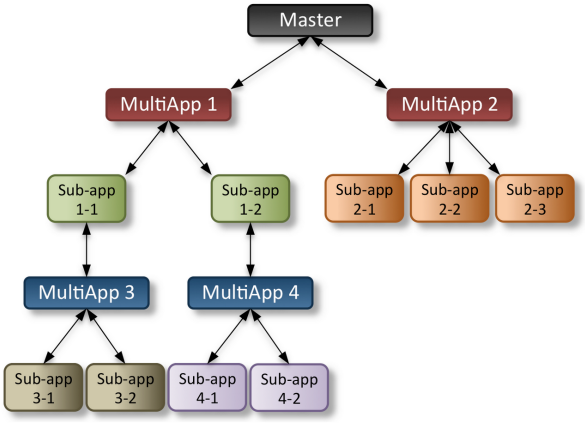


Fig. 3. A view of the tree-like design of MOOSE's MultiApp system [12]. Each MultiApp solve consists of a master application that drives the overall solve, with sub-applications contributing solutions of individual physics of interest.

create a new physics application simply create a new MooseApp subclass for their MOOSE-based physics application. For example, BisonApp is a subclass of MooseApp for the BISON Fuel Performance MOOSE-based application. Each new MooseApp can register or declare extensions for other parts of the MOOSE framework, such as the registration of new physics *Kernels* or *Material* properties. Another key extensible system in MOOSE is the *Executioner* class, which acts to direct the entire solve for a given multiphysics simulation. There are a number of different types of default Executioners: *Steady* for steady-state solves, *Transient* for time-dependent solves, and *PetscTSExecutioner* (for use of the PETSc TS object in MOOSE) are just a few.

MOOSE provides an extensible system for coupling disparate physics codes that are provided as MOOSE-based applications. The *MultiApps* system lets users of MOOSE define a simulation that uses physics contributions from any number of existing MOOSE-based applications, for example coupling BISON to the MARMOT microscale fuel modeling code. This system is designed as a N-ary tree of MooseApps, with the root being the application executing the master solve (Fig. 3). This tree-like structure enables MOOSE to execute solves using loosely-coupled operator split or tightly-coupled picard iteration methods. In addition to the MultiApp system, MOOSE provides a corresponding *Transfers* system that enables the transfer of solution fields between MultiApp solves. This system is extensible, and provides transfer algorithms like L_2 projection and interpolation by default [9].

With the above mentioned extensible systems, and many more, MOOSE enables the rapid development of finite-element JFNK coupled, multiphysics applications. To standup a new application, one simply creates a new MooseApp, registers any new objects created for the simulation (new physics Kernels, Materials, etc., and of course, one could use any of the many existing objects), constructs the input file for a given execution, then start using that application.

WARTHOG

The goal of the Warthog application is two-fold: (1) to enable the direct code coupling of BISON and PROTEUS to improve overall solution fidelity, and (2) to leverage as much existing work as possible from the extensive NEAMS toolkit into one cohesive whole. This second point is important, the work done over the years by NEAMS developers has been tremendous, and relying on that work when possible efficiently facilitates the rapid development of any potential BISON-PROTEUS coupled application.

There is a strong desire in the NEAMS community to see the direct code coupling of PROTEUS and BISON. First and foremost, this coupled capability will provide an accurate power density feedback for the overall BISON thermomechanics solve, therefore improving overall solution fidelity. Besides this, the desire to see these applications coupled also stems from the desire to see the best from the Reactors and Fuels Product Lines working in concert to provide a rich tool for nuclear fuel studies. The program benefits immensely from the connection of these disparate and separate efforts as it shows that tools from MOOSE and SHARP/CouPE can be interoperable.

Aside from relying on the BISON and PROTEUS applications themselves, Warthog builds on top of a number of existing technologies from NEAMS. Warthog utilizes the MOAB, libMesh, MOOSE, and the primarily CASL-developed DataTransferKit. The remainder of this section will detail how each of these technologies enables Warthog's direct code coupling of BISON and PROTEUS.

BISON Fuel Performance

The BISON Fuel Performance application is a MOOSE-based application authored primarily at Idaho National Laboratory. It is a C++, parallel solver for coupled thermomechanics and species diffusion in nuclear fuel rods. BISON seeks to solve the following equations in a fully-coupled manner:

$$\rho C_p \frac{\partial T}{\partial t} + \nabla \cdot \mathbf{q} - e_f \frac{dF}{dt} = 0, \tag{1}$$

$$\frac{\partial C}{\partial t} + \nabla \cdot \mathbf{J} + \lambda C - S = 0, \tag{2}$$

$$\nabla \cdot \sigma + \rho \mathbf{f} = 0, \tag{3}$$

where the first equation models heat conduction with T, ρ, C_p, $\mathbf{q}$, e_f, and F being the temperature, density, and specific heat, heat flux, energy per fission, and volumetric fission rate, respectively [21]. The second equation models species conservation, where C, λ, $\mathbf{J}$, and S are concentration, radioactive decay constant, mass flux, and source rate, respectively. The third equation is Cauchy's equation and models momentum conservation, with σ and $\mathbf{f}$ the stress tensor and body force per unit mass, respectively.

BISON has been applied to a wide range of fuel types, including metallic rod, TRISO particle fuel, LWR fuels, and plate fuel. As an application built on top of the MOOSE framework, BISON provides a MooseApp subclass that registers specialized MOOSE Kernels, Materials, and Boundary Conditions.

PROTEUS Core Neutronics

The PROTEUS Core Neutronics application is primarily developed at Argonne National Laboratory (ANL) as part of the NEAMS Reactors Product Line. It is written in Fortran 90 with C preprocessor definitions, and uses an even-parity discrete ordinates approximation (SN2ND) to solve the steady-state neutron transport equation [17]. It runs on a variety of architectures, and scales from a couple cores on a

laptop, to over 10^5 cores on a machine such as Mira at ANL. It relies heavily on PETSc matrices and vectors to acheive this parallelism and provide such scalability.

Crucially, PROTEUS provides conversion code to map its internal mesh data structures to MOAB, and vice versa. This functionality was developed as part of work done to integrate PROTEUS into SHARP and CouPE, and enables the use of PROTEUS coupled to NEK5000 and DIABLO. It is this MOAB mesh layer that will quickly and efficiently enable solution communication between BISON and PROTEUS.

Warthog Architecture

The coupling of various applications to BISON has been done before [10]. BISON has successfully been coupled to applications like MARMOT, RattleSnake, and RELAP-7. To acheive this coupling, the MultiApps and Transfers system was utilized to provide both loose and tight coupling strategies. The success of these past code coupling attempts provides a key insight for the coupling of PROTEUS and BISON - utilize the MultiApps system to enable their coupling. However, to do that, we need to provide PROTEUS to the MultiApps system in way that it expects, i.e., as a MOOSE-based application.

Warthog provides just that - it is a MOOSE-based application that executes a PROTEUS steady-state solve. It acts as a wrapper for PROTEUS that makes it compatible with the rest of the MOOSE ecosystem. A high-level view of the Warthog architecture is shown in Figure 4. The first thing to note is that Warthog enters into the MOOSE framework by providing a new MooseApp object called *WarthogApp*. This enables the creation of a new MOOSE-based application that can declare or register any number of MOOSE-system extensions. Specifically, Warthog registers a new Transient Executioner, called *MoabCodeExectioner*, as well as realizations of a new interface called *IMoabCode*.

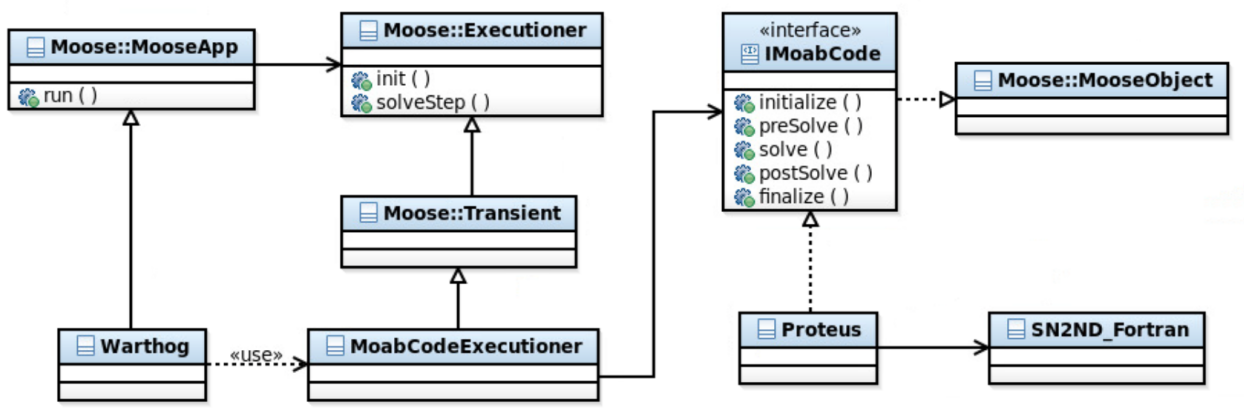


Fig. 4. A high-level view of the Warthog Architecture in the Unified Modeling Language (UML). This diagram demonstrates the class hierarchies developed by Warthog, with MoabCode-Executioner subclassing Transient and having an IMoabCode reference, and Proteus realizing the IMoabCode interface and using the PROTEUS SN2ND Fortran subroutines.

The IMoabCode interface provides a communication protocol for Warthog and external MOAB-based physics applications, such as PROTEUS. This interface was created to enable the integration with BISON of not only PROTEUS, but potentially NEK5000 and DIABLO. It provides methods that any MOAB-based physics codes should be able to implement to efficiently execute a given solve and transfer data to and from MOOSE. Importantly, this interface subclasses the *MooseObject* class, and in doing so picks up the use of

the MOOSE factory system. So, creating a new realization of IMoabCode is as simple as declaring that realization as a string in the Warthog input file.

The MoabCodeExecutioner subclasses the behaviour of a Transient Executioner to provide custom *init* and *solveStep* methods that initialize and execute an IMoabCode realization, respectively. It acts to drive the entire IMoabCode execution by invoking the correct methods to initialize, solve, and finalize a given MOAB-based physics application solve. The MoabCodeExecutioner *solveStep* implementation executes the *preSolve*, *solve*, and *postSolve* methods on the given IMoabCode realization. That concrete realization is specified to the MoabCodeExecutioner through a required MOOSE parameter, *moab_code*, defined in the input file. In the future, one could imagine this *moab_code* parameter being a vector of strings for specifying the use of many MOAB-based physics applications in a given coupled BISON solve.

The only implementation of IMoabCode currently available in Warthog is a class called *Proteus*. This class provides concrete implementations of the methods defined on IMoabCode that execute a steady state solve for PROTEUS' SN2ND solver. It provides the connection between the C++ MOOSE and Fortran 90 PROTEUS subroutines. It's *initialize* method passes a reference to MOOSE's MPI communicator to PROTEUS and invokes a number of subroutines used to initialize PROTEUS' solver data structures. The *preSolve* method acts to transfer temperature data from MOOSE to PROTEUS for its cross-section calculation. The *solve* implementation simply executes the PROTEUS SN2ND solve, and the *postSolve* method transfers the computed power density from PROTEUS to MOOSE.

The only remaining question is how does this temperature and power density solution transfer happen? To acheive this we must map solution fields from two completely different mesh libraries, MOAB and libMesh. Warthog accomplishes this through DataTransferKit (DTK).

DataTransferKit

To enable interoperability between Proteus and Bison, the DataTransferKit library (DTK) was used to provide solution transfer services [18]. Several development activities were needed to provide DTK services for the libMesh data structures used by MOOSE and Bison for mesh and field data and the MOAB data structures used by Proteus for mesh and field data. This development included the implementation and testing of DTK interfaces for these data structures. In addition, an L2 projection data transfer operator was added to DTK to support accurate mesh-based solution transfers. Finally, a number of integration tests were developed in addition to the unit tests developed for each implementation which demonstrate the mathematical correctness of the L2 projection operator when used for parallel solution transfer between MOAB and libMesh data structures. This section of the report documents these development efforts.

DTK Data Transfer Concepts

The data transfer problem in DTK is defined as the translation or reconstruction of a function represented by a discretization on one mesh to a representation on a potentially different mesh with a different discretization. The mesh on which the function is initially represented is defined as the *source mesh* and the mesh onto which the function will be transferred is defined as the *target mesh*. In its representation on the source mesh, the function will be defined as, f, the *source function*. The representation of the function on the target mesh will be, g, the *target function*. We define the source function over M *support locations* as:

$$f = \sum_{i=1}^{M} f_i \phi_i \,, \tag{4}$$

where:

$$f_i = f(\mathbf{s}_i)\,, \tag{5}$$

is the source function evaluated at the i^{th} support location in the source mesh, $\mathbf{s}_i$, and:

$$\phi_i = \phi(\hat{\mathbf{s}}_i)\,, \tag{6}$$

is the source basis function evaluated at the parametric coordinates of the i^{th} support location in the source mesh, $\hat{\mathbf{s}}_i$. Equivalently, we have the target function defined over N support locations as:

$$g = \sum_{i=1}^{N} g_i \psi_i\,, \tag{7}$$

where:

$$g_i = g(\mathbf{t}_i)\,, \tag{8}$$

is the target function evaluated at the i^{th} support location in the target mesh, $\mathbf{t}_i$, and:

$$\psi_i = \psi(\hat{\mathbf{t}}_i)\,, \tag{9}$$

is the target basis function evaluated at the parametric coordinates i^{th} support location in the target mesh, $\hat{\mathbf{t}}_i$. A support location is defined as a geometric entity supporting the function. It may be a node, edge, face, or element in the mesh depending on the underlying discretization of the function.

The data transfer problem is then to find the values of the source function on the target mesh given the function discretization on both meshes and the source function values on the source mesh support locations. We define a *data transfer operator*, $\mathbf{H}$, such that:

$$\mathbf{g} \leftarrow \mathbf{H}(\phi, \psi)\mathbf{f}\,, \tag{10}$$

with $\mathbf{H} : \mathbb{R}^M \rightarrow \mathbb{R}^N$, $\mathbf{f} \in \mathbb{R}^M$ the vector of source function values at the source support locations, and $\mathbf{g} \in \mathbb{R}^N$ the vector of target function values at the target support locations. The notation $\mathbf{H}(\phi, \psi)$ indicates that $\mathbf{H}$ is potentially constructed from the basis functions of the source and target discretizations.

To enable this numerical description, DTK has a set of interfaces defined as C++ classes which client applications (i.e. libMesh or MOAB) implement to access DTK services. There are 6 such interfaces: Entity, EntitySet, EntityLocalMap, EntityShapeFunction, EntityIntegrationRule, and Field. We will briefly outline each of these to provide context for the libMesh and MOAB implementations created for Warthog. These implementations are now included in the DTK library.

Entity An implementation of the *Entity* class gives a general description of a geometric object. For example, all MOAB mesh entities such as faces and elements have an implementation for this class. To allow DTK to query the state of the entity, the *Entity* interface gives the following information:

1. The physical dimension of the Entity. This is the number of coordinates needed to locate a point in physical space contained within the entity.

2. The topological dimension of the Entity. This is the number of coordinates needed to locate a point in the reference frame of the entity.

3. The unique global id number of the entity.

4. The parallel process id uniquely owning the entity.
5. The bounding box of the entity.
6. The block of the mesh in which the mesh entity resides.
7. The boundary (if any) on which the entity resides.

EntitySet The *EntitySet* interface provides DTK with a set of functions that describe all entities in the problem. In the case of libMesh and MOAB, this describes the entire mesh in the problem. Subsets of the mesh are accessed through predicate functions that modify iterators. The following information is provided by this interface:

1. The parallel communicator (i.e. MPI communicator) for the entity set.
2. The physical dimension of the entity set.
3. The bounding box of all entities on the local parallel process.
4. The bounding box of all entities on all processes.
5. Retrieve an entity from the set given its topological dimension and global identifier.
6. Provide an iterator over entities in the set that satisfy a given predicate (i.e. an iterator over entities on a given boundary).
7. Given an entity, retrieve the entities that are adjacent to it of a given topological dimension.

EntityLocalMap Through the *EntityLocalMap* interface, a user application provides DTK with functions describing the reference frame of the entity. These include mapping a point to and from the reference frame of an entity as well as functions that use the reference frame for geometric computations. The following information is provided by the *EntityLocalMap* interface:

1. The measure of an entity. For an entity of topological dimension 1 this is the length, area for a dimension of 2, and volume for a dimension of 3.
2. The centroid of an entity.
3. Map a point from the physical frame to the reference frame.
4. Determine if a point in the reference frame is in the entity.
5. Map a point from the reference frame of an entity to the physical frame.

EntityShapeFunction The *EntityShapeFunction* interface gives access to the shape functions discretizing fields defined on entities. This includes both shape function evaluations as well as degree of freedom id numbers. We will often want the actual shape function values for certain solution transfer algorithms. The following information is provided by this interface:

1. The global identifiers of the objects supporting the shape function. For example, this is either a set of node ids for an element or a set of different degree of freedom ids.

2. A function for evaluating the shape function of an entity at a given reference point.

3. A function for evaluating the gradient of the shape function of an entity at a given reference point.

EntityIntegrationRule For numerical integration, the *EntityIntegrationRule* interface provides DTK with numerical quadrature rules for entities. This interface provides:

1. Given an entity and order of numerical integration, provide integration points in the reference frame of the entity and the weights of those points in the quadrature.

Field Solution data is extracted from applications through the *Field* interface on an entity-by-entity basis. The interface provides a means for establishing parallel vectors on top of application field data which can then be used to transfer solutions between applications using operators applied to the vectors. An implementation of the interface provides the following capabilities:

1. The dimension of the field. If the field is a scalar (e.g. temperature) the dimension is 1. If the field is a vector quantity the dimension may be larger. For example, a flow velocity vector field in a 3D calculation will have a dimension of 3.

2. The global ids of the objects supporting the field that are owned owned by the local parallel process. This is the set of ids, such as degree of freedom ids, that would be returned by the *EntityShapeFunction* interface which are also uniquely owned by the local parallel process.

3. Given a support id and a field component dimension, get the value of the field from the application.

4. Given a support id, a field component, and a field value, write the field value to the application.

5. Place the field in a parallel consistent state. The functionality gives the user application the option to update, for example, ghosted field values after solution transfer as DTK will only operate on non-ghosted data.

libMesh Adapters

To enable interoperability with Bison and other MOOSE applications within the NEAMS Fuels Product Line, adapters were written for the interfaces described in the previous section. These adapters currently reside within the DTK repository. However, previous users of DTK with libMesh will note that this is in contrast to the DTK code residing both the MOOSE and libMesh repositories. These new adapters specifically implement what we are terming the new, "Version 2" API of DTK. The implementations that currently exist with MOOSE or libMesh, however, are still fully supported as the "Version 1" API of DTK by use of the *ClassicAdapters* package in the library. These classic adapters provide all "Version 1" functionality with the same API but implementation details have been replaced to use the new "Version 2" code.

For the purposes of Warthog, the approach of adding new libMesh adapters to the DTK repository is sufficient as all MOOSE/Bison builds are executed only with a PETSc dependency, eliminating the Trilinos and DTK dependency from the MOOSE/Bison build. At a higher level, Warthog coordinates the use of DTK and compilation of third-party libraries including DTK and Trilinos. Users access these new adapters in DTK by building the library with libMesh as a dependency, thus avoiding cyclic dependencies.

The implementation of DTK interfaces using libMesh was relatively straightforward. Table 1 provides a reference of which libMesh classes and interfaces map to which DTK interfaces. Additional data structures to create a full adjacency graph between elements, faces, and nodes in libMesh were needed to satisfy all DTK interface requirements. In particular, the extra adjacency data structures permit, for example, the immediate extraction of all nodes in the mesh that compose a particular mesh element. All implementations of DTK functionality are entirely parallel, utilizing native libMesh parallel functionality. A suite of unit tests that check the libMesh implementation of all DTK interfaces and additional data structures was developed as well. These tests, which run automatically with the rest of the DTK unit test suite when the libMesh adapters are enabled, provide coverage for initial Warthog use cases.

Table 1. Relationships between DataTransferKit interfaces and libMesh interfaces.

DTK Interface	libMesh Interface
Entity	libMesh::Node, libMesh::Elem
EntitySet	libMesh::MeshBase
EntityLocalMap	libMesh::FEInterface, libMesh::Elem
EntityShapeFunction	libMesh::FEComputeData, libMesh::FEInterface
EntityIntegrationRule	libMesh::QBase
Field	libMesh::System, libMesh::Variable

As indicated in the previous section, DTK provides the opportunity to select only a subset of the mesh for solution transfer using the concept of mesh blocks or boundaries which logically subdivide volumes or surfaces. In libMesh, the concept of blocks is represented by subdomains. Each element in the mesh has a given subdomain id and nodes are considered to be in a subdomain if one of their parent elements is in the subdomain. The libMesh library equivalently has a boundary concept. A node can be directly on a boundary while an element is considered to be on a boundary if any of its sides (faces) are on the boundary. We have demonstrated solution transfer using DTK over both selected subdomains and boundaries of a libMesh mesh.

MOAB Adapters

For interoperability with several physics tools in the NEAMS Reactors Product Line, DTK interface adapters for the MOAB mesh database library were created for this work. Compared to libMesh, MOAB has had no prior dependency on DTK and does not have a Trilinos dependency. Therefore, the logical location for these adapters is in the DTK repository. These adapters, if enabled, use MOAB as a third-party library to satisfy the dependency.

Table 2 gives the relationships between the DTK interfaces and MOAB interfaces. Although still relatively straightforward, implementation of the DTK interfaces using MOAB was more nuanced than the libMesh implementation. Management of the parallel mesh state in MOAB is provided through the *moab::ParallelComm* interface. Some work was needed to understand how the parallel mesh is initialized and how global indices are constructed for mesh elements using this interface. In addition, the bulk of discretization capability is handled through the *moab::ElemEvaluator* interface which handles point location, function evaluation, and numerical integration. For shape function evaluation, the

moab::ElemEvaluator uses tag data to provide field evaluation in a single function. To extract only the shape function values and gradients for the DTK interfaces, artificial field values are used and the *moab::Tag* interface is used to access field data when needed. For numerical integration, the *moab::ElemEvaluator* interface could not be manipulated to extract the quadrature rule embedded in the implementation. Instead, the Intrepid package of Trilinos was used to provide quadrature rules compatible with the discretization provided for various topologies in the *moab::ElemEvaluator* interface [11].

Table 2. Relationships between DataTransferKit interfaces and MOAB interfaces.

DTK Interface	MOAB Interface
Entity	moab::EntityHandle
EntitySet	moab::ParallelComm, moab::Interface
EntityLocalMap	moab::ElemEvaluator
EntityShapeFunction	moab::ElemEvaluator
EntityIntegrationRule	moab::ElemEvaluator
Field	moab::Tag

Like the libMesh implementation, a suite of unit tests was developed for the MOAB implementation to provide coverage for all Warthog use cases and to verify the correctness of the implementation. In particular, they ensure that the data extracted from the *moab::ElemEvaluator* interface as well as the use of Intrepid shape functions with MOAB discretizations compute the correct values. When the MOAB adapters are enabled in DTK, these tests execute with the other unit tests. It was found that these tests were critical for verifying correctness of the code before proceeding to use the adapters in a larger coupled framework under Warthog.

Both block and boundary concepts in the DTK implementation are represented using MOAB mesh sets. These mesh sets are user defined collections of entities and are commonly used to represent logical subdivisions of the mesh. Like libMesh, the DTK implementation considers a MOAB entity to be in a block or on a boundary if that entity is contained within the mesh set defining the block or boundary. In particular, predicate functions have been developed for MOAB which enable easy selection of entities in a single or multiple mesh sets for solution transfer with DTK. We have also demonstrated this functionality with solution transfers over specified MOAB mesh sets.

L2 Projection

For the accurate transfer of mesh-based fields in Warthog, an L2 projection operator was added to DTK. To construct the data transfer operator for this method, the weighted residual problem based on L_2 minimization is formed as follows per [14]. We define the following minimization problem:

$$\frac{\partial}{\partial g_i}\left[\int_\Omega (g-f)^2 d\Omega\right] = 0\,. \tag{11}$$

Expanding the squared term and substituting in Eq (7) for g we have:

$$\frac{\partial}{\partial g_i}\left[\int_\Omega (g-f)^2 d\Omega\right] = \frac{\partial}{\partial g_i}\left[\int_\Omega \left(\Big(\sum_{j=1}^{N}\psi_j g_j\Big)^2 - 2\sum_{j=1}^{N}\sum_{k=1}^{M}\psi_j g_j \phi_k f_k + \Big(\sum_{k=1}^{M}\phi_k f_k\Big)^2\right) d\Omega\right]. \tag{12}$$

Performing the differentiation and separating the integral gives the following linear system to solve for the target function:

$$\sum_{j=1}^{N} \int_{\Omega} \psi_i \psi_j d\Omega g_i = \sum_{k=1}^{M} \int_{\Omega} \psi_i \phi_k d\Omega f_k \,, \tag{13}$$

or

$$\mathbf{Mg} = \mathbf{Af} \,, \tag{14}$$

where $\mathbf{M}$ is defined as the *mass matrix* with individual elements:

$$M_{ij} = \int_{\Omega} \psi_i \psi_j d\Omega \,, \tag{15}$$

$\mathbf{A}$ is defined as the *coupling matrix* with individual components:

$$A_{ik} = \int_{\Omega} \psi_i \phi_k d\Omega \,, \tag{16}$$

$\mathbf{g}$ the vector of unknown target function values and $\mathbf{f}$ the vector of known source function values. The action of the data transfer operator on the source function is then:

$$\mathbf{H}f = \mathbf{M}^{-1}\mathbf{Af} \,, \tag{17}$$

where now a symmetric positive-definite linear system must be solved at every application of the data transfer operator from the current values of the source function. Although the weighted residual problem can be weakly formulated as in [3], we find this definition to be more general as it permits minimizing the data transfer residual over other norms[1].

Building the mass matrix only requires integrations of the target basis functions and therefore those integrations can always occur exactly over the target grid. However, constructing the coupling matrix requires numerical integration of the target basis functions and the source function containing the source basis functions. As both of those functions are likely to be defined over grids of different topologies and/or mesh size, the question becomes how to perform those integrations in a way that is both accurate and conservative. A first choice for building the coupling matrix is numerical integration over the source grid while a second choice is numerical integration over the target grid [3, 4]. In general, integration of the source grid provides better conservation while integrating over the target grid provides better accuracy. In DTK, we choose integration of the target grid for improved accuracy. If needed in the future, more accurate and conservative numerical integrations could be constructed via mesh intersection using the common-refinement scheme [14] or the nearly identical super-mesh scheme [5].

DTK Integration and Verification Tests

To demonstrate interoperability between MOAB and libMesh as well as the numerical correctness of the L2 projection operator specifically developed for this work, several integration and verification tests were developed and exist as examples in the DTK repository. This are in addition to the individual unit test suites developed for each adapter implementation to test using the the new code in the same manner as the Warthog use case for both serial and parallel calculations.

To test interoperability, a libMesh and MOAB database is constructed from an exodus mesh file. The DTK interface subclass implementations are then created from the libMesh and MOAB data structures. For

[1] Such as the Sobolev norm as in [14].

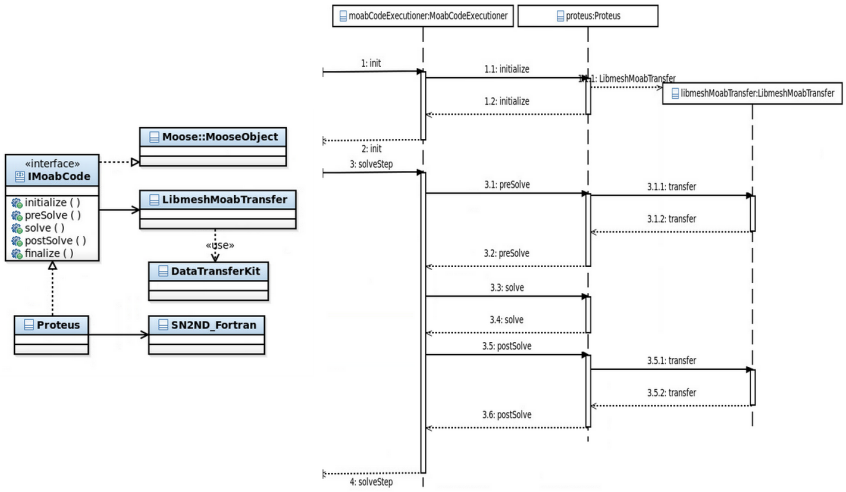


Fig. 5. The LibmeshMoabTransfer architecture in UML (left) with a sequence diagram describing its use in Warthog (right).

verification, a known function is assigned to either mesh depending on the direction of transfer (either from libMesh to MOAB or from MOAB to libMesh), and the transfer is executed. The resulting solution is checked against the values of the known function at those spatial locations. Accuracy is checked by computing the difference between the transferred value at each node and the expected value with a norm of this vector giving a global measure of accuracy. Conservation is assessed by comparing the difference between the known function integrated over the source mesh to the transferred function integrated over the target mesh. These tests show that for both transfer directions and in serial and parallel, both interoperability between MOAB and libMesh and expected measures of conservation and accuracy using the L2 projection operator are achieved.

LibmeshMoabTransfer Black Box Solution Transfer

To efficiently incorporate the aforementioned tools and technologies found in DataTransferKit, Warthog provides an object that acts as a black box for pertinent solution transfers using DTK. The *LibmeshMoabTransfer* object, shown in Figure 5, provides this black box solution transfer capability by wrapping the essential DTK functionality for creating and executing transfers to and from MOAB and libmesh. Every realization of IMoabCode inherits a reference to this object and can use it to create a DTK transfer mechanism and execute it when its needed. For example, the Proteus implementation of IMoabCode creates transfer operators for the temperature and power, and executes them in the preSolve and postSolve methods, respectively. The sequence of events for a given PROTEUS execution and the utilization of the LibmeshMoabTransfer object is demonstrated in the sequence diagram in Figure 5.

The power of DataTransferKit is exhibited in the example transfer code snippet in Figure 6. This is essentially what the LibmeshMoabTransfer executes to create and invoke DTK transfer maps. First, manager objects are created for the MOAB and libMesh mesh instances. From those, field vectors are created for the Tag and Variable corresponding to the solution field of interest. The transfer operator is then constructed in an extensible manner (operator types and parameters determined by XML file). The operator

```
// Create the MOAB Mesh
Teuchos::RCP<moab::ParallelComm> source_mesh = getMoabMesh();

// Create the Libmesh Mesh
Teuchos::RCP<libMesh::Mesh> tgt_mesh = getLibmeshMesh();

// Create a manager for the source set elements.
DataTransferKit::MoabManager src_manager(source_mesh, source_set);

// Create a manager for the target set nodes.
DataTransferKit::LibmeshManager tgt_manager(tgt_mesh,
                                            Teuchos::rcpFromRef(tgt_system));

// Create a solution vector for the source.
Teuchos::RCP<Tpetra::MultiVector<double, int, std::size_t> > src_vector =
    src_manager.createFieldMultiVector(source_node_set,
                                       source_data_tag);

// Create a solution vector for the target.
Teuchos::RCP<Tpetra::MultiVector<double, int, std::size_t> > tgt_vector =
    tgt_manager.createFieldMultiVector(tgt_var_name);

// Create a map operator.
Teuchos::ParameterList& dtk_list = plist->sublist("DataTransferKit");
DataTransferKit::MapOperatorFactory op_factory;
Teuchos::RCP<DataTransferKit::MapOperator> map_op =
    op_factory.create(src_vector->getMap(), tgt_vector->getMap(),
                      dtk_list);

// Setup the map operator.
map_op->setup(src_manager.functionSpace(), tgt_manager.functionSpace());


// SOLUTION TRANSFER
// -----------------

// Apply the map operator.
map_op->apply(*src_vector, *tgt_vector);
```

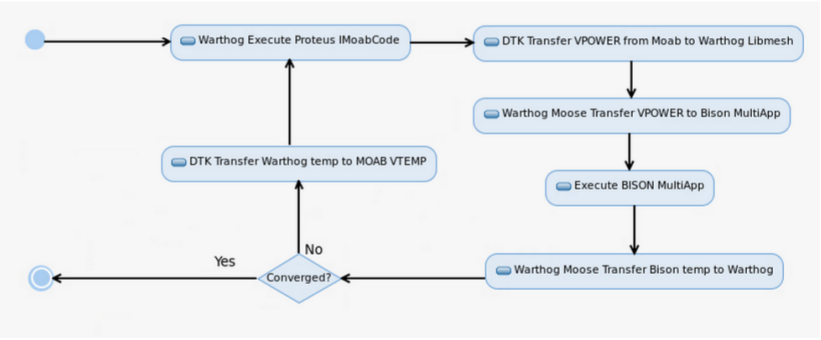
Fig. 6. Code used by LibmeshMoabTransfer to execute MOAB to libMesh solution transfers. This code takes solution data on a moab::ParallelComm instance and maps it to a libMesh::Mesh instance.

is then setup and a simple apply operation is executed to perform the solution transfer from MOAB to libMesh.

Warthog-BISON Coupling Mechanism

With PROTEUS wrapped in a MOOSE-based application, and a convenient MOAB-libMesh data transfer mechanism in place, the act of coupling to BISON is simple thanks to the MOOSE MultiApps and Transfers systems. Figure 7 shows a high level view of the overall workflow for a coupled BISON-PROTEUS calculation using Warthog, and the actual input file declaration to make it happen. A user simply declares Warthog as a MultiApp in a BISON solve, and declares Transfers that take calculated power from Warthog to BISON, and temperature from BISON to Warthog. These transfers occur between the BISON and Warthog libMesh instances within the MOOSE framework, with the MOAB-libMesh transfer between Warthog and PROTEUS happening at the lower, IMoabCode level.

A BISON solve that contains these blocks in the input file can perform a coupled thermomechanics-neutronics solve. The overall workflow begins with an execution of Warthog, which in turn executes a PROTEUS steady-state solve with the current temperature field. This solution is transfered from MOAB back to libMesh using DTK, and that is then transferred from Warthog to BISON, which acts as the master application in this MultiApp solve. BISON is then executed with the updated power density data, and is followed by a transfer of the computed temperature back to Warthog. A Warthog solve is executed again, with the updated temperature field thanks to a transfer invocation for the LibmeshMoabTransfer object in the Proteus preSolve method. This continues interatively until convergence is reached. Furthermore, the MOOSE Transient Executioner provides a hook in the input file to specify a Picard iteration for even tighter coupling.




```
[MultiApps]
  [./sub_warthog]
    type = TransientMultiApp
    input_files = 'warthog_fumex.i'
    app_type = WarthogApp
    positions = '0 0 0'
    execute_on = 'timestep_end'
  [../]
[]

[Transfers]
  [./vpower_to_bison]
    variable = power
    source_variable = VPOWER
    multi_app = sub_warthog
    direction = from_multiapp
    type = MultiAppInterpolationTransfer
  [../]
  [./temp_to_warthog]
    variable = VTEMP
    source_variable = temp
    multi_app = sub_warthog
    direction = to_multiapp
    type = MultiAppInterpolationTransfer
  [../]
[]
```

Fig. 7. High-level flow for a given Warthog execution (left) and example MOOSE input file blocks for enabling BISON-PROTEUS coupling with Warthog (right).

Integration with the NEAMS Integrated Computational Environment

The NEAMS Integrated Computational Environment (NiCE) is a product from the Integration Product Line in NEAMS that provides a end-to-end workflow platform for NEAMS toolkit simulation technologies. It provides a graphical interface and associated integrated tools for a large array of pertinent scientific computing tasks such as local or remote simulation job launch, simulation input generation, and data visualization and analysis. Throughout the 2015 fiscal year, NiCE support for MOOSE-based applications has matured immensely. NiCE provides an integrated *MOOSE Workflow* tool for interaction with *any* MOOSE-based application. NiCE now has support for embedded visualizations across the entire platform, and was leveraged in the MOOSE Workflow tool to provide detailed and dynamic views of the problem mesh, solution mesh, and XY plots of Postprocessor data. Furthermore, NiCE now supports real-time Postprocessor visualizations. As MOOSE simulations are running, they can now report back Postprocessor data at each time step to NiCE via a web socket, and NiCE updates a XY plot in real-time.

NiCE has also been updated to provide support for actual MOOSE application development. Users can now leverage a myriad of tools in ICE for C++ development, application building, and even GitHub integration. For MOOSE, NiCE now provides custom buttons and widgets to *Fork the Stork* [20], and clone and fork MOOSE itself. MOOSE application developers can now create their application, keep it version controlled, execute it, and analyze simulation results in one holistic and integrated environment.

Warthog development leveraged NiCE as much as possible. Warthog was actually developed in NiCE using the aforementioned developer tools, and Warthog executions are performed using the MOOSE Workflow tool (see Fig. 8 and 9). Using NiCE, a Warthog user can efficiently and quickly interact with Warthog and execute simulation launches, view embedded mesh visualizations and simulation results, and generate input file using a custom tree view.

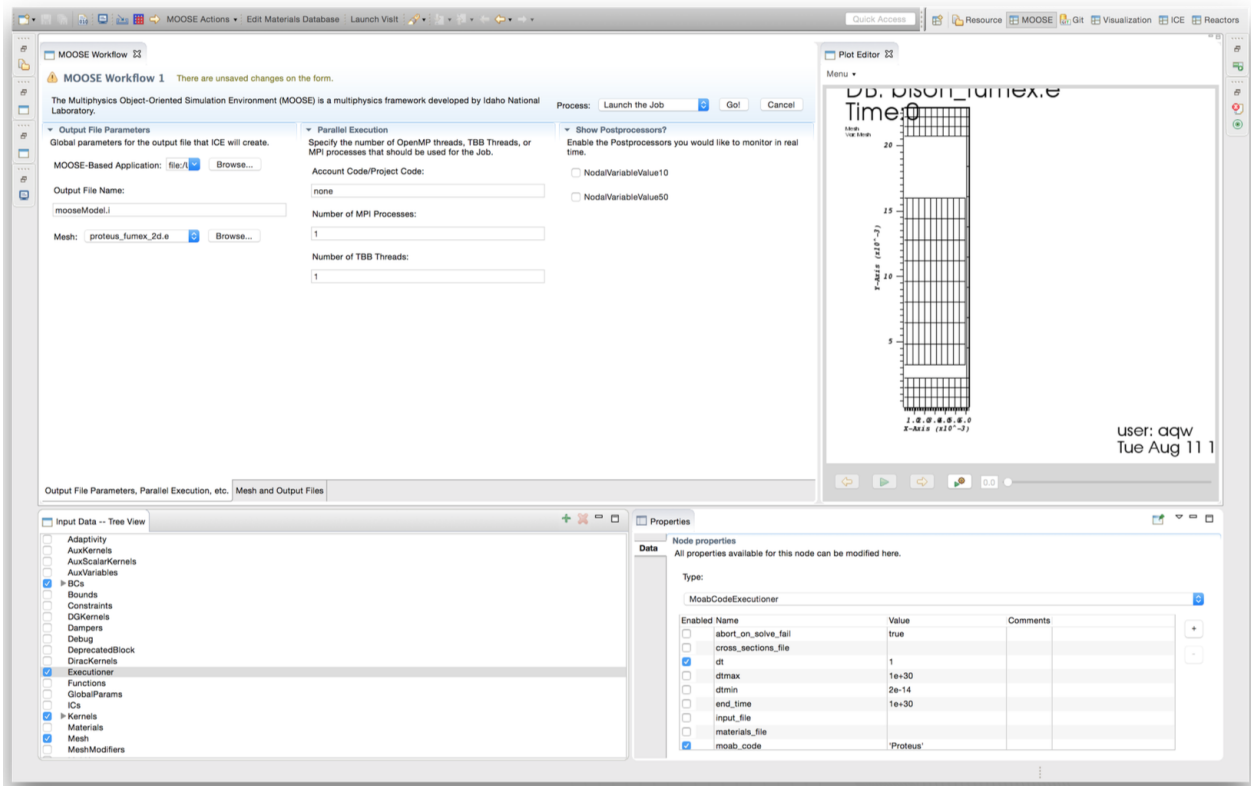


Fig. 8. A view of a MOOSE Workflow Item for interacting with a Warthog simulation. NiCE provides embedded views of the mesh and solution using VisIt or Paraview, and provides a graphical tree view for creating a Warthog input file.

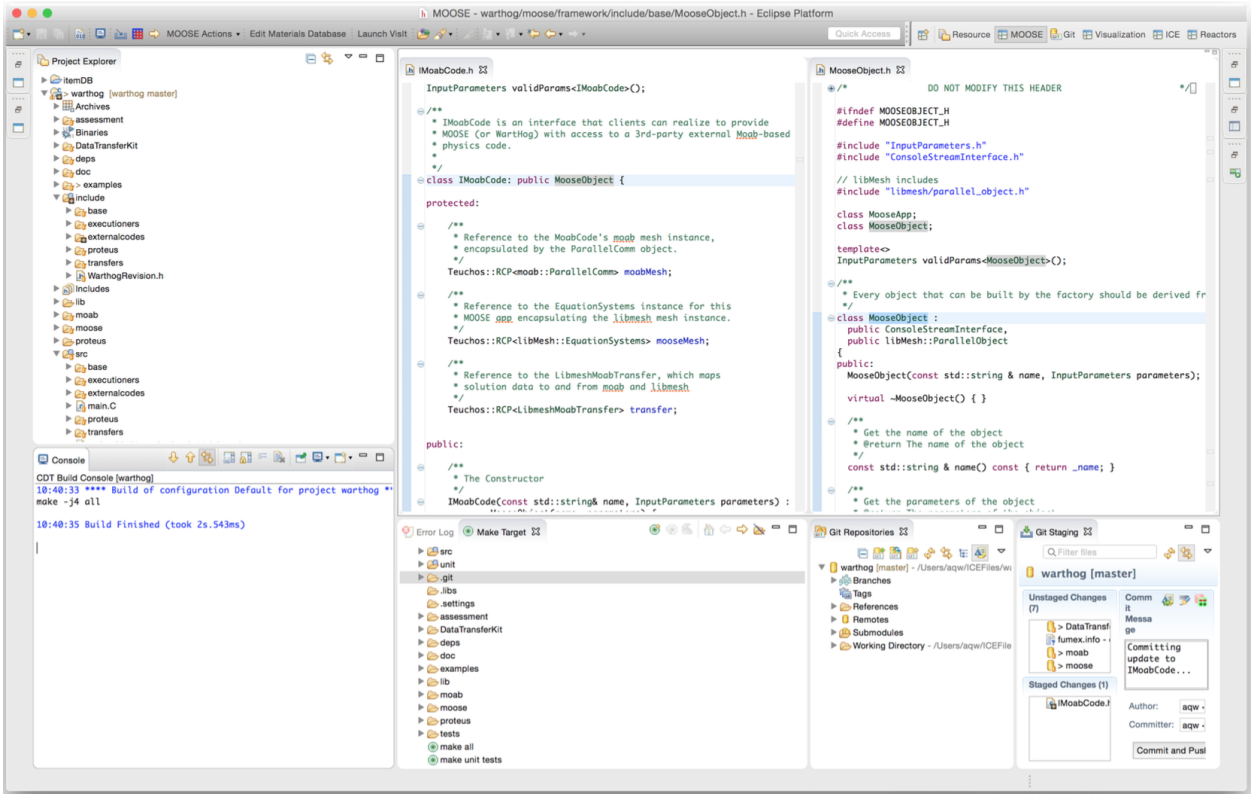


Fig. 9. A view of NiCE tools for actual Warthog development. NiCE provides code editing tools, as well as version control with Git.

Current Software Coupling Results

The ability to perform adequate software coupling has been demonstrated with Warthog using a simple debug problem from the SHARP test system. This problem represents an idealized simple hexagonal assembly (SAHEX) for a sodium cooled fast reactor [15]. Cross section data parameterized by temperature and density was generated by the SHARP team for this problem using the MC^2 library [1]. First, to simply demonstrate that Warthog correctly wraps the functionality in PROTEUS, we ran Warthog with the SAHEX input deck to compare against an actual PROTEUS execution for that input. We found that Warthog correctly produced the same power profile as PROTEUS, as shown in the left plot of Figure 10.

Finally, to demonstrate the complete workflow for our direct software coupling, we ran this calculation as part of a MultiApp execution with BISON. We constructed a BISON input file that attempted to mimic the SAHEX input model for PROTEUS. This input model was not meant to be physically accurate, as we just want to demonstrate data transfer. We constructed a BISON mesh corresponding to the fuel pins in the SAHEX PROTEUS model using Cubit, and simply assigned the MOOSE ThermalUO2 Material to the fuel block. We added a TransientMultiApp to the BISON input file that executed the sub-app Warthog, and transfered data to and from BISON and Warthog using provided MOOSE Transfer objects. The results are shown in Figure 10, where one can clearly see that the correct workflow of data transfer from MOAB to libMesh through DTK, and a MOOSE Transfer from Warthog to BISON for the final, correct power distribution on the BISON mesh. This demonstrates successful and direct software coupling between PROTEUS and BISON.

Future Work and Directions

Warthog has been demonstrated to succesfully enable the software coupling of BISON and PROTEUS. As of the time of this writing, and moving forward in the near future, the Warthog team will be focusing on the execution of physically realistic, and experimentally validated, BISON assessment cases. We have already begun work on executing the BISON-Warthog MultiApp system for the simplified FUMEX-II

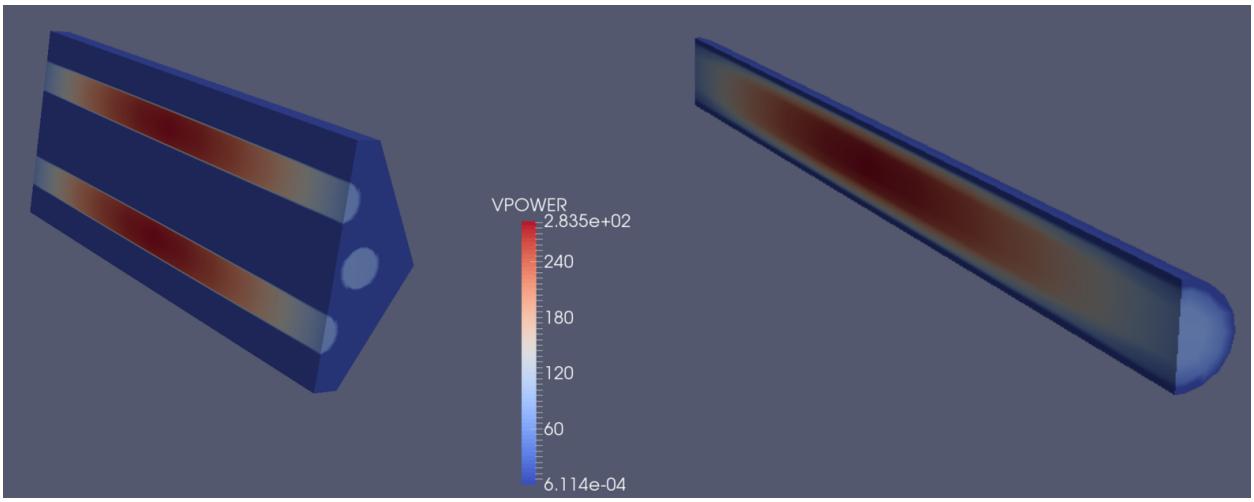


Fig. 10. A view of the power density result for PROTEUS (left) and its mapped solution to the BISON fuel pin mesh (right).

27_2b case [16]. So far, we have been able to demonstrate data transfer between the applications, but have not put together a physically realistic input deck for execution.

The primary issue moving forward for executing physically realistic models is the lack of adequate thermal fuel cross section data for PROTEUS executions. PROTEUS primarily works with fast reactor fuel, and therefore does not have an extensive library of cross section data for thermal fuel. We have begun work on generating the necessary cross section data with Ugur Mertyurek and Kang Seog Kim from the ORNL SCALE team. We have considered solving the FUMEX-II 27_2b case assuming a 40% void fraction and average core conditions [2]. However, while this problem was designed to facilitate comparison between codes, it does so by specifying a power profile and was never intended to be simulated with an actual neutronics simulator. The team is actively reviewing the FUMEX-II problems to identify one that is a good fit for a multiphysics simulation with neutronics and fuel performance calculations.

There are other issues we would like to investigate going forward. Mapping input decks between BISON and PROTEUS has proven difficult to do manually. We would like to incorporate some automatic mapping in the overall Warthog execution that hides this complexity and successfully maps BISON inputs to the files that PROTEUS expects. Perhaps we could investigate streamlining cross section generation for PROTEUS as part of the overall Warthog solve. Additionally, with the extensibility of the MoabCodeExecutioner and IMoabCode interface, we could begin to investigate coupling additional components of the SHARP framework into Warthog. It would be interesting to experiment with adding a Nek5000 implementation of IMoabCode, for example, to couple BISON and PROTEUS to a thermal hydraulics solve.

Conclusion

We have demonstrated a viable methodology for the direct software coupling of BISON and PROTEUS. This methodology could be extended to the coupling of MOOSE applications and other components of the SHARP framework. In enabling this coupling, we have developed a new MOOSE-based application called Warthog, which wraps the functionality of a PROTEUS steady-state solve and provides a means of utilizing PROTEUS in the MOOSE MultiApps and Transfers systems. Crucially, this work has leveraged a wide array of existing NEAMS technologies, as well as developed new tools for future use. We have leveraged all that is available in the MOOSE framework and the MOAB bindings in the SHARP/PROTEUS system to enable this coupling. We have provided a means for MOOSE-SHARP, MOAB-libmesh communication through developed extensions to DataTransferKit. This work enables future work in NEAMS to seamlessly communicate solution data between these two disparate mesh formats.

There is still much work to be done in the construction and execution of physically realistic reactor models. Primarily, work needs to be done in the generation of thermal fuel cross section data for PROTEUS to be utilized in BISON assessment case executions. Going forward, the Warthog team will focus on this data generation and utilize the newly developed technologies to study models with experimental data to validate against.

Acknowledgements

The Warthog team would like to acknowledge the MOOSE and BISON development teams at Idaho National Laboratory, and the PROTEUS development team at Argonne National Laboratory. Their input and advice greatly advanced this work, and will continue to do so in the future. Additionally, we would like to acknowledge the funding agency for this work, the U.S. Department of Energy Office of Nuclear Energy's Nuclear Energy Advanced Modeling and Simulation (NEAMS) program and the Advanced Modeling and Simulation Office (AMSO) within DOE-NE.

REFERENCES

[1] MC^2-2: A Code to Calculate Fast Neutron Spectra and Multigroup Cross Sections. *Argonna National Laboratory, ANL-8144*.

[2] J. J. Billings, A. McCaskey, U. Mertyurek, K. Kim, and R. Williamson. private email communication, 2015.

[3] Juan Raul Cebral and Rainald Lohner. Conservative load projection and tracking for fluid-structure problems. *AIAA Journal*, 35(4):687–692, 04 1997.

[4] C. Farhat, M. Lesoinne, and P. LeTallec. Load and motion transfer algorithms for fluid/structure interaction problems with non-matching discrete interfaces: Momentum and energy conservation, optimal discretization and application to aeroelasticity. *Computer methods in applied mechanics and engineering*, 157:95–114, 1998.

[5] P.E. Farrell, M.D. Piggott, C.C. Pain, G.J. Gorman, and C.R. Wilson. Conservative interpolation between unstructured meshes via supermesh construction. *Computer methods in applied mechanics and engineering*, 198:2632–2642, 2009.

[6] RM Ferencz. Technical Spotlight: NEAMS Structural Mechanics with Diablo. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013.

[7] P Fischer, J Kruse, J Mullen, H Tufo, J Lottes, and S Kerkemeier. Nek5000 - Open Source Spectral Element CFD Solver. *Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL, see https://nek5000. mcs. anl. gov/index. php/MainPage*, 2008.

[8] Derek Gaston, Chris Newman, Glen Hansen, and Damien Lebrun-Grandié. MOOSE: A Parallel Computational Framework for Coupled Systems of Nonlinear Equations. *Nuclear Engineering and Design*, 239(10):1768–1778, 2009.

[9] Derek R. Gaston, Cody J. Permann, John W. Peterson, Andrew E. Slaughter, David AndrÅą, Yaqi Wang, Michael P. Short, Danielle M. Perez, Michael R. Tonks, Javier Ortensi, Ling Zou, and Richard C. Martineau. Physics-based multiscale coupling for full core nuclear reactor simulation. *Annals of Nuclear Energy*, 84:45 – 54, 2015. Multi-Physics Modelling of {LWR} Static and Transient Behaviour.

[10] Frederick N. Gleicher, Richard L. Williamson, Javier Ortensi, Yaqi Wang, Benjamin W. Spencer, Stephen R. Novascone, Jason D. Hales, and Richard C. Martineau. *The coupling of the neutron transport application RATTLESNAKE to the nuclear fuels performance application BISON under the MOOSE framework*. Oct 2014.

[11] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, September 2005.

[12] MOOSE Development Team: http://mooseframework.org/wiki/MooseSystems/MultiApps/. Moose multiapp architecture, 2015.

[13] MOOSE Development Team: http://mooseframework.org/wiki/MooseTraining/Overview/. Moose high-level architecture, 2015.

[14] Xiangmin Jiao and Michael T. Heath. Common-refinement-based data transfer between non-matching mesh in multiphysics simulation. *International Journal for Numerical Methods in Engineering*, 61:2402–2427, 10 2004.

[15] Vijay S Mahadevan, Elia Merzari, Timothy Tautges, Rajeev Jain, Aleksandr Obabko, Michael Smith, and Paul Fischer. High-Resolution Coupled Physics Solvers for Analysing Fine-Scale Nuclear Reactor Design Problems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2021):20130381, 2014.

[16] D. Perez, R. Williamson, S. Novascone, G. Pastore, J. Hales, and B. Spencer. Assessment of BISON: A Nuclear Fuel Performance Analysis Code. *Tech. Rep. INL/MIS-13-30314, Idaho National Laboratory*, 2013.

[17] E. R. Shemon, M. A. Smith, C. H. Lee, and A. (Nuclear Engineering Division) Marin-Lafleche. *PROTEUS-SN User Manual*. Aug 2014.

[18] SR Slattery, PPH Wilson, and RP Pawlowski. The Data Transfer Kit: A geometric rendezvous-based tool for multiphysics data transfer. In *International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering (M&C 2013)*, pages 5–9, 2013.

[19] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst. MOAB: a mesh-oriented database. SAND2004-1592, Sandia National Laboratories, April 2004. Report.

[20] MOOSE Development Team. Fork the Stork - http://mooseframework.org/create-an-app/, 2015.

[21] R.L. Williamson, J.D. Hales, S.R. Novascone, M.R. Tonks, D.R. Gaston, C.J. Permann, D. Andrs, and R.C. Martineau. Multidimensional multiphysics simulation of nuclear fuel behavior. *J. Nuclear Materials*, 423(1-3):149–163, 2012.