Fiscal Year 2011 Infrastructure Refactorizations in AMP

September 2011

Mark A. Berrill, Bobby Philip, Rahul S. Sampath, Srikanth Allu, Pallab Barai, William K. Cochran, and Kevin T. Clarno (Oak Ridge National Laboratory)

Gary A. Dilts (Los Alamos National Laboratory)

1 Introduction

In Fiscal Year 2011 (FY11), the AMP (Advanced MultiPhysics) Nuclear Fuel Performance code [1] went through a thorough review and refactorization based on the lessons-learned from the previous year, in which the version 0.9 of the software was released as a prototype. This report describes the refactorization work that has occurred or is in progress during FY11.

Several major sections of AMP have been refactored to improve the functionality and user interface. These include:

1. High-level documentation was incorporated to define the vision, stakeholders, and requirements. (Section 2)
2. The build system has been upgraded to use CMake/CTest/CDash, from KitWare Inc., to simplify the process of building and testing AMP. (Section 3)
3. The Materials component has been refactored to allow for a vector based interface, better organization, and to make it easier for users to add their own materials. (Section 4)
4. The Mechanics component has been refactored to support both small deformation analysis and large deformation analysis using the Updated Lagrangian approach. (Section 5)
5. The TrilinosMLSolver class has been refactored to allow the use of the ML solver without having to create Epetra matrices. (Section 6)
6. The Mesh component is currently being redesigned to allow for multiple mesh packages using a common interface. (Section 7)
7. The AMP Nuclear Fuel Performance code is being split into two codes: a base package that will provide general purpose routines and design and can leverage other work, and a nuclear fuel performance code that contains all of the fuel specific and export controlled code. (Section 8)
8. An AMP-MPI class was added to manage all parallel communication and allow for serial build. (Section 9)

2 Vision, stakeholders, and requirements

During the prototyping effort of Fiscal Year 2010 (FY10), the concept and requirements for the software were discussed and the software was developed with an agile approach that allowed the team to adjust the plans and direction of the software as the team developed cohesiveness. However, the vision, stakeholders, and requirements were never explicitly defined and documented, which created misunderstanding within, and beyond, the team.

Significant effort was placed on developing a vision statement [2] to clearly communicate the direction the team and the software would take. Within this document, the key stakeholders were identified and their high-level requirements specified. In addition, a detailed requirements document [3], was developed, which allows for proper tracking, as required for effective software management and quality assurance practices. These documents are maintained under revision control and distributed with the software.

The requirements document is composed of five major sections:

- Project Requirements that are related to the execution of the project as a whole;
- Nuclear Fuel R&D Requirements, which are derived from the Project Requirements, that are related to the nuclear fuel software research and development that is necessary to meet the overall requirements of the project;
- Computational Science R&D Requirements, which are derived from the Project and Nuclear Fuel R&D Requirements, that are related to the research and development requirements of the computational science software infrastructure;
- Software Design Requirements, which are derived from the Project, Nuclear Fuel R&D, and Computational Science R&D Requirements, define the requirements on the design of the software to meet the needs of the project; and
- Human Factors, because the project will be executed by individuals, and for individuals, with specific needs that must be acknowledged and accounted for.

The high-level requirements for the project were identified as:

1. Deliver a product useful to nuclear fuel designers and for embedding in reactor simulations;
2. Proactively address the future needs of stakeholders through education, strategic research, and preparatory development;
3. Utilize well defined processes for efficient, high-quality software development and research;
4. Maintain effective collaborations and interfacing with user community and stakeholders; and

However, the mission for the AMP Nuclear Fuel Performance code changed significantly in the middle of the fiscal year, and the vision and requirements will need to be updated accordingly.

3 Build system

The build system of AMP has been upgraded to use CMake [4], with the associated testing harness (CTest) and reporting dashboard (CDash) [5], from KitWare, Inc [6]. The Nuclear Energy Advanced Modeling and Simulation (NEAMS) Enabling Computational Technologies cross-cutting element identified these KitWare Inc. tools as a high-quality set of tools that the NEAMS Integrated Performance and Safety Codes (IPSCs) should migrate towards.

In FY10, during the initial prototyping stage of AMP, an internal (called Nemesis, and associated with Scale/Denovo [7]) build, configure, test environment was used that was based on AutoTools, but incorporated many "homegrown" python scripts for testing and reporting errors. This had become cumbersome to maintain and enhance and the configuring of AMP took up to 20 minutes. Nemesis did not allow for regular testing reporting the results to a common location, as is easily integrated with the suite of tools from KitWare, Inc. By upgrading the build system to CMake, the build scripts are significantly simplified, and execute much more quickly (under one minute). An example script is shown here:

```
 cmake                                                  \
     -D AMP_DATA:PATH=${AMP_DATA_DIR}                   \
     -D CMAKE_C_COMPILER:PATH=${CC}                     \
     -D CMAKE_CXX_COMPILER:PATH=${CXX}                  \
     -D CMAKE_Fortran_COMPILER:PATH=${F90}              \
     -D TRILINOS_DIRECTORY:PATH=${TRILINOS_DIR}         \
     -D PETSC_DIRECTORY:PATH=${PETSC_DIR}               \
     -D PETSC_ARCH:STRING=${PETSC_ARCH}                 \
     -D LIBMESH_DIRECTORY:PATH=${LIBMESH_DIR}           \
     -D LIBMESH_HOSTTYPE:STRING=${LIBMESH_ARCH}         \
     -D LIBMESH_COMPILE_TYPE:STRING=dbg                 \
     -D COMPILE_MODE:STRING=debug                       \
     -D SUNDIALS_DIRECTORY:PATH=${SUNDIALS_DIR}         \
     -D X11_DIRECTORY:PATH=/usr/                        \
     -D MPI_DIRECTORY:PATH=/usr/                        \
     -D BLAS_DIRECTORY:PATH=${BLAS_DIR}                 \
     -D BLAS_LIB:STRING=blas_LINUX.a                    \
     -D LAPACK_DIRECTORY:PATH=${BLAS_DIR}               \
     -D SILO_DIRECTORY:PATH=${SILO}                     \
     -D HDF5_DIRECTORY:PATH=${HDF5}                     \
     -D HYPRE_DIRECTORY:PATH=${HYPRE_INSTALL_DIR}       \
     ../../../AMP-source/
make
```

This is both much simpler for the user and less error prone. Additionally, a significant amount of checking occurs within the build system to make sure the necessary libraries and dependencies are properly installed before building.

Adding new files and directories to AMP has also been simplifies significantly by migrating to CMake. Previously, each directory had to have a number of files used by the configure system to configure the directory and set the dependencies. All of the tests had to be included in a Makefile for them to be caught by the build system. Now, each source directory needs a very simple CMakeLists.txt file that indicated how to add files. An example file is:

```
INCLUDE (macros)
BEGIN_PACKAGE_CONFIG ( ampmesh )
INSTALL_AMP_TARGET ( ampmesh )
ADD_SUBDIRECTORY ( test )
```

Though this requires only four lines, it tells the build system to add all files in the current directory as part of the “ampmesh” library, and then as the “test” sub-folder to the build. The “test” sub-folder includes its own CMakeLists.txt file that will describe how to add each test, but it’s structure is also very simple. The build system is then able to properly sort the dependencies. If we compare this with the original build system, each folder needed several files totaling more than 200 lines per folder. Additionally, the dependencies were manually set within each file. This required the developers to know and maintain the dependencies for each directory as other directories were added or modified, which made the process of adding new folders and restructuring folders very difficult and error prone.

In addition to improvements to the build system, the build refactorization has allowed the use of CTest to automate testing on a regular basis. The user can run all the tests using a simple “ctest” or “make test” command to verify correct installation. Regular testing is performed on a nightly and continuous basis to ensure changes to the code do not affect the tests. These results are then automatically submitted to a website running CDash for summary results. This allows for rapid identification of problems that may or may not be platform dependent. Additionally, the testing system performs a coverage analysis on the entire code, and Valgrind [8] testing of most of the tests on a nightly basis. The results of which are also posted to CDash. Because the CDash webpage is outward facing, regression testing on machines that are outside the Oak Ridge National Laboratory internal network, such as Los Alamos National Laboratory, Idaho National

Laboratory, the University of Tennessee-Knoxville, and the National Center for Computational Science. The nightly reporting and summarized results have proven valuable and have significantly improved development productivity.

4 Materials

The material interface has been redesigned to simplify the interface, bring it in line with the basic design of AMP, and allow for a vector- based interface, including multi-vectors. The original material interface had a number of issues that needed to be reworked, specifically:

- All properties were identified by a traits class that was defined in one place and did not allow for a user to easily add additional material properties. The given property was selected by a switch statement that needed to be modified each time a property was added or removed, which made it difficult for users to add properties and created the potential for user error.
- The approach to passing arguments (temperature, stoichiometry, burnup) to the property function were hardwired. All properties needed to take the same arguments in the same order. For the small set of properties and arguments considered in the prototyping stage, this proved workable. However, as the number of properties and potential input arguments (fission gas constituents, actinide concentrations, etc.) increased, this required changes to all materials because of the addition of a single argument to any property.
- The material properties were evaluated through a single eval call on a single point, though it could be extended to a standard vector, while most of AMP works with AMP vectors, that are parallel aware and may contain multiple vectors within a single object. As a result the operator would regularly loop through the vector and call the material property function many times. This produced excessive amounts of repeated code, potential coding errors, and a significant performance issues because of the amount of overhead associated with calling the property function for each point. For more complex models, such as gap conductance, this would easily create a major bottleneck.

To address these deficiencies, the material interface has been redesigned to be easily modified and provide a consistent vector interface. The new design allows for each property to define the inputs required, and provides the desired vector interface that simplifies the connection with AMP and can be better optimized for performance.

A summary of the original and new interface is provided below:

4.1 Original Architecture

The architecture of the materials library was developed from `Property` and `Material` classes. A `PropertySpec` object contains all the information necessary to characterize a material property:

```
template<typename Number>
class PropertySpec {
public:
    PropertyType d_type; ///< integer identifier from enum PropertyType
    std::string d_name; ///< string name
    std::string d_source; ///< journal or report reference: where did model come from?
    std::valarray<Number> d_params; ///< parameters
    unsigned int d_nparams; ///< number of parameters
    unsigned int d_n_arguments; ///< number of arguments to evalv function
    std::vector<std::string> d_arguments; ///< names of the arguments in the function
}
```

It was meant to act as a `struct` with defaults, like a `Parameters` object for `AMP::Operator`. The `Number` template argument allowed for the definition of `float` or `double` material properties, although at the time only `double` properties are used. A `PropertyBase` object contained a `PropertySpec` object and accessors (get and set) to the items in it. The `Property` class was derived from `PropertySpec` and merely adds evaluators; the only one of which originally used was:

```
template<typename Number>
class Property: public PropertyBase<Number> {
public:
    virtual void evalv(std::vector<double> &r,
                       const std::vector<std::vector<double> > &vv);
}
```

It was intended wherever possible, but not required, that each `Property` also have a non-virtual `eval` function that was fast and inlineable, whose arguments are scalars, should anyone so wish to use it. The virtual `evalv` function is meant to "evaluate a vector of results". The "v" at the end of "evalv" stands for "vector". The `eval` function is layered on in the `Prop` class defined in `Helpers.h` and looks like:

```
template<PropertyType type> class Prop: public Property<double>
{
public:
    double eval(const double a0=0, const double a1=0, const double a2=0);
}
```

The `PropertyType` template argument was an `enum` that enumerates all the possible different property types. The function `eval` was defined in the usable material definition files such as `UO2_MATPRO.h`:

```
template<> inline
double Prop<ThermalConductivity>::eval(const double T,
                                       const double u,
                                       const double burn){
    double dpor, gadoln;
    dpor = 0.955;
    gadoln = 0.;
    double ax = p[11];
    double bx = p[12];
    double T2 = T*T;
    double fact = pow( burn, p[5] ) / (1.0 + p[6]*exp( -p[7]/T ) );
    double phonon = ax + bx*T + p[0]*gadoln + p[1]*burn
                  + ( 1.0 - p[2]*exp( -p[3]*burn ) ) * p[4] * fact;
    phonon = 1./phonon;
    double electron = p[8]/T2 * exp( -p[9]/T );
    double fm  = p[10] * dpor / ( 1. + 0.5*(1. - dpor) )    ;
    double thcond = ( phonon + electron ) * fm;
    return thcond;
}
```

The intermediary `Prop` class was intended to enforce uniformity among all the properties for a material, such as a common base name, source and names and numbers of arguments.

A material object would provide access to one or more related, pre-defined, internal property objects through the `evalv` function. The following interface was used by the current operators (such as mechanics or diffusion):

```
class Material {
public:
    virtual void evalv(const PropertyType type,
                       std::vector<double> &r,
                       const std::vector<std::vector<double> > &vv);
}
```

Where the vector-of-vectors vv contains the input arguments.

4.2 New Materials Interface

There were four primary changes made to the materials interface, as discussed below.

4.2.1 Merging of base classes

First, the `PropertySpec`, `PropertyBase`, and `Property` classes in file `Property.h` were all merged. Separating them only caused confusion amongst developers, and it only required changes to `Helpers.h`. The new `Property` class acquired new data and methods, and is now abstract:

```
template<Number>
class Property {
public:
    /**
     * \brief Names of arguments of evalv that will be defaulted at eval time.
     * A host code does not need to supply all the arguments to evalv.
     * This list tells the property what default arguments will not be supplied.
     * Selected from d_arguments.
     */
    std::vector<std::string> d_defaultArguments;

    /**
     * \brief The default values for defaulted arguments to evalv.
     * Size and order must match names in d_defaultArguments.
     */
    std::vector<Number> d_defaultArgumentValues;

    /**
     * \brief The names of arguments that will actually be supplied to evalv.
     * Order is defined here and will be remembered.
     * Selected from d_arguments.
     * The union of d_sequence and d_defaultArguments must be d_arguments
     * (except for order).
     * There must be no overlap of d_sequence and d_defaultArguments.
     */
    std::vector<std::string> d_sequence;

    /**
     * \brief Evaluate a vector of results.
     * Template argument:
     *
     * VTYPE   A vector type. Must support
     *         VTYPE::iterator
     *         VTYPE::const_iterator
     *         VTYPE::iterator begin()
     *         VTYPE::iterator end()
     * Constraints: type of *VTYPE::iterator is Number.
     *              type of *VTYPE::const_iterator is const Number.
     *
     * \param r vector of results
     * \param args input arguments, corresponding to d_sequence.
     */
     template <class VTYPE>
     virtual void evalv(VTYPE &r, const std::vector<boost::shared_ptr<VTYPE> > args)=0;
}
```

Constructor arguments with defaults, and accessors were also added. This allowed host code to access individual properties in a material with different sets of non-defaulted arguments, and all other versions of `evalv` were removed.

4.2.2 AMP naming conventions were incorporated

The class `Prop` was renamed the `PropertyHelper` class, and the `Helpers.h` file was renamed the `PropertyHelper.h` file, to conform to AMP naming schemes. The `eval` and `evalv` functions in the `PropertyHelper` class are rewritten:

```
class PropertyHelper: public Property<double> {
public:
    double eval(const std::vector<double> args);

    template <class VTYPE>
    void evalv(VTYPE &r, const std::vector<boost::shared_ptr<VTYPE> > args){...}
}
```

4.2.3 Input of a single argument list

All properties in the usable material definition files were changed to accept a single argument of `std::vector<double>`, such as in `UO2_MATPRO.h`:

```
double PropertyHelper<ThermalConductivity>::eval(std::vector<double> args){
    double T=args[0], u=args[1], burn=args[2]; ... }
```

The `args` variable is full-length; all defaults and non-defaults are supplied in the correct order (specified by `d_arguments`) by `evalv`. It may have different lengths for different properties even within the same material.

4.2.4 Pointer versions were removed

The pointered versions of `evalv` are removed from `AMP::Material` and `AMP::MaterialBase` classes, and accessors for the new property data members `Property::d_defaultArguments`, etc. are added. New versions of `evalv` are created:

```
class Material {
    template <class VTYPE>
    virtual void evalv(VTYPE &r, const std::vector<boost::shared_ptr<VTYPE> > args)=0;
}

template<class Traits>
class MaterialBase: public Material {
    template <class VTYPE>
    virtual void evalv(VTYPE &r, const std::vector<boost::shared_ptr<VTYPE> > args);
}
```

This is the only version of `evalv` that is now allowed.

4.2.5 Material operator

A new material operator is in the process of being created in `trunk/source/src/operators/MaterialOperator.h`:

```
class MaterialOperatorParameters: public Parameters{...};

class MaterialOperator:: public Operator {
public:
    virtual void reset(const boost::shared_ptr<OperatorParameters>& params);

    virtual void apply(const Vector::shared_ptr &f,
        const  Vector::shared_ptr &u, Vector::shared_ptr  &r,
        const double a = -1.0, const double b = 1.0);

    virtual boost::shared_ptr<OperatorParameters>
    getJacobianParameters(const boost::shared_ptr<Vector>& );
private:
    std::vector<boost::shared_ptr<Material> > materials;
}
```

The `MaterialOperatorParameters` class contains specifications for defaults, specific usable material class names, etc. It will handle multiple materials, so that there needn't be multiple copies in a calculation. The `reset` function will allow these to be changed, as well as which materials are active. The `apply` function will return `b f - a evalv(u)`, roughly, for the active materials. It will exercise the `MaterialBase::evalv` function with `VTYPE=AMP::Vector`. The `getJacobianParameters` function will be null in the initial version, but eventually could supply derivatives of material properties for the construction of more accurate Jacobians.

5 Mechanics

The mechanics component of AMP was extended to support large deformation analysis using the Updated Lagrangian approach, in addition to supporting small deformation analysis. The small deformation analysis is a basic capability that is sufficient for some problems; this capability was developed as part of the initial prototyping effort in AMP. The large deformation analysis is more accurate for many problems and is necessary to simulate problems involving large deformations, such as clad ridging, clad thinning, assembly bowing and grid deformation and to simulate problems involving long time periods such for studying cladding integrity for long term storage. The large deformation analysis is also required for updating the geometry/mesh in coupled multi-physics simulations.

6 TrilinosMLSolver

The TrilinosMLSolver class in AMP provides an interface to the ML solver, which is part of the Trilinos library. The prototype version of this class was only using the Epetra (also part of Trilinos) interface to ML and not directly interfacing to ML. This had the following restrictions.

- It required access to Epetra matrices. This made it unusable in situations when the matrix is stored in some other format for example as a Petsc matrix or when it undesirable/infeasible to assemble the matrix.
- The Epetra interface to ML did not support all the options of the ML solver package.

The new refactorization lifted these restrictions by adding a direct interface to ML inside the TrilionsML-Solver class.

7 Mesh

AMP is currently using Libmesh [9] for all of the mesh interface and the finite element formulation. This was originally done to save time and to get AMP working as a prototype. However, there are external collaborators, which require that AMP interface with other mesh packages for efficient communication (e.g. MOAB [10], STKmesh [11]) The design of AMP is flexible and will allow for a consistent mesh interface that could be based on a large number of possible mesh databases and packages. Additionally, we will separate the concepts of mesh and discretization so that different mesh packages could be mixed with different discretizations. For example, using a STKMesh from Sierra with the finite element formulation from Libmesh. To do this we are in the process of redesigning and implementing a new mesh interface that will be used within AMP. Behind the interface, there may be a number of mesh programs, including the possibility for AMP to have it's own mesh capabilities, if ever required. The existing mesh design is largely a direct copy of Libmesh functionality, which is not discussed in this report.

Fig. 1 shows the flow diagram for the new mesh and discretization interface.

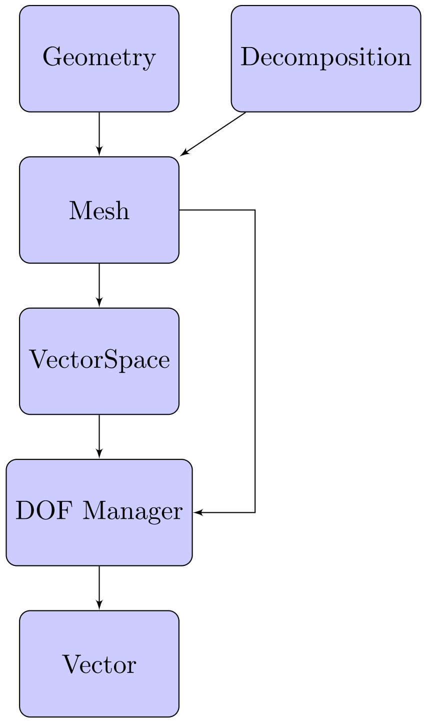

Figure 1: The mesh and descritization flow diagram

7.1 Geometry

The geometry defines the object in space. The functionality provided is undetermined at this point and intentionally left open to allow for future needs, where the integration of geometry and mesh occur. This is regularly seen in geometry-aware adaptive mesh refinement and geometry-based physics modules (Monte Carlo radiation transport).

7.2 Spatial Domain Decomposition

The decomposition defines how a mesh will be distributed across a given communicator. It is mesh-type specific and flexible based on the specific partitioner and mesh formulation.

7.3 Mesh

A mesh is a locally supported discretization of a continuous domain defined by the geometry. By discretization, we mean the values of a function on the continuous domain are sampled at a finite number of locations in the domain. By locally supported, we mean a low-order piecewise interpolant through nearby points can be consistent with the continuous function throughout a subset of the domain defined by the points.

The resulting geometric objects are referred to as vertex (points in space), edges (line connectivity between two verticies), faces (convex surface defined by 3 or more verticies), and volumes (simple volume element defined by 4 or more surface elements). The different geometric objects can be identified by their enum value (or resulting dimension), and can exist in equal or higher spatial dimension. For example, a vertex is a simple point with an enum value of 0, and can exist in any spatial dimension. An edge is a line between 2 points, has an enum value of 1, and can exist in 1D, 2D, or 3D space. A face is convex object with an enum value of 2 and can only exist in 2D or 3D. A volume is a simple volume object with an enum value of 3 and can only exist in 3D space. Note that generalization to higher dimension objects is straightforward and supported through their enum value (their names have not been defined at this point). The mesh can be viewed as the complete collection of all geometric objects and thier relations.

There are several different methods for storing meshes, the most common for scientific applications being "connectivity lists." A connectivity list is two arrays, one that provides locations of nodes and another that provides the nodes that make up cells. Packages such as libMesh rely on connectivity lists to store and manipulate meshes. These packages provide abstractions to intuitive mesh types such as sides, faces, cells, and nodes.

Other packages such as the SIERRA ToolKit rely on two abstract concepts: a mesh object and a relationship. A mesh is defined as a collection of these objects and relationships. In these packages, a mesh object has data, such as displacement, temperature, etc., and dimensionality, a tag that indicates where in the hierarchy of mesh objects this object is placed. A relationship is just that, a member of the set of possible relations of mesh objects. In order to provide support for computational science, the set of possible relations is well-ordered.

These two approaches both provide the basic concepts of cell and node. These approaches also lend themselves to use of the iterator popularized by C++. While random access of nodes and elements may require $O(\log n)$ computation, iterative access is accomplished in $O(1)$. The AMP mesh relies on this idiom to abstract away from the developer the underlying storage mechanism.

Mesh Class

The mesh base class provides the basic interface for accessing and creating different mesh objects. All mesh objects are inherited from this base class. A brief description of the interface is given below:

Constructors:

There are several constructors that will create a new mesh. The first constructor takes a MeshParameters object that contains the necessary information for the construction of a new mesh. The second constructor takes an existing mesh and returns a new mesh. This is useful for converting one mesh object to another. Finally, the subset command subsets the mesh given a MeshIterator, returning a new mesh.

```
Mesh ( const MeshParameters::shared_ptr & );
Mesh ( const Mesh::shared_ptr & );
boost::shared_ptr<Mesh> Subset ( MeshIterator::shared_ptr &iterator );
```

Mesh iterators:

MeshIterators provide a method of iterating over a given set of mesh elements. The mesh provides a number of default iterators of different types. The first method will iterate over all of the objects of a given size including the specified ghost-cell-width (gcw). GeomType is an enum that specifies the type of object that we want to iterate over (Vertex, Edge, Face, Volume). The second iterator will iterate over the objects of a given type on the surface. Finally, the last command will perform basic set operations on given iterators to generate new iterator types. SetOP is an enum that specifies the desired set operation (Union, Intersection, Complement).

```
MeshIterator getIterator ( GeomType &type, int gcw=0 );
MeshIterator getSurfaceIterator ( GeomType &type );
MeshIterator getIterator ( SetOP                     &OP,
                           MeshIterator::shared_ptr &A,
                           MeshIterator::shared_ptr &B);
```

Multimesh:

Multiple meshes will be supported through the use of a MultiMesh object. This object will inherit from the base mesh class, and will provide additional functionality by combining multiple mesh objects. In addition to the basic iterators, an iterator over the mesh objects will also be provided.

```
MultiMeshIterator getIterator ( );
```

For most applications, data will be needed that exists outside the mesh defined on the local processors. The method of accessing this data is through the use of a ghost cell width (*gcw*). In this context, the ghost cell width specifies the number of neighboring objects outside the current local portion of the mesh. The mesh is responsible for returning an iterator that includes the desired number of ghost neighbor elements. For example, given a $gcw = 1$, the returned iterator should be able to iterate over all local elements and all immediate neighbor elements. To limit the communication that is necessary when creating an iterator, the mesh may require a maximum ghost cell width that should be supplied by the user during construction. Then when a ghost iterator is requested, it must be less than or equal to this maximum value. It is the mesh's responsibility to be able to create all iterators up to and including this maximum value ($0 <= gcw <= MAX\_GCW$).

MeshIterator Class

The MeshIterator class is a C++ iterator over a group of elements. It is provided by the Mesh base class and is the basic method that will be used to build more complex objects (such as the Discretization iterators and the Degree Of Freedom (DOF) Manager). The functions provided are those of a standard iterator.

```
Constructor:
   MeshIterator(const MeshIterator& mit);
Increment/Decrement:
   MeshIterator& operator++();
   MeshIterator operator++(int);
   MeshIterator& operator--();
   MeshIterator operator--(int);
Comparison:
   bool operator==(const MeshIterator& rhs);
   bool operator!=(const MeshIterator& rhs);
Dereference:
   MeshElement& operator*();
```

MeshElement Class The concept of a MeshElement was previously discussed in the discussion of Mesh, and represents the storage of a simple geometric object that composes the most basic piece of a mesh. The type of the mesh element is identified by it's enum value (vertex=0, edge=1, face=2, volume=3), and can exist in any dimension space that is greater than, or equal to, it's current dimension. The MeshElement class provides routines for accessing and identifying mesh elements and the connectivity.

```
Return the element type (enum value):
   GeomType getElementType();
Return the unique global id:
   ID getGlobalId();
Return the local id:
   int getLocalId();
Return the elements composing the current element:
   vector<MeshElement> getElements(GeomType &type);
```

7.4 Discretization

The discretization namespace provides classes needed for constructing data iterators that are used to constructing vectors on a mesh. It represents the intermediate step of relating the mesh elements to a discretization of the function values on the given mesh. For example, it allows the creation of basic DOF managers like a nodal scalar value and more complex operations like a finite-element discretization. These capabilities are provided through an option VectorSpace Class, and a DOF Manager which in turn creates the DataIterators necessary for the construction of the Vectors. The DOF Manager is also responsible for providing the relationships between a MeshElement, the degree of freedom (DOF), and the storage element in the vector.

VectorSpace Class

The VectorSpace class provides additional functionality for discretizing the function on the mesh. It contains information like the basis set used, and feeds into the DOF Manager.

DOF Manager Class

The DOF Manager class provides the necessary relations between the mesh element the VectorSpace, the degree of freedom, and the storage of the data. It provides the necessary DataIterators for Vectors to allocate the underling storage as well as the means for identifying and accessing a particular data element. Internally it requires a pointer to the underlying mesh, and the vector space (if used). Optionally, it may provide for a different parallel decomposition from the underlying mesh.

DataIterator Class

The DataIterator is a basic iterator that allows for the construction of data objects (like Vectors). The relationship between a particular data element, a mesh element, and the degrees of freedom is determined by the DOF Manager which created the given DataIterator

8 Seperation of AMP and FUEL

Currently, the AMP Nuclear Fuel Performance code is a single code and application infrastructure used to simulate the behavior of nuclear fuel during irradiation. However, much of AMP can be used for a wide variety of problems that involve coupled complex physics, and can take advantage of the capabilities designed in AMP. Doing so would allow multiple projects to leverage common resources, and increase productivity. To this end, we are in the process of splitting the AMP Nuclear Fuel Performance code into two codes: the AMP package that is responsible for providing the application infrastructure and common operations like basic physics operators (mechanics, diffusion, flow), and the Nuclear Fuel Performance code (commonly referred to as AMPFuel). The fuel code will contain all of the export and proprietary pieces that cannot be distributed as part of the AMP package. The AMP package by contrast will be an open source code containing the interfaces, routines for mesh management and mapping operations, physics operators like thermal transfer, general purpose solvers and time integrators, and the linear algebra operations. This split is being done so that AMP can be distributed as a stand-alone package, while AMPFuel will still appear nearly seamless to the fuel performance code developer or end-user. Part of this is done through careful management of the repository and build system. The AMP build system integrates the necessary external packages, creates a standalone build, and creates a general purpose cmake file that subsequent builds (like AMPFuel) can use to link AMP. The AMPFuel package then uses a very simple build script and CMake file to get all of the packages that were used to build AMP to create the AMPFuel build. An example of the AMPFuel build script is shown below. As can be seen, very few variables are set, most of them are set when building AMP, and only fuel-specific variables are needed. This makes it very easy to build the combined package.

```
cmake                                                \
   -D AMP_DIRECTORY:PATH=${PWD}/../AMP/ampdir       \
   -D FUEL_DATA:PATH=${FUEL_DATA}                   \
   -D ORIGEN_DATA_PATH:PATH=${FUEL_DATA}/origen     \
   -D USE_ORIGEN=1                                  \
   -D USE_DENOVO=0                                  \
   -Wdev                                            \
   ../../../FUEL-source/
```

To further simplify the union of AMP and AMPFuel, the documentation for AMPFuel will integrate the documentation of both AMP and AMPFuel into a single documentation build so that the user does not need to maintain two sets of documentation. Currently, the AMP and AMPFuel source code and build systems have been fully separated, and we are in the process of separating the repositories and make AMP into an open-source code.

9 AMP_MPI

All MPI calls and routines were refactored into a single object-based AMP_MPI class. Previously, MPI communications were being done inconsistently through three different methods (calling MPI routines directly, using an existing AMP_MPI struct that was based on a single (global) communicator, and using routines for unit testing from the nemesis build system). Not only was this inconsistent, it created problems when working with multiple communicators (AMP_MPI is designed to effectively use multiple communicators), resulting in the potential for bugs and memory leaks involving the MPI communicators. Additionally, all builds required MPI to be present. While most users will want to use MPI, the possibility for a serial build is useful for some users.

To overcome these issues the AMP_MPI struct was redesigned to be an MPI-based class object. Internally an AMP_MPI object contains an MPI communicator (for parallel builds) or a reference integer (for serial builds), and then wraps all of the MPI calls within similar member functions. These member functions are designed to be very lightweight will and work for both parallel and serial builds. Additionally, we are able to template many of the communication routines based on the data type. This significantly reduces the complexity of communication, reducing the likelihood of new bugs. All of AMP has been modified to use this new capability, and builds of AMP in parallel and series have been completed successfully, and are part of the nightly regression.

10 Acknowledgement, Access, and Citations

The development of the AMP Nuclear Fuel Performance code was funded by the Fuels Integrated Performance and Safety Code (Fuels IPSC) element of the the Nuclear Energy Advanced Modeling and Simulation (NEAMS) program of the U.S. Department of Energy Office of Nuclear Energy (DOE/NE), Advanced Modeling and Simulation Office (AMSO).

The export controlled AMP Nuclear Fuel Performance code is available through the Radiation Safety Information Computational Center (RSICC) for limited users, by returning the completed forms at this web address: `http://rsicc.ornl.gov/rsiccnew/CFDOCS/amprequest_form1.cfm`. The source code is maintained on a repository at the Oak Ridge National Laboratory.

At present, citations related to the AMP Nuclear Fuel Performance code should refer to reference [1]. Past citations included references [12] and [13].

References

[1] K. Clarno, B. Philip, W. Cochran, R. Sampath, S. Allu, P. Barai, S. Simunovic, L. Ott, S. Pannala, P. Nukala, G. Dilts, B. Mihaila, C. Unal, G. Yesilyurt, J. Lee, J. Banfield, and G. Maldonado, The AMP (Advanced MultiPhysics) Nuclear Fuel Performance Code, Technical Report ORNL/LTR-2011/42, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 2011.

[2] K. Clarno, B. Philip, W. Cochran, J. Billings, and G. Dilts, Vision Document for the AMP Nuclear Fuel Performance Code, Technical Report ORNL/LTR-2011/41, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 2011.

[3] B. Philip, W. Cochran, K. Clarno, and J. Billings, Requirements Document for the AMP Nuclear Fuel Performance Code, Technical report, Oak Ridge National Laboratory, 2010, Within the source documentation of the AMP Nuclear Fuel Performance code.

[4] `http://www.cmake.org`.

[5] `http://www.cdash.org`.

[6] `http://www.kitware.org`.

[7] T. Evans, A. Stafford, and K. Clarno, *Nuclear Technology* (2009), submitted for publication.

[8] `http://www.valgrind.org`.

[9] `http://libmesh.sourceforge.net/`.

[10] T. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst, MOAB: A Mesh-Oriented Database, Technical Report SAND2004-1592, Sandia National Laboratory, 2004.

[11] `http://trilinos.sandia.gov/packages/stk`.

[12] G. Dilts, B. Mihaila, C. Unal, K. Clarno, B. Philip, W. Cochran, R. Sampath, S. Allu, P. Barai, G. Yesilyurt, J. Lee, S. Simunovic, L. Ott, J. Turner, S. Pannala, P. Nukala, and J. Billings, The AMP (Advanced MultiPhysics) Nuclear Fuel Integrated Performance and Safety Code, Technical Report LA-UR-10-8450, Los Alamos National Laboratory, Los Alamos, NM, USA, 2010.

[13] J. Turner, K. Clarno, and G. Hansen, Roadmap to an Engineering-Scale Nuclear Fuel Performance and Safety Code, Technical Report ORNL/TM-2009/233, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA, 2009.