Technical Note

Subject: LIME 2.0 Architecture and Design : Version 0.5 (Rev. 1)

Executive Summary

The architecture and design of LIME 2.0 is described. LIME 2.0 provides the foundation for multi-physics coupling in the CASL VERA software collection and will be reused in a number of other software efforts outside of CASL.

Contents

1 Introduction 2
2 Additional background 2
3 Subpackage and dependency structure for LIME 2.0 2
4 LIME 2.0 subpackages 2
4.1 LIMELite 3
4.1.1 Properties of LIMELite 3
4.1.2 Basic LIMELite interfaces and supporting classes 4
4.1.3 Other possible issues to consider for LIMELite 8
4.2 LIMEImplicit 9
5 Coding and development standards for LIME 2.0 10
6 Summary and next steps 11
7 Endorsers of LIME 2.0 design plan 11
1 Introduction

The domain model for multi-physics coupling that the design for LIME 2.0 is based on is given in [16]. This domain model provides the terminology and concepts used in this design document.

There are a number of drivers for the creation of a LIME 2.0 package but the primary driver is the CASL project. LIME 2.0 will form the foundation for the VERA Base collection of packages [1].

The existing LIME 1.0 software package has a number of limitations that require the creation of a new version in order to satisfy the growing requirements of multi-physics coupling in VERA and other related projects. However, this document will not provide a code review of the existing LIME 1.0 implementation. Instead, a listing of desired properties of LIME 2.0 will be stated and these can be compared to LIME 1.0.

The CASL VERA project is partitioned into two major tracks of development. The first track of development is to assimilate a number of existing legacy simulation codes and provide the most basic type of coupling to create an incrementally improved simulation capability. However, very little refactoring of these existing codes will take place and therefore very little extra functionality will be expressed in these codes. The second track of CASL VERA development will be to couple together more modern and more actively developed simulation codes that can provide greater functionality. These two development tracks drive and shape the design of LIME 2.0 and will drive its implementation.

The requirements for LIME 2.0 will be listed implicitly with the properties each of the subpackages of LIME 2.0 described below.

2 Additional background

In addition to the domain model for multiphysics coupling provided by the LIME Theory document [16], a little more background needs to be covered before describing the design of LIME 2.0.

One important concept to describe is the idea of Abstract Numerical Algorithms (ANAs). ANAs are comprised of algorithms that only deal with abstract operators, vector spaces, functions on those spaces and algorithms built up from those abstractions [6, 2]. This software knows nothing of parallel computing, meshes, or PDE discretizations. Examples of this type of software from Trilinos include NOX, Rythmos, and MOOCHO. All of these Trilinos packages accept problems expressed in the Trilinos standard ANA API called Thyra [2]. Any algorithms expressed in pure Thyra objects almost automatically leads to ANA software. In addition, given the black-box nature of most DAKOTA algorithms, DAKOTA would also be categorized as a type of ANA software since it does not see the underlying mesh, discretizations, or details of the parallelization of the physics solves. All of the core interface and solver classes described for LIME 2.0 will all meet the basic requirements of ANAs; some the DAKOTA variety and others more the Thyra variety.

3 Subpackage and dependency structure for LIME 2.0

Figure 1 shows the main LIME2 package broken down into two primary subpackages: LIMELite and LIMEImplicit. The first subpackage is a light-weight version of LIME (i.e. LIME “Lite” or LIMELite) for steady-state and transient solvers using Black-Box Picard Iterations. The second subpackage LIMEImplicit contains all of the additional interfaces and more advanced support algorithms for Block-Implicit and Nonlinear-Elimination Newton Methods.

4 LIME 2.0 subpackages

The two major subpackages comprising lime, LIMELite and LIMEImplicit, are described below.
4.1 LIMELite

The LIMELite subpackage is designed to provide only the minimal support needed to couple together black-box legacy codes. In black-box mode, all of the state equations $f_i(\ldots)$ and state variables $x_i$ are eliminated using the model-specific solvers to yield the reduced response functions

$$\hat{g}_{i,j}(\{z_{i,k}\}, \{p_{i,l}\}), \text{ for } i = 0, \ldots, N_f - 1, \ j = 0, \ldots, N_{\hat{g}_i} - 1. \tag{1}$$

The system coupling parameters $\{z_{i,k}\}$ defined in (1) are determined through the transfer functions

$$z_{i,k} = r_{i,k}(\{x_m\}, \{p_{m,n}\}), \text{ for } i = 0, \ldots, N_f - 1, \ k = 0, \ldots, N_{z_i} - 1, \tag{2}$$

Note that the transfer functions $r_{i,k}(\ldots)$ in (2) still need to “see” the state variables $x_m$ from each given physics model $f_m(\ldots)$ but this is part of the elimination that is occurring to give the final set of fully reduced response functions

$$\hat{g}_{i,j}(\{p_{i,l}\}), \text{ for } i = 0, \ldots, N_f - 1, \ j = 0, \ldots, N_{\hat{g}_i} - 1. \tag{3}$$

In actuality, the reduced response functions from the different models will be combined or selected in some way to give a global set of responses and parameters

$$\hat{g}_j(\{p\}) = 0, \text{ for } j = 0, \ldots, N_{\hat{g}} - 1. \tag{4}$$

The form of the fully reduced and processed response functions in (4) after performing the full multi-physics solve is what would be seen by a higher-level driver like DAKOTA to perform parameter, UQ studies, and other automated studies.

The basic design of LIMELite is shown in Figure 2. Before describing the classes in Figure 2 in detail, we first state some of the basic properties of LIMELite software.

4.1.1 Properties of LIMELite

- LIMELite only supports black-box solves of the underlying model equations using built-in model-specific solvers (i.e. only the reduced response functions $\hat{g}_j(\ldots)$ are exposed, not the full equations $f_i(\ldots)$).
- LIMELite supports basic parameters and response functions throughout so as to facilitate parameter studies, UQ studies, design optimization, etc. (e.g. driven by DAKOTA).
LIMELite interfaces or algorithms will not support any parameter/response sensitivities or other types of more advanced calculations.

LIMELite only has dependencies on minimal utility software needed for safe memory management, object control, object introspection, and to provide a more general foundation for more advanced features (e.g. a dependence on core Teuchos software part of the Trilinos coding and documentation guidelines described in Section 5).

LIMELite interfaces and algorithms are ANAs and will therefore not have any direct concept of a communicator in the highest level interfaces or support algorithms.

LIMELite supports the hierarchical aggregation of solvers to allow embedding one multi-physics solve as a single model in higher-level multi-physics solvers, etc.

LIMELite supports arbitrary data transfers from any number of physics models to any other physics models. These interfaces will have no concept of a mesh or other complex concept of the underlying applications.

LIMELite will not define a single (or even a closed set) of concrete multi-physics solver classes or algorithms. The individual physics model adapters will not know about or point to any specific multi-physics coupling class. This is critical to allow for independence, testability, and reusability of the individual physics application adapters. (For example, the SIERRA Solution Control module could be refactored to work in terms of LIMELite interface classes.)

4.1.2 Basic LIMELite interfaces and supporting classes

The different classes in LIMELite shown in Figure 2 are described below.

**BlackBoxModelEvaluator** is an abstract interface for a black-box eliminated physics model and represents the reduced response function \( \hat{g}_{i,j} \) shown in (1). This is a statefull interface where the client can set current values of the parameters using a Parameters objects (see below) and then call solve() to converge the internal hidden state equations. If the solve is successful (as determined by isConverged()) then the values of the given response functions can be computed/extracted using the getResponses() function as a Responses object (see below). The solve() function takes an optional tolerance that can be chosen by the client to control how tightly the state equations are solved and can be used by an outer driver algorithm to progressively tighten the tolerance as fixed-point Picard iterations are performed. Not shown but a getValidParameters() function could be included that provides a listing of valid parameters and their names to allow clients to identify and pass in a subset of parameters. The class Parameters would then provide a subset of the full set of supported parameters leaving the other parameters as their default values.

**SomeSteadyStateApp1Adapter** represents an ADAPTER subclass that implements the basic BlackBoxModelEvaluator interface in terms of an existing steady-state legacy code shown as SomeLegacyApp1. Much of the work involved in integrating a legacy code into this system includes refactoring the legacy application (SomeLegacyApp1) and then in writing the adapter (SomeSteadyStateApp1Adapter) to set parameters, drive the solution process, and then extract response functions.

**TransferOperator** is an abstract interface that aggregates one or more compatible transfer functions \( r_{i,k} \) shown in (2) to extract the state variables \( \{x_m\} \) from a given set of converged “upstream” BlackBoxModelEvaluator objects in order to set one or more coupling variables \( z_{i,k} \) to a set of “downstream” BlackBoxModelEvaluator objects that will have solve() called on them. This approach provides complete flexibility in how physics models are coupled together allowing the state from one or more physics models to provide input to one or more other physics models. Subclasses such as SomeTransferOperator would use dynamic casting to get at the converged states from upstream BlackBoxModelEvaluator objects and set the coupling parameters on downstream BlackBoxModelEvaluator objects. Any number of mix-in interfaces or other approaches could be used to make these TransferOperator subclasses more or less general.
Figure 2: Core LIMELite interfaces and classes. (UML Class Diagram.)
very specific implementations could be hard-coded to a given set of concrete model subclasses (e.g. dynamic casting directly to SomeSteadyStateApp1Adapter) or more general implementations could dynamic cast to more general mix-in interfaces that utilize a common mesh/field interface to allow for greater reuse. How the transfers are done is of no concern to the higher-level ANAs implemented in terms of the LIMELite interfaces. However, a major task in coupling legacy application codes will be in refactoring the underlying legacy codes to expose states and coupling variables and in implementing these transfer subclasses.

TransientBlackBoxModelEvaluator is an abstract interface that derives from BlackBoxModelEvaluator and adds functions needed to drive a transient multi-physics time-step solve. The function getDesiredTimeStepSize() returns the time step size that the physics model wants to take next (i.e. to satisfy its stability or LTE targets). The function getMaxNextTimeStepSize() returns the maximum time step size that a physics model can handle, such as to satisfy a CFL limit for an explicit time integration method. The set of these two time step sizes could then be used by a black-box Picard solver to negotiate a single time step for all of the physics models. The function setNextTimeStepSize(dt) is called by the black-box solver to set the next time step and then solve() is called iteratively in a black-box Picard iterative method to converge the time-step equations for a single time step.

SomeTransientApp2Adapter is an ADAPTER subclass that implements the TransientBlackBoxModelEvaluator interface in terms of an existing transient legacy application code shown as SomeLegacyApp2. This is identical to the SomeSteadyStateApp1Adapter subclass described above but now for a transient application code that exposes its inner time-step solver. All of the same issues of difficulties refactoring the legacy code apply equally.

SteadyStateBlackBoxPicardSolver is an example of a steady-state black-box Picard solver implementation that LIMELite could contain. This is just one example of a concrete implementation of such an algorithm in LIMELite since LIME 2.0 will not mandate a single implementation. In order to construct a fully determined multi-physics system, a set of $1, \ldots, N_f$ model objects of type BlackBoxModelEvaluator are needed along with a set of $0, \ldots, N_r$ transfer objects of type TransferOperator and connectivity information about how the transfer objects map to model objects. It is this connectivityGraph (represented as the ConcreteBlackBoxModelEvaluator class) that defines the computational workflow for each iteration of a black-box Picard iterative method. For a simple 2x2 system of equations, the SteadyStateBlackBoxPicardSolver’s solve() function could perform the Picard iterative method as shown in Figure 3 (shown in Python-like pseudo code for the sake of readability and compactness). The Jacobi Picard iterative method shown in Figure 3 does not pass in tolerances or check residual norms but another version could (if all of the supporting physics models supported returning residual norm estimates and accepted tolerances). Also, the real implementation would contain loops and use the info in connectivityGraph to match up model objects with transfer calls but one should get the idea.

Figure 4 shows an example concrete instantiation of a 2-by-2 system of coupled physics models that is driven by an analysis study using DAKOTA. In this example, both of the steady-state physics models are legacy application codes wrapped as BlackBoxModelEvaluator objects. Also, in this example, the two TransferOperator subclass objects are specifically designed to transfer data for these concrete legacy application codes. When transferData(...) is called, dynamic casts are be performed to the concrete adapter subclasses in order to access the underlying legacy application code in order to extract converged states and to set coupling variables.

Finally, the class TransientBlackBoxPicardSolver provides a concrete implementation of a black-box transient solver that derives from the BlackBoxModelEvaluator interface (and can therefore be driven by a DAKOTA or other analysis study) and uses black-box Picard solves to converge the time-step equations.

---

1. The TimeRange class was defined in the Trilinos Rythmos package and defines a careful system for representing time ranges and inclusion or exclusion in a give time range taking into account floating point issues. Any piece of software for transient simulation that does not carefully handle time ranges will be fragile, unpredictable, and potentially inefficient as compared to an implementation that is careful. The class TimeRange simply and elegantly standardizes the handling of time and resolves these tricky issues.
SteadyStateBlackBoxPicardSolver::solve()

# Set parameters (splitting the global parameters into two subsets)
models[0].setParameters(parameters(subset 0))
models[1].setParameters(parameters(subset 1))

# Perform Picard iteration (Jacobi in this case)
converged = false
while not converged:
    # Perform transfers
    transfers[0].transferData(models[0], models[1])
    transfers[1].transferData(models[1], models[0])
    # Check for convergence
    if models[0].isConverged() and models[1].isConverged():
        converged = true
        break
    # Solve each physics model
    models[0].solve()
    models[1].solve()
    # Compute responses (taking from models[0] in this case)
    responses = models[0].getResponses()

Figure 3: Listing of a simple implementation of a steady-state black-box Picard solver.

Figure 4: Example instantiation of a 2-by-2 coupled system where BBME=BlackBoxModelEvaluator and TO=TransferOperator. (UML Object Diagram.)
TransientBlackBoxPicardSolver::solve()
  # Set up the stead-state Picard solver that will converge time-step equations
  steadyStateBlackBoxPicardSolver.initialize(models, transfers)
  # Set parameters for the transient solve
  steadyStateBlackBoxPicardSolver.setParameters(parameters)
  # Do the time-step loop
  currentTime = fullTimeRange.lower()
  while compareTimeValues(currentTime, fullTimeRange.upper()) < 0:
    # Find the next time step size to try
    maxTimeStep = fullTimeRange.upper() - currentTime
    maxTimeStep = min(models[0].getMaxNextTimeStep(), maxTimeStep)
    maxTimeStep = min(models[1].getMaxNextTimeStep(), maxTimeStep)
    desiredTimeSteps[0] = models[0].getDesiredNextTimeStepSize()
    nextTimeStepSize = Some value less than maxTimeStep while considering desiredTimeSteps[]
    # Set the next time step size and solve using picard iterations
    models[0].setNextTimeStepSize(nextTimeStepSize)
    models[1].setNextTimeStepSize(nextTimeStepSize)
    steadyStateBlackBoxPicardSolver.solve()
    # Update the current time
    currentTime += nextTimeStepSize
    # Extract the response at the final time (a terminal response in this case)
    responses = models[0].getResponses()

Figure 5: Listing of a simple implementation of a transient black-box Picard solver.

This class utilizes the time-step specific functions in the TransientBlackBoxModelEvaluator interfaces of each transient model to coordinate and set up the time-step equations for before a time-step solve. An embedded SteadyStateBlackBoxPicardSolver object is then (re)used to converge the time-step coupled equations for each time-step. An example of the implementation of the solve() function for a transient time-step loop for a simple 2-by-2 set of equations might look like shown in Figure 5 (again in Python-like pseudo code). The transient solve in Figure 5 has many simplifications over a more general implementation. It assumes that the time-step equations can always be solved and that the only response is a terminal response (however, every distributed response can be reformulated as a terminal response by adding more ODES to the model's transient equations). Also, again, of course the real implementation would use loops over the model objects. What this example does show is how the the steady-state Picard solver class can be used unchanged in the transient solver class and how the interface all work well together.

4.1.3 Other possible issues to consider for LIMELite

Additional issues that might be addressed in the design of LIMELite in the near future include:

- Concurrent evaluations of single-physics solves and transfer operators: For example, it might be desirable to distribute the first model's solve onto one set of 100 processors and another model's solve onto a different separate set of 1000 processors (load balanced so that they should both finish at about the same time) and then execute the two physics solves in parallel of each other on the two different sets (i.e. clusters) of processors. This could provide up to a factor of two reduction in wallclock time for parallel codes (and less so for serial codes but still useful). Supporting concurrent operations can be easily accommodated by providing “begin” and “end” versions of the solve() and transferData() functions. For example, all of the solves can be started with a call to non-blocking beginSolve() and then blocking wait operations can be invoked on all of the model solves by calling endSolve().
This allows all of the solves to take place in parallel of each other and the synchronize at the end. These functions can transparently be implemented on the backend in SPMD mode using (MPI) communicators or using threading but the LIMELite algorithms would be completely oblivious to this and therefore remain true ANAs.

- **Support for dampened fixed-point iterations**: A simple fixed-point Picard iteration may not converge without some dampening. If all of the models support the `getResidualNorm()` function and accept tolerances in the `solve(tol)` function, then one type of dampening can be accomplished by starting out with a loose solve tolerance and then reducing the tolerance as the multiphysics system converges. A more general approach to dampening would be for the transfer operators to not fully transfer the the state information from upstream models to the coupling variables of the downstream models but to instead use some type of continuation method to control how strong the transfers would be. This could be accomplished in a general way by adding a continuation parameter $\alpha$ (i.e. where $\alpha = 0.0$ is no transfer and $\alpha = 1.0$ is full undampened transfers) to the `TransferOperator` interface. Then, the outer dampened Picard solver would start with a small $\alpha$ and then over a number of iterations increase it to $\alpha = 1.0$ before converging the whole set of coupled physics equations.

- **Support for just first-order response/parameter sensitivities $\frac{\partial \hat{g}}{\partial p}$**: To support some of the more efficient and robust sensitivity, UQ and optimization methods, we might consider adding support for just the response/parameter sensitivity matrix for $\frac{\partial \hat{g}}{\partial p}$ to the `Responses` class. In this way, if an underlying `BlackBoxModelEvaluator` object could support this basic sensitivity matrix then it could be passed through the interface.

- **Support for text processing tools for setting parameters and extracting response functions by modifying input files and reading output files, respectively**: These utilities would be very useful in helping to couple black-box application codes that would need to read and write output files. Such a coupling approach would not be very high performance but could be a quick initial implementation that could be later refined to remove file-based manipulation. These file processing utilities may not go into the LIMELite subpackage itself but they could go into another utility subpackage in LIMELite or in some other part of Trilinos (perhaps in Teuchos) to be readily available for user of LIMELite to exploit.

### 4.2 LIMEImplicit

The LIMEImplicit subpackage would contain (or rely on other provided) interfaces and support software for implementing more sophisticated implicit multi-physics solution methods. Some of the critical properties of LIMEImplicit are given below.

**Properties of LIMEImplicit**

- LIMEImplicit will support Block-Implicit Newton Methods, Nonlinear-Elimination Newton Methods, and arbitrary combinations of these methods nested in with one another.

- LIMEImplicit core algorithms will be represented as ANAs and therefore not have any specific mention of communicators, meshes, or any other non-general mathematical entity that is inconsistent with ANAs.

- LIMEImplicit will define (or leverage) a set of tools for constructing various implicitly based multi-physics coupling algorithms and support software for various implementations but will not define a master solution class. This collection of solver algorithms will be extensible by outside developers without having to touch any LIMEImplicit source code (i.e. the Open-Closed Principle (OCP) [14]).

- LIMEImplicit constructed solvers will be representable as black-box models and can be incorporated into LIMELite black-box coupling algorithms.
LIMEImplicit will support full 64 bit address spaces to allow the largest simulations possible on the highest end super supercomputers to be constructed and used through the year 2020 or later (e.g. using Tpetra not Epetra).

LIMEImplicit core interfaces and algorithms will be 100% agnostic to the computing platform and will work, unchanged, in newer multi-core computing paradigms (e.g. OpenMP through GPUs using CUDA).

LIMEImplicit core ANA interfaces and algorithms support software will not have any required dependency on any concrete (parallel or serial) linear algebra implementation. However, adapters to necessary implementations can live in other subpackages (e.g. for Epetra, Tpetra, PETSc, SUNDIALS, etc.).

LIMEImplicit will leverage (and be leveraged by) external solver software for preconditioners, linear equations, nonlinear equations, stability analysis, time integration, embedded optimization, and embedded UQ and other advanced algorithms. LIMEImplicit will not itself contain complex generic algorithms would be more suited to exist in more general software collections (e.g. Trilinos, PETSc, etc.)

LIMEImplicit will support fully general parameters and response functions throughout so as to facilitate parameter studies, UQ studies, design optimization, etc. (e.g. driven by DAKOTA).

LIMEImplicit model interfaces will support the full range of extended derivative quantities such as Jacobians, gradients, Lagrangians, Hessians, etc. that are needed to efficiently implement a wide range of advanced embedded analysis and optimization methods.

LIMEImplicit will contain or leverage basic interfaces and support software to construct various operator split methods for time integration.

In future versions of this document, a design for LIMEImplicit will be defined that will describe the toolbox of classes and interfaces that will support the creation of a variety of block-implicit and nonlinear eliminating Newton methods and a limitless number of combinations of implicit and black-box solve methods.

5 Coding and development standards for LIME 2.0

In order for LIME 2.0 to provide a solid foundation for VERA Base (and therefore CASL) and for collaborative development, a number of different standards need to be considered. Several different categories of standards that need to be considered for LIME 2.0 include the following.

LIME 2.0 should follow accepted object-oriented and agile design best practices: Without using solid agile OO design principles, LIME 2.0 will be fragile, hard to modify and extend, and hard to maintain.

LIME 2.0 should adhere to the Trilinos Lifecycle Model 2.0: The new lean/agile consistent lifecycle model being developed for Trilinos should be used for the development of LIME 2.0 (as well as for all of VERA). The standards and practices outlined in this lifecycle model allow for a smooth transition from research-based code to solid production code, all the while keeping the software clean and exceptionally well tested.

LIME 2.0 should follow the Trilinos coding and documentation guidelines: These coding guidelines are based on expected best practices in design and C++ development and are augmented with items more specific to CSE codes building on the foundation of core Trilinos utilities in the Teuchos package. An important part of this is the Teuchos Memory Management standard.
LIME 2.0 should be developed and released as a BSD-licensed Trilinos package: Developing LIME 2.0 inside of the Trilinos system as a Secondary Stable add-on package will result in more portable code and greater feedback than what can be achieved just through CASL-specific efforts. Once LIME 2.0 is copyrighted, it can go into the main Trilinos repository and can then become Primary Stable code. Also, developing LIME 2.0 as a Trilinos package and copyrighting as BSD software provides an automatic quarterly release mechanism for the software (i.e. regular releases four times a year).

6 Summary and next steps

The design for LIME 2.0 described here lays out a strategy for well supporting the immediate needs of CASL to couple existing legacy codes with black-box solution methods (e.g. ANC/VIPRE/BOA) and for a second track of development to couple more modern codes with fully implicit approaches (e.g. Fully coupled Drekar and Denovo).

The development of LIME 2.0 can be initially carried out in a separate “LIME2” namespace and package as to not conflict with the existing LIME 1.0 code and coupling efforts. The existing LIME package can be made to optionally depend on this separate LIME2 package and adapters and drivers for the LIME2 interfaces and support software can be built side-by-side with the existing LIME 1.0 adapters and drivers. The LIME 1.0 based software will be refactored to reduce duplication with the LIME2 software. Eventually, all usage of the black-box solvers and interfaces in LIME 1.0 will be replaced with LIMELite in LIME2 for all usage in CASL.

The initial thrust of development will be to create LIMELite software using Teste Driven Development (TDD) (i.e. with excellent unit tests) and replicate and then replace the existing LIME 1.0 black-box adapters and drivers. A parallel development thrust will be to design LIMEImplicit and decide how to leverage and reuse software from Trilinos, LIME 1.0, and AMP to create the initial and growing capability. This initial implementation of LIMEImplicit will be driven by demonstration calculations with early modern VERA codes such as Denovo and Drekar.

The high-level design of LIME2 will evolve over time and this high-level architecture and design document will be kept up to date to reflect the current state of the system. However, the lower-level details will be documented in the software itself in the form of Doxygen documentation and various executable tested examples and tests.

7 Endorsers of LIME 2.0 design plan

The following individuals have read this LIME 2.0 design plan and have explicitly provided their endorsement to the design and implementation plan:

- Brian Adams (SNL)

References


[3] R. A. Bartlett, “Teuchos C++ memory management classes, idioms, and related topics: The complete reference (a comprehensive strategy for safe and efficient memory management in C++ for high per-


Distribution
CAUTION

This document has not been given final patent clearance and is for internal use only. If this document is to be given public release, it must be cleared through the site Technical Information Office, which will see that the proper patent and technical information reviews are completed in accordance with the policies of Oak Ridge National Laboratory and UT-Battelle, LLC.