Survey of Software Quality Assurance and Code Verification Practices in CASL

Michael Pernice
Idaho National Laboratory

Matt Sieger
Oak Ridge National Laboratory

March 26, 2013
DISCLAIMER

This information was prepared as an account of work sponsored by an agency of the U.S. Government. Neither the U.S. Government nor any agency thereof, nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness, of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the U.S. Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the U.S. Government or any agency thereof.

While every effort has been made to ensure correctness of the findings in this report, minor mistakes are inevitable when assimilating and transcribing such a large volume of material. These errors are unintentional and apologies are extended where needed. Corrections are welcome to ensure accuracy of the findings.
Please complete sections appropriate for this record.

**REVISION LOG**

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Affected Pages</th>
<th>Revision Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Document pages that are:**

Export Controlled __________________________________________________

IP/Proprietary/NDA Controlled____________________________________________________

Sensitive Controlled____________________________________________________

**Requested Distribution:**

To:      Doug Kothe, CASL Director  
         Douglas Burns, CASL Deputy Director  
         Paul Turinsky, CASL Chief Scientist  
         Jeff Banta, CASL Program Manager

Copy:    John Turner, VRI/ORNL  
         Randall Summers, VRI/SNL  
         William Martin, RTM/University of Michigan  
         Tom Evans, RTM/ORNL  
         Chris Stanek, MPO/LANL  
         Brian Wirth, MPO/UTK  
         Mark Christon, THM/LANL  
         Emilio Baglietti, THM/MIT  
         Jess Gehin, AMA/ORNL  
         Zeses Karoutas, AMA/WEC  
         Steve Hess, AMA/EPRI  
         Matt Sieger, QM/ORNL  
         James Stewart, VUQ/SNL  
         Vince Mousseau, VUQ/SNL  
         William J. Rider, VUQ/SNL
ABSTRACT

The Consortium for Advanced Simulation of Light Water Reactors (CASL) is developing capabilities for multiphysics simulation of light water reactors that couples state-of-the-art software components for neutron transport, thermal-hydraulics, structural mechanics, coolant chemistry, and fuel performance. CASL will establish confidence in its ability to simulate the performance of light water reactors through extensive software quality assurance, including verification and validation of selected challenge problems. In preparation for initial test stand and alpha releases of CASL software, a survey of software quality assurance and code verification practices was undertaken during the first half of Fiscal Year 2013. This report documents the findings of the survey and identifies best practices and opportunities for improvement in both software quality assurance and code verification practices.
The Consortium for Advanced Simulation of Light Water Reactors (CASL) is developing capabilities for multiphysics simulation of light water reactors that couple state-of-the-art software components for neutron transport, thermal-hydraulics, structural mechanics, coolant chemistry, and fuel performance. CASL will establish confidence in its ability to simulate the performance of light water reactors through extensive software quality assurance, including verification and validation of selected challenge problems. In preparation for initial test stand and alpha releases of CASL software, a survey of software quality assurance and code verification practices was undertaken during the first half of Fiscal Year 2013. This will be followed by a second survey of solution verification and validation practices for one of CASL’s challenge problems during the second half of Fiscal Year 2013. These surveys are intended to focus effort needed to support the test stand and alpha releases of CASL software.

The survey was conducted as a series of interviews with the developers of each of CASL’s physics components. Both best practices and opportunities for improvement were identified by the survey. It was found that the institution-specific software quality practices followed by the physics component software development teams are generally adequate for ensuring an acceptable level of built-in quality. Opportunities for improvement in defining and documenting component life cycles and measuring test coverage have been identified, and several cases where considerable effort will be needed to provide documentation required for public releases of CASL software have been noted.

It was also found that code verification is not widely practiced in CASL. In cases where code verification is being performed, it is not well publicized and/or is focused on numerical benchmark problems and code–code comparisons. A set of suggested guidelines for documenting code verification problems have been provided and it is strongly recommended that these be widely discussed and a finalized set of guidelines be adopted. In addition, it was found that no plans are in place for performing code verification on coupled components. It is strongly recommended that CASL clearly define responsibility for code verification of coupled components and begin planning for this activity as soon as possible.
ACKNOWLEDGEMENTS

Timely completion of this survey of code verification practices would not have been possible without the participation of CASL’s physics software component development teams, who took time from their busy schedules for the initial interviews, reviews of interview notes, and reviews of the survey findings.
CONTENTS

ABSTRACT ................................................................................................................................................. vi

SUMMARY ............................................................................................................................................... viii

ACKNOWLEDGEMENTS .......................................................................................................................... x

ACRONYMS .............................................................................................................................................. xii

1. INTRODUCTION .................................................................................................................................. 1
   1.1 Scope ..............................................................................................................................................
   1.2 Document Organization ............................................................................................................

2. ELEMENTS OF THE CODE VERIFICATION SURVEY ....................................................................... 3
   2.1 Software Quality Assurance ......................................................................................................
   2.2 Code Verification ....................................................................................................................... 5

3. CODE VERIFICATION PRACTICES BY COMPONENT ...................................................................... 8
   3.1 Cobra-TF ......................................................................................................................................
   3.2 Hydra-TH .....................................................................................................................................
   3.3 Denovo ....................................................................................................................................... 9
   3.4 MPACT .......................................................................................................................................... 10
   3.5 Peregrine ..................................................................................................................................... 11
   3.6 MAMBA ...................................................................................................................................... 12
   3.7 MAMBA-BDM ............................................................................................................................. 13
   3.8 DTK ............................................................................................................................................. 13
   3.9 DAKOTA ....................................................................................................................................... 14

4. BEST PRACTICES AND OPPORTUNITIES FOR IMPROVEMENT .................................................... 16
   4.1 Best Practices ............................................................................................................................. 16
       4.1.1 Requirements and Software Design .................................................................................... 16
       4.1.2 Management of Third-Party Libraries .............................................................................. 16
       4.1.3 Software Documentation .................................................................................................. 16
       4.1.4 Framework Leverage ........................................................................................................ 16
       4.1.5 Code Verification Documentation .................................................................................... 17
       4.1.6 Code Verification Milestones ............................................................................................ 17
   4.2 Opportunities for Improvement .................................................................................................. 17
       4.2.1 Code Verification ................................................................................................................ 17
       4.2.2 Documentation ................................................................................................................... 17
       4.2.3 Testing ............................................................................................................................... 17

5. SUMMARY AND CONCLUSIONS ........................................................................................................ 18

6. REFERENCES ......................................................................................................................................... 19
# ACRONYMS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMA</td>
<td>Advanced Modeling Applications</td>
</tr>
<tr>
<td>ASC</td>
<td>Advanced Simulation and Computing</td>
</tr>
<tr>
<td>CASL</td>
<td>Consortium for Advanced Simulation of Light Water Reactors</td>
</tr>
<tr>
<td>CI</td>
<td>continuous integration</td>
</tr>
<tr>
<td>CRUD</td>
<td>Chalk River Unidentified Deposits</td>
</tr>
<tr>
<td>CSE</td>
<td>computational science and engineering</td>
</tr>
<tr>
<td>FY</td>
<td>fiscal year</td>
</tr>
<tr>
<td>INL</td>
<td>Idaho National Laboratory</td>
</tr>
<tr>
<td>MMS</td>
<td>method of manufactured solutions</td>
</tr>
<tr>
<td>MOC</td>
<td>method of characteristic</td>
</tr>
<tr>
<td>ODE</td>
<td>ordinary differential equation</td>
</tr>
<tr>
<td>ORNL</td>
<td>Oak Ridge National Laboratory</td>
</tr>
<tr>
<td>PCMM</td>
<td>Predictive Capability Maturity Matrix</td>
</tr>
<tr>
<td>PDE</td>
<td>partial differential equation</td>
</tr>
<tr>
<td>PSU</td>
<td>Pennsylvania State University</td>
</tr>
<tr>
<td>RHEL</td>
<td>Red Hat Enterprise Linux</td>
</tr>
<tr>
<td>SNL</td>
<td>Sandia National Laboratories</td>
</tr>
<tr>
<td>SQA</td>
<td>software quality assurance</td>
</tr>
<tr>
<td>TPL</td>
<td>third party libraries</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>verification and validation</td>
</tr>
<tr>
<td>VERA</td>
<td>Virtual Environmental for Reactor Applications</td>
</tr>
<tr>
<td>VRD</td>
<td>VERA Requirements Document</td>
</tr>
<tr>
<td>VRI</td>
<td>Virtual Reactor Integration</td>
</tr>
</tbody>
</table>
SURVEY OF SOFTWARE QUALITY ASSURANCE AND CODE VERIFICATION PRACTICES IN CASL

1. INTRODUCTION

The Consortium for Advanced Simulation of Light Water Reactors (CASL) is developing a multiphysics software environment for science-based simulation of light water reactors. The Virtual Environment for Reactor Applications (VERA) couples state-of-the-art physics software components for neutron transport, thermal-hydraulics, structural mechanics, coolant chemistry, and fuel performance. These modeling and simulation capabilities address essential issues in the design and operation of light water reactors, such as life extension, power uprates, and assessment of design enhancements.

Modeling and simulation capabilities that are planned for VERA have the potential to play a transformational role in risk-informed decision-making for nuclear energy applications. It is, therefore, critical for CASL to establish confidence in the predictive capability of VERA through extensive software quality assurance (SQA), including verification and validation (V&V). Verification is the process of determining the accuracy of computed results. It addresses mathematical aspects of software, and seeks to answer the question: “Are the equations being solved correctly?” Validation is the process of determining how accurately the computed results represent the modeled phenomena through comparison with experimental measurements (NRC 2012). It addresses physical fidelity of the software and seeks to quantify the relationship between model and reality by addressing the question: “Are the right equations being solved?”

VERA is a large and complex suite of physics simulation components, utilities, and tools. This survey is focused on the software components that implement VERA’s physics capabilities and their coupled behavior. V&V of VERA will be demonstrated by conducting validation studies of specific challenge problems that CASL has identified to guide its efforts. These challenge problems include Chalk River Unidentified Deposits (CRUD) -induced power shifts, CRUD-induced localized corrosion, grid-to-rod fretting, pellet-cladding interaction, and fuel assembly distortion. V&V of VERA will be facilitated through integration of DAKOTA (Adams et al. 2011). A key aspect of DAKOTA is that it is non-intrusive and requires no modification of the target application. With this capability, VERA users will be able to directly conduct their own analyses (e.g., calibration or propagation of uncertainties), guided by examples of DAKOTA-based V&V that will be included in each release of VERA.

To help guide efforts and to quantify progress toward the goal of science-based, predictive simulation capability for light water reactors, CASL has adopted an augmented version of the Predictive Capability Maturity Matrix (PCMM) (Oberkampf, Pilch, and Trucano 2007). The PCMM was developed in response to the need to quantify the predictive maturity of modeling and simulation capabilities, and emphasizes peer review, metrics, evidence, and documentation over expert judgment. It addresses the essential elements of predictive modeling and simulation: physics modeling fidelity, code verification, solution verification, and model validation and uncertainty quantification. CASL has added elements addressing software modularity and extensibility and the capability to execute efficiently on current and future high-performance computing systems. Note in particular that an appropriate level of code verification and its supporting software quality assurance (SQA) practices are key elements of the PCMM. CASL has developed quality requirements for VERA and its components. These requirements are specified in the VERA Requirements Document (VRD) Hess (2012, which is primarily intended to address both functional and quality requirements for VERA.

a. The terms “verification” and “validation” have different interpretations in different disciplines. We follow the definitions established by the National Research Council (NRC 2012).
CASL is preparing test stand and alpha releases of VERA during Fiscal Year (FY) 2013. Requirements for this release are currently under development (Hess and Montgomery 2013). In addition to an expected set of technical capabilities, the requirements include:

- Documentation of how to install, build, test, and execute VERA, and to post-process its results
- A collection of demonstration problems along with input files and reference output files to check results and to verify correctness of the installation
- Code, theory, users, and V&V manuals for each of the component physics codes included in the test stand and alpha distributions.

In particular, the installation documentation is required to contain enough information for a user to check that the installation passes all tests (i.e., unit, regression, verification, and demonstration problems). Unit and regression tests are by-products of SQA practices, while verification problems are by-products of code verification practices. The demonstration problems are drawn from CASL’s collection of benchmark progression problems, some of which are documented in detail in Godfrey (2012).

In CASL’s software development process, project teams at partner institutions develop the physics software components and follow institution-specific SQA requirements. In addition, software components currently planned for the test stand and alpha releases of VERA are at different stages of maturity. To ensure availability and to determine the status of all required documentation, CASL conducted a survey of current SQA and code verification practices performed by the physics code development teams in order to document the current state of practice and to identify best practices and opportunities for improvement. The survey constitutes an initial project-wide study of one row of the PCMM and consisted of a set of interviews with the lead developers of VERA physics software components. Both SQA and code verification practices were covered in the survey. Most of these interviews were conducted in person or via CASL’s Vidyo™ teleconference facilities during collocation week in December 2012; three remaining interviews that could not be scheduled at that time were done in January 2013 via telephone. Narratives were derived from the interviews and reviewed by the participants. Findings were communicated to code developers in Vidyo teleconferences during collocation week in February 2013. Summaries of current practices in SQA and code verification were distilled from those narratives.

This report summarizes the results of the code verification survey and is organized as follows. Section 2 discusses aspects of SQA and code verification addressed by the survey. The role of code verification in the chain of SQA practices and its contribution to validation is also discussed. Section 3 summarizes SQA and code verification practices that are being performed for each of VERA’s physics software components. Best practices and opportunities for improvement are addressed in Section 4, and Section 5 contains a summary of the findings. Appendix A provides the survey questionnaire, while Appendix B documents the narratives of each of the interviews that were conducted.
2. ELEMENTS OF THE CODE VERIFICATION SURVEY

As mentioned in the introduction, the code verification survey covered both SQA and code verification. The objective of SQA is to ensure an acceptable level of quality in software. Effective SQA practices either prevent the introduction of software defects or detect defects soon after they are introduced. Because developers spend less time addressing defects and more time implementing new functionality, effective SQA practices can reduce development time. Current CASL SQA practices are based on requirements of DOE O 414.1D “Quality Assurance”; ISO 9001:2008, “Quality management systems – Requirements”; and NQA-1-2008, Part IV, Subpart 4.2 “Guidance on Graded Application of Quality Assurance (QA) for Nuclear-Related Research and Development.” However, independent industry, government, and academic organizations are developing VERA’s physics software components under a variety of SQA plans. CASL expects to support commercial-grade dedication efforts by Nuclear Regulatory Commission licensees who apply VERA to safety-related computations, and to that end, has endorsed the guidance of EPRI (2012). A first step in providing this support is to survey and comprehend the level of SQA rigor that has been applied to VERA components. SQA practices provide confidence of an acceptable level of built-in quality by assuring that the software was carefully designed and implemented, and that controls are in place to detect and repair defects during development.

While SQA is necessary for ensuring general software quality, conventional SQA processes alone are not sufficient to guarantee the correctness of scientific and engineering modeling and simulation software. Algorithms and the numerical solution of partial differential equations (PDEs) dominate activities in computational science and engineering (CSE). This introduces several distinct additional domain-specific sources of error that SQA cannot identify, including lack of resolution in space and/or time, insufficient accuracy in solution algorithms, and finite precision arithmetic (Oberkampf and Trucano 2002). These sources of error are the result of the subtle and complex relationship between the governing equations (i.e., the mathematical representation of the physical problem being studied, generally a system of nonlinear PDEs), the discrete formulation of the governing equations (i.e., primarily a mapping of derivatives and integrals to systems of algebraic equations), and translation of the discrete formulation into software. Validation efforts that seek to assess the adequacy of the model described by the governing equations must account for the difference between the solution of the governing equations and the solution of the discrete problem, which is called the discretization error. While discretization errors cannot be eliminated, they can be controlled; doing so requires knowledge of the behavior of the discretization errors. Other desirable properties of the numerical scheme (e.g., monotonicity and conservation) should also be tested, as well as other numerical aspects (e.g., exactness of interpolation and quadrature rules). Consequently, any activities directed at ensuring quality of CSE software must also include explicit testing of the numerical aspects of the application.

Following Oberkampf, Pilch, and Trucano (2007), ASME (2009), and NRC (2012), CASL distinguishes between code verification and solution verification. This is not merely a semantic distinction. Code verification is an exercise in demonstrating that the software reproduces the correct mathematical behavior and behaves as expected under mesh refinement. In particular, software that solves PDEs should reproduce the correct rate of convergence of the discrete solution to the true solution of the
governing equations. Because it is exceedingly difficult to prove this correctness for anything other than the simplest software, it must be demonstrated by evaluating the error in a known solution. On the other hand, knowledge of numerical error in the solution to a specific problem is critical to the overall assessment of its uncertainty, and validation efforts must account for the contribution of numerical error. Solution verification estimates the error in an unknown solution to a specific problem. Ideally solution verification also provides information about the exact solution for use in validation.

Code verification helps to rule out encountering mathematical problems while performing solution verification studies. For example, when assessing the error in a computed solution, code verification exercises can uncover the need for tighter convergence tolerances in iterative methods, stronger numerical coupling between physics components, improved treatment of boundary conditions, better mesh quality, finer mesh resolution to obtain asymptotic behavior of the error, or an alternative discretization scheme. These or other numerical difficulties can be regarded to be defects that were introduced upstream of validation efforts, which cannot proceed until the numerical difficulties are identified and addressed. Failure to do so may produce misleading results, over-reliance on calibration, and, in the worst case, a false sense of confidence. Decomposing overall SQA into the distinct processes of SQA, code verification, solution verification, and solution validation allows each stage of the QA chain to be documented and repeated independently, and facilitates re-verification of VERA when defects are discovered and after installation in new operating environments. Examples of both code and solution verification are provided in the ASME (2009). In CASL, Copps (2011) provides an example of code verification and Rider and Kamm (2012) provides an example of solution verification.

2.1 Software Quality Assurance

SQA is focused on ensuring that software is reliable (implemented correctly and with minimal defects) and produces repeatable results in specified hardware/software environments. SQA practices have been developed in the computer science and software engineering communities, and numerous SQA standards exist. Effective SQA supports early detection, communication, and correction of software defects, which can lead to reduced development time and costs. It relies heavily on practices (e.g., configuration management, configuration control, peer review, and unit/regression testing) to develop documented and repeatable evidence of software correctness and conformance to requirements. The basic elements of SQA comprise the following:

- Project management and quality planning
- Project risk management
- Requirements management
- Design
- Configuration management
- Procurement and supplier management
- Developer testing
- Software failure analysis
- V&V
- Problem reporting and corrective action.

In support of commercial-grade dedication of VERA, the emphasis is on surveying and comprehending the SQA provenance of independently developed component codes, identifying CASL

c. This is especially important for multiphysics simulations in which the single physics components are loosely coupled numerically.
requirements and critical characteristics of the codes, and providing evidence that those characteristics have been correctly implemented through documented V&V reports, automated tests, and audits (code reviews).

2.2 Code Verification

Code verification is directed at demonstrating that software reproduces the mathematical properties of the methods and algorithms that it implements. This section briefly discusses procedures for code verification and includes a recommendation for documenting benchmark problems used for code verification. Rider, Kamm, and Weirs (2010) provide a full description of a code verification workflow.

Code verification must begin with a clear statement of the types of problems that the software can solve and the methods used to obtain solutions. This includes the mathematical statement of the problem, the discretization scheme that is used, and tolerances for any iterative methods that are used. In multiphysics problems, the numerical coupling between single physics components must be specified and, where data is exchanged between different grids, the interpolation methods that are used should also be included. This information can then be used to determine the theoretical stability and accuracy properties (in particular, a convergence rate) that the software is designed to deliver through selection of the underlying discretization scheme.

The essential idea behind code verification is simple to state: identify a test problem with a known reference solution, solve the test problem on a sequence of successively finer grids, and compare the computed solutions to the reference solution. The results of such mesh convergence studies are then combined to determine the numerical convergence rate. The numerical convergence rate is compared to the theoretical convergence rate. Any discrepancies should then be resolved by modifying the software, reconsidering the test problem, or re-evaluating theoretical understanding. An important side benefit of this process is a deeper understanding of the behavior of the software and the methods it implements. While simple to state, code verification is difficult and time-consuming to perform. For example, selecting an appropriate sequence of grids is still very much a research topic, and different techniques for comparing computed to reference solutions can produce different results. Code verification for coupled multiphysics problems is largely unexplored. More detailed discussions can be found in Rider, Kamm, and Weirs (2010) and Oberkampf and Trucano (2008).

An important issue to consider is the selection of the reference solution. Errors that may be present in evaluating the reference solution can obscure small errors in the software being tested, making it difficult to draw sensible conclusions on the numerical convergence rate. Four different approaches to selecting a reference solution are described in Oberkampf and Trucano (2008):

1. The method of manufactured solutions (MMS) (Roache 2002)
2. Analytical solutions to special cases of the governing equations
3. Numerical solutions to ordinary differential equations (ODEs) derived from special cases of the governing equations

Each approach has its strengths and weaknesses.

MMS uses a prescribed “exact” solution, which is substituted into the governing equations to generate a source term. Solutions in MMS must be chosen carefully to reflect solution features that are typically encountered in practice (e.g., boundary layers, singularities, and discontinuities) and to exercise all the terms in the governing equations. MMS is the preferred method of ASME (2009). While considerable

---

d. Some methods used in VERA (e.g., method of characteristics and subchannel flow) do not readily lend themselves to conventional notions of discretization error. The relationship between computed solutions and governing equations must, nevertheless, be characterized.
effort has been made in applying MMS to single physics components, this can be very difficult for complex multiphysics software (e.g., VERA). MMS is intrusive in the sense that an artificial source term must be inserted into the code. Furthermore, MMS can generate very complex source terms that must be subjected to SQA and verification practices to ensure accurate evaluation. Symbolic manipulation software can be helpful for generating source terms for MMS. Open source software for generating manufactured solutions to specific problems is available through the MASA (i.e., Manufactured Analytical Solution Abstraction) website.

Analytical solutions to special cases of the governing equations most often take the form of infinite series or complex integrals. This raises issues in accurate evaluation that are similar to those encountered in MMS source terms. Another weakness of this type of benchmark is that analytical solutions exist only for relatively simple problems.

ODE formulations of simplified problems can be obtained, for example, through reduction of the governing equations to one dimension or through similarity transformations. The ODE formulation may not reflect the modeling assumptions of the original model; this can lead to small differences that fail to differentiate between modeling assumptions and software defects. Further, the reference solution must be computed to high accuracy by an ODE solver that has been rigorously verified to assure correctness.

Numerical benchmarks are useful to check software against functional requirements by demonstrating software capabilities, but these must be carefully evaluated for purposes of code verification. This category includes what is usually referred to as “code-to-code comparisons,” but such comparisons do not constitute code verification. Numerical benchmark problems published in the open literature often do not provide numerical values for the benchmark solutions, which often leads to subjective, non-rigorous comparisons (aka, the “view graph norm”). Comparing the computed and reference solution almost always requires mapping data between meshes at different resolutions (and possibly different mesh topologies). This introduces another source of numerical error (which must also be tested) that can contaminate the results of mesh convergence studies. The reference solution must be computed on a mesh that is much finer than planned for the mesh convergence study, and the numerical reliability of the reference solution must be demonstrated to be the highest quality through documented, rigorous SQA of the software that produced it. This includes code verification in the sense discussed herein, developing as much objective evidence as possible, and minimizing the amount of expert judgment needed to evaluate the benchmark. Given these considerations, a more reliable approach would be to simply adopt the verification tests of the software that was used to generate the numerical benchmark. If documentation of those tests is not available, then the value of the numerical benchmark for the purposes of code verification is questionable.

Most software development, as currently practiced in CASL, already involves some aspects of code verification. CASL must proceed with more formality, developing a defensible record in the form of objective reproducible evidence, in order to document the accuracy of VERA results. In particular, code verification benchmark problems must be carefully documented to unambiguously communicate those aspects of the software that are tested and how the benchmark solution was obtained. Benchmark specifications should be complete and specific enough to enable others to understand the process followed to develop and execute the benchmark comparison. Examples of verification benchmark problem documentation that conforms to these guidelines may be found in Kamm et al. (2009). The following guidelines for documenting code verification benchmark problems are distilled from recommendations described in Oberkampf and Trucano (2008).

1. Conceptual description of the verification benchmark. The following details should be included:

e. Some argue that numerical benchmarks and highly accurate numerical reference solutions are simply unsuitable for the purposes of code verification.

f. Introduced in Oberkampf, Trucano, and Hirsch (2004), strong-sense benchmarks are essentially engineering standards that should be maintained by professional societies, academic institutions, and nonprofit organizations.
a. Physics that is modeled in the benchmark problem  
b. Initial and boundary conditions, spatial domain (problem geometry)  
c. Examples of applications where the benchmark problem is relevant  
d. Type of benchmark problem (MMS, analytic, reduced problem, numerical benchmark)  
e. Algorithms and code features tested by the problem; single physics or multiphysics test.

2. Mathematical description of the benchmark problem:  
   a. Governing equations, including all secondary models and sub-models.  
   b. If the benchmark problem is based on MMS or an analytic solution that is not in closed form, specification of the MMS source term or evaluation of the analytic solution must be provided. A code fragment showing the implementation should also be provided.  
   c. Output value(s) being assessed for accuracy. This could be a single scalar value (e.g., heat flux through a surface) or a solution variable (e.g., fluid pressure on a surface). For CASL, it is particularly useful to test outputs that serve as inputs to other VERA components in a coupled simulation (in particular, for challenge problems).

3. Accuracy assessment:  
   a. Specification of the sequence of meshes used to assess accuracy  
   b. Specification of how solution error is measured  
   c. Additional calculations (e.g., interpolation) needed to implement comparison to the reference solution.

4. Additional user information, including:  
   a. Computing system(s) on which the benchmark was performed  
   b. Operating system and version  
   c. Compiler, version, options used  
   d. Precision (single or double)  
   e. Programming language  
   f. Execution time for each benchmark problem  
   g. Authorship and contact information  
   h. Additional information specific to the benchmark problem  
   i. Relevant peer-reviewed publications.
3. CODE VERIFICATION PRACTICES BY COMPONENT

The following subsections summarize the survey findings for each physics software component that is anticipated for inclusion in the test stand and alpha releases of VERA.

3.1 COBRA-TF

Interviewees: Robert Salko, Pennsylvania State University; Rod Schmidt, Sandia National Laboratories.

COBRA-TF is a thermal-hydraulics sub-channel code that is widely used for evaluation of safety margins in nuclear reactors. Originally developed at Pacific Northwest Laboratory in the early 1980s, this software has been adapted, modified, and developed by numerous academic and industrial organizations. There is no *de facto* standard version of COBRA-TF. This particular version is currently in use at Pennsylvania State University. The provenance of this software is discussed in (Avramova, 2007). CASL selected COBRA-TF for use because of a need for a non-proprietary sub-channel code.

Some benchmark comparisons have been made with COBRA-TF. Some comparisons of results to experiment have also been performed. Current development work at the Pennsylvania State University (PSU) is focused on performance optimization and is under source control. These existing benchmarks are used in a set of regression tests to ensure that no new defects are introduced. More tests are being developed.

Virtual Reactor Integration (VRI) is working to establish a software quality pedigree for COBRA-TF. COBRA-TF has been integrated into the VERA development environment, bringing any changes to this version under CASL SQA practices. In particular, regression tests are run automatically when code changes are committed and reported using standard practices for VERA. Unit tests and acceptance tests have yet to be developed. A synchronization server has been set up to ensure consistency between versions at PSU and VERA. With establishment of this synchronization, work is proceeding to devise further tests to obtain full coverage.

Formal code verification for COBRA-TF presents an interesting challenge. The nature of sub-channel flow precludes mesh refinement studies in the usual sense. Some additional analysis of sub-channel flow models will be needed to devise approaches for developing evidence of mathematical correctness of the implementation.

3.2 Hydra-TH

Interviewee: Mark Christon, Los Alamos National Laboratory.

Hydra-TH is a hybrid finite element/finite volume incompressible/low-Mach number fluid dynamics code. Hydra-TH uses cell-centered transport variables and conservative discretization that features a high-resolution monotonicity-preserving advection algorithm and capabilities for both explicit and implicit advection. Time integration methods include the unconditionally stable first-order backward Euler method (used primarily to drive solutions to steady state) and the neutrally dissipative second-order trapezoidal method. The trapezoidal method provides transient calculations that are unconditionally stable for the scalar transport equations and conditionally stable for the momentum transport equations. A second-order incremental projection algorithm forms the basis for the Hydra-TH solver, with development of a fully implicit approach currently underway. Several turbulence models are implemented, including Spalart-Allamaras, $k$-$\varepsilon$, implicit large-eddy simulation, and detached-eddy simulation. The Hydra-TH Theory Manual (Christon 2011) provides details.

Hydra-TH development follows SQA practices based on experiences in commercial software development, but these practices are not documented. The project uses *GanttProject* to track...
requirements and progress. Requirements are determined by CASL needs; software features that are needed to satisfy these requirements are identified through group discussion and then scheduled for implementation. Every iteration of the design and development process includes a design review, and every commit of new code must pass all regression tests on two different platforms. Two other team members perform a code review after the potential commit passes these regression tests. This process often identifies a revision or a potential software defect that does not get introduced into the repository and is strictly enforced in the Hydra project. These extensive review processes ensure that every member of the Hydra development team is familiar with all of the software.

Source code and serial and parallel regression tests are currently maintained in an internal subversion repository, which periodically gets pushed to the central VERA repository. There are some unit tests, but current efforts are focused on integrated tests. Current plans call for moving the Hydra repository to an external-facing server to support continuous integration (CI). Both serial and parallel tests are executed on a nightly basis on three platforms, and test results are reported internally through a custom testing and reporting capability. Test coverage is not measured, but the developers consider feature coverage in the regression tests to be good.

Doxygen is used to generate a developers’ manual from annotated source code, which is used daily. A comprehensive Theory Manual (Christon 2011) documents Hydra’s discretization and solution algorithms for single-phase flow; theory and documentation for multiphase flow is being developed and may appear in a separate manual for manageability. There is also an extensive User’s Manual (Christon, Bakosi, and Lowrie 2012) that includes documentation of how to run the software, code input, and several sample problems. Instructions for building the code are provided with the source distribution.

Verification of Hydra-TH is proceeding along several different paths. The Theory Manual includes mesh convergence studies for a pair of pure advection problems. Milestone-driven assessment of single-phase large-eddy simulation is underway for a set of well-documented large-eddy simulation test cases, using available experimental data and references in the open literature. Additional verification studies are currently underway using sample problems from the User Manual and the regression tests. These are numerical benchmark problems that are extensively studied in the fluid dynamics literature.

3.3 Denovo

Interviewee: Tom Evans, Oak Ridge National Laboratory

Denovo is a three-dimensional discrete ordinates ($S_n$) deterministic radiation transport code. Features of Denovo include use of Cartesian grids, efficient Krylov subspace solvers, diffusion synthetic acceleration for preconditioning, multigroup energy approximation, multiple spatial discretization schemes (including several variants of diamond differencing and discontinuous Galerkin finite element discretization), and a wave front parallel sweep algorithm for efficient parallel scaling. Evans et al. (2010) provides more details, including an early numerical benchmark comparison.

Denovo follows ISO-9000 standards and the SQA policy that is internal to the Reactors and Nuclear Science Division at Oak Ridge National Laboratory (ORNL). The developers use fogbugz and kanban sites at ORNL to track deliverables and requirements from CASL and other customers. An informal planning process is documented in a calendar-based schedule. An electronic notebook application is also used. Formal documentation follows a “tech-note” process, in which all the major features to be

h. subversion is an open source version control system. [http://subversion.apache.org/](http://subversion.apache.org/)

i. Doxygen is open source software for generating documentation directly from source code. [http://www.stack.nl/~dimitri/doxygen/index.html](http://www.stack.nl/~dimitri/doxygen/index.html)

j. fogbugz is a commercial bug-tracking system. [http://www.fogcreek.com/fogbugz/](http://www.fogcreek.com/fogbugz/)

k. kanban is an agile technique for efficiently managing the software development process.
implemented are documented: derivation of equations, linear algebraic structure, and algorithmic components (e.g., energy decomposition). The tech-note process is also useful for tracking changes in requirements. Simple Unified Modeling Language is used in informal code design, and unit tests are used to control changing requirements. Code standards that principally address clarity, consistency, and completeness are documented in a development environment manual and are enforced in code reviews. Third-party libraries (TPLs) are qualified as optional (i.e., hdf5, silo, brl-cad, sprng, SCALE, and SuperLU) or required (i.e., Trilinos, LAPACK, and BLAS), and TPLs under development are updated continuously. This is supported by writing tests directly to the TPL’s application programming interface for the functionality required by Denovo. Source code, documentation, tests, and examples are maintained in a git repository.

Denovo uses the TriBITS build system (Bartlett, Heroux, and Willenbring 2012) for consistency with VERA. CI tests, consisting of 343 unit tests and 110 python tests, are run every 4 hours on an internal cluster at ORNL. Tests are also run on VERA platforms. Logs of test results are maintained, and failed tests trigger notification emails. BullseyeCoverage measures test coverage at greater than 84% function points. An additional set of more stringent acceptance tests are run weekly and consist of extensive numerical tests of limiting cases that either have closed form solutions or known solution structure (e.g., symmetry or parity).

Doxygen is used to generate developers, methods, and algorithms (aka, theory), and user documentation. This documentation leverages content generated by the tech-note design process. The user documentation includes a development environment and standards manual. Currently, documentation for installation is in flux, as Denovo transitions to the CMake-based build system under TriBITS. With adoption of CI, any commit that doesn’t trigger a failed test is, in principle, releasable; periodically issued release notes associated with a tagged version of Denovo principally documents the current set of software dependencies. A SCALE release of Denovo occurs every 6 to 12 months.

Denovo has several approaches to code verification. Convergence studies are run for problems with semi-analytic solutions, and recent work evaluating eigenvalue convergence with respect to angular quadrature will be published soon. Some of the acceptance tests use MMS. Denovo can generate verification problems with its $S_n$-MC module, which runs Monte Carlo on the exact same discretization.

3.4 MPACT

Interviewees: Ben Collins, Brendan Kochunas, University of Michigan.

MPACT is a two- and three-dimensional radiation transport code for high-fidelity light-water reactor analysis using the method of characteristics (MOC) for whole-core transport calculations with neutron flux information provided at the sub-pin level. An initial evaluation of MPACT (Godfrey, Franceschini, and Palmtag 2012) showed good code-to-code agreement for the second Advanced Modeling Applications (AMA) benchmark progression problem (Hess 2012; Godfrey2012).

MPACT has defined its own informal SQA standard that is documented on an internal Trac site. Feature requests are reviewed to determine requirements, and code design is built from this. The process is documented using Trac’s ticketing system. When requirements change, the developers review these for impact on old requirements and repeat this process. Fortran coding standards to get uniform implementation are documented on the internal Trac site and enforced in code reviews. TPLs are selected for their functionality and are automatically tested when they are optionally enabled. Processes for managing TPLs are still being worked out. Source code, documentation, tests, and examples are maintained in a git repository.

For consistency with VERA, MPACT uses TriBITS. A CI server checks for changes every 10 minutes. If changes are found, the code is rebuilt, and both serial and parallel unit tests are run. Nightly

---

1. git is an open source distributed software version control system. [http://git-scm.com/](http://git-scm.com/)
unit and regression tests are run on a variety of compilers and configurations. Due to their expense, the regression tests are run for one configuration. Test coverage is measured at about 90% lines of code by \texttt{gcov}.

Developers and users manuals are generated from annotated source code using \texttt{Doxygen}. Installation procedures are documented on an internal website, and a README file provided with the distribution documents build options. While there is no detailed tutorial or examples, the regression tests provide examples of input files. A theory manual is in the planning stages. Releases are planned for every 1 to 3 months, and are annotated with \texttt{git} tags.

Code verification is not being performed for MPACT. There may be some open questions about the convergence behavior of MOC that must be addressed before code verification can be done. The developers are considering generating fixed source solutions on a Cartesian grid for comparison, MMS, and comparing MOC calculations to highly resolved Monte Carlo calculations. There is also the opportunity to leverage Denovo’s suite of verification problems.

### 3.5 Peregrine

Interviewees: Robert Montgomery, Pacific Northwest National Laboratory; Chris Stanek, Los Alamos National Laboratory.

Peregrine is a fuel performance code that is being developed to provide three-dimensional fuel performance modeling capability for CASL. Initial code-to-code comparisons with FALCON (EPRI 2004), which is a two-dimensional axisymmetric industry-standard fuel performance code, demonstrates that Peregrine is able to calculate thermal expansion, cracking, and relocation of the fuel pellet; heat transfer across the pellet cladding gap; and gap closure leading to pellet-cladding contact (Montgomery et al. 2012). These studies have highlighted the need for improved material and behavior models in Peregrine, particularly for fission gas release.

Peregrine uses the MOOSE framework (Gaston et al. 2009), which follows NQA-1 software quality practices (Lackner and Schulmeyer 2012). Consequently, Peregrine leverages MOOSE software quality practices and software dependencies. This includes, in particular, an extensive set of verification tests that are under revision control and execute automatically after nightly builds, dashboard reporting of test results in an internal Trac site, a central \texttt{subversion} code repository, MOOSE-based code design and standards, defect identification and tracking via Trac’s ticketing system, an automated build system, a \texttt{Doxygen}-generated developer’s manual, and documented installation procedures. Code design is strictly enforced to conform to MOOSE’s software architecture, and design reviews are conducted in collaboration with the MOOSE development team to ensure compliance. MOOSE leverages a considerable amount of third-party software, including discretization capabilities from libMesh (Kirk et al. 2006), nonlinear solver capabilities from PETSc (2012), and scalable algebraic multigrid capabilities from hypre (Falgout, Jones, and Meier Yang 2006). Other significant third-party tools include CUBIT for mesh generation, Exodus for input/output, and Paraview for visualization.

Requirements are determined by CASL’s need for modeling and simulation of nuclear fuel performance. Selection of models is determined by experience, and many models used in FALCON have stood the test of time. When the adequacy of a FALCON model comes into question, alternatives are sought in the literature. New model components (e.g., constitutive model frameworks for mechanical and thermal behavior, currently being developed) are incorporated into Peregrine by first determining the required inputs, implementing the component, and developing tests for the component in a local software repository. The tests are generally single element tests for correct evaluation of the model component, and once these are deemed satisfactory, the new capabilities are integrated back into the central MOOSE software repository. Unit tests are often hard to compose. For example, current results for fission gas...
release are not satisfactory, but there is no unit test for fission gas release because the large number of inputs into fission gas release makes it hard to isolate. Material properties for fuel and cladding are taken from MATPRO, which is a material properties database that has been used extensively in fuel performance and severe accident codes.

A user manual for Peregrine is currently being written. A theory manual does not yet exist. While code-to-code comparisons with FALCON have been largely positive and extremely useful, code verification studies in the sense discussed in Section 2 have not yet been devised. There is some question about whether responsibility for this effort should reside with the framework (MOOSE), with the application (Peregrine), or shared in some manner.

3.6 MAMBA

Interviewee: Brian Kendrick, Los Alamos National Laboratory.

MAMBA is an engineering-scale code for modeling CRUD deposition. Mass and heat are transported via convection, evaporation, charge-driven ionic movement, and diffusion. These phenomena are modeled at the engineering scale by numerically solving an appropriate set of coupled transport equations. Boundary conditions are provided by coupling to other VERA components and include thermal and radiation fluxes at the surface of the fuel cladding, the thermal-hydraulics at the interface between the CRUD and the coolant, and the particulate/soluble concentrations of the various chemical species in the coolant.

MAMBA is based on ChemPac and follows an internally documented SQA plan for ChemPac that also follows DOE O 414.1D for SQA. Development roughly follows a spiral life cycle, occurring in stages of design, code, test, validate, and enhance. Requirements are determined by the CASL MPO CRUD group for fundamental models of CRUD deposition. Software interface requirements are also obtained from other VERA physics components that couple to MAMBA (in particular, MAMBA-BDM). The requirements are captured informally in a set of action items and meeting minutes. Software design is based on a combination of ChemPac experience and MOOSE requirements, and implementation consists of modifying existing code and writing new code. Code reviews for ChemPac are done within the development team, but MAMBA is still too immature to follow this process. Code designs are documented in README files associated with the source code and provide an idea of the code structure. There are no formal coding standards. Besides ChemPac, MAMBA relies on MUMPS (MUMPS 2012) for direct solution of sparse systems of linear equations. The primary considerations in selecting TPLs are licensing and efficiency.

Software configuration control is done manually using a directory-based structure with version-based names. This is modeled on the approach currently used for ChemPac. Tests are associated with each version and are controlled in the same manner. MAMBA’s build system uses make, but dependencies and configuration are managed manually. Slightly different versions of the makefile are needed for different platforms. The frequency of builds depends on the level of development activity and occurs on a daily to weekly basis on a variety of Linux-based platforms (e.g., workstations, laptops, and high-performance computing systems).

Validation of MAMBA, using different levels of tests, is just getting underway. Some of the tests have known solutions, while others are regression tests. Simple, inexpensive tests are selected with efforts made to exercise the entire code; however, test coverage is not measured. Regression tests are run after active periods of code development, and results are written to output files that summarize results. Failures are either fixed or documented and reported to the team.

Releases of MAMBA are tagged with version numbers associated with the directory-based version control practices and occur approximately yearly. When a defect in a released version is reported, either a patch or a pre-release of the next version of MAMBA is provided, depending on the severity of the defect.
Instructions for building and using MAMBA are documented in a README file. In addition, a subdirectory of examples is available, along with input files and sample results.

A theory manual is under development, and manuscripts describing applications of MAMBA are in preparation. MAMBA uses the Crank-Nicolson method for time stepping, second-order finite volume discretization for the diffusion operator, and tight \((10^{-9})\) tolerances for nonlinear solve operations, so the code is expected to be second-order accurate in space and time. Advective velocities normal to the fuel rod are obtained from local mass evaporation and boiling rates and could be large enough to shorten the residence time scale; time steps must be selected to properly resolve this. While properties of the numerical methods are understood, MAMBA solves complex advection-diffusion-reaction problems with precipitation and deposition, and it is difficult to identify representative verification problems. When analytic solutions for simplified test cases are available, they are used to quantify errors; otherwise, code-to-code comparisons are done. A Courant stability condition, together with regular refinement, is used to select mesh resolution and time step sizes in mesh and time step convergence studies. A maximum percentage difference is used to measure differences in the computed and reference solutions, and a rate of convergence is found by plotting this maximum percentage difference against mesh size, and measuring the slope. However, these test problems are not formally documented, and no reports on these verification activities have been written.

3.7 MAMBA-BDM

Interviewee: Michael Short, Massachusetts Institute of Technology.

MAMBA-BDM is a first-principles, physics-based, mesoscale model that focuses on simulating temperature, fluid velocity, and species concentration profiles inside the CRUD around a single boiling chimney. This level of resolution is too fine for MAMBA to handle efficiently, and MAMBA-BDM can be used to provide MAMBA with quantities calculated at the mesoscale, such as overall CRUD temperature, surface CRUD temperature, peak cladding temperature, boron mass loading, and the total fraction of heat flux due to wick or nucleate boiling.

MAMBA-BDM uses the MOOSE framework, and, like Peregrine, leverages MOOSE’s software quality practices. Tests are written for every piece of physics that has been finalized and are structured to ensure that the proper physics is being reproduced. Test coverage is low since MAMBA-BDM is still under development, with current efforts focused on migrating from single-phase flow to multiphase flow.

A detailed user manual is available (Short et al. 2012) and is updated every month. In addition to detailed information on how to run MAMBA-BDM (along with a full input file), this includes an extensive review of prior work on CRUD models, a description of the MAMBA framework, specification of correlations used for physical and chemical properties, and a set of results on simulating CRUD scrapes. In addition, some parameter sensitivity studies are included. Some limited convergence studies have been performed to check the convergence properties of MAMBA-BDM, but these have not been documented.

3.8 DTK

Interviewees: Roger Pawlowski, Sandia National Laboratories; Stuart Slattery, University of Wisconsin-Madison.

DTK is a data-transfer toolkit that enables mesh searching and data transfer between different physics components in VERA. Because each physics software component in VERA typically defines its own computational mesh—which is often tuned to the requirements of the physics captured in that component—some interpolation operations are needed to move data from one mesh to another. DTK handles the case of overlapping meshes, and implements this volumetric exchange of data using a geometric rendezvous algorithm (Plimpton, Hendrickson, and Stewart 2004).
DTK was implemented during a CASL summer internship at ORNL, following SQA drawn from both the Trilinos and Denovo projects. Requirements are determined by the need to transfer data between meshes having different topology and different mappings onto large parallel computing systems. Requirements for DTK are clearly defined by data requirements for volumetric interpolation and mapping of data onto parallel computing systems, and are documented in Plimpton, Hendrickson, and Stewart (2004). DTK currently resides in VERA and uses VERA automated build, testing, test reporting, and installation procedures. A strict set of unit tests is run under CI. The data transfer problem is conceptually simple, the underlying interpolation theory is well understood, and a small number of exact analytic tests are provided. A user manual is generated using Doxygen from annotated source code. A theory manual (Slattery 2012) describes the domain model for mesh, fields, and parallel topology maps based on the concept of geometric rendezvous.

3.9 DAKOTA

Interviewees: Dena Vigil, Michael Eldred, and Brian Adams, Sandia National Laboratories.

DAKOTA is a toolkit for large-scale engineering optimization and uncertainty analysis. It provides methods for optimization, propagation of uncertainty, parameter estimation, and sensitivity/variance-based analysis that can be used in a completely non-intrusive manner (i.e., no changes to the target simulation code are needed). This is accomplished through use of scripting and hierarchical input specifications, which allows for additional composite analyses (e.g., hybrid optimization, surrogate-based optimization, and optimization under uncertainty). See Adams et al. (2011) for details.

DAKOTA development does not follow a specific SQA standard, but SQA processes are documented internally and are assessed against the Advanced Simulation and Computing (ASC) Software Quality Plan (Minana et al. 2009). Funding programs (e.g., ASC, U.S. Department of Energy Office of Science, the Nuclear Energy Advanced Modeling and Simulation program, CASL, and Cooperative Research and Development Agreements) provide programmatic requirements. Other users request requirements, and DAKOTA developers may identify requirements through code reviews, helping customers, or routine development and maintenance activities. All requirements are collected and reviewed in a more formal planning process, and most are followed in an issue tracking system. Requirements are reviewed and prioritized and must include a test plan. Code reviews are performed at different levels of formality, depending on the importance or priority of the feature request, and on an as-needed basis. Designs are reviewed in the context of a “design notebook,” which documents the design.

Source code, tests, documents, examples, and software project infrastructure reside in a subversion repository. Third-party components are first integrated into the development environment and evaluated against the test suite. Source code for TPL components that are in active development are pulled into the source tree using the subversion externals feature; otherwise, snapshots are used. Builds are fully automated and under CI, and any commit triggers a build job within an hour on a Red Hat Enterprise Linux (RHEL) 6 CI server. Nightly tests are also run on other platforms (e.g., Mac, Windows, internal Sandia National Laboratories (SNL) clusters). Integration with other software (e.g., Trilinos and VERA) is also tested on a nightly basis.

A full suite of 1,500 regression tests is run, with pass/fail status closely monitored on the RHEL 6 CI server. A tight relative numerical tolerance ($10^{-10}$) is used to compare against a previously computed gold standard to determine success, and small numerical differences are expected on the other testing platforms. The tests are determined at the design phase and mostly involve problems with known (typically scalar-valued) solutions. Tests are reported through email notification. A report is generated every morning, with links to dashboards that display the results. When found, software defects are tracked using Trac’s issue tracking system and a ticket is created. Defects are prioritized, and high priority defects in released codes are fixed, and patches are distributed.
User, reference, theory, and developers’ manuals are available on the DAKOTA website. A stable (i.e., nightly development version) and two versioned releases of DAKOTA and corresponding documentation are maintained. Doxygen is used to generate the reference and developers’ manuals from annotated source code. The users’ and theory manuals are updated when releases are done to reflect new capabilities. Releases use major/minor version numbers and are roughly annual, but efforts are being made to shorten this to semiannual releases. Extensive references documenting DAKOTA are listed on the public website. Tutorials and examples are provided with each DAKOTA distribution, and installation instructions are provided in a text file that accompanies distribution.

DAKOTA performs code verification to the extent possible. It provides a single interface to a wide variety of analysis capabilities, some of which are not amenable to code verification practices. Some methods (e.g., genetic algorithms for global optimization) are not well analyzed. Others (optimization algorithms) have expected rates of convergence that are tested. Still others (e.g., numerically generated polynomial bases for histograms and gradient-enhanced Hermite polynomials) are known to produce bad results under certain use cases (e.g., very high order polynomials can produce oscillatory results). Many of DAKOTA’s methods are not mesh-based. For example, for collocation methods, selection of mesh points is mostly determined by theoretical results that identify optimum location of quadrature points. When possible, well-known problems in peer-reviewed publications with known analytic solutions are used for verification. DAKOTA acknowledges a hierarchy of test problems: known analytic solutions, a middle ground that is compared against highly resolved Monte Carlo calculations, and a weakest level that can be characterized as “best we’ve found.” In this latter case, testing is performed to ensure that performance does not degrade, with the intent to push toward greater rigor with continuous improvement over time. Many reference problems and solutions are documented in peer-reviewed publications that document a baseline that is used as a gold standard in making comparisons.

4. BEST PRACTICES AND OPPORTUNITIES FOR IMPROVEMENT

This survey of code verification practices performed by CASL’s physics software development teams illustrates the wide range of maturity levels of individual physics components across the project. VERA physics components with lower levels of maturity can benefit from improvements in clearly identifiable areas, and components with higher levels of maturity can provide examples of best practices. This section identifies best practices and opportunities for improvement for SQA and code verification.

4.1 Best Practices

4.1.1 Requirements and Software Design

Verification and validation of scientific and engineering software cannot proceed without specification of the equations that are being solved. This, along with translation to discrete form, must be carefully documented. Moreover, in a dynamic environment where new requirements are being generated and new models are being incorporated, it is important to have a flexible and efficient way to document requirements and translate them to software. Denovo’s tech-memo process (see Section 3.3) illustrates one way to do this in a manner that generates material that can be re-used in other contexts (e.g., publications and presentations). DAKOTA follows a similar “design notebook” process (Section 3.9).

4.1.2 Management of Third-Party Libraries

TPLs are essential to the efficient development of advanced applications that target high-performance computing platforms. This approach leverages considerable past investment in software tool development but carries inherent risk for TPLs that are under active development. One way to manage this is to use snapshots of the TPL, sometimes placing the snapshot under source control when local customizations are needed. This runs the risk of embedding existing bugs or not taking advantage of new optimizations and capabilities of TPLs that are actively being developed. Upgrading to a new version can be painful or impossible if the snapshot lags the development of the TPL by too much. Another approach is to tap into a continuous feed of the developer’s version of a TPL. This requires close coordination between the project and the TPL developers and exposes the project to increased maintenance costs in case software defects are discovered in the development version. An alternative is to identify the functionality needed in a TPL, and develop tests to the native application program interface for this functionality. This practice has the advantage of familiarizing the developers with the TPL capabilities and allowing detection of defects, changes in functionality, or changes in application program interface. A new version of the TPL can be adopted once it passes all the tests.

4.1.3 Software Documentation

The need for clear and complete documentation to accompany VERA releases has recently been articulated. Currently, the physics components are in various states of readiness for this. The Hydra-TH theory (Christon 2011) and users (Christon, Bakosi and Lowrie 2012) manuals and the DAKOTA theory and users manuals provide examples of high quality documentation that meets these requirements.

4.1.4 Framework Leverage

Stand-alone codes must define and create their own SQA practices and tools to support these practices. Each set of practices must be evaluated for its compliance with requirements, leading to a good deal of duplication of effort and consequent increased cost. Leveraging a common software framework such as MOOSE minimizes both software development efforts and software engineering activities. This gives SQA by inheritance. The trade-off lies in ownership and control.
4.1.5 Code Verification Documentation

DAKOTA documents its code verification results in peer-reviewed publications. Publication-based documentation of baseline code verification is highly recommended. It performs several distinct functions, and it has built-in incentives. By its very nature, a peer-reviewed code verification study has received some external examination and a degree of approval from the scientific community. The motivation to publish the study encourages innovative approaches to code verification and helps ensure an appropriate level of effort. The results are archived in the scientific literature, and these documented results can be a gold standard that ensures through regression testing that performance does not degrade over time.

4.1.6 Code Verification Milestones

One motivation for conducting this survey of code verification practices was the observation that there is little evidence in CASL of code verification activities. Because it is a time-consuming and resource-intensive activity, code verification should not be relegated to a set of background activities. Defining and executing code verification milestones can result in greater visibility of code verification activities and results.

4.2 Opportunities for Improvement

4.2.1 Code Verification

With a few exceptions, formal code verification in the sense discussed herein is not being widely practiced in CASL. Even in cases where code verification is being practiced, it is not being called out. In some cases, disciplinary notions of code verification (not those described in ASME (2009) and NRC (2012)) are being practiced, or developers are not aware of an expectation for code verification. In other cases, there is some disagreement about where the responsibility for code verification lies. In particular, when a CASL physics component is built on top of another framework, it is unclear whether the responsibility for code verification lies with the component developers or the framework developers. It is certainly the case that every CASL code development team has some practices in place to convince themselves of the correctness of their software; the challenge is to develop evidence that is convincing to others, both technical and non-technical. CASL must formulate requirements for code verification and schedule the resources needed to meet these requirements, possibly adjusting the scope of other project technical deliverables. The risk is that CASL does not implement all of the quality measures needed to support rigorous V&V consistent with ASME (2009) and NRC (2012).

4.2.2 Documentation

AMA is in the process of drafting requirements for documentation of software to be included in the test stand and alpha releases of VERA. Generally speaking, development of theory manuals lags development of user and installation manuals, especially for the less mature codes. Resources need to be allocated to ensure timely availability of the required documentation. Resource constraints may require trade-offs with the amount of available physics capabilities. Requirements for documentation of verification benchmarks should be discussed and defined; an initial set of guidelines given in Section 2.2, “Code Verification,” provides a starting point for discussion.

4.2.3 Test Coverage

While unit and regression tests, together with automated reporting, are widely used across CASL, some additional effort is needed to measure test coverage. Tools for measuring test coverage are available and could readily be used more widely.
5. SUMMARY AND CONCLUSIONS

This report has summarized the findings of a survey of code verification practices in CASL that was undertaken in the first half of FY 2013 in advance of test stand and alpha releases of VERA. The principal finding of this survey is that code verification in the sense of ASME (2009) and NRC (2012) is not widely practiced in CASL. In cases where code verification is being performed, it is not well publicized and/or is focused on numerical benchmark problems and code-code comparisons. For some capabilities (e.g., MOC and sub-channel flow), theoretical work may be needed to more fully articulate the relationship between computed solutions and the governing equations. A set of suggested guidelines for documenting code verification problems were provided, and it is strongly recommended that these be widely discussed and a finalized set of guidelines be adopted. Project scope may need to be adjusted to accommodate the time-consuming nature of code verification.

The survey also addressed SQA practices in CASL. Although there is considerable variation in the level of maturity of the components slated for inclusion in the test stand and alpha releases of VERA, the institution-specific SQA practices followed by the physics software component development teams are generally adequate for ensuring that VERA’s physics software components possess an adequate level of built-in quality. Opportunities for improvement in defining and documenting component life cycles and measuring test coverage have been identified. In particular, while a high-level specification of requirements is provided in Hess (2012), features and functionality for individual VERA components are not always well documented. Several cases were noted where considerable effort will be needed to provide documentation required for the test stand and alpha releases of VERA. One case was identified where use of automated software configuration control would be beneficial. Finally, CASL should consider criteria and mechanisms to identify and track the level of maturity of each physics component in VERA.

Moving forward, this initial survey of code verification practices in CASL identified additional opportunities for improvement that are not specific to a single physics component in VERA. Coupling of single physics components is essential for modeling and simulation of reactor systems, and currently, extensive activity is being directed at achieving several pair-wise and three-way couplings. The question of code verification of these couplings is not being addressed. CASL needs to clearly define where responsibility for this aspect of code verification resides. Code verification of coupled multiphysics software, which may require extension of techniques (e.g., MMS) is currently an open area of research. Because code verification is, in many senses, an additional aspect of SQA directed at the mathematical aspects of software quality, VRI should drive this effort, with substantial input and support from the Validation and Uncertainty Quantification Focus Area to help develop techniques and participation from the focus areas responsible for the individual codes being coupled.
6. REFERENCES


Montgomery, R. (PNL); D. Sunderland, W. Liu, H. Wallin (ANATECH Corp.); C. Stanek (LANL); N. Capps, B. Wirth (University of Tennessee); R. Williamson, J. Hales, and B. Spencer (INL), *Peregrine: Initial Verification and Benchmark Evaluations Against Halden Fuel Rod Data and Falcon*, Consortium for Advanced Simulation of LWRs, August 2012.


Appendix A
Code Verification Survey

Following is the survey of software quality assurance and code verification practices that was used to solicit information about these practices from the VERA physics software component development teams.

I. Software Quality Assurance

Code verification is one part of the larger SQA process. This section focuses on discovering SQA practices and identifying available documentation of these practices.

General
1. Does the project adhere to an SQA standard (NQA-1, ISO 9001, DOE 414.1x, IEEE 1492, etc)?
2. Is there a documented SQA plan?
3. Does the software development follow a defined life-cycle process?

Requirements and Planning

1. How are software features or requirements determined?
2. How are requirements documented?
3. How are new features or requirements reviewed and approved before being implemented?
4. How are changes in requirements handled?

Design

1. How is the software architecture determined?
2. How are requirements transformed into code?
3. What reviews are conducted on designs?
4. How is the design documented?

Coding

1. Are coding standards used, and if so how are they enforced?
2. What types of code reviews or inspections are conducted?
3. How are third party components determined to be acceptable for use?

Configuration Control

1. How is source code controlled?
2. How are tests controlled?
3. How are third party component versions controlled?

Building and Testing

1. Is the build process automated?
2. How frequently do you build the software?
3. What platforms do you test your build process on?
4. How are tests determined?
5. How frequently are regression tests run?
6. Is system testing automated?
7. How are test results reported?
8. Is test coverage measured, if so what coverage is achieved?
9. What types of compare criteria are used to determine pass or fail?
10. Are there coverage requirements for unit tests?
11. How is reporting of unit test results accomplished?
12. How frequently are unit tests run?
13. Do you have estimates of the code coverage of these tests?
14. How are defects found in the software recorded?
15. What process is used to address any defects found?
16. What types of tools are used?

Documentation
1. Is a user manual available?
2. How frequently is the user manual updated?
3. Is the user manual verified against the software?
4. Is a tutorial or examples provided with the software?
5. Are installation or build instructions provided?

Release Management and Support
1. How are code/binary releases qualified?
2. How often are releases made?
3. How is versioning done?
4. How are bugs and other issues dealt with?

II. Supporting Theory

Computer simulations are built from software implementations of mathematical models that specify the problem being solved. This section focuses on discovering the cited supporting theory for the model and identifying available documentation of the theory.
1. Is a theory manual available?
2. How frequently is the theory manual updated?
3. Are there publications documenting the models or methods used in the code?
III. Code Verification

Code verification focuses on the correctness and accuracy of software that solves a mathematical model. This section focuses on discovering current practices for code verification and identifying available documentation of these practices and the results of verification.

1. An analysis of the accuracy and stability properties of the numerical methods you are using?
2. How does the accuracy and stability of numerical methods change with the character of the problems being solved?
3. How do you identify a problem to solve for verification purposes?
4. How do you identify a reference solution as a basis for comparison?
5. Is the reference solution itself verified, and on what basis?
6. Has the error in the reference solution been determined?
7. Is the code implementation for reference solutions itself subjected to code SQA?
8. Are the reference problems and solutions documented?
9. Do you use the method of manufactured solutions for code verification?
10. How do you select grids for conducting the convergence analysis?
11. What metrics do you use to compare the numerical and reference solutions?
12. How do you determine the rate of convergence from these metrics?
13. How do you address any discrepancies between the theoretical and observed rates of convergence? What difference in rate of convergence is considered a discrepancy?
14. How do you report the results of code verification activities?
15. Do you measure lines of code or features covered by code verification problems?
16. Is verification done automatically? If not, how often is code verification conducted?
17. Are the code verification problems, documentation, reference solutions, and associated code under version control?
Appendix B
Interview Narratives

Following are the narratives of the interviews with each of the VERA physics software component developers. In some cases the narratives are closer to the raw notes taken while completing the survey.

Appendix B1: COBRA-TF (CTF)

The current COBRA-TF effort is focused on performance optimization (without changing any of the models and physics) as well as setting up validation and verification cases.

COBRA-TF was originally developed by Pacific Northwest Lab in 1980 under sponsorship of the Nuclear Regulatory Commission. This original version was implemented into the COBRA-TRAC code system and further validated and refined as part of the FLECHT-SEASET 163-Rod Blocked Bundle Test and analysis program. This version was transferred to PSU for analysis of the Rod Bundle Heat Transfer Testing Facility data. This version has been adopted by the Reactor Dynamics and Fuel Management Group at PSU under the name CTF. Other versions of COBRA-TF do exist at other institutions and other research groups. There is no de-facto standard version of COBRA-TF but, rather, a scattered collection of individually maintained versions; however, the CTF version is the most updated, maintained, and further developed.

There have been many advances at various places and the genealogy of the code is complex and probably not traceable. Versions are mostly forked (that is: v4 does not mean that v3 is part of it). A lot of validation work has been done on it, but it has passed hands so many times some of the validation work has been lost and/or can’t be traced to any specific version. NRC uses this extensively.

The most recent verification that has been done for the PSU version was done in 2005 as part of work contracted by AREVA. The verification involved PWR steady-state cases, PWR flow-reduction cases, PWR power-rise transients, PWR pressure-reduction transients, and a PWR main-steam-line-break transient. These were verification cases, which did not compare results to experimental data, though some results were compared to a different COBRA version, COBRA 3C. The cases used in this study may be useful for testing COBRA-TF between source code changes and also to increase code coverage.

Validation work was also done for AREAVA-NP on GE 3x3 and ISPRA 4x4 experiments and results have been documented in publications. Recently, RDFMG, PSU has performed validation of CTF on OECD/NRC BFBT and PSBT benchmarks.

Current code development, being performed by the CASL participants is all under configuration control using the git version tracking system. No specific code development standards are currently in place.

A growing set of problems that test certain aspects of the code is used to assure results obtained are compared against previous results. This problem set started with the 17x17-pin assembly found in the VERA benchmark progression plan (Problem 3). Multi-assembly problems were built off of this single-assembly problem, leading to a 2x2-assembly case, a 4x4-assembly case, and a 7x7-assembly case. A 3x3-rod case with a central guide tube has also been created for the purpose of getting quick results. The 3x3-rod case and 17x17-rod case are used to test for changes in code results after source code changes are made.

Most often, a source code optimization leads to no results changes. Code results are compared by performing a diff on the standard COBRA-TF output files. However, some changes, such as enabling the user to specify the energy boundary condition as temperature instead of just enthalpy, did lead to some significant changes in code output. Note, though, that this leads to a different steam table being used,
therefore changing the boundary conditions. There is currently no criteria in place for determining acceptable levels of change in results after performing source code changes which lead to results changes.

There are plans to remedy this problem by including more test cases in the COBRA-TF repository. To date, the Series 5, Series 6, and Series 7 PSBT tests have been modeled using COBRA-TF (open data only). These 35 tests include experimental data for void fraction at 3 axial locations in 5x5 rod bundle tests. The bundle geometry is changed by replacing the central rod with a guide tube and the axial and radial power distributions are also varied between test series. The PSBT tests also include data for DNB tests (location of CHF) as well as transient tests (power increase, flow reduction, depressurization, temperature increase) where void and DNB was measured. In addition to the PSBT tests, there are also BFBT tests which provide open data that can be used to increase the number of test cases in the COBRA-TF validation matrix. The BFBT tests include void measurements, pressure drop measurements (single-phase and two-phase), as well as critical power measurements on an 8x8 rod bundle geometry containing several configurations of guide tubes.

COBRA-TF was selected because CASL needed a non-proprietary sub-channel code. Developing the capability ourselves was judged too expensive. CASL is aware of the complex genealogy of the code and selected this PSU version because it was determined to be the most up-to-date and well maintained. VRI has made some modifications to enable interoperability and integration in VERAs build/test system. These modifications include breaking the main program into setup and solve subroutines to allow COBRA-TF to be built as a library and linked into an executable containing other applications codes. The transient solve routine has also been broken into subroutines that allow operator split time integration alongside other transient applications codes, eg neutronics. In CASL, the build system has been augmented with cmake text files to allow COBRA-TF to build under Tribits (a cmake-based Trilinos configure, build and test capability) which allows for continuous and nightly integrated testing.

Configuration control uses a password-protected account on GitHub. It builds with Intel’s Math Kernal Library for BLAS and SparsKit for solving the pressure correction equation. Other than that it is self-contained.

On GitHub, the utility is used to build the code, but the configuration step has to be done by hand. The GitHub site offers a README, which provides instructions for how to do this. COBRA-TF is now integrated into the VRI build system Cmake/TriBITS, pulling source from GitHub, and manually merging conflicts that arise in the merge process. Efforts are being made to automate the process, but there is essentially only one developer. Russ, Rod & Scott also make changes and perform tests on the CASL side. There is no check-in test script to automatically run tests.

The user manual that is updated whenever a change is made to anything through the input deck. A separate user manual is used for the preprocessor.

The theory manual covers such information as conservation equations, how they’re set up, solution algorithm, and correlations. Again: no changes in the models/physics, so no need to update the theory manual. Models are taken from the literature, and accuracy of the model is determined by comparison. The theory manual references 80 papers and studies, and most of the models have references in the literature.

No studies are being performed regarding grid convergence. There should be plenty of validation studies in places (e.g., American Nuclear Society). We don’t know whether the validation studies were done on this particular fork? The problem here is we don't have a record back to the origin of the source that's being used. CASL will have to develop a suite of tests using the current version of COBRA-TF as a baseline.

We may be able to cobble together a chain of validation, Making sure COBRA-TF can do the problems that we care about. VRI may be developing tests and code verification studies to more fully
assess COBRA-TF. Some feel the verification process should not have any problems, and that COBRA-TF is probably in better shape than VIPRE-W.

John explains channel flow to the uninitiated. Matt finds NRC has no problems using COBRA-TF. Jim and Mike are unsure whether code verification in the strict sense can be done on this code.

B1.1 List of Publications on CTF Validation:


Appendix B2: Hydra-TH

Multiphase flow will be very expensive and very difficult to fully verify and validate.

A small team performs Hydra development. They follow formal practices that closely follow what Mark has done for Abaqus/CFD, but they are not documented. Software design documentation is maintained for the Hydra development team, but is not public.

Features needed to satisfy requirements are identified and then scheduled for implementation. Design reviews are done on the implementation at each iteration of the design/development process.

Every code commit must pass all serial and parallel regression tests on two distinct platforms (generally from a pool of available machines). Two other persons on the team perform a code review after the potential commit passes regression tests. A code comparison tool is used, every file gets “diffed” for code inspection. This process often finds a revision or a potential bug that does not get introduced into the repository. There is no tolerance for not following this process: “commit anyway, back it out if needed” creates more work for everyone. (VRI created problems by not following this process and committing to the ORNL git repository for Hydra-TH.)

Effort is being made to keep the development and documentation process agile. Unfortunately, a single development, V&V process doesn’t scale for all code teams, i.e., the many “academic” models are too heavy and monolithic for the Hydra team which is small and trying to remain agile.

The review process means that everyone gets a chance to look at all parts of the code.

There is an implicit life cycle in the above.

When THM pushes code out to the repository, their life cycle ends. They rely on VRI to handle the deployment. THM is trying to be a good CASL citizen, but they have other customers. Hydra-TH is a subset of Hydra. Hydra-TH is a branch off the main trunk (identified with a set of tags) and gets pushed to the ORNL repository. At that time the 2 repos are in sync. Hydra will go on an external-facing server, and pulling from that server will support VRI’s continuous integration practices. They also push out abbreviated serial and parallel regression tests. Note that the tests are abbreviated relative to Hydra, but include the complete suite of serial/parallel/long regression tests for Hydra-TH. There was intent to use CTest and CDash, but ended up creating their own testing/reporting capability. All of this is in the repository, test results are posted to a dashboard, creating a record of the tests that are run. The results get posted to a website internal to LANL.

Progress is tracked via GanttProject. At the start of each performance period, start and end dates are defined, and the Gantt chart is updated periodically. Publications are tracked here as well, to try to balance the work. New feature requests get added to the list. Priorities are also provided mostly by the SLT and other FA leads. This is used to develop milestones before putting them in Trac.

subversion is used for configuration control, though this will probably change. They have had serious problems using git, which seems better for small chunks of source code but not big binaries. To be clear here, git is not good for large binaries, however, we have seen problems even with the small binary Exodus-II files used for regression testing. The overall cumulative data size seems to be an issue for git. Refactoring of the repository is expected when it is moved to the open network.

Everything in the code is annotated with Doxygen. The result amounts to a developers manual, which is used on a day-to-day basis. Control flow is not necessarily documented well, however, the top-level flow is in a single file and quite simple.
There is a theory manual specific to Hydra-TH, available in the VERA Component Info table under VRI on CASLPedia. This is updated continuously; the software release number (LA-CC) is updated less frequently to reduce overhead. Theory manual describes discretization, solution algorithms. Multi-phase theory is being developed. This may need to be distilled or a separate manual for multiphase developed. Parts of it might be considered to be a design specification, laying out code architecture. But currently not fit for human consumption.

Properties of the projection algorithm are laid out in the theory manual. The method is not CFL-constrained. The projection method has some problems with Stokes flow, but this is not representative of engineering applications. (To be specific, the P2 projection algorithm has problems with very large time-steps for Stokes flow leading to a limit cycle. This was well-documented in work by Gresho, Christon, Chan in 1995.) The k-epsilon turbulence model or temperature-dependent properties may also cause problems for this method. Some pseudo-timestep continuation is implemented as a first pass to compute steady states. Some considerations are underway for time-step control based on accuracy. A fully implicit version is being developed that will be added to the theory manual. The projection method is used as a preconditioner for the fully implicit method. The designed accuracy is second-order in time, and second-order in space. For reaching steady-states, first-order in time, i.e., backward-Euler is used for both the projection and fully-implicit methods.

THM is resource-constrained for developing more complete documentation. If more documentation is a priority, milestones will need to be reworked to cover it.

Verification problems are very expensive to run. Something on the order of 20-25 will be documented for FY13. Verification process is hard to automate and labor-intensive.

Regression tests are derived from the verification problems, e.g. coarser resolution versions. Verification tests set up a gold standard for comparisons. If the low-level functions haven't changed, everything that depends on it upstream won't break. Unit tests run quickly, compiling and linking take a lot of time, so Hydra tend to use more integrated tests. Some of the verification tests are really validation, and it’s hard to separate in some cases.

THM won't be able to do verification for all of the capabilities in Hydra-TH (let alone all of Hydra) and they will be lucky to be rather complete for one of the models. Some turbulence models won't get fully tested and documented, although feature coverage is good in the regression tests. Probably won't use the method of manufactured solutions, the focus is more on solution verification than code verification for the FY13 V&V documentation. Limited convergence analysis is being done. Some grid sequence studies provide estimates of rates of convergence for some problems. Some limited code-code comparisons are also used.

The main constraint is lack of manpower, and as a result they are trying to catch whatever low-hanging fruit they can get. THM expects that multiphase flow will lead to explosive growth in verification and testing requirements.
Appendix B3: Denovo and MPACT

MPACT is not following any defined SQA standard, but have defined their own informal standard. This is documented in a wiki on an internal Trac site at the University of Michigan. They can set up access for external users (e.g., Scott Palmtag has access).

Likewise for Denovo, there is a document that covers their practices. Technically, they are conforming to ISO-9000, though documentation of each step in verification is missing. A general outline is available, but the work is being done. There is an Reactor and Nuclear Systems Division SQA policy that is being followed. Rigorous compliance to backward compatibility is satisfied with release testing.

CASL needs and other users determine MPACT features. After features requests are made, they perform a review to determine the requirements, and build a design from that. This is documented via tickets in the MPACT wiki. The design process is documented there as well. As requirements change, the developers review old ones and discuss what the changes mean, then start over with a new high-level design, followed by a formal review, which is also documented on the internal wiki. MPACT is young, and the developers haven't yet had to deal with changing requirements.

Denovo has a fogbugz site at ORNL. ORNL uses their own kanban site, which includes deliverables and requirements from CASL as well as other customers. The developers participate in integrated programming sessions two to three times per week in the VOCC) facility. An informal planning process is documented in a calendar-based schedule. The wiki page is more code documentation, but is not accessible outside ORNL. In addition an electronic notebook app, is used to document all work. Formal documentation is based on a "tech-note" process, which is archived in their git repository. Major features to be implemented are documented: derivation of equations, linear algebraic structure, algorithmic components (e.g. energy decomposition). There is also a methods manual and these design documents end up being integrated into that. The tech-note process has also proven useful for tracking changes in requirements. Code design is informal (simple UML) and unit tests are used to control changing requirements. Their kanban board is updated to reflect any changes.

MPACT has Fortran coding standards to get uniform implementation. Code reviews ensure adherence to the standards. These are documented at their wiki.

Denovo coding standards are documented in a development environment manual. There are four pages of standards, but the most important ones address clarity, consistency, and completeness. The desire is to keep it to a minimum to avoid high-level enforcement processes. Editor macros are available to set their formatting standard. Commit-time scripts do some code cleanup. Source code reviews are done.

Both MPACT and Denovo use git for configuration control.

MPACT requires 6-7 third-party libraries that they wrap to get a uniform API. When an optional feature is turned on, they automatically get tested. They can also compile without MPI. They write their own solvers, mostly relying on PETSc for large-scale problems. Their tests cover the interface between MPACT and PETSc, not the PETSc tests. The PETSc option is new, so they haven't needed to manage it. The developers likely will stay with the latest version. When their tests fail they identify the issue. These processes are still being worked out.

Denovo has two levels of requirements: tool-chain and TPLs. The tool-chain consists of a compiler (various compilers are supported), MPI, python, and swig. TPLs are qualified as optional (hdf5, silo,brlcad, spring (random number generator), KGTLIB, lava, scale, SuperLU) or required (Trilinos, LAPACK, BLAS). They have moved to a full CI process, which includes the TPLs. When tests pass, report includes information on which versions were used. TPLs are updated as they go along. There are unit tests for each TPL that checks their functionality without invoking any part of Denovo. This is an easy way to learn how to use the TPL and allows them to diagnose problems caused by updating a TPL.
MPACT and Denovo both use TriBits, which uses cmake. This is for consistency with VERA. Both groups also note some customers working on Windows platforms. MPACT has a CI server that checks for changes every 10 minutes; when changes are found the code is rebuilt and both serial and parallel unit tests are run. MPACT runs nightly tests on a variety of compilers and configurations; the tests are both unit and regression. Unit tests are run for all configurations, the regression tests are more expensive and are run for one configuration. Using gcov, MPACT measures test coverage at about 90% lines of code. Denovo also does CI testing on Tom's cluster. A cron job runs 343 unit tests and 110 python tests. All these are very fast, run every 4 hours. The SCALE version uses SCALE CI servers on Windows, Linux, and Macs, which runs every time there is a commit. Finally the Denovo team runs tests on VERAs platforms. These last two (SCALE and CASL) run tests specific to the distribution, but the Denovo team runs tests on all features. Any time a test fails a notification email is issued, test logs are also maintained. Bullseye is used to measure test coverage at more than 84% function point coverage. Acceptance tests are numerical tests of various limiting cases that either have closed form solutions or known solution structure (such as symmetry or parity) to check known behavior of the problem. The acceptance tests are 10 minutes to an hour, run on Sundays all day on the latest build. Denovo is looking at Google test as a testing framework for unit tests. Denovo uses design-by-contract.

MPACT and Denovo both use Doxygen to generate documentation from annotated source. MPACT uses this to generate developers and users manual. HTML versions are linked to wiki, all on the same server. Because of their use of source annotation, documentation is automatically changed when input changes. The MPACT build process is documented on wiki. There is also a README that documents build options. MPACT provides scripts for building and have added some cmake options such as enable all regression tests. There is no detailed tutorial; examples provide inputs for validation/regression tests. There is an outline but no MPACT theory manual yet. Various options for composing a theory manual are being considered, including Doxygen-based and leveraging dissertation contents.

Denovo uses Doxygen to generate three levels of documentation: developers, methods and algorithms (these are the tech notes and methods manual), and user documentation. For user documentation, Denovo also has a development environment and standards manual, which is based on texinfo. Sphinx gives a markup language, gives really nice websites. The build documentation is in a state of flux, currently documented in a README while converting from autoconf tools to cmake. Quickstart manual is up-to-date in the infobook, this is also still in transition. There are a lot of Python examples, their acceptance tests are useful for this. There is a generator script that can help users get started. Denovo has aspirations to put a list of problems on their wiki page that describe a collection of problems, how they are set up, and the solutions they’ve obtained. However there’s no external-facing server for posting this. The combination of a 45 page methods manual and the tech-notes provides pretty complete documentation.

For release management, MPACT uses git tags to annotate different version number releases of the code. There is currently a limited set of users (Scott and Andrew). The MPACT team installs the code for their users. Once you have the tags you can check out a release. They try to do a release about once every three months. They are considering options for distributing binaries.

Denovo used to do regular feature-based releases, but now with CI any commit of the code is “releasable,” unless a test fails. Release notes are periodically issued with a tagged version. This is mostly to document that Denovo works with specified sw stacks. SCALE releases occur maybe 6 months to a year.

MPACT has an outline but does not have a theory manual. Their intent is to do the theory manual in Doxygen, leveraging Latex source written for a dissertation. Theory is in-line with developers work.

Denovo has an extensive theory manual that is continuously updated, leveraging the tech-note process described above.
There is not a good understanding of the convergence behavior of the method of characteristics (MOC). There is a “Corrected path length” fix up that is not mathematically consistent. There is an asymptotic limit where it is. MOC doesn't preserve diffusion limit. Extensive verification with respect to some AMA benchmarks is planned. It might be possible to perform order of convergence studies with respect to ray spacing and angles. MPACT developers are looking at some analytic solutions for purely absorbing media, which should get exact answer. This is one of the unit tests for the MOC kernel.

Denovo has a lot of code verification tests. Convergence studies are run for problems with semi-analytic solutions. Recent work looking at eigenvalue convergence with respect to angular quadrature will be published soon. Some of the acceptance tests use the method of manufactured solutions (MMS).

MPACT can do all the kinds of acceptance test problems that Denovo does. MMS for MOC is consistent with some suggestions made by Bill Martin. MPACT reproduces a constant solution and has also solved an eigenvalue problem. An intern might be able to look at what we can say about convergence in space for MOC.

Denovo has an Sn-MC module which runs Monte Carlo on the exact same discretization and can use that to generate verification problems. Correctness of the MC solution can be determined by a code-code comparison.

MPACT has performed some comparisons to MC, and other comparisons are being made. One idea is to generate some fixed source solutions on a Cartesian grid, and compare MPACT results to that.
Appendix B4: Peregrine

A development plan is being followed. Peregrine is based on MOOSE. The underlying solver is part of MOOSE, so SQA of this is taken care of by MOOSE. This includes extensive nightly regression tests. SQA for Peregrine uses MOOSE SQA infrastructure and practices. Peregrine mostly employs correlational models derived from other codes or literature. These are implemented and tested to check for bugs and to check whether the implementation works in MOOSE. There are currently about 10 distinct Peregrine-specific tests. Once they feel these are implemented correctly, they are integrated into Peregrine and verification studies are performed on a number of cases. MOOSE follows NQA-1, and an assessment was recently done. (Rich Martineau forwarded a copy of the assessment to us.

While Peregrine tests its own components, there are some questions about how coupling of components in MOOSE are tested.

There is a documented SQA plan for MOOSE.

Efforts are focused on verification and making comparisons to data from separate effects and integrated effects experiments. The recent completion report for milestone L2:MPO.P5.03 serves as an initial benchmark, focusing on integral behavior, coupling structural and thermal, and determining whether Peregrine is behaving properly. This is the first of several benchmark studies. The report contains code-code comparisons with FALCON validation database, with mostly good agreement except for the fission gas release model. FALCON is a two-dimensional axisymmetric code and is the industry-standard fuel performance code. Because of this, access to FALCON and its validation database is a distinct advantage. Peregrine is still in early stages of development and these benchmark comparisons provide assurance that they are on the right track.

There was some discussion of distinguishing between verification and validation. From their perspective code verification is properly in the MOOSE domain. There is some flavor of validation in these evaluations. A primary benefit of CASL is the ability to access proprietary/NDA data.

MOOSE determines software design of Peregrine. Software development and coding standards follow the process defined by MOOSE. subversion and git are used for configuration control. The development process is to update from the main MOOSE repository at INL into a local repository, develop and integrate a model component, develop tests for the new capabilities working with the local repository, then committing the new code to the INL repository. MOOSE has requirements for when we check into their repos, test cases are also provided for any new additions. Informal biweekly code reviews are done. This involves walking through the code, discussion of how to use it, and determining what the inputs are. Most of the code written for Peregrine is just a couple of lines of code, plus extracting/inserting data. Altogether Peregrine is about 300 lines of code at most. Everything but the physics is hidden. Some constitutive model frameworks for mechanical, thermal behavior are being developed; these will be used to obtain material responses.

MOOSE is under continual development and enhancement, and they frequently update from the main MOOSE repository. Building and installation on a platform uses the MOOSE system and requires a considerable software stack. Tests for each component in MOOSE are run after build/install to ensure correct installation. Any problems they find are fixed about as soon as they are reported.

There is not much in the way of regression tests. When logic changes, the unit tests are re-run. Benchmark cases are currently being run, they take a long time to run. If the benchmarks are performing as expected, they move on to the next task, otherwise stop and correct, then return to benchmark cases. The benchmark suite is run quarterly, with about 2 months spent on writing new code. Regression tests are run when updating from main line. These are single element problem to make sure appropriate values are passed in during the computation; results are plotted. These tests are performed whenever an update is done, to make sure system changes did not break anything.
Unit tests may be hard to compose. For example, current results for fission gas release are not satisfactory, but there is no unit test for fission gas release because there are so many inputs into fission gas release that it is hard to isolate. NEAMS fuels has been leveraged for both Bison and Peregrine.

Deciding what models to use is based on experience. FALCON uses models that have stood the test of time. For example EPRI has consistently used these models. If experience suggests a FALCON model is outdated, alternatives are identified from the literature. Matpro is an old DOE capability that provides many of the material properties they use. The science is selecting the right combination of models to get the desired results. EPRI partnership is important because the NDA provides access to that. The owners are Anatech. Access to the code is not sufficient, we also need the people with the historical knowledge. There has been a huge positive impact resulting from interactions with EPRI and they seek clarification from EPRI on a regular basis. As long as FALCON source code is not exposed, Anatech is okay with Peregrine. The revised IP management plan will hopefully clarify the situation, particularly with respect to code distribution. Access to proprietary models has been extremely valuable.

Other third party tools such as Cubit for mesh generation, exodus for I/O, Paraview for visualization, are frequently used. Beneath MOOSE, there are TPL dependencies on PETSc, hypre, and libmesh.

There is no theory manual yet. A user manual is being developed.

There was some discussion of open source designation. CASL needs to consider what the value of this is. Access to proprietary data makes the difference, but you lose access to the proprietary data if you if the code is designated open source.

No sensitivity studies on nodalization are being done; MOOSE provides tests for solving PDEs. Code verification studies have not been devised, Peregrine is currently relying on other work in MOOSE. Peregrine relies on this functionality.
Appendix B5: MAMBA

MAMBA was discussed briefly but the developers were not available for the initial meeting and another teleconference has to be scheduled. There are two flavors, MAMBA for resolving pin-scale phenomena and MAMBA-BDM for operating at a mesoscale length scale. For both, the above SQA and development practices applies. Brian Kendricks is building his off ChemPack. Most answers will be the same, but details will differ. While FALCON and Peregrine are very similar, BOA and MAMBA are very different. FALCON is not being actively developed, BOA is. BOA accounts for steam generators, etc., but MAMBA only considers pins, and works with different information than is in BOA. The interaction with the industrial codes is different.

One benefit of the structure is that they at least interact with the BOA development team, which brings benefits to them as well. CASL needs to define a type of success that accounts for impact of CASL technology on industrial code.

Following is the narrative from the rescheduled MAMBA interview:

The project follows DOE Order 414.1D on quality assurance and interface requirements imposed by MOOSE. The SQA plan for models follows a plan for an internal Los Alamos National Laboratory ChemPac. But a plan for CASL/MAMBA is not documented. The life-cycle process comprises design, code, test, validate, and enhance (like spiral).

Requirements are determined by modeling needs for CRUD deposition. These are fundamental models, but a requirements interface to other codes also must be met. The CASL MPO CRUD group identifies the physics requirements. Interactions with other FAs determine the interface requirements. The requirements are captured informally in a set of action items and meeting minutes. The same process is followed for new features and changes in requirements.

Software design is based on a combination of ChemPac experience and MOOSE requirements. ChemPac is under active development, MAMBA design in based in part on this. Requirements are transformed into code through a combination of modifying existing code and writing new pieces. Reviews are done internally with team through discussion for ChemPac, MAMBA is still too immature to follow this process. Design is documented in a README associated with source code, and provides an idea of the code structure.

There are no formal coding standards, this is self-policed within a small development team. Code reviews are done prior to release. MAMBA uses ChemPac and TPL solver libraries (MUMPS) (these are subject to licensing issues) and other, public domain software. ChemPac is LANL proprietary. Outside ChemPac, the primary considerations are licensing and efficiency.

The MAMBA development team is small, so they have not implemented full revision control. Instead, configuration control is done manually, using a directory structure named with different versions. The same approach is used for tests: each version has a subdirectory of tests associated with it. Comparison of test results is done manually. ChemPac is under active development, and has a similar but separate configuration control process.

MAMBA is built using make, but Machine-specific environment variables are defined in the Makefile, so there are slightly different versions for different platforms. These platform dependencies are determined manually. The frequency of builds depends on level of activity. This is done weekly to daily on a variety of Linux-based platforms (workstations, laptops, HPC systems).
The MAMBA team is just starting to perform validation using different levels of tests. Some have known solutions and solutions obtained from previous versions of the code. They choose simple test cases that run fast and try to exercise the whole code. Test coverage is not measured. Regression tests are run whenever major changes are made (after periods of active coding). The frequency depends on the amount of changes introduced. Test results are written to output files summarizing results. If a failure is detected through test reporting process for a particular version, it is either fixed or documented and reported to the team.

Usage of MAMBA is documented in a README, as well as instructions for building and running, and the source code is 'self documented'. There are plans to generate documentation from annotated source code using **Doxygen**. This is updated with every release. This documentation is verified within the team. An examples subdir comes with the source code along with input files and sample results.

Releases are managed with version numbers associated with directory-based version control practices. There are currently two versions 1.0 (12.2011, 2D model) and 2.0 (02.2012, 3D model). Version 2.1 is under development. MAMBA is released about yearly. The approach to handling defects in released versions depends on the severity of the defect. In some cases could just send a patch in response to a request, or do a pre-release of another version.

There is no theory manual, currently it is under development and will evolve depending on the pace of changes made to the code or the models. A couple manuscripts are in preparation.

Numerical properties of the numerical methods are understood, but MAMBA solves a complicated problem combining convection, diffusion and reaction. While individual pieces can be examined, it is non-trivial to quantify convergence in mesh and time stepping. MAMBA uses Crank-Nicolson for time stepping finite volume in space, with second-order accuracy for diffusion, so they expect second-order accuracy in space/time. A nonlinear solver is used with residual tolerance of about 1.e-9. MAMBA does not solve the full flow field, which is being looked at a more fundamental level, at the mesoscale (BDM). Instead, MAMBA used velocities normal to the fuel rod that come from local mass evaporation and boiling rates. One difficulty in modeling CRUD is that these normal velocities may shorten the residence time scale, which has to be accounted for in the numerical scheme. As the CRUD layer thickens, the problem gets harder to solve. Iteration counts and residuals are monitored as diagnostics. Identifying verification problems is tricky. Currently a simple test case (e.g. no boiling, no flow/convection in the CRUD, simplified geometry and boundary conditions) with a known answer is used. They try to pick problems with analytic solutions, but when these are not available they perform code-code comparisons. Some reference solutions are compared against MOOSE calculations coming from BDM (which follows MOOSE SQA practices). Solutions from ChemPac with small residuals are considered to have small error, but that error is not quantified for crud applications. When analytic solutions for simplified test cases are available, they are used to quantify errors. (MMS is not being used.) However these problems are not formally documented (there is a plot of convergence in an earlier milestone report, but this is just an example.), and no reports on these verification activities have been written.

To do convergence and stability studies with respect to grid size and time step size they use a Courant condition for stability to pick mesh size and time step, then do regular refinement with that. Cylindrical coordinates are natural in this geometry. A maximum percentage difference is used to measure differences in the computed and reference solutions, and a rate of convergence is found by plotting this maximum percentage difference against mesh size, and measuring the slope. So far, the code appears to behave as expected, but problems could occur when they start to push the envelope with high boiling or thicker CRUD deposits. However this hasn't been explored yet. Currently this is not ‘reported’ through documentation, but MAMBA developers have done enough to convince themselves things are right. Coverage of these tests isn’t measured, and these verification exercises are done manually after major changes. A CASL milestone report documents initial model validation. The verification tests are controlled via the directory-based version control practice.
Appendix B6: MAMBA-BDM

General

Runs tests every time he builds the codes, moose tests. But makes sure new code development
doesn’t mess up the physics. Simple solutions are easy to cook up, but haven’t done much, sole
developer.

Some pieces of physics that are there are subjected to regression tests.

MAMBA and BDM have been written separately. BDM is a microscale model that sits on top of
Moose and provides microstructural support for MAMBA. BDM can see the hot spots in between the
boiling chimneys in the CRUD. Also supports the CILC model. MAMBA will call BDM when it sees a
hot spot.

Heat transfer+fluid flow+chemistry.

Once in a while will do a mesh convergence study, have a handle on the required mesh size and
parameters needed to get a converged solution.

Requirements: peak clad temperatures, fluid flow, chemistry driven by the problem. The requirement
is to pass information up the length scales to MAMBA and/or Peregrine. Some atomistic work being done
as well.

Working alone, milestones to Chris and Brian. No code review.

Close adherence to Moose standards. Reviews are occasionally done by Moose development team.

Using Moose repository. Write some things on top of that, in C or shell scripting. All this is in the
Moose repo.

Build and configuration are automated through MOOSE. Software is built every time a change is
done. Committed changes are tested every night. Also this is in the VERA repos so it is building and
tested nightly. Tests are written for every piece of physics that has been ‘finalized’. Tests are structured to
ensure that the proper physics is being reproduced. Debug mode in BDM will output every intermediate
stage of the calculation.

Test coverage is low since much is still in development, changing over from single phase to
multiphase. Plan to pick up some tests from Falcon.

How are defects dealt with? Enable the debug mode. Resort to gdb.

Detailed user manual is available, updated every 6 months. Usually connected to his L4 milestones.
Includes a tutorial, but not fleshed out examples, but example results are provided. Can distribute input
files if needed.

Release management: VERA wants every code to primarily live and work in the VRI repo. The
Moose guys would rather not. Still trying to work this out. Exact snapshot can be retrieved with some
work.

Under continuous development, providing capabilities as they become available. Versioning doesn’t
make sense yet.

By end of May want to start on a serious validation exercise. Ping Brian and have him ping you.

Theory: this is in the user manual. Intend to publish in the Journal of Nuclear Materials soon and
produce a conference paper.
Code Verification

Accuracy and stability of the methods in Moose are documented in the literature.

Some example problems for solution verification have been identified. No particular methodology, try to construct something that is representative of the CRUD problem being studied. It’s a CYA methodology: have backup evidence to support the correctness of the results. These don’t have known solutions.

Moose guys have talked about MMS, once code up and running will want to use this for verification. Not ready to say this is a realistic code without it being two-phase. Verification efforts with single phase flow pointed to this.
Appendix B7: DTK

Matt has done an SQA study of VERA so we skipped the SQA part of the discussion. Currently, there are few VERA component couplings done through LIME; code verification of the couplings will need to be done. Otherwise SQA covers LIME functionality for enabling interoperability. The discussion focused on code verification for DTK.

There is a theory manual for DTK, which encapsulates the rendezvous algorithm (developed in conjunction with Sierra), doing it in a robust, scalable fashion. There has been extensive code verification during its development. In the rendezvous algorithm, the destination component asks source components to provide values at certain specified locations. A rendezvous mesh is built to provide the optimal communication pattern for transferring the data. DTK provides tools to enable each VERA component to perform the data transfer.

The domain model describes all the mathematics and is software agnostic. There is no user manual but a strict set of unit tests, some with analytic solutions are used for verification. This theory manual is updated whenever changes/additions are made to the code. Currently one of three mappings is documented, with the remaining to be documented as part of L3 VR1.PSS.P5.07 milestone that should be closed at the end of the week.

Verification consists of prescribing a field to transfer to another grid and match values to an analytic result to ensure exactness to machine precision. The interpolation formulas are simple enough to derive analytic results. Tests are only documented in the code for the unit tests and the analytic solutions are coded in the test.

Tests are for overlapping domain transfer, which is a volume-to-volume mapping. There’s a source and a target, source asks for data. Tests are run on both the continuous integration server and nightly test server and are reported to the CASL CDash server. All the interpolations are thoroughly tested; though it is not measured there is probably a high level of coverage.

All this is under version control in the source repository.
Appendix B8: DAKOTA

DAKOTA does not follow a specific SQA standard; rather, its processes are assessed against an ASC software quality plan. DOE Order 414.1D also applies. The SQA plan is documented internal to SNL. An agile life cycle process is loosely followed.

Three levels of requirements are collected: program, user, and developer. Programmatic requirements are determined by funding programs such as ASC, DOE/SC, NEAMS, CASL, and CRADA work; these are sometimes formally communicated, sometimes not. DAKOTA is open source software so external or internal users can request requirements. DAKOTA developers may identify requirements through code reviews, helping customers, or routine development and maintenance activities. These are all collected and reviewed in a more formal planning process. An issue tracking system is used to follow most requirements.

Software architecture is not documented in any of the SQA documents, though it is described at a high level in the Developer’s manual, with pointers to implementing classes and design documents have been used at various points in the project’s 15 year life. The project PI is the gatekeeper. Requirements undergo a review and distillation process, and tasks to implement the requirements are then identified. Code reviews are done at different levels of formality, depending on the importance or priority of the feature request. The review looks at what needs to be done, and develops a design, which is reviewed in the context of a "design notebook," which documents the design. This gets iterated.

Recommended practices for coding are in a developer’s manual that is published on the web. Code reviews are done on an as-needed basis. Developers may request code review after a commit. Code reviews are also done as part of the design/implementation process referenced above. Third party components are first integrated into the development environment and evaluated against the test suite, which mostly use problems with known solutions.

Source code, along with tests, documents, examples, and software project infrastructure are controlled using subversion. For third-party libraries that are in active development, the "subversion externals" feature is used to pull the source into the DAKOTA source tree. Those not in active development are just taken from snapshots.

Builds are now fully automated and under continuous integration supported by a Jenkins continuous integration server. Any new commit triggers a build job within an hour on a RHEL 6 platform. Nightly tests on a broader set of platforms, selected by platforms targeted for release: Mac, Windows, RHEL 6, internal SNL clusters. Also integration with other software like SNL’s Trilinos and CASL’s VERA is tested on a nightly basis.

A full suite of 1500 regression tests, measure of success is the build platform of choice (RHEL 6, for CI), however the tests are regularly run on other platforms, where numerical differences are expected. A list of known pass/fails is maintained. Tests are determined back at the design phase, problems with known solutions for the most part. Requirements have to include a test plan. Tests are designed to address the requirements and are implemented after the code is written. Some tests are run under continuous integration, others that require more time are run nightly. Tests are reported via email notification. A report is generated every morning, with links to dashboards that display the results. A file that contains all the test results is generated for each platform where the tests are run. They show pass, fail, or a set of diffs. Fail is something catastrophic (didn't run to completion). Pass is based on a relative numerical tolerance (1e-10). One platform is used as a gold standard for comparisons. Unit test suite is not regularly exercised, but the regression tests are. Trac is used for issue tracking. When a defect is found a ticket is created. Defects are prioritized, high priority defects in released codes are fixed and patches are distributed.
A user manual is available on the DAKOTA website. A stable (nightly development version) and two versioned releases of Dakota and corresponding docs are, maintained, so documentation tracks development. Minimally, manuals are updated at release time to reflect new capabilities. DAKOTA relies on its user community to exercise new capabilities that are documented in the user manual. Tutorials and examples are provided with each DAKOTA distribution. There is also a set of slide-based presentations. Installation instructions are provided in a text file with the distribution, with more extensive examples on publically available development Wiki pages.

DAKOTA tries to keep to an annual release schedule (though is striving toward a 6 month release cycle with patches in between). Each release has major/minor tags. Releases during the year are mostly minor, and patches are issued to address minor releases.

A theory manual is available on the DAKOTA website, with stable, release, and VOTD (soon to be stable and two released) versions available. It is updated when releases are done to reflect new capabilities. Extensive references documenting DAKOTA are listed on the public website.

DAKOTA collects a lot of different methods under one interface. Some methods (e.g. genetic algorithms for global optimization) are not well-analyzed. Others (optimization algorithms) have expected rates of convergence that are tested. Still others (numerically-generated polynomial bases for histograms, gradient-enhanced Hermite polynomials) are known to produce bad results under certain use cases (e.g. very high order polynomials will produce oscillatory results). Well-known problems in the literature with known analytic solutions are used for verification. There are published benchmarks in the UQ community. Other problems are mostly drawn from civil engineering literature, which provide good benchmarks for reliability. Most verification problems are not PDEs. Some problems use highly over-resolved Monte Carlo to compare against, to ensure that the reference solution regressed against is consistent and can be used as a gold standard. DAKOTA acknowledges a hierarchy of test problems: known analytic solutions, a middle ground that is verified against exhaustive Monte Carlo calculations, and a weakest level that is "best we've found." In this latter case testing is done to ensure performance doesn’t degrade with the intent to push towards greater rigor with continuous improvement over time. MC methods generally are inadequate for rigorous convergence studies because they converge so slowly, in particular can’t determine an error for an MC solution. For some of the test problems, errors in the reference solution are not known. Reference problems and solutions are documented in publications, where a lot of work has been done for order of convergence studies to known reference solutions for specific DAKOTA capabilities (such as stochastic expansions). These peer-reviewed publications document a baseline that is used as a gold standard in making comparisons.

Specification by CASL of use cases for DAKOTA to address would be helpful to reduce the amount of documentation provided.

Many of DAKOTA’s methods are not grid-based. For collocation methods, selection of grid points is mostly determined by theory determining location of quadrature points. Convergence studies are in the literature for uniform and adaptive refinement. Geometry is not an issue in stochastic space.

Numerical and reference solutions are tested to a tight relative tolerance (1.e-10) by the test harness, which reports differences for further investigation. Strict 1-norm and 2-norm tests on probability distribution functions are used; the literature often looks at convergence in moments but these are problematic, as the tails of the distribution might not converge under these metrics. Expected convergence behaviors are documented in the literature (e.g., exponential convergence rates for smooth problems using global basis functions; algebraic rates for piecewise basis functions); if there is a discrepancy then the code is worked until the expected behavior is obtained. This approach allows comparing against a peer-reviewed baseline, and these verification tests are part of the regression suite. Problems satisfying these tests get reported through the regression test mechanism (email notification and dashboard).
Test coverage right now is probably less than 80% (last measured at 70% in 2008), but this isn’t automatically measured. They are looking at unit testing to get higher coverage of lines of code.

Code verification is not yet fully automated, but will be once the move to continuous integration is completed.

All code verification artifacts are under version control.