

“Software Engineering Intersections with Verification and Validation (V&V) of High Performance Computational Science Software: Some Observations”

*T. G. Trucano**, *D. E. Post[#]*, *M. Pilch** and *W. L. Oberkampf**

* Sandia National Laboratories,
P. O. Box 5800, Albuquerque, New Mexico 87185-0370

[#] DoD High Performance Computing Modernization Program
1010 North Glebe Road, Suite 510, Arlington, VA 22201

Principal Author E-mail: tgruca@sandia.gov

Abstract: In this paper we briefly consider the role that software engineering has in performing verification and validation of high performance computational science software. We focus on three areas where the intersection of software engineering methodologies and the goals of computational science verification and validation clearly overlap. These topics are (1) the evidence of verification and validation that adherence to formal software engineering methodologies provides; (2) testing; and (3) qualification, or acceptance, of software. In each case we emphasize some challenges and opportunities presented by consideration of the overlap.

1. INTRODUCTION

We remind the reader that the IEEE (IEEE, 1991) has defined verification and validation in the following way:

- Verification – “(1) The process of evaluating a [software] system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (2) Formal proof of program correctness.”
- Validation – “The process of evaluating a [software] system or component during or at the end of the development process to determine whether it satisfies specified requirements.”
- Verification and Validation (V&V) – “The process of determining whether the requirements for a [software] system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final [software] system or component complies with specified requirements.”

Oberkampf and Trucano (2002) pointed out that it requires some care and subtlety to understand these definitions from the perspective of computational science. Here, we simply take these definitions as given and clear, as the thrust of our discussion is on software engineering issues for which these definitions can be strictly applied.

Our starting point for this discussion is consideration of three statements in the latest Advanced Simulation and Computing (ASC) Program Plan of the United States Department of Energy's (DOE) National Nuclear Security Administration (USDOE, 2003).

- The Mission of the ASC program is to “Provide leading edge, high-end simulation capabilities needed to meet weapons assessment and certification requirements.”
- The Vision of the ASC program is to “Predict, with confidence, the behavior of nuclear weapons, through comprehensive, science-based simulations.”
- The Strategic Goal of the ASC program is to provide “Predictive simulations and modeling tools, supported by necessary computing resources, to sustain long-term stewardship of the stockpile.”

Additionally, it is stated in this DOE program plan that “Verification and Validation (V&V) will provide high confidence in computational accuracy by systematically measuring, documenting, and demonstrating the predictive capability of codes...”

The thrust of these quotes is startling. That is, the ASC program has the goal of building, deploying and applying complex computational science tools for the purpose of providing high-consequence predictions. V&V is understood by ASC to be key to understanding the confidence in these computational tools and for establishing sufficiency of this confidence for the intended applications to the U.S. nuclear weapons program. No less is implied.

One of the elements of the V&V strategy identified by the ASC Program Plan is “improve software engineering tools and practices for application to [ASC] simulations.” It is this point, made in the context of the above mission, vision and goals of the ASC program, which brings us to our purpose in the present paper. From the perspective of a high-consequence HPC-dominated national program, software engineering is an important component of achieving success. Why do we make this statement? In what ways does software engineering influence, or intersect, V&V for HPC computational science?

In this paper, we separately consider three key challenges that speak to this intersection. We use the term *code* to mean a HPC computational science software system and then express these challenges as follows:

- How to build high-integrity codes.
- How to test high-integrity codes.
- How to accept high-integrity codes.

In the following, we will briefly consider each of these challenges within the context of V&V. This considerably limits our discussion but emphasizes certain ideas that are especially relevant to V&V.

2. CODE CONSTRUCTION

It is clear that techniques used for constructing software have a strong influence on the reliability and success of that software. This awareness has led to increasingly formal software development models addressing software engineering in apparently every field *except* computational science. In computational science, which we view as the numerical solution of systems of complex partial differential equations in this paper, we observe resistance to formal software engineering models, especially to highly structured software

development processes and documentation requirements (such as Fairley, 1985; Sanders and Curran, 1994; Paulk et al., 1994; Phillips, 1997; IEEE, 1998; Vliet, 2000; Pressman, 2001).

A fundamental component of V&V is the rigor with which it can be demonstrated that a software implementation is correct, that is free from bugs and accurately achieves the design requirements. In ASC computational software the issues of absence of bugs and correctness of design requirements are centered on complex solution algorithms for partial differential equations. In a perfect world, algorithms as designed are correct, so software bugs and failures to achieve requirements center strictly on software implementation of these algorithms. This would then likely intensify the scrutiny of computational scientists upon programming models that increased the likelihood of correct software implementations and it would be less of a challenge to argue for this need. The world is not perfect however, especially in HPC (High Performance Computing) codes, and it is often a complex but fair question as to whether an observed bug or requirements failure in a code is due to algorithms or software or both.

Since we cannot prove in some mathematically rigorous sense that a code is correct, belief in correct software implementation is supported in principle through the *incomplete* accumulation of positive evidence that the software implementation is correct. Even the notion of *positive evidence* – that is, evidence that bugs and requirements failures *don't* exist – is challenging in practice. The history of HPC software verification, for example in computational fluid dynamics, is dominated rather by the response to *negative evidence* – that is, detection of bugs and requirements failures during code operation. This makes the understanding of the HPC software development endeavor all the harder because HPC papers are not written that emphasize the detection of bugs. Papers are often written instead that emphasize apparent computational successes in which, quite frankly, all that can really be claimed is that software bugs have not been evident, and are therefore *believed* to not be present. Clearly this approach has been sufficient for three generations and more of scientific publication in computational science, but it is somewhat distant from the needs underlying the mission, vision and goals of the ASC program stated above.

In our view, a key reason to use formal software engineering methodologies (also called *software quality engineering* – SQE) in HPC software development is because of their potential for creating additional positive evidence of software verification. Generically, this evidence is strongly correlated with two major constructs that result from a deployment of useful software engineering formalism; that is (1) an appropriate *software development model* and (2) an appropriate *software life cycle model*. There is currently no accepted standard for either a software development model or a software life cycle model for the codes that are created under that ASC program (and for HPC computational science in general). This fact creates an opportunity for software engineering as a discipline to strongly influence the future of high-integrity HPC verification, at least in the ASC program.

Despite the absence of a decreed formal SQE standard, the United States DOE has recognized the important role of software engineering in contributing to the integrity of HPC. High-level guidance that emphasizes software engineering as a component of evidence of V&V for ASC codes has been created and documented (USDOE, 2001). That brief document emphasizes the way that software engineering formalisms, including software development techniques, software verification techniques and software project management techniques, contribute to valued dimensions of ASC software – fidelity, functionality, reproducibility, traceability, manageability and supportability. Traditionally the focus on the *science* in

computational science has emphasized fidelity and functionality, with the other factors simply supporting these supreme goals as best they could and all but invisible in subject-matter discourse. It is now understood, as evidenced by (USDOE, 2001) that “high-integrity” computational science raises the level of importance of these other dimensions. In particular, the conduct and conclusions of V&V are directly influenced by software reproducibility, traceability and supportability, as well as fidelity and functionality. It remains an important *technical* software engineering problem to understand how to best accomplish the appropriate passage to a more broadly based emphasis on all of these dimensions in the context of physical-science dominated HPC.

The key DOE labs building ASC codes are expected to respond to this high-level DOE guidance with a more detailed implementation of the key expressed principles. An example of a detailed implementation strategy is that of Zepper et al. (2003). In this document a series of specific SQE practices that are believed to support development of reliable *software products* under the ASC program are defined. These practices address broad SQE principles, including documentation, testing, SQE – software life cycle linkages, project management processes, and roles and responsibilities versus support elements, such as third party software packages. Practices have defining elements that include needed inputs, expected outputs, and metrics. Quality of the deployment of a given practice, for example documentation of software requirements, is quantized into generally four categories, with the intent of supporting graded deployment of the practice, assessment of the deployment, and quality improvement of the practice over time. Minimal expectations for a core set of identified practices are also stated in this document.

An assessment mechanism coupled with this software engineering deployment strategy was also designed and implemented. Assessment is essential for improvement; and annual internal assessments of the ASC code development program at Sandia are performed and documented. The latest example of this assessment and documentation is found in (Ellis et al., 2004). These assessments are formal, requiring trained assessment teams and significant documentation from the twenty different software projects that participated in the 2003 assessment documented in (Ellis et al., 2004). The products of these assessments include (1) a common measurement of the degree to which the practices identified in Zepper et al. (2003) are implemented; (2) comparison with past assessment to quantify the nature of SQE improvement over time, both for particular software projects and for the overall Sandia ASC program; (3) identification of lessons-learned, both best practices and challenges, that influence the overall execution of the ASC code development projects at Sandia. This information also influences the nature of defined SQE practices and the means of assessment (as we write, the initial definition of Zepper and colleagues is undergoing modification based on assessment lessons-learned and overall deepening of knowledge).

In theory, an important quantification of the SQE processes should be the degree to which these practices are contributing to the development of “Predictive simulations and modeling tools...” and how this factor is improving in time. This has not yet been accomplished. Scoring versus defined practices and the quality of deployment is performed, plotted and compared with the previous assessment in (Ellis et al., 2004). The expectation is that improvement in one or more practices, hopefully of special importance to given code projects, takes place on yearly time scales. It is also expected that performance on any given practice will not worsen.

As an example of how such an assessment might be conceivably used by those who may be perceived to be only interested in the calculations such software delivers, we emphasize that the V&V program at Sandia uses this information as part of the general challenge of determining that a code is “suitable” for validation. Code suitability for validation was defined in Trucano, Pilch and Oberkampff (2002), and operationally means that there is evidence that money will not be wasted in a dedicated experimental validation activity targeting a specific application of the code. Before committing significant resources to validation some evidence that the code can function well enough to allow comparison with experimental data is desirable. If no such evidence is demanded, then all codes at any state of their development can reasonably expect validation resources to be directed their way. This situation is impossible because of resource limitations and priorities. The SQE assessment process at Sandia mentioned above provides relatively minimal evidence of code suitability for validation, but it is evidence nonetheless. The existence of this evidence does not guarantee that the code will be fully functional for all aspects of an elaborate experimental validation project, but it is useful as a necessary condition for managing a V&V program. This is not an unreasonable example of a larger class of decisions that might sensibly rely upon the existence of SQE evidence as a necessary condition for application of an identified code.

Post (2004) has observed that in the HPC community, resistance to SQE arises when perceived purely formal requirements, for example for seemingly unnecessary documentation, take precedence over correctly functioning scientific software. It is also onerous to implement practices for which evidence of their effectiveness when applied to HPC software is missing. These observations are also echoed by others, for example the “Agile Manifesto” community (Agile Manifesto, 2003). While this concern is real, it partly arises from psychological issues associated with computational scientists being first and foremost physical scientists and second software engineers. We emphasize this point strongly. No amount of software engineering will fix incorrect algorithms, or perform the research required to find “the” correct equation required to model a given physical phenomenon. However, it is also probably true that properly defined, implemented and constrained software engineering is a necessary condition for computational science to develop high-integrity software appropriate for high-consequence applications. In particular, appropriate software engineering methodologies contribute V&V evidence that goes beyond waiting for the next bug or requirements inadequacy to surface. Past scientific success with an HPC code is of undoubted importance and compelling in itself, but in high-consequence computing one must always confront the question “How do you know the answer that was calculated is the right answer?” and seek any relevant information wherever it may be found.

To conclude this section we return to a point we raised earlier, that compelling software development and life cycle models for HPC codes, such as the codes the ASC program is developing, are unclear. We are aware of little work on this particular topic. A standard approach that we have observed is to decree that a software development quality model, such as the Capability Maturity Model CMM (Paulk et al., 1994), will be used with little available real experience of the application of the model to, say, computational physics codes.

Even less is apparently known about the life cycle of important HPC codes. For example, HPC life cycles often boil down to anticipation of major code rewrites at fixed periods of time governed by the acquisition of new computers or by the loss of key personnel. There is little

confrontation with the fundamental concept of *retirement* in the HPC code world; retirement is perceived as happening when projects are cancelled or funding is eliminated, or when key personnel leave a project. SQE formalism firmly linked to life cycle models specifically recognizes the importance of managing software retirement. It is believed by most practitioners that the standard waterfall model (Fairley, 1985), or an iterated waterfall model, or any number of other such models, is not appropriate for HPC codes. Beyond that point, there are wildly differentiated approaches to development and life cycle across the spectrum of ASC codes and the broader U.S. HPC software community.

We therefore take this opportunity to advocate several opportunities for future work on the intersection of formal models of software development and software life cycle for high-integrity HPC software.

- First, we believe that it remains a rich area for applied research to define an appropriate SQE development model that is broadly applicable for the ASC codes. We do not advocate development of a model that will be rigidly and ruthlessly applied. Rather, we advocate identification of a model that provides richer opportunities for measurement of the software development process, for V&V to be a more integral part of the development process, for better life cycle planning, and for improved documentation.

An interesting example of work along these lines is Ambrosiano and Peterson (2000). These authors discuss a software engineering model, called the Exploratory Process Model, that emphasizes exploratory work flow characteristics of computational science software projects. The model has some elements in common with Extreme Programming (Beck, 2000) and also the Agile Manifesto. A particular focus of their study is the impact that exploratory research has on the otherwise rigid framework for developing requirements and subsequent software project planning present in more conventional software engineering models. It is also of interest that Ambrosiano and Peterson's Exploratory Process Model comfortably embraces some of the psychological elements of "Scientist" versus "Software Developer" that we mentioned above. We do not believe that this work proceeded beyond the initial report these authors published. We consider this to be an excellent example of taking a deeper look at SQE issues in computational science and deriving useful unobvious conclusions. More work along these lines is desirable, especially if accompanied with careful empirical studies.

Much is assumed within the HPC community about the limitations of software engineering formalisms but little published analysis of these limitations is found in the computational science literature. Above, we implicitly assume that a model like CMM is not really appropriate for HPC code development. Is this a fair assumption? More precisely, what can be understood empirically about apparent limitations of specific software engineering methodologies for HPC? It is unlikely that this will be a study conducted by HPC code activities like those in the ASC program; rather, this is a research opportunity for the software engineering community itself

- Second, we believe that life cycle concepts are relevant to HPC code development and need to be better understood. Post and Kendall (2003) have undertaken the only study that we are aware of that examines certain life cycle dimensions and parameters for

ASC codes, centered on ASC codes developed at Los Alamos National Laboratory and Lawrence Livermore National Laboratory. These factors include code complexity measures, qualitative and quantitative; product characteristics; user base definition; configuration management statistics; hardware platform characteristics; funding parameters; software lifetime characteristics; development team characteristics; and project planning characteristics. The discussion is framed within an overall iterative life cycle model, as well as tied to accepted software measurements (Jones, 1997) for predicting software evolution within the life cycle and future resource requirements. Throughout their paper, Post and Kendall emphasize that *requirements drive resources and software development within the life cycle*, not vice versa. In point of fact, this amounts to pointing out that ignoring software life cycle issues is tantamount to ignoring a fact of life. Post and Kendall also consider the impact of schedule factors other than software development in the evolution of ASC code project, often a somewhat disconcerting issue because it too may be incorrectly managed.

Among other things, life cycle concepts are important for cost estimation and project planning and influence quantitative measurement techniques that contribute to the objective quantitative evaluation of software correctness as well as software development management (Fenton and Pleeger, 1997; Jones, 1997). This was clearly revealed in the study of Post and Kendall. Post (2004) has emphasized the point we essentially made above, that *requirements drive costs*, not vice versa, or a code project is doomed, even if one wants to think of it as only computational science. Unfortunately, the largest hidden (or at least hard to measure) cost in current ASC projects, in our opinion, is V&V. This is not surprising, since these costs are recognized by the HPC community as being the hardest to estimate with good fidelity. Failure to focus on HPC software life cycle increases our inability to discern and estimate HPC V&V costs. Since V&V is a critical requirement for ASC software (as we emphasized in our introduction), failure to properly estimate and measure these costs within a life cycle model increases the potential for doom of ASC code projects.

- Third, it is worth studying how software engineering influences, and is influenced by, the psychological problem of “physical scientist” versus “software engineer” that is prominent in large-scale HPC code development projects. In our opinion, the HPC-as-physical-science philosophy contributes to the practice of V&V as a reaction to discovered problems, rather than as a directed organic activity within HPC code development projects. Reacting to discovered bugs is inadequate when the consequences of a bug are great. The consequences of a late-discovered bug in HPC software intended “to meet weapons assessment and certification requirements” are indeed great!

3. CODE TESTING

The next intersection between HPC and V&V that we wish to discuss is *software testing*. In our view, the frontiers of high-integrity HPC are dominated by testing. Testing remains the most essential contributor to the collection of evidence that is required for establishing confidence in HPC code applications, independently of whether the testing is deliberate or inadvertent through code application not intended to be a “test.” For high-integrity HPC *sufficient* confidence in the credibility of the code is crucial. Sufficient confidence in software

firmly rests upon the idea of sufficient testing. Inadequate testing increases the risk of dire consequences of applying malfunctioning software in important circumstances.

How sufficient testing may be defined and how it may be achieved is influenced by software engineering principles. Testing infrastructure, that is, the design of testing, the technology required to properly conduct it, and the methodologies for properly interpreting the results remains a huge technical challenges for the entire software industry, not simply for HPC and the ASC program. Inadequacy of the testing infrastructure has been estimated to contribute billions of dollars in additional costs to software development projects nationwide (NIST, 2002), as well as to decreased quality of delivered software products. The NIS study analyzes testing infrastructure and economics with a relatively strict alignment to SQE principles and dimensions. To the degree that this alignment is incomplete, for example due to an inadequate recognition of SQE factors in a given software product, it is hard to even properly frame the economic questions that the NIS study poses. While the case studies presented by NIS were in non-HPC applications (CAD/CAM/CAE systems; product data management software; financial services software), there is little reason to doubt the relevancy of the qualitative conclusions for HPC software.

The NIST study implies in its detail that to the degree that a formal SQE framework is missing for a software product, the more likely the software testing is “art” (ad hoc and expert judgment driven) than “science.” Beizer (1990) expounds exactly upon this issue in a treatise of over 500 pages devoted to testing in SQE. Beizer’s basic message is that good testing requires good test design; and good test design is rooted in the characteristics of a good software development process. Details will vary, of course, depending on whether the software being tested in a CAD/CAM system or solves a system of partial differential equations.

Software testing for HPC is especially difficult because of the problems created by the strong coupling of the science and mathematics of complex algorithms with code success. Formal logic of testing (Beizer, 1990) first and foremost depends upon having tests that a code passes or fails. In HPC such tests are most easily designed as structural tests, say at the unit testing level. Unfortunately, a large set of unit tests can be successfully passed in a CFD code, yet the resulting computations can fail to accurately simulate a desired problem. At more complex integral software levels, structural tests that have clear cut pass/fail assessment are very difficult to devise, while more easily designed and broadly applied functional tests have more difficult assessment issues. At the integral software level we tend to see a somewhat implicit transformation from SQE-dominated unit-type structural testing to user-dominated integral functional testing, with attendant assessment principles that may be best characterized as “In the Eye of the Beholder.”

Most really severe problems with HPC codes that have achieved initial release are not hard faults that create clear compiler failures, dramatic code crashes, or other clearly interpreted symptoms. Rather, these problems center on lack of accuracy of numerical solutions of partial differential equations that is directly (or indirectly) traceable to algorithmic failures. Algorithmic failures are mathematical problems, but also typically strongly correlated with the *science* in computational science practice. Detecting a lack of accuracy, as opposed to an outright code crash, can be very difficult (hence the importance of the “In the Eye of the Beholder” assessment). Extensive experience may be required to recognize a true algorithm failure. Lack of accuracy may also be due to lack of discretization

resolution in a given calculation; the algorithm may be correct, but the number of finite elements, or finite difference resolution, or number of iterations specified for an iterative solver, may not be sufficient. Since HPC is dominated by the need to increase numerical resolution for hard problems where current achievable discretizations are known to be insufficient, we see how difficult it can be to detect true algorithmic faults or other code problems that don't result in hard software failures. Detecting an algorithmic failure may have to be deferred until numerical resolution issues can be dealt with. But notice that believed numerical resolution adequacy is itself coupled to some belief in current understanding of solution algorithms. To the extent that the algorithms may be wrong, the perception of needed numerical resolution may be wrong. Untangling these complex problems is not easy.

For these and other reasons, HPC testing ends up looking quite ad hoc over the very long run and differs very little from the *lowest* level of testing Beizer (1990) identifies, that is *debugging*. This kind of testing will not likely originate in formal test design described in detail by Beizer and other processes advocated by SQE, nor will it rely upon a systematic testing infrastructure. This means that the issues raised by NIS are probably magnified, not absent. In computational science, only a few of the existing test problems are agreed to be standards of simulation performance in given fields. Where a standard test problem is identified (very rare) or implicitly exists (for example, the Sod problem in compressible fluid flow; Woodward and Colella, 1984) while the correct solution is known there is no universal standard for how close to that answer a code must be for a given resolution to define success as opposed to failure.

There sometimes appears to be an active distaste for this kind of discourse, which is particularly apparent in published comparisons with such “test” problems. This issue has also been firmly raised recently by Quirk (2004). In principle, the matter could be made more objective through an approach such as convergence studies applied with a rigorous formalism that draws a firm conclusion as to whether one achieves the correct solution in the limit as discretization fidelity increases (Oberkampf and Trucano, 2002). In practice, one rarely sees this kind of testing logic published, even if one is lucky enough to have a test problem that could be considered (informally we emphasize) to be a community standard. Nor is this kind of study typically required by editorial constraints in journals devoted to publishing the progress and results of HPC.

Key HPC testing centers on usage of the code, hence is clearly functional testing at an integral software level. This is effectively unplanned software testing and it certainly helps develop a sense of the correct functioning of the software over a long period of time. This sense of correctness – correct requirements correctly implemented – may indeed run exceptionally deep within a particular community of users of the code, but it is also strongly correlated with the knowledge of experienced individual users. The result is often a basis for belief in the correctness of HPC software that simply cannot be decoupled from an individual user. This is not satisfactory in our view, since software should either be functioning correctly or not as an abstract principle divorced from any particular user. This is not philosophical hairsplitting when high-consequence application of the code is the goal.

While accumulated experience applying complex computational science codes is very important to understanding the problem of correctness, it is also unsatisfactory from the perspective of knowing when a code is adequate for a particular application. This is also important for high consequence applications. Rigorous testing should form an important

contribution to decisions about usage of software. Ad hoc testing seems singularly unable to accomplish this. The problem of adequacy, or *acceptance*, of HPC is further discussed in Section 4.

It is important to emphasize the three important dimensions in testing that must be encompassed by formal testing methodologies integrated with SQE elements for HPC software. First, a precise understanding of the purpose of the test must be developed. This is very straightforward for structural unit tests, and considerably harder for functional integral tests. An entire spectrum lies between these two extremes in HPC software. Second, specific definition of a test that achieves the purpose is required. Again, this is relatively straightforward for a unit-level functional software test. It can be virtually impossible if the goal is to devise an integral test for combined algorithms solving a complex system of partial differential equations. Third, this test must be assessable. Or, another way to put this is that the test must be accompanied by a precise assessment specification. The goal of this specification is quite simply to eliminate, or at least minimize, the “In The Eye Of The Beholder” effect in computational science software testing. It is not clear to what degree this ideal can be achieved; therefore, this is clearly an attractive target for dedicated research.

The importance of testing is further illuminated by a comment on software product liability. The purpose of testing is to remove defects from software, whether they are programming language, algorithmic, or physics in origin. Product defects in worlds other than software produce vulnerability to legal liability. Due diligence in product testing is an important principle in this case, because it is generally recognized that a product cannot be guaranteed to function properly through testing alone. Due diligence typically reflects the existence of standards guiding that product testing. Where such standards don't exist, the presence of due diligence is harder to establish. It is unclear what due diligence means when we are discussing the testing of HPC software. A default then appears to be the use of standard cautionary language, such as the statement appearing in Press et al. (1992)

“We make no warranties, express or implied, that the programs contained in this volume are free of error, or are consistent with any particular merchantability, or that they will meet your requirements for any particular application. They should not be relied upon for solving a problem whose solution could result in injury to a person or loss of property. If you do use the programs in such a manner, it is at your own risk. The authors and publisher disclaim all liability for direct or consequential damages resulting from your use of the programs.”

In another large book on software testing, Kaner, Faulk and Nguyen (1999) devote an entire chapter to a discussion of potential liability issues that trace to a greater or lesser extent to inadequate software testing. The title of the chapter in question is “Legal Consequences of Defective Software.” We heartily recommend that readers of this paper scrutinize that chapter. Among other thing, these authors point out that it is not clear that cautionary language of the above type is sufficient protection. This kind of warning is called a *restrictive warranty disclaimer*, and does not necessarily protect software product developers from suits based on non-contract legal theories, such as claims of fraud (for example, an unsound basis for making a claim about the software performance) or property damage (for example, software that destroys real property – i.e. data – through malfunction). From the testing perspective, we simply note that one potential unsound basis for claiming software performance is inadequate testing, for example failure to conform to available industry standards. As it happens, there is an ANSI standard for software testing that as far as we know is not usually followed in the development of HPC codes.

Our purpose is not to emphasize litigation potential in inadequately tested HPC software. Rather, it is to emphasize that formal testing procedures rooted in SQE methodologies are worth considering for claimed high-integrity HPC software. The *legal* themes of negligently supported claims of software capability (inadequate testing) and property loss due to inadequate software quality (time, money, information) are perfectly appropriate as *technical* themes in the HPC world. Due diligence – rigorous testing – is not simply a legal recourse, but a scientific necessity.

Given these observations about HPC testing, we offer the following recommendations for building on the role of SQE specifically in the area of testing:

- Developing software testing methods that are specialized for complex computational science software and deployable within formal SQE-centric test infrastructures is an important area of research. One attempt at a general approach is the Method of Manufactured Solutions (MMS), a testing procedure of some generality that is specifically aimed at software for solving partial differential equations. A complete discussion of this methodology is given by Knupp and Salari (2003), extensively building on an earlier exposition of Roache (1998). Setting aside the interesting challenge of finding other testing methodologies that reflect to some extent the generality claimed by MMS, a fair question is how to properly meld a testing methodology like MMS with the software development process. MMS has characteristics of structural testing and therefore would appear to be strengthened by greater attention to its needs and requirements during even the design phase of an HPC code. Actually confirming this speculation is of interest.
- We believe that there should be increased reliance upon major structural testing procedures, mirroring SQE procedures, to achieve measurably high-reliability HPC software components, and less reliance upon ad hoc experience alone. More generally, there should be increased emphasis on formal testing as product development, not just as computational science experience. In SQE, test plans, infrastructure and test conclusions couple to the software life cycle elements. TO the degree that a reasonable HPC software life cycle framework is missing, it is hard to achieve the goal of a systematic SQE-focused test process. Constructing a recognized and standardized test discipline for high-integrity HPC is a difficult but worthwhile problem for research by the SQE community.
- We recommend the exploration of increased use of testing methods not traditionally applied in computational science. In particular, we advocate the increased use of statistical testing. This is correlated with a methodology currently applied to many ASC codes, systematic regression testing, that was not even visible as recently as 15 years ago. As pointed out by a National Research Council study (NAS, 1996) the problem of adequately covering complex software systems with structural tests is formidable, at least if coverage is defined to be of common use profiles or execution paths as well as discrete software elements. Statistical testing methodologies offer a means of addressing this problem. However, to accomplish this, an underlying framework of statistical software reliability must also be conceived.

Statistical testing, correlated with statistical specification of important use profiles, is a truly new idea in computational science software and strikes us as a highly important

step forward. From the perspective of the traditional computational science community, however, the difficulty posed by statistical testing methodologies is that their presumed results are also statistical. In particular, conceptions of reliability or effectiveness of the software based on statistical testing must, of necessity, be statistical. While this is new for the computational science community it is certainly not new for the broader software engineering community (Singpurwalla and Wilson, 1999). Certainly the use of statistical reliability concepts in software measurement is now rather conventional (Fenton and Pfleeger, 1997). A traditionalist would argue that the current ad hoc and evolutionary accumulation of evidence that an HPC code works is not statistical and therefore *better*. Our rejoinder is that the ad hoc and evolutionary accumulation of evidence is in truth uncertain, and therefore statistical in one sense or another whether the computational science community wished to admit it or not. Explicitly acknowledging the statistical aspects of computational science software reliability offers fresh insights and additional quality measures that are themselves of interest.

To take full advantage of this requires new work in characterizing faults in computational software as accuracy failures and other sorts of soft failures (code runs but results are not sufficient), rather than hard faults (code dies). This is theoretically what V&V is really all about, so there is an opportunity here for building on the failure information that V&V provides as well as the successes.

Finally, from the software reliability view, we need stochastic process models of faults in software that are indexed by spatial variables (typically high-dimensional, we are not referring only to the three dimensions of physical space but to the dimensions provided by all the various parameters that enter into the specification of specific simulations in a given computational science code). The easiest software reliability formulation develops stochastic processes and predicts software fault characteristics as functions of time. It is of interest to go beyond that for HPC.

- An important opportunity for the SQE community is presented by the need to research and develop predictive cost (time, money, level of effort) estimates for V&V of HPC software, especially test costs. The real goal is to estimate the cost of development and deployment of a sufficient testing infrastructure for HPC codes. By the cost of V&V for software, we mean the cost required to achieve a believed level of reliability as measure by specific metrics. Sufficiency dominates the concept of code acceptance (accreditation) discussed in Section 4, which is why we cast this point in this way. V&V costs are hard to estimate for highly formal software development activities, let alone for HPC with its lack of life cycle constructs and historical reliance on ad hoc, and essentially unbounded, user testing. Sufficiency is also governed by appropriate quantitative metrics. So, coupled with our call for better cost predictors is of course the need to develop appropriate measures of testing impact for HPC codes.

4. CODE ACCEPTANCE

The purpose of V&V is to develop a knowledge base that allows objective assessment of whether or not an HPC code can provide a good enough answer for a stated problem. For high-consequence applications, an HPC code must be a high-integrity software system and the

V&V must be performed in a way and to an extent that allows determination and warranting of the high-integrity characteristics that are required. This is the problem that we call *acceptance* or *accreditation*, and it remains the most difficult problem for computational science.

The problem has not traditionally been an issue when science is emphasized in HPC because of the accepted evolutionary character of the code (worse yesterday, better tomorrow) that matches the progress of science. When a high-integrity software product must be delivered on a finite time scale through the application of less than unlimited resources the question is dominant.

The Department of Defense recognizes the need to address these issues. That is why they have developed and deployed a VV&A (Verification, Validation and Accreditation) program to support DOD software efforts, rather than only a V&V program (DMSO, 2000). There, *accreditation* is defined as “The official certification that a model, simulation, or federation of models and simulations and its associated data is acceptable for use for a specific purpose.” This is also what we mean by acceptance in this paper.

Various questions enter into the judgment of acceptance, all providing opportunities for potential consideration of SQE methodologies.

- Was the code built “good enough?” How do you measure “good enough?”
- What are metrics for determining sufficient V&V? How many verification and validation tests do you need? How should the testing infrastructure be designed to support an evaluation of software acceptance for HPC? How should code performance on a collection of test problems be evaluated?
- When does the science in computational science end and SQE begin? When does the SQE end and the science begin?
- How is risk quantified for high-integrity HPC?
- Does the notion of formal acceptance actually make sense for HPC?

5. CONCLUSION

There are several factors that we have not discussed to any degree in this paper. For example, there are unique problems posed by the requirement to design mathematically correct solution algorithms for complex systems of partial differential equations that are accurate with available computing resources. These algorithms must also be efficient, that is, allow code executions on problems of interest on *required* time scales. There are also unique problems posed by experimental validation. Both of these factors add great complexity to the purely SQE factors that enter into a rational discussion of verification and validation. In the first case, a problem that is essentially mathematical must be solved. In the latter case, a problem that is essentially physical must be solved. Further discussion of these problems can be found, for example, in Oberkampf and Trucano (2002).

The issues we have discussed in this paper, HPC code development, testing, and acceptance, are neither academic nor trivial. Ad hoc evolution of ASC computational science software from a past state of “worse” to a future state of “better” is insufficient and

inappropriate because of the fundamental goal for its application that has been stated in the Introduction – to “Predict, with confidence, the behavior of nuclear weapons, through comprehensive, science-based simulations.” This is a high-consequence, demanding deliverable that will not likely be achieved through software engineering principles that aim low because of fear of inappropriate formality. In our opinion, the challenges we have raised that center on overlaps between computational science and software engineering should be studied and resolved to minimize the risk of future computational science catastrophic blunders.

REFERENCES

1. Agile Manifesto (2003), “Principles Behind the Agile Manifesto,” <http://agilemanifesto.org>.
2. J. Ambrosiano and M.-M. Peterson (2000), “Research Software Development Based on Exploratory Workflows: The Exploratory Process Model (ExP),” Los Alamos National Laboratory report, LA-UR-00-3697.
3. K. Beck (2000), *Extreme Programming Explained*, Addison-Wesley.
4. B. Beizer (1990), *Software Testing Techniques*, 2nd Edition, International Thomson Computer Press, London.
5. DMSO (2000), Verification, Validation and Accreditation Homepage; see <https://www.dmsomil/public/transition/vva/>.
6. DOE – United States Department of Energy (2001), “ASCI Software Quality Engineering: Goals, Principles, and Guidelines,” DOE/DP/ASC-SQE-2000-FDRFT-VERS2.
7. United States Department Of Energy (2003), “Advanced Simulation and Computing Program Plan,” Sandia National Laboratories report, SAND2003-3130P.
8. M. A. Ellis, et al. (2004), “2003 SNL ASCI Applications Software Quality Engineering Assessment Report,” Sandia National Laboratories, SAND2004-0075.
9. Fairley, R. E. (1985), *Software Engineering Concepts*, McGraw-Hill, New York, NY.
10. N. E. Fenton and S. L. Pfleeger (1997), *Software Measurement*, PWS Publishing Company, New York.
11. IEEE (1991), *IEEE standard glossary of software engineering terminology*, IEEE Std 610.12-1990, New York.
12. IEEE (1998), *Software Life Cycle Processes – Life Cycle Data*, IEEE Std. 12207.1-1997.
13. C. Jones (1997), *Applied Software Measurement*, McGraw-Hill, New York, NY.
14. P. Knupp and K. Salari (2003), *Verification of Computer Codes In Computational Science and Engineering*, Chapman and Hall/CRC, Boca Raton.
15. National Research Council (1996), *Statistical Software Engineering*, National Academy Press.
16. NIST (2002), *The Economic Impacts of Inadequate Infrastructure for Software Testing*, prepared by RTI, Research Triangle Park, North Carolina.
17. W. L. Oberkampf and T. G. Trucano (2002), “Verification and validation in computational fluid dynamics,” *Progress in Aerospace Sciences*, Volume 38, 209–272.
18. M. C. Paulk, C. V. Weber, B. Curtis, and M. B. Chrissie (1994), *The Capability Maturity Model: Guidelines For Improving the Software Process*, Addison-Wesley, Reading, MA.
19. D. Phillips (1997), *The Software Project Manager’s Handbook*, IEEE Computer Society, Los Alamitos, CA.
20. D. Post (2004), “The Coming Crisis in Computational Science,” Los Alamos National Laboratory Report, LA-UR-04-0388.
21. D. Post and R. Kendall (2003), “Software Project Management and Quality Engineering Practices for Complex, Coupled Multi-Physics, Massively Parallel Computational Simulations,” Los Alamos National Laboratory Report, LA-UR-03-1274.
22. W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery (1992), *Numerical Recipes*, 2nd Edition, Cambridge University Press.

23. J. J. Quirk (2005), "Computational Science – 'Same Old Silence, Same Old Mistakes' – 'Something More is Needed'," in *Adaptive Mesh Refinement – Theory and Applications*, Eds. T. Plewa, T. Linde, and V. G. Weirs, Springer, Berlin Heidelberg New York.
24. P. J. Roache (1998), *Verification and Validation in Computational Science and Engineering*, Hermosa, Albuquerque.
25. J. Sanders and E. Curran (1994), *Software Quality*, Addison-Wesley, Reading, MA.
26. N. D. Singpurwalla and S. P. Wilson (1999), *Statistical Methods in Software Engineering*. Springer-Verlag, New York.
27. T. G. Trucano, M. Pilch, and W. L. Oberkampf (2002), "General Concepts for Experimental Validation of ASCI Code Applications," Sandia National Laboratories Report, SAND2002-0341.
28. H. v. Vliet (2000), *Software Engineering, Principles and Practice*, John Wiley & Sons, Chichester.
29. P. Woodward and P. Colella (1984), "The numerical simulation of two-dimensional fluid flow with strong shocks," *Journal of Computational Physics*, Volume 54, Number 1, 115-173.
30. J. Zepper, et al. (2003), "Sandia National Laboratories ASCI Applications Software Quality Engineering Practices, Version 2.0," Sandia National Laboratories, SAND2003-0962.

ACKNOWLEDGEMENTS

We thank David Peercy, Steve Lott, Karen Jefferson, Mike McGlaun, Mike Eldred, Ann Hodges and Robert Heaphy of Sandia, and Jamileh Soudah of NNSA, for reviewing versions of this manuscript. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. The views presented in this paper are not intended to reflect official positions of Sandia National Laboratories, Los Alamos National Laboratory or the U. S. Department of Energy's National Nuclear Security Administration.