SANDIA REPORT

SAND98-2420

Unlimited Release

Printed November 1998

21

MS-0619

Review & Approval Desk
For DOE/OSTI, 12690

RECEIVED

NOV 27 1998

OSTI

Certainty in Stockpile Computing: Recommending a Verification and Validation Program for Scientific Software

James R. Lee

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express 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. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

DISCLAIMER

**Portions of this document may be illegible
electronic image products. Images are
produced from the best available original
document.**

SAND98-2420
Unlimited Release
Printed November 1998

Certainty in Stockpile Computing:
Recommending a Verification and Validation Program
for Scientific Software

James R. Lee

Applied Physics Center

Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-1179

Abstract

As computing assumes a more central role in managing the nuclear stockpile, the consequences of an erroneous computer simulation could be severe. Computational failures are common in other endeavors and have caused project failures, significant economic loss, and loss of life. This report examines the causes of software failure and proposes steps to mitigate them. A formal verification and validation program for scientific software is recommended and described.

Acknowledgments

The ideas presented in this report were developed during numerous discussions with dozens of staff members at Sandia who are in the trenches doing scientific computing every day. In particular, I thank Tim Trucano, Bill Oberkamp, Bob Cochran, and Len Lorence for their repeated discussions and suggestions. I am grateful to Jaime Moya, Tom Bickel, Jim Assay, and Wendell Kawahara for their insight into MAVEN, and to Paul Hommert and Bob Thomas for the cooperation and support of the ASCI Applications Program. I thank Gerry Yonas for sponsoring this study. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Contents

1. Introduction	4
2. Sources of Computational Failure.....	6
3. Principles of Verification and Validation.....	12
4. Requirements.....	14
5. Code Verification	16
6. Model Validation.....	19
7. Protocols for Stockpile Calculations.....	21
8. Conclusions: Managing a V&V Program	23
9. References	26

1. Introduction

The national laboratories increasingly rely on scientific computing to resolve technical issues that arise in managing the nuclear stockpile. This is the natural consequence of two powerful trends, one political and one technological. The first is the evolution of the nuclear weapons complex in the post cold war era. In the absence of a superpower threat that is visibly aggressive and hostile to the United States, there will be pressure to reduce the nuclear inventories and shrink the complex. The public will grow increasingly intolerant of risk, will not allow the resumption of nuclear testing, and will demand reductions in the cost of maintaining a nuclear deterrent. The second trend is the exponential improvement in computing technology. The goal of the Accelerated Strategic Computing Initiative (ASCI) is to further accelerate this development, leading to computers that run at tens to hundreds of teraflops.

The challenges of stockpile management in the twenty-first century will be many, and they are not widely appreciated by those outside the industry. A new weapon probably will not be designed for many years. This will push existing systems far beyond their design lifetimes, well past the limits of our practical engineering experience. The lack of nuclear tests means that system performance must be inferred from tests of the various subsystems and observations of degradation in the nuclear explosive package. Inevitably, subsystem deterioration eventually will force the replacement of components. In many cases, components will be redesigned from scratch because of sunset technologies. Some of the weapon technologies are quite exotic. They were developed specifically to meet the extreme design requirements of nuclear weapons. We will have to deal with the loss of specialized engineering expertise. For many critical components, most of the people who designed our newest system have left the labs or are near retirement. As we do this job, we must be ever mindful of the consequences that could result from errors in judgment regarding nuclear weapons.

The possibility that a revolution in computing will provide easy answers is enticing. Computer modeling is invisible. It doesn't generate fire, smoke, noxious odors, hazardous waste, or loud noises. Protesters don't line up at the gate in the morning when you run a

simulation. While the cost certainly is not insignificant, it is cheaper than digging tunnels at the Nevada Test Site and setting off nuclear devices. Virtually any question regarding the stockpile can be investigated on a computer. Aggressive efforts are underway to substantially improve our modeling capability in the areas of primary and secondary design, aging of materials, structural vibration, aerodynamics, fire simulation, shock physics, electromagnetic phenomena, production processes, electrical circuit design, and radiation effects. The only problem is that inferences drawn from computer simulations sometimes are wrong.

The danger that a serious mistake will be made in the design, construction, or handling of a nuclear weapon, based upon an erroneous computer simulation, is very real. In this report, I will examine the root causes that lead to failure in scientific calculations and propose steps to mitigate them. In particular, I will outline the essential elements of a program to verify and validate computer simulations. I will make recommendations for managing such a program in a systematic, comprehensive, and rigorous manner. It is not possible to eliminate all modeling errors and software bugs and to prove beyond all doubt that calculations are accurate. There is a great deal that we can do, however, to manage the risks inherent in computing. The goal is to demonstrate a level of certainty that is consistent with our responsibilities for managing the nuclear stockpile.

2. Sources of Computational Failure

Nearly all computer programs contain large numbers of bugs that, at least occasionally, lead to unintended behavior. Computers have become so ubiquitous that this fact is generally recognized. Most people are accustomed to dealing with fairly frequent manifestations of imperfection in their word processing, spreadsheet, and presentation software. Facing a deadline, they find that fonts are corrupted, tables get reformatted, e-mail from the boss disappears, and the operating system crashes. Life goes on. What is much less generally recognized is that safety-critical software sometimes fails with catastrophic consequences. These are programs that have been written with enormous attention to detail and subjected to extensive testing, and yet, they fail.

Examples of significant computational failures can be found in virtually every area where software has been used in critical applications. Faulty flight-control software has contributed to airplane crashes in the FA-18, the F-16, the Swedish Gripen, the RAF Tornado, the Fokker F.100, and the Airbus A320[1]. The inability of a Patriot missile to intercept the Iraqi Scud missile that killed 28 people during the Gulf War was caused by a floating point error[2]. A similar mistake caused the failure of the European Ariane 5 rocket. Software running on the navigation computer took into account the motion of the rocket swaying in the wind while initialization calculations were completed prior to launch. The calculations continued for a short time during launch, when the much greater horizontal velocity caused a variable to overflow. Detecting a trivial, irrelevant error, the computer shut down, because it had a twin backup. Unfortunately, the backup computer had just failed from the same problem. The result was characterized in French newspapers as a 37-billion franc (\$7 billion) fireworks display. This quirk in the algorithm was benign to the Ariane 4, for which the software was developed, and the problem was not caught when the code fragment was reused[3]. People have been killed in accidents caused by industrial control software. In a near miss, a software upgrade caused a robot on an automobile production line to pick up a vehicle and drop it on top of the safety engineer's hut[4]. Computer errors have caused financial chaos and economic loss. Montgomery Ward was shocked to discover that a data input mistake had caused an entire warehouse full of goods in Redding, California to disappear from its distribution

system. No trucks visited the warehouse to pick up or deliver goods for three years! The warehouse staff faithfully reported to work and was paid during this period[4]. Our own space program has had its share of problems. NASA was inconvenienced when programmers made gravity repulsive (Apollo 11) and forgot that the Earth moves (Gemini V), and the Mariner I mission was destroyed because a handwritten formula was incorrectly transcribed into the radar system software[1]. It could have been worse. Software errors have caused the North American Air Defense Command (NORAD) to go to full alert on at least three separate occasions, once when the Moon popped up over the horizon[4].

Scientific software presents several challenges beyond those that are common to all code development. The underlying phenomena usually are quite complex, as are the mathematical and numerical methods used to describe them. Applications frequently push the limits of available computer hardware, and new computing environments take time to become stable. Calculations depend upon the accuracy of input data. Scientific software is more susceptible than most to inappropriate use by those who are unfamiliar with the physical models and numerical methods. Hatton has published an extensive analysis of defects in scientific software from more than a dozen technical fields[5]. He concludes that scientific software typically is riddled with defects, and he suggests that calculations should be treated with severe skepticism.

The computing environment

The first documented bug was an unfortunate moth in Grace Hopper's Mark I computer[6]. Ever since, a programmer's first impulse has been to suspect the inner wiring of the computer as the cause of unexpected behavior in a program. On rare occasions, this is justified. The most notorious recent example was in the Intel Pentium Processor, in which the table used by the divide routine was not completely initialized. There are many other examples. The IBM System 360 did not correctly handle guard bits, which occasionally caused mistakes in subtraction[7]. The Intel 486 had flaws in the trigonometric functions[1]. Other computers have had problems with multiplication (returning 0.00482 for $y = x \times x$ when $x = 6 \times 10^{-79}$) and with the square function (returning $\text{sqr}(2.0) = 2$) [4]. Intel devotes prodigious effort to validating the designs of their microprocessors, at a cost of several hundred million dollars per year. However, the complexity of modern processors has reached a level where complete validation of the design is essentially impossible. A totally bug-free processor may never be manufactured again. More than 50 errata have been found in the Pentium II since it was shipped to customers[8].

The development of massively parallel computers places extreme demands on the reliability of the individual components. Even if the mean time between failures (MTBF) of a processor-memory module is as long as 20 years, a computer constructed of 9000 modules can have a MTBF of no longer than about 20 hours. It easily could be shorter if hardware failures are dominated by failures of the interconnections between modules.

This could limit the effective computing capability more than processor speed or memory size[9].

A crucial aspect of the computing environment is the stability of the compiler. This is often a problem when programs contain statements, perhaps inadvertently, that are formally unspecified, undefined, or implementation dependent in the language standard. Code that does not manifest itself in a computational error in one situation can do so when the compiler is changed or different compile options are selected. Compilers for massively parallel computers are especially complex and relatively new, so they tend to be updated frequently.

Another issue is the reliability of function libraries. In general, routines found in standard libraries are very robust, but if there is an error it can affect many applications. These are seldom checked thoroughly by the end user.

Coding errors

The classic form of computer bug is the coding error, where unintended behavior is generated by a mistake in the logic or syntax of the program. There are many ways to make such errors, and they are much easier to create than to find.

Computer languages are full of features that can trap the unwary. For instance, C contains 22 constructions for which the program behavior is unspecified, 97 for which the behavior is undefined, 76 for which the behavior is implementation dependent, and 6 for which the behavior is locale dependent[4]. An example is the expression

```
x = (i++) + a[i];
```

Whether *i* is incremented before or after the *i*th element of *a* is selected is unspecified and will depend on the compiler. Other constructions are perfectly legal and well defined, but they are prone to error. Perhaps the most common C bug is the test

```
if( i = j )
{
    do_something();
}
```

where the assignment operator `=` is used mistakenly instead of the equality operator `==`. The fact that everyone knows this is a common error doesn't stop it from happening.

Sometimes the program syntax is simply so complex that it invites error. The class hierarchy in C++ programs can become forbidding, especially when features such as multiple inheritance are used. What is perfectly obvious to the experienced programmer

creating the class structure may seem decidedly less obvious to her replacement, who is trying to modify the program and learn C++ concurrently.

Coding errors become more likely as the complexity of the structure and control logic of a program increases. Various metrics have been proposed for describing complexity. These include the number of decision points in a subroutine, the deepest level of nesting, the number of external variables in a subroutine, the number of static paths through a program, and the fan-in/fan-out product. While there is no consensus on what values signal danger, astoundingly high scores have been observed in commercial scientific software[5].

Formal programming standards, style guides, code analysis tools to detect static and dynamic faults, and independent software inspection are not widely used in the development of scientific software. The atmosphere for code teams working at the forefront of science can be tremendously exciting. Highly productive, clever teams with a "can do" attitude are the norm. Unfortunately, code development characterized by prodigious effort under deadline, caffeine consumption, and sleep deprivation is unlikely to produce bug-free software.

Mathematical and numerical complexity

If there is one characteristic that makes scientific calculations especially problematic, it is the exceptional complexity of the mathematical techniques and numerical methods that are embodied in the software. This is unavoidable. The only question is how to systematically recognize and mitigate the risks.

While each application may have a unique set of difficulties, they generally involve issues such as the numerical stability and convergence of iterative techniques, discretization error, complicated boundary conditions, estimation of uncertainties, and the accumulation of floating-point errors. Sometimes algorithms that improve computational efficiency make uncertainty estimation more difficult. Biasing techniques in Monte Carlo codes can suppress high-impact but low-probability histories called zingers, and convergence theorems may not exist for non-uniform grids. Oberkampf and Blottner have described some of the subtleties in applying Richardson's extrapolation method to problems in computational fluid dynamics. They suggest that in many simulations, convergence is much slower than expected[10]. While great advances in computing can be enabled by the development of new mathematical and numerical techniques, the limitations and pitfalls of new methods are discovered over time. This is particularly relevant now, as massively parallel computers become available. The standard algorithms that have been optimized for serial computation frequently don't run efficiently on the new platforms. New algorithms designed expressly for parallel processing are being researched intensively.

Code developers are not always expert in the techniques embodied in their software. If the project is large enough, there can be a separation of function between the development of numerical techniques and coding. Even in a small development team, subroutines developed by others may be borrowed. If the code developers are not familiar with the limitations of the algorithms, this can lead to trouble. The borrowed software also may contain outright errors. Code swapping propagates errors from project to project, and it limits the utility of code-to-code comparisons. Of course, reusing code also can enhance software reliability. If the software modules have been developed carefully and tested rigorously, reuse can be safer than redevelopment. A decision should be made mindfully after assessing the scope of the algorithms, the quality of the code, and the risks. The history of the borrowed code, the testing procedures that it has undergone, and any known problems or limitations should be understood fully and documented.

Nature is complicated

Except in rare instances, computer simulations are merely approximations of nature. Often, these approximations are crude. In developing a code, judgments are made about which physical phenomena to include and which to ignore. In the best case, an informed decision to exclude an effect is made after it is demonstrated to be unimportant to the intended application. In the worst case, its importance is unknown, but inclusion would make the problem intractable. Obviously, there is no accounting for effects that are unknown. Even when a phenomenon is included, a complete mathematical description may not be known. The best available model may have known deficiencies or limited applicability. Numerical accuracy in the solver will not help if the underlying equation is wrong.

In a typical application of simulation software, problem simplification is essential. The software user must systematically eliminate unnecessary detail to keep from overwhelming the capacity of the computer. A 100-teraflop computer will not render this step obsolete. If the user is not intimately familiar with the physical models and numerical algorithms in the code, essential detail may be discarded.

Simulations often use empirical models that depend on measured parameters. These include material properties and particle cross sections. In this case, the simulations can never be better than the database of parameters. A parameter may be valid only over some physical range, beyond which it changes value or the underlying model becomes invalid. For example, consider the elastic properties of a material. When the material becomes plastic, the coefficient of elasticity does not merely change. A completely different description of the material behavior is required. Extending a simulation beyond the regime where the models are valid can lead to results that are grossly incorrect. There also may be some intrinsic variability in a parameter, dependent upon manufacturing processes or sample histories that are unknown. Insuring the accuracy of data and the applicability of empirical models are daunting challenges in computer simulation.

Another is quantifying the impact of uncertainties in the data, especially when there are large numbers of parameters.

Sometimes nature is more than merely complicated. It can be chaotic[11]. The mathematical manifestation of chaos is instability in the differential equations that describe nature. Small changes in input parameters can cause large swings in results. Chaos is ubiquitous in nature. Normally, one of the design goals of an engineered system is to prevent chaotic behavior, but it cannot be avoided without understanding. The possibility of chaos presents two problems in computer modeling. The first is the chance that a non-linear phenomenon that drives instability has been discarded in simplifying the mathematical equations imbedded in the simulations. Non-linear terms that may be small corrections for one choice of input parameters can dominate the problem for other choices, and so these terms might mistakenly be thought insignificant and be neglected. The second problem is that an analyst may not be aware that the input variables lie in a chaotic region, even when the essential physics is included in the model. Unless the appropriate variations are made in the sensitive parameters, there may be no obvious clue to this behavior.

Inappropriate use

Distance, time, education, experience, organization, computing environment, retirement, and death may separate software users from developers. The result is simulation software being applied to problems that the developers never intended to support. The worst problems occur when an inexperienced analyst uses software that he doesn't understand. Unless there is detailed oversight and review by experts who are thoroughly familiar with both the codes and the applications, there are many opportunities for egregious error. Essential details are ignored, parameters explore uncharted regions, algorithms compute but never converge, and long-dormant coding errors awaken with devastating, albeit unnoticed, effect. The results can seem entirely reasonable, and yet be utterly wrong.

This is another distinguishing feature of scientific simulation software. Bugs certainly exist in all types of codes, but when errors are generated in system software or typical commercial applications, the fact that something is wrong usually is fairly obvious. With scientific simulations, severe errors can be insidiously difficult to detect. The code can execute without crashing, but simply give answers that are numerically incorrect. As long as curves are smooth and plots look plausible, there may be no obvious warning sign that something is amiss.

3. Principles of Verification and Validation

A formal verification and validation (V&V) program is fundamental to successfully realizing the ASCI vision, which is to "shift promptly from nuclear test-based methods to compute-based methods"[12]. There is no other way to demonstrate convincingly that high-consequence software has been developed and applied without significant error. The ASCI goal is to "create the leading-edge computational modeling and simulation capabilities that are *essential* for maintaining the safety, reliability, and performance of the US nuclear stockpile and reducing the nuclear danger" (*italics added*). A computational failure that would compromise the safety, reliability, or performance of a nuclear weapon or increase the nuclear danger obviously is of high consequence.

My recommendation is that every code development project sponsored by ASCI shall maintain detailed, written V&V plans. These plans shall be updated frequently and subjected to rigorous peer review. There are four essential elements that must be addressed thoroughly:

1. Requirements
2. Code verification
3. Model validation
4. Protocols for stockpile computations

These will be described in the following sections of this report.

Clear accountability must be established for delivering calculations that satisfy the requirements and are demonstrably accurate. This accountability extends to all aspects of V&V. In most cases, the logical choice for the accountable individual will be the project leader of the code development team.

While full accountability rests individually with each development project, the V&V plans should build upon global efforts to achieve goals common to all code development. These would include establishing stable computing environments, maintaining

thoroughly tested and documented software libraries, promoting standard code development tools, configuration management tools, code architectures, and interfaces between codes, and developing programming standards and style guides. A carefully focused effort to leverage resources will be critical to a successful V&V program.

Code development projects differ greatly in scope, complexity, and purpose. The V&V program should be flexible enough to manage sensibly the whole spectrum of activities. Flexible does not mean out of date, undocumented, or not reviewed.

4. Requirements

The requirements for software development derive from our fundamental responsibilities in managing the nuclear stockpile. Code requirements must be tied explicitly and accurately to specific stockpile activities. If requirements are understated, software will not include the necessary features, be developed with the required rigor, or be ready in time to support the stockpile. If requirements are overstated, the resources needed to implement a comprehensive V&V program will not be bounded sufficiently. A V&V program cannot be managed rationally until the requirements have been thoughtfully negotiated and documented.

The questions that should be addressed in developing software requirements include the following:

1. Who are the customers?
2. What are the essential applications that must be supported? The truly critical applications should be differentiated from those that are merely important.
3. When must the software be ready to support key milestones in the weapons program?
4. What is the impact of having this software? What development or testing activities that were done previously can now be avoided?
5. What are the major technical risks in developing this software?
6. How accurate does this software have to be? What are the consequences if calculations done with this software are erroneous?
7. Which computer platforms must be supported?
8. Who will use the software? Will the code developers or those expert in the physical algorithms and numerical techniques be setting up all critical applications, or will this be done by novices? Developing bulletproof software for use by non-experts is extraordinarily challenging.
9. What other software is needed to support the critical applications? What interfaces exist with other codes?

The detail with which these questions may be answered will vary depending on the project. In many software development projects, the answers evolve as the scope is broadened to include more challenging or different applications. Considerable effort is warranted in developing the requirements in as much detail as possible. This will pay off in a more tightly focused and efficient V&V program.

5. Code Verification

Code verification means insuring that software correctly solves the equations that form the mathematical representation of nature in the program. Whether these equations constitute an accurate and complete description of nature is the subject of model validation. The V&V plan should describe the steps that are taken to minimize the introduction of errors, specify procedures for identifying and correcting mistakes, document software features that monitor the performance and convergence of the algorithms, discuss the quantification of uncertainty, and describe tests that are done to assess the reliability of the finished product. The scope of the verification effort should be consistent with the documented requirements.

Processes for code development

Finding and fixing errors is difficult, and so anything that is done to minimize bugs in the first place has great value. While there are some unique difficulties in developing scientific software, there are also concerns that have very much in common with software development in general. Substantial work has been done to identify processes for developing code that are seen to improve the quality of the product.

One of the most comprehensive models for software development is the Capability Maturity Model (CMM) from the Software Engineering Institute (SEI) at Carnegie Mellon University[13,14]. This is a five-level model, ranging from level 1 (initial) to level 5 (optimizing). The operational level of an existing software development process is determined with questionnaires, which are filled out independently by the development staff and management. The model presents a prioritized roadmap for incrementally improving the process. At each level, there are specific objectives that must be met to advance to the next level. There is a balanced approach in the areas of standards and procedures, organization, tools and technology, and process metrics. This is key, because optimizing a deficient process can be counterproductive. A strategy that is designed to address the challenges of a level-3 process, for example, may bring an immature process operating at level 1 (characterized as *ad hoc* and chaotic) to a grinding halt. The dominant

challenges facing a level-1 organization are management deficiencies. These must be solved to provide a solid foundation for further growth.

There are many specific processes that can improve software quality, and they should be considered carefully in constructing a V&V plan. A *programming standard* may be developed to document the reasons for choosing a particular programming language, to develop a subset of that language that avoids unsafe features, to set limits on the complexity of the code elements, and to isolate platform dependencies with function wrappers to enhance portability. A *style guide* may address conventions for naming identifiers, guidelines for comment statements, indentation standards, and other issues that tend to be more emotionally charged than those dealt with in the programming standard. Formalized *code inspections* are widely used in developing safety-related software. These are most valuable when human inspection is used in conjunction with software tools that automatically detect as many defects as possible. Code inspection procedures should guarantee adherence to the programming standard. *Unit testing* and *regression testing* may be done to specific standards for code coverage and the use of particular fault-detection tools. In some cases, individuals who are not part of the development team do the testing. Procedures for *tracking bugs* will keep them in sight until they are resolved. *Change control* formalizes the process of proposing improvements to software, evaluating them for technical feasibility, validating the proposal against written requirements, changing code components or adding new components, testing the code, documenting the changes, updating the production libraries, and releasing the software. *Configuration control* manages different versions of the software and allows calculations to be repeated much later in exact detail. This can be particularly important in managing the stockpile, where decades may elapse between modifications of a weapon. Thorough documentation is critical.

A V&V plan should describe the code development environment. It should specify the development platforms, the programming languages, the compilers, the code development tools, and the software libraries. The code development processes should be described in detail. Adherence to the processes should be transparent and permit auditing. There should be procedures defined for modifying the code development process.

It may be entirely appropriate that code development processes vary widely from one project to another. Judgement is needed in assessing which software-engineering techniques are most effective, and these may be project dependent. What is essential is that mindful choices be made that are consistent with the requirements, the choices are documented, they are followed, they are defended in peer review, and accountability is clear.

Numerical methods

The sophisticated mathematical models and numerical methods used in scientific software make code verification intrinsically difficult. It also is difficult to make

generalizations about how to do it well. There is enormous variety in the problems being considered and the approaches taken.

At the very least, the approach must be clear. The equations being solved and the algorithms used to solve them should be documented explicitly. This includes any auxiliary equations, such as equations of state, cross-section models, turbulence models, interpolation methods used in look-up tables, convergence criteria, and anything else that can influence the results.

Performance metrics should be built into the algorithms wherever possible. Quantities that are conserved exactly in nature, such as energy, momentum, and charge, are only conserved approximately in many simulations. These must be tracked during calculations. In iterative algorithms, parameters that indicate the degree of convergence must be tracked. In Monte Carlo codes, the statistical uncertainties must be tracked. Software should alert users when parameters in a simulation exceed validated limits. In short, any significant source of numerical error should be monitored during the calculation, including those caused by the auxiliary equations. Estimates must be made of the ensuing uncertainty in the final results of the simulation. This includes quantifying discretization errors caused by the spatial grid resolution, the magnitude of the time step, or linear approximations to differential equations.

It is a major challenge to perform integrated tests of the equation solvers, where many algorithms interact in a complicated way and the problem geometry is complex. When analytic solutions exist, they can be extremely valuable. Code-to-code comparisons also are valuable, especially when another code contains the same physical models but uses different numerical techniques. The widespread practice of borrowing software must be considered in evaluating the results of code-to-code comparisons. There is a high probability that two codes that solve the same problems have identical code incorporated in some of the algorithms. Software coverage and relevance to the critical applications are serious issues in developing a set of test problems.

6. Model Validation

Model validation means demonstrating that the equations used to describe nature are sufficiently complete and accurate to support the intended applications. A V&V plan should document the physical basis of the mathematical description, identify parameters which must be experimentally determined, assess the accuracy and integrity of existing databases, identify the need for additional or confirming data to support the requirements, and document the systematic comparison of the code with experiments at all levels of integration.

The physical phenomena that are included in the model should be described in detail. Fundamental assumptions that are implicit in the model should be documented. Any simplifications, approximations, or known limits of applicability should be described. Related phenomena that are not included also should be described, along with the justification for their exclusion.

If calculations depend on measured quantities, it must be demonstrated that these are known well enough to support the applications. If data already exists, their accuracy must be assessed. If not available, a plan to acquire necessary data must be developed. The range over which parameters are valid must be determined. Any code calibration that must be done experimentally in the normal course of using the code should be described.

There should be a comprehensive validation of the code with experiments. The first step is to isolate each phenomenon in turn and do simple experiments that confirm the basic models. These experiments should be very simple to validate the physics with few complicating factors. The next step is to incrementally increase the experimental complexity. Experiments where multiple phenomena are important, or which stress the two and three-dimensional capabilities of the code, should be done. Finally, the code must be tested in highly integrated experiments that are comparable in complexity to the critical applications. A V&V plan should describe experiments that have been done as well as further experimentation needed to acquire confidence. The relevance of the experiments to the intended applications must be established.

Resources necessary to implement the code validation should be identified. This includes experimental facilities, specialized diagnostics and test equipment, and experienced personnel. Estimates should be made of the funding and time needed to carry out the experiments. Program partners that are expected to sponsor the code validation should be identified.

Uncertainty in the measured parameters, limitations in the code calibration procedures, and simplifications and approximations in the physical models all will lead to uncertainty in calculations. A V&V plan should describe how these uncertainties are estimated.

7. Protocols for Stockpile Calculations

Both the promises and the dangers of code development are realized when software is used to perform stockpile calculations. This is when computer simulations provide the basis for decisions about nuclear weapons that either enhance or compromise the safety and security of the nation. These simulations should not be done in a haphazard, *ad hoc* fashion.

The purpose of defining protocols for stockpile calculations is to insure that software is used properly in making decisions about nuclear weapons. Even codes that have been verified and validated thoroughly may be misused. All simulations have limitations, and these must be understood and considered.

Appropriate circumstances for doing stockpile calculations should be designated. The user must have the necessary expertise. Test problems must be run successfully on the computing platform. Code verification and validation must demonstrate that the application is within the scope of the software. In some cases, further experiments will be necessary to address validation issues that are specific to the problem at hand.

The entire simulation process should be documented in reproducible detail. This includes the version of the code used in the calculation, the computing platform, and the compiler. In most simulations the analyst makes a series of problem simplifications. These should be documented and justified. In some cases, the calculations must be linked to configuration control in the production of weapon components. This prevents production changes from invalidating analyses without due consideration.

Extreme care is warranted for the most critical applications. Various measures should be considered to reduce risk:

1. Use codes that have been formally accredited to be suitable for particular applications.

2. Formally accredit analysts to insure that they have the necessary expertise to use these tools for particular applications.
3. Perform independent simulations by separate teams of analysts. This provides a check on the problem simplification and engineering judgment that is inherent in computer simulations. There is additional benefit if the simulations are done with different codes or separate computing environments.
4. Scrutinize the simulations with peer review.
5. Audit the simulation process to guarantee that test problems are successfully completed, configuration control is maintained, necessary information is archived, and documentation is completed.

A V&V plan should describe and justify the protocols for doing stockpile calculations. The scope and formality of the protocols must be consistent with the documented requirements.

8. Conclusions: Managing a V&V Program

Demonstrating that calculations are correct is difficult. The consequences of misleading simulations could be severe. There are many opportunities for error, and much must be done to gain confidence. A V&V program is unlikely to succeed unless it has high priority and receives sustained attention.

Establish accountability

The first step is to establish accountability unambiguously. This is not as trivial as it might seem, since there are numerous players in a comprehensive V&V program. They include the software developers, the analysts who use the codes for specific applications, the customers who are responsible for the applications, scientists who do validation experiments, those who provide experimental facilities, and the numerous managers who fund various activities or provide people to implement the program. Splitting accountability is a bad idea. It guarantees that something important will not get done.

In most cases, the project leader of the code development team is in the best position to succeed in the long run. Knowledge and continuity are the deciding factors. This individual should know as much as anyone about the details of the software implementation and will have ready access to those who develop algorithms or run applications. Moreover, a project leader tends to be involved with a particular code for an extended period of time. Management structures, funding chains, and weapon programs with milestones are shifting sand, but there are project leaders who have experience exceeding three decades in developing a particular code or class of codes. Of course, accountability passes through the management chain of the project leader. The management responsibility can move as organization charts evolve, without undue disruption.

The scope of accountability must be defined carefully, since the project leaders usually do not control all of the resources needed to *implement* all aspects of a comprehensive V&V plan. However, it is reasonable to hold them accountable for the following:

1. Negotiate the requirements for the software.
2. Develop and implement the code verification plan.
3. Develop the code validation plan.
4. Become an advocate for the code validation experiments. Negotiate partnerships with programs that have a vested interest in the software to provide experimental resources. Bring shortfalls to the attention of management.
5. Develop the protocols for stockpile calculations.
6. Write the V&V plan, and keep it up to date.
7. Lead the defense of the V&V plan and its implementation in periodic peer reviews.
8. Demonstrate the accuracy and quantify the limitations of calculations that are done to support the nuclear stockpile.

These all can be done as an integral part of developing software. They should be supported by the same source of funds used to develop the code. This will increase the cost of developing code, probably substantially. However, I believe that this will be more efficient and more successful than attempting to graft a V&V effort onto code development, funded separately and managed externally.

Peer review

Rigorous peer review is a critical element of a V&V program. No other process is as efficient in exposing gaps in planning or deficiencies in execution. All aspects of V&V should be scrutinized in peer review, from defining requirements to stockpile computations.

Members of a peer review panel should encompass a breadth of experience. They should include experts in the physics incorporated in the code, experts in the applications of the code, customers, program managers, program partners, and those with experience in high-consequence computing. The perspective of people from outside Sandia can provide valuable insight in a peer review.

Metrics

There are several metrics that could be used by executive management to assess progress in a V&V program. Initially, these could include the fraction of code development projects for which V&V accountability has been established, the fraction that have written V&V plans, and the fraction that have been peer reviewed. Milestones could be defined for increasing these fractions to 100% within a defined period.

Another metric could be the establishment of program partnerships to provide experimental resources. This is an area where management involvement is likely to be especially helpful.

Understand the risks

My expectation is that an aggressive V&V program can make a dramatic difference in refining software requirements, improving software quality, developing better physical models, and minimizing application errors. I do not expect that any V&V program can make the risk of computational failure zero.

The scientists who are creating the revolution in modeling and simulation understand this viscerally. Managers have to work at it. A V&V program can reduce and quantify computational uncertainties, but it cannot make them disappear. Even as we celebrate the new computational tools, some small degree of humility is warranted as we apply them to the sobering task of managing the nuclear weapons stockpile.

There will be some instances where we are forced to make stockpile decisions based largely or entirely on computations, with little or no supporting test data. These cases demand special attention and should be the highest priority of the V&V program. There are many other problems that can be addressed through a variety of computational and experimental approaches. We should avoid becoming completely dependent upon calculations, except under the most compelling circumstances and after exhaustive consideration of the risks and potential consequences.

9. References

1. P. Neumann, *Computer Related Risks*. Addison-Wesley, 1995.
2. GAO, Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical Report GAO/IMTEC-92-26, GPO, February 1992.
3. R. Kunzig, "Europe's Dream," *Discover Magazine*, Vol. 18, No. 5, May 1997, p. 96.
4. L. Hatton, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. McGraw-Hill, London, 1995.
5. L. Hatton, The T Experiments: Errors in Scientific Software. *IEEE Computational Science and Engineering*, pp. 27-38, April-June 1997.
6. "IEEE Annals of the History of Computing," Vol. 3, No. 3, July 1981, pp. 285-286.
7. D. E. Stevenson, *Stories for Computation: Why Care is Needed*. Department of Computer Science, Clemson University, 1998 (unpublished).
8. C. Metz, "The Big Test", *PC Magazine*, Vol. 17, No. 18, October 20, 1998, p. 28.
9. J. Gustafson, Computational Verifiability and Feasibility of the ASCI Program. *IEEE Computational Science and Engineering*, pp. 36-45, January-March 1998.
10. W. L. Oberkampf and F. G. Blottner, *Issues in Computational Fluid Dynamics Code Verification and Validation*, Sandia National Laboratories, Albuquerque, NM, SAND95-1352, September 1997.
11. J. Gleick, *Chaos*. Penguin Books, New York, 1987.
12. *Accelerated Strategic Computing Initiative (ASCI) Program Plan*, DOE/DP draft.
13. M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, *Capability Maturity ModelSM for Software, Version 1.1*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, CMU/SEI-93-TR-024, February 1993.
14. M. C. Paulk, C. V. Weber, S. M. Garcia, M. B. Chrissis, and M. Bush, *Key Practices of the Capability Maturity ModelSM, Version 1.1*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, CMU/SEI-93-TR-025, February 1993.

Distribution:

1-MS-0513	R. J. Eagan, 1000	1-MS-9036	M. C. Hinckley, 2254
1-MS-0513	T. J. Allard, 1002	1-MS-9033	C. L. Knapp, 2262
1-MS-1427	S. T. Picraux, 1100	1-MS-9034	D. J. Beyer, 2263
1-MS-1056	B. L. Doyle, 1111	1-MS-9035	R. D. Monson, 2265
1-MS-0960	J. Q. Searcy, 1400	1-MS-9132	D. P. Green, 2265-1
1-MS-0953	W. E. Alzheimer, 1500	1-MS-9013	R. G. Miller, 2266
1-MS-1452	J. G. Harlan, 1500	1-MS-9014	T. R. Harrison, 2271
1-MS-0953	T. J. Cutchen, 1501	1-MS-9039	V. E. Byfield, 2282
1-MS-0613	D. H. Doughty, 1521	1-MS-0509	W. D. Williams, 2300
1-MS-0614	D. E. Mitchell, 1522	1-MS-0535	T. B. Lucy, 2301
1-MS-0614	K. R. Grothaus, 1523	1-MS-0505	D. L. Eilers, 2304
1-MS-0613	P. C. Butler, 1525	1-MS-0537	M. J. Mundt, 2314
1-MS-0665	L. J. Azevedo, 1541	1-MS-0501	S. M. Kohler, 2334
1-MS-0665	R. B. Pettit, 1542	1-MS-0503	B. D. Shafer, 2335
1-MS-0665	J. M. Simons, 1544	1-MS-0505	P. A. Molley, 2336
1-MS-1452	J. A. Merson, 1552	1-MS-0501	M. K. Lau, 2338
1-MS-1453	F. H. Braaten, Jr., 1553	1-MS-0533	W. H. Schaedla, 2343
1-MS-1454	L. L. Bonzon, 1554	1-MS-0533	D. R. Weiss, 2343
1-MS-0515	F. M. Bacon, 1561	1-MS-0537	R. M. Axline, 2344
1-MS-0521	T. J. Young, 1567	1-MS-0529	B. C. Walker, 2345
1-MS-1079	A. D. Romig, Jr., 1700	1-MS-0529	M. B. Murphy, 2346
1-MS-1083	P. S. Winokur, 1727	1-MS-0301	D. J. Rigali, 2400
1-MS-0525	P. V. Plunkett, 1734	1-MS-0303	W. E. Williamson, 2414
1-MS-1434	G. E. Pike, 1800	1-MS-0842	C. M. Hart, 2500
1-MS-0333	W. C. Moffatt, 1803	1-MS-0507	J. H. Stichman, 2600
1-MS-0457	G. N. Beeler, 2000	1-MS-0319	D. J. Gangel, 2601
1-MS-0457	W. J. Tedeschi, 2001	1-MS-0987	R. J. Longoria, 2611
1-MS-0429	R. D. Andreas, 2100	1-MS-0519	T. R. Perea, 2612
1-MS-0435	F. F. Dean, 2102	1-MS-0535	L. J. Dalton, 2615
1-MS-0427	W. R. Reynolds, 2103	1-MS-0329	D. W. Plummer, 2643
1-MS-0453	D. L. McCoy, 2104	1-MS-0319	R. E. Kreutzfeld, 2645
1-MS-0475	R. C. Hartwig, 2105	1-MS-0986	V. P. Salazar, 2663
1-MS-0447	J. O. Harrison, 2111	1-MS-0987	D. E. Ryerson, 2664
1-MS-0487	J. D. Mangum, 2121	1-MS-0986	M. E. Partridge, 2665
1-MS-0486	M. E. Bleck, 2123	1-MS-0311	J. W. Hole, 2671
1-MS-0436	G. L. Maxam, 2147	1-MS-0328	J. A. Wilder, Jr., 2674
1-MS-0479	P. A. Sena, 2151	1-MS-0986	S. C. Roehrig, 2691
1-MS-0482	A. B. Cox, 2161	1-MS-0469	R. J. Detry, 2900
1-MS-0483	T. D. Hernandez, 2165	1-MS-1231	R. L. Hagenruber, 5000
1-MS-0481	M. A. Rosenthal, 2167	1-MS-1231	W. M. Knauf, 5001
1-MS-0481	K. D. Meeks, 2168	1-MS-1233	J. F. Ney, 5100
1-MS-9005	J. B. Wright, 2200	1-MS-1233	J. B. Woodard, 5131
1-MS-9032	M. A. Dremalas, 2211	1-MS-1237	C. G. Thomas, 5132
1-MS-9014	W. G. Wilson, 2221	1-MS-0865	J. L. Moya, 9105
1-MS-1223	M. W. Callahan, 5133	1-MS-0865	J. R. Garcia, 9106
1-MS-1393	J. A. Ang, 5134	1-MS-0826	W. Hermina, 9111
1-MS-1237	J. L. Meinhardt, 5134	1-MS-0826	D. K. Gartling, 9111
1-MS-1237	R. K. Wilson, 5135	1-MS-0826	P. R. Schunk, 9111
1-MS-1237	A. K. Hays, 5136	1-MS-0834	A. C. Ratzel, 9112
1-MS-1223	D. M. Dominguez, 5138	1-MS-0835	R. J. Cochran, 9113
1-MS-1233	W. I. Klein, 5139	1-MS-0835	S. N. Kempka, 9113
1-MS-9003	D. L. Crawford, 5200	1-MS-0835	B. F. Blackwell, 9113

1-MS-9006	D. J. Havlik, 5201	1-MS-0835	R. E. Hogan, 9113
1-MS-0501	P. J. Wilson, 5202	1-MS-0835	D. W. Kuntz, 9113
1-MS-1211	T. A. Sellers, 5300	1-MS-0835	V. J. Romero, 9113
1-MS-1221	J. S. Rottler, 5400	1-MS-0827	R. Griffith, 9114
1-MS-0421	W. C. Hines, 5401	1-MS-0827	J. E. Johannes, 9114
1-MS-0421	T. D. Woodall, 5402	1-MS-0825	W. H. Rutledge, 9115
1-MS-1221	R. D. Bentley, 5404	1-MS-0825	V. A. Amatucci, 9115
1-MS-0415	K. J. Almquist, 5411	1-MS-0825	R. S. Baty, 9115
1-MS-0419	R. G. Easterling, 5412	1-MS-0825	F. G. Blottner, 9115
1-MS-0417	W. H. Ling, 5413	1-MS-0825	K. V. Chavez, 9115
1-MS-0425	S. K. Fraley, 5415	1-MS-0825	W. T. Gutierrez, 9115
1-MS-0423	G. S. Brown, 5417	1-MS-0825	B. Hassan, 9115
1-MS-0631	C. A. Yarnall, 5500	1-MS-0825	M. W. Kniskern, 9115
1-MS-0724	J. B. Woodard, 6000	1-MS-0825	M. A. McWherter-Payne, 9115
1-MS-0736	N. R. Ortiz, 6400	1-MS-0825	A. R. Lopez, 9115
1-MS-0739	K. D. Bergeron, 6421	1-MS-0825	W. L. Oberkamp, 9115
1-MS-1139	M. M. Pilch, 6423	1-MS-0825	J. L. Payne, 9115
1-MS-1136	P. S. Pickard, 6440	1-MS-0825	K. Salari, 9115
1-MS-1146	T. F. Luera, 6442	1-MS-0825	T. M. Sterk, 9115
1-MS-0977	S. M. DeLand, 6524	1-MS-0836	C. W. Peterson, 9116
1-MS-9001	T. O. Hunter, 8000	1-MS-0836	S. P. Burns, 9116
1-MS-9004	M. E. John, 8100	1-MS-0836	L. A. Gritzo, 9116
1-MS-9007	R. C. Wayne, 8400	1-MS-0836	G. F. Homicz, 9116
1-MS-9405	T. M. Dyer, 8700	1-MS-0836	D. D. McBride, 9116
1-MS-9402	G. J. Thomas, 8715	1-MS-0836	J. H. Strickland, 9116
1-MS-9161	S. M. Foiles, 8717	1-MS-0836	W. P. Wolfe, 9116
1-MS-9042	E. P. Chen, 8742	1-MS-0443	H. S. Morgan, 9117
1-MS-9405	P. E. Nielan, 8743	1-MS-0443	C. R. Adams, 9117
1-MS-9044	W. A. Kawahara, 8746	1-MS-0443	S. N. Burchett, 9117
1-MS-9214	M. L. Koszykowski, 8950	1-MS-0443	A. F. Fossum, 9117
1-MS-9011	J. C. Meza, 8950	1-MS-0443	G. D. Sjaardema, 9117
1-MS-0151	G. Yonas, 9000	1-MS-0557	T. J. Baca, 9119
1-MS-0337	R. D. Skocypec, 9002	1-MS-0828	J. H. Biffle, 9121
1-MS-0841	P. J. Hommert, 9100	1-MS-0828	L. M. Taylor, 9121
1-MS-0835	T. C. Bickel, 9101	1-MS-0841	M. C. Maurer, 9131
1-MS-0841	C. M. Hartwig, 9102	1-MS-0557	K. Roach, 9132
1-MS-0828	R. K. Thomas, 9104	1-MS-0555	M. S. Garrett, 9133
1-MS-1135	D. B. Davis, 9134	1-MS-1190	D. L. Cook, 9500
1-MS-0555	R. A. May, 9135	1-MS-1195	L. G. Benavides, 9501
1-MS-0827	J. Zepper, 9136	1-MS-1191	J. P. Quintenz, 9502
1-MS-0321	W. J. Camp, 9200	1-MS-1181	J. R. Asay, 9511
1-MS-0316	P. F. Chavez, 9204	1-MS-1188	R. A. Hamil, 9512
1-MS-0321	P. R. Graham, 9208	1-MS-1182	B. N. Turman, 9514
1-MS-0828	E. D. Gorham, 9209	1-MS-1193	J. E. Maenchen, 9515
1-MS-0318	G. S. Davidson, 9215	1-MS-1178	J. A. Hands, 9516
1-MS-1111	S. S. Dosanjh, 9221	1-MS-1182	D. D. Bloomquist, 9530
1-MS-1110	D. E. Womble, 9222	1-MS-1194	D. H. McDaniel, 9540
1-MS-1110	D. S. Greenberg, 9223	1-MS-1152	M. L. Kiefer, 9542
1-MS-0321	A. L. Hale, 9224	1-MS-1152	J. D. Kotulski, 9542
1-MS-1111	G. S. Heffelfinger, 9225	1-MS-1152	D. J. Riley, 9542
1-MS-0441	R. W. Leland, 9226	1-MS-1152	C. D. Turner, 9542
1-MS-0819	J. S. Peery, 9231	1-MS-1152	L. X. Schneider, 9543
1-MS-0819	T. G. Trucano, 9231	1-MS-1194	R. B. Spielman, 9544
1-MS-0819	E. S. Hertel, 9231	1-MS-1187	M. K. Matzen, 9570

1-MS-0820	P. Yarrington, 9232	1-MS-1187	J. L. Porter, 9573
1-MS-0439	D. R. Martinez, 9234	1-MS-1186	T. A. Mehlhorn, 9574
1-MS-0439	K. F. Alvin, 9234	1-MS-1196	R. J. Leeper, 9577
1-MS-0439	M. S. Eldred, 9234	1-MS-1178	J. J. Ramirez, 9580
1-MS-0439	J. R. Red-Horse, 9234	1-MS-1178	P. S. Raglin, 9581
1-MS-1165	J. Polito, Jr., 9300	1-MS-9999	V. S. Roberts, 9591
1-MS-1155	W. Beezhold, 9300	1-MS-1002	P. J. Eicker, 9600
1-MS-1165	D. M. Rondeau, 9301	1-MS-1005	R. W. Shaum, 9601
1-MS-1170	G. A. Zawadzka, 9302	1-MS-1003	R. D. Robinett, III, 9611
1-MS-1170	M. A. Robertson, 9309	1-MS-1008	J. C. Fahrenholtz, 9621
1-MS-1168	W. A. Boyer, 9321	1-MS-1010	M. E. Olson, 9622
1-MS-1156	R. Tachau, 9322	1-MS-1004	R. W. Harrigan, 9623
1-MS-1392	V. Gabbard, 9324	1-MS-1125	A. K. Miller, 9652
1-MS-1153	M. T. Buttram, 9330	1-MS-1006	P. Garcia, 9671
1-MS-1166	G. J. Scrivner, 9332	1-MS-1007	A. T. Jones, 9672
1-MS-1166	C. Drumm, 9332	1-MS-0428	W. C. Nickell, 12300
1-MS-1166	W. C. Fan, 9332	1-MS-0829	K. V. Diegert, 12303
30-MS-1179	J. R. Lee, 9341	1-MS-0829	B. M. Rutherford, 12303
1-MS-1179	W. P. Ballard, 9341	1-MS-0635	G. W. Kuswa, 12365
1-MS-1179	L. J. Lorence, 9341	1-MS-0867	J. L. Martinez, 14000
1-MS-1179	J. L. Powell, 9341	1-MS-0868	K. G. McCaughey, 14400
1-MS-1179	J. A. Halbleib, 9341	1-MS-1179	R. P. Kensek, 9341
1-MS-1179	G. D. Valdez, 9341	1-MS-9018	Central Technical Files, 8940-2
1-MS-1179	D. E. Beutler, 9341	2-MS-0899	Technical Library, 4916
1-MS-1106	A. W. Sharpe, 9342	1-MS-1167	E. F. Hartman, 9343
1-MS-1159	M. A. Hedemann, 9344	1-MS-0619	Review & Approval Desk for DOE/OSTI, 15102

1 Mike Jones
MS B259, X-PA
LANL
Los Alamos, NM 87545

1 Jay Boettner
MS P915, DX-DO
LANL
Los Alamos, NM 87545

1 Stephen Lee
MS F663, X-CI
LANL
Los Alamos, NM 87545

1 Rendell Carver
MS P940, DX-3
LANL
Los Alamos, NM 87545

1 Tom Bement
MS F600, TSA-1
LANL
Los Alamos, NM 87545

1	Linda Verdoorn Powers DP18 Department of Energy 19901 Germantown Road Germantown, MD 20874	1	John Reynders MS B260, CIC-DO LANL Los Alamos, NM 87545
1	Chris Keane Department of Energy DP18 19901 Germantown Road Germantown, MD 20874	1	Steven Ashby LLNL P. O. Box 808, L-561 Livermore, CA 94551
1	Rose Mary Baltrusaitis Group XTA, MS B220 LANL Los Alamos, NM 87545	1	Ken Hanson MS P940, DX-3 LANL Los Alamos, NM 87545
1	Gary Carlson LLNL P. O. Box 808, L-170 Livermore, CA 94551	1	Richard Klein LLNL P. O. Box 808, L-012 Livermore, CA 94551
1	David Luginbuhl DP 51 Department of Energy 1000 Independence Ave. SW Washington, DC 20585	1	William Reed DP 51 Department of Energy 1000 Independence Ave., SW Washington, DC 20585
1	Kirk Levedahl LLNL P. O. Box 808, L-016 Livermore, CA 94551	1	Robert Benjamin MS P940, DX-3 LANL Los Alamos, NM 87545
1	John Cerutti MS D413, X-HM LANL Los Alamos, NM 87545	1	Tom Adams LLNL P. O. Box 808, L-098 Livermore, CA 94551
1	Randy Christensen LLNL P. O. Box 808, L-066 Livermore, CA 94551	1	Bob Tomlinson MS B287, CIC-8 LANL Los Alamos, NM 87545
1	M. Peterson-Schnell MS B295, CIC-12 LANL Los Alamos, NM 87545	1	Allan McPherson MS B287, CIC-ACL LANL Los Alamos, NM 87545

1 Cynthia Nitta
LLNL
P. O. Box 808, L-096
Livermore, CA 94551

1 Robert Weaver
MS B220, X-TA
LANL
Los Alamos, NM 87545

1 Mary Zosel
LLNL
P. O. Box 808, L-561
Livermore, CA 94551

1 William Krauser
MS B220, X-DO
LANL
Los Alamos, NM 87545

1 Karnig Mikaelian
LLNL
P. O. Box 808, L-097
Livermore, CA 94551

1 Kris Winer
LLNL
P. O. Box 808, L-096
Livermore, CA 94551

1 Richard Bowers
MS B259, X-PA
LANL
Los Alamos, NM 87545

1 Jeff Brown
MS B272, CIC-8
LANL
Los Alamos, NM 87545

1 Christian Mailhot
LLNL
P. O. Box 808, L-055
Livermore, CA 94551

1 Donald Burton
MS F663, X-CI
LANL
Los Alamos, NM 87545

1 Richard LeSar
MS G755, MST-8
LANL
Los Alamos, NM 87545

1 John Pedicini
MS F664, X-NH
LANL
Los Alamos, NM 87545

1 Donald Sandoval
MS F664, X-NH
LANL
Los Alamos, NM 87545

1 Don McCoy
MS F670, ALDNW
LANL
Los Alamos, NM 87545

1 Robert Shea
MS F664, X-NH
LANL
Los Alamos, NM 87545

1 Art Dana
MS F664, X-NH
LANL
Los Alamos, NM 87545

1 Dean Preston
MS B218, X-DO
LANL
Los Alamos, NM 87545

1 Steven Girrens
MS P946, ESA-EA
LANL
Los Alamos, NM 87545

1 Charles Farrar
MS P946, ESA-EA
LANL
Los Alamos, NM 87545

1 Rollin Whitman
MS P941, DS-DO
LANL
Los Alamos, NM 87545

1 John Budzinski
MS F664, X-NH
LANL
Los Alamos, NM 87545

1 Paul Rightley
MS P940, DX-3
LANL
Los Alamos, NM 87545