

A CRITICAL LOOK AT QUALITY IN LARGE-SCALE SIMULATIONS

There is a disconnect between ASCI science and management. The author proposes an alternative paradigm. Simulation quality depends on the quality of insights gained, but software engineering is not the discipline to develop those insights. The author also proposes a definition for validation and differentiates it from current internal quality and verification approaches.

The Jan.–Mar. 1998 *IEEE Computational Science & Engineering* presented two articles on the Department of Energy's Accelerated Scientific Computing Initiative, from two respected authors. John Gustafson from Ames Lab at Iowa State University discussed ASCI's computational problems.¹ Alex Larzelere, the past Director for Strategic Computing and Modeling for Department of Energy Defense Programs, discussed software engineering and management.² The content of the articles could not have been more diametrically opposed.

Herein lies one of the biggest problems in large-scale simulations: management thinks that the “techy stuff” is a major annoyance, and the technical people do not understand the managerial need to keep costs and schedules under control. Both sides speak in the language of metrics: lines of code, test coverage, and dollars spent on one hand, and error and convergence on the

other. But these metrics measure different things for different purposes. Discussions about lines of code tested and rates of convergence do not lead to meaningful communications.

The role of management and science in simulation development must be changed. Software engineering is meant to produce software by a manufacturing paradigm, but this paradigm simply cannot deal with the scientific issues. This article examines the successes and failures of software engineering. I conclude that *process* does not develop software, *people* and their tools do. Second, software metrics are not meaningful when the software's purpose is to guarantee the world's safety in the nuclear era. Finally, the quality of simulations must be based on the quality of insights gained from the revealed science. Aristotle talked about it 2,300 years ago; it is time to listen. I propose a category-theory definition of validation.

Is there a problem?

This is not just an academic discussion. Problems such as the disconnect between management and technical people can lead to inaccurate or incomplete simulations. Failure has real consequences. Imagine this CNN news report:

[Jan 28, 2003.] This just in. From Makkah, Saudi Arabia. An FA-18 fighter carrying a tactical nuclear weapon has crashed in the center of this holiest of all Islamic sites. Although the weapon was not armed, it detonated. US experts cannot explain how this explosion could have occurred. The city was filled with people making the hajj. Because of the pilgrimage, there is no estimate of the number of dead. Indeed, we may never know.

Can current simulations be held to the high standard that ASCI demands? Can we guarantee the above story never becomes a reality? Is such an accident possible? Believe it! Here are just a few of the problems in technical systems that have surfaced in the literature:

- On October 5, 1960, the North American Defense Command went to 99.9% alert because the moon came up—the designers “forgot” that the moon rising over the horizon would show on radar. Forgiven. Except that 20 years later and twice in one month, NORAD threatened to shoot everything because of computer glitches.³
- The software in Apollo 11 had the sign wrong on the gravitational constant: some programmer made gravity repulsive instead of attractive.⁴
- Patriot missiles missed a Scud over Dhahran, Saudi Arabia, during the Gulf War. One problem among many was that their software used two different binary versions of the number 0.1. This led the Patriots to improperly compute the closing speed.

Can American management practices and software engineering guarantee that millions of lines of code will be without significant defects? If Les Hatton is correct that the number of fatal errors is proportional to the log of the number of lines of code,³ then a million-line code has approximately 10 fatal errors. Million-line simulations are common.

Also, we have some hints from reading W. Edward Deming’s *Out of the Crisis*.⁵ The concept of quality he proposes has yet to take hold of American manufacturing. Because software engineering uses the same metaphors as manufacturing, we can expect software engineering to not adhere to Deming’s thoughts. Even more disconcerting is that software is nowhere nearly as well-crafted as an automobile.

Besides the disconnect between management and technical people and the failure to embrace

quality, two key factors will likely contribute to future simulation failures:

- Scientific computing is subject to any number of failures: scientific and engineering judgments that get modified; numerical techniques that change due to new algorithms, environments, or precision requirements; and vagaries in computers and computer programming that lead to an unstable environment.
- Reliance on simulations represents a huge cultural change for all concerned. We cannot expect this transition to be simple or expect the world to wait while it occurs.

The modeling cycle and validation

Our primary concern is about models, not simulations. Models refer to the systems of assumptions, functions, and relations that make up a scientific or engineering discipline. In my article “Science, Computational Science, and Computer Science: At a Crossroads,”⁶ I review the Baconian scientific method cycle of modeling, experimentation, and insight, including simulation as a form of experimentation. We validate simulations for the same reason we rerun experiments. The validation exercises often produce deep insights, which we then use to modify the models.

Modeling and insight

But let’s be clear about the goals. We’re not interested in the codes per se. Richard Hamming said it best: “The purpose of computing is insight, not numbers.” Insight is a very elusive commodity and might take years to develop. Insight is what scientists and engineers count on to guide their research. Nobel prizes are given for insight, not necessarily for details.

Technical disciplines are not the only ones that need insight. A story about music illustrates several points. First, how valuable is experience and insight? Vladimir Horowitz was convinced that one note of the many thousands in Beethoven’s *Appassionata* piano sonata was wrong. Going to the original, he found he was right. The score had been miscopied almost from the beginning.³

The moral is clear: Horowitz had the insight born of years of study and experience—something that seems scarce these days. Beethoven wrote wonderfully structured music; Horowitz knew from the structure what the note had to be.

These insights are crucial to modeling. Larzelere points this out in ASCI’s case: the people

with first-hand knowledge of nuclear munitions will most likely be gone by 2010. This problem is not restricted to ASCI. As the older engineers and scientists retire, their knowledge, insights, and intuitions are lost.

History shows that you must be prepared to accept insights. Röntgen is quoted as saying, “Things must be believed to be seen.” Our present university system does little to educate for insight. Universities are under increasing pressure to prepare students for that first job. History shows that most engineers never advance formally past their undergraduate education. We should be teaching models and thought processes, not facts. Modeling starts and ends with insight. But much hard work in the middle might or might not be usable. To the goal-oriented, non-science-trained manager, the insights do not count. Therefore, management sees validation as a place to cut costs. The consequence for science—no insights, no model improvement, no economic improvement.

Simulations and validations

Management hopes that simulations will take the place of costly and lengthy development and testing. We must be able to validate that the science and engineering are “correct enough” from currently available or reasonably cheap test data. This usually means consistency among the many numerical experiments. We must also verify that the codes accurately reflect the best that science and engineering have to offer for the problem at hand.

Simulations evolve and are subject to long, costly developments. As understanding of the model improves, the simulation must change to reflect this. “Full science” simulations place extraordinary demands on machine and code. Consequently, the simulations are rewritten for any number of reasons. Large portions might not be reusable because of changes in the numerical method or computing environment; almost certainly, the detailed scientific codes will change, perhaps drastically. But each rewrite requires validation.

Modeling asks and answers questions about a system, using a particular paradigm. Validation answers the question “How well does the model reflect objective observations?” The operations research (OR) community has long had a paradigm for validation—Figure 1 shows Sargent’s Circle, which depicts the components of the development cycle as problem, conceptual model, and computer model. The inner arcs represent the development process, and the outer arcs, the V&V process.

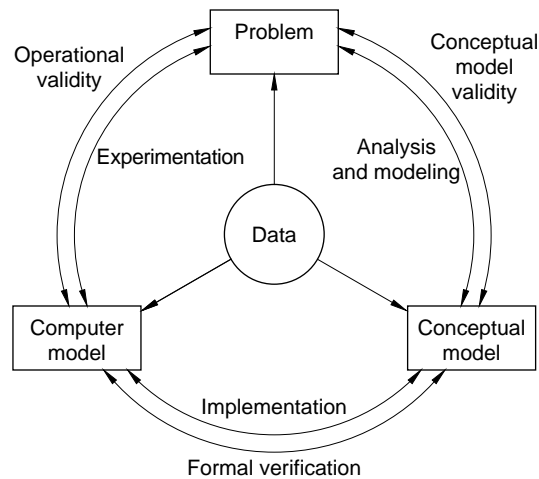


Figure 1. Sargent’s Circle.

There are two separate circles because development and V&V might evolve at different rates. If the V&V organization is totally independent of the development organization, these evolutions might be badly out of synchronization.

Sargent’s Circle fails to capture that V&V plays a vital role in the self-correcting nature of the process of science, and is not just the product. In OR, this might not be a problem, but in science this is exactly the problem.⁶ Validation should guarantee a model’s usefulness and upgrade the process by which science and engineering proceed.

Also, Sargent’s Circle is too old (1979) to recognize modern software-development processes. But far too many ways exist to graphically display the various phases of software development and team organization. I leave this to others.

Quality explained

Having looked at the end product—modeling and insight—we can now address the question of what quality means in science, mathematics, and simulations. The concept of quality in science and mathematics is not new; it goes back 2,300 years to the *Posterior Analytics* of Aristotle. I discussed its evolution in another article.⁶ Frederick Suppe presented a wonderful review of the state of the philosophy of science; his remarks give us much to contemplate on how validations should be conducted.⁷

I can think of no better way to start any discussion about quality than to look at Deming’s 14 Points (see Figure 2). After all, they’ve worked well for Japan. Some of these ideas are pretty radical, especially in an organization that

Figure 2.
Deming's 14
Points.⁵

1. Constancy of purpose.
2. Everybody wins.
3. Design quality in.
4. Cease doing business on price tag alone.
5. Continuous improvement.
6. Training for skills.
7. Institute leadership (not supervision).
8. Drive out fear.
9. Break down barriers.
10. Eliminate slogans.
11. Method (get rid of numerical goals/quotas).
12. Joy in work (abolish merit systems, as they promote competition rather than cooperation).
13. Continue education (not just about your job).
14. Accomplish the Transformation.

has been quite successful over the years doing something else. Deming's main emphasis is that people, not process, are responsible for developing quality products. Engineering education today stresses these principles because some organizations succeed when they adopt these attitudes. Unfortunately, engineering proponents of Deming's principles fail to appreciate that they apply to manufacturing, and not design. Simulations are not bolts turned out on an assembly line.

What would be a good prototype of the high-quality, highly innovative organization? What is that organization about? How do they do it? I propose that an excellent prototype is the famous Lockheed Martin Skunk Works (originally Lockheed's Advanced Development Projects) (www.lmsw.external.lmco.com/lmsw/text/body.html). The Skunk Works developed the U-2 and SR-71 airplanes. A look into how Clarence L. "Kelly" Johnson and Ben Rich ran the Skunk Works reveals an organization to which I'm sure Deming would have given his stamp of approval.

If the purpose of computing is insight, then we can measure the quality of the computing by the quality of the insight. Because insight leads to knowledge, we judge quality by the knowledge. But the major players in the technical game all have decidedly different views of knowledge:

- Scientific knowledge is inductive and requires experimentation and observation.

- Mathematical knowledge is classically deductive proof.
- How computer science will define its epistemology is not clear—if it knows what the problem is at all.

Knowledge requires justification; quality assurance is basically knowledge justification. The point is that if we cannot agree as to what constitutes knowledge, we cannot agree what constitutes quality. Nor can we agree on quality assurance until we can agree on justification.

I have proposed three principles for CSE knowledge:⁶

- *Physical exactness.* We must strive to eliminate nonphysical (mathematically convenient) assumptions.
- *Computability.* We must identify noncomputable relationships. Most mathematical relationships turn out to be approximate, not exact.
- *Bounded errors.* No formulation is acceptable without a priori error estimates or a posteriori error results.

These ideas plus the existing ideas of technical-knowledge justification are a start.

Impediments to quality

How do we gain quality? Unfortunately, it's not something that you find in the ground or on a tree. Quality is produced by people, so the lack thereof comes from people. How do we get poor quality?

Cognitive complexity

V.A. Vyssotsky of Bell Labs often gave insightful talks about his world; I was fortunate to hear him on many occasions. One of his comments was, "I've never seen a poorly performing underloaded system." This is especially true for people: overloaded, confused people lead to disasters.

Charles Perrow advances a thesis that accidents such as Three Mile Island have two causes: mind-boggling component complexity and mind-boggling interconnectivity of components.⁸ Is this not what is happening in simulations with very complex code spread across 9,000 processors? Looking at the spectrum of activities in the design, development, deployment, and maintenance of nuclear weapons, do they contain enough complexity and connectivity to harbor another Three Mile Island? We should expect that at some point

the complexity of the process will overwhelm our cognitive ability to understand it. Let me call this the issue of *cognitive complexity*.

Cognitive complexity is the difficulty in understanding a concept, thought, or system. Ultimately, the validity of code comes from our ability to understand the entire simulation. Cognitive complexity is an attempt to quantify mind-boggling. We know that this complexity is 7 ± 2 “things” and is closely related to Pareto distributions, but where is it for each person? For the programming group? How do we get a handle on this?

The cost of quality

Historically, the cost of quality has been thought to be extremely high in both time and money. Deming attacks this view much better than I could.⁵ But because “time is money,” costs make it hard to sell quality in our frantic, market-driven economic system. Computer science researchers now concentrate on “safety,” “safety-critical,” or “mission-critical” systems (such as NASA shuttles and nuclear power plants) because you can make a case for the resources to insure high quality. But these costs are high: Nancy Leveson reports a cost of \$15 million for inspecting a 1,200-line program for a nuclear reactor in Canada. How much more so for ASCI and other high-profile simulations, as in the aerospace and automotive industry?

Standards

There is no dearth of advice; the documents and organizations shown in Table 1 figure prominently in software-engineering literature. Not all this advice is worth listening to. For example, ISO9126, on quality in simulations, lists six attributes: efficiency, functionality, maintainability, portability, reliability, and usability. They do not list correctness, validatability, and verifiability. You might counter with, “But reliability is the same thing.” The previous Horowitz story shows that reliability is not equal to correctness: the score was reliably copied—except once.

To read the literature on the ISO 9xxx standards and the Capability Maturity Model, you would think all is well. Interestingly enough, Japanese industry has no ISO standards because their own national standards exceed the ISO/CMM standards. So why rush to ISO/CMM? Consider this about ISO: you can turn out junk as long as you know you are turning out junk.

Even so, standards are assumed to be the solution to software quality. For example, the UK

Table 1. Quality in software engineering.

Organizations	Documents
ISO	ISO 9xxx IEC 1508 (was ISO/IEC SC 65A [65A Secretariat 122])
NIST	ANSI/ASQC Q 90-4
US Dept. of Defense	MIL-Q 9858A, MIL-I 452058
NATO	NATO AQAP 1-3
Software Eng. Inst.	Capability Maturity Model (CMM)
IEEE	Total Quality Management
UK Defence Dept.	UK Def Stan 00-55
SAME	
NASA	
ACM	
AIAA	
ad nauseam	

Defence Department chose five rules to guide the choice of a programming language. The language should

1. have a formally defined syntax,
2. have a means of enforcing subsets,
3. have well-understood semantics and a formal means of relating code to design,
4. be block structured, and
5. be strongly typed.

Obviously, the current stable of languages does not stack up well. For example, Rule 2 requires an ad hoc tool, if we knew what the subset was supposed to be. Rule 3 is really two items. The first is the language semantics, which is often incompletely understood, even by very experienced programmers. The second is the tracking of code to documents, which is completely outside the purview of the compilers—another ad hoc tool.

Miracle cures

The software-engineering literature is filled with magical solutions to development problems. Fred Brooks said it best:

There is no single development in either technology or management technique which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.⁹

For ASCI, a major focus is on what are known in the literature as *problem-solving environments*. Much of the PSE work is being performed by John Rice and his colleagues at Purdue. It is too early to tell exactly what form these systems will take or what their exact place in the scheme of things will be. However, we must be wary that management will see PSEs as a magic cure-all (a silver bullet in Brooks' sense). M.M. Lehman further cautions,

To expect process models of themselves to improve the quality of software ... permitting, for example, total mechanisation of the process, is as futile as the search over the last three decades for automatic programming; [It] is, in fact, part of the same mirage.¹⁰

Academe

But things get worse. In a scathing indictment, Greg Wilson pointed out that academic computer science is flawed in that the students always play with toy projects.¹¹ My observation, both in and out of academia, is that science and engineering students “pick up” computer knowledge on their own. This makes Wilson's observation even more chilling: the players have neither the skill set nor experience to design, implement, or validate a large, complex simulation.

Industry

Where computer science is practiced for real—in industry—the groups simply are not given time or resources to develop sound practices that the groups can live with.

Certainly one of the outputs of software engineering has been tools. There is a large collection of software tools—sometimes called *shelfware*—that industrial groups attempt to use but soon discard, for whatever reasons, as irrelevant. “The shelfware syndrome is singular, if not unique, to the software world.”³ So the hunt continues for quality design disciplines and methods for measuring quality.

Artsy programming

Some will counter the drive to understand quality in terms of the code as written. They use the “artsy” argument: “Programming is an art and you can't constrain an artist.” Horowitz's story belies that argument. Anyone who thinks that great paintings occur without immense periods of training and preparation simply have not read their art history.

Speed freaks

My observation, first at Bell Labs and again as moderator of comp.parallel, has been that programmers love to talk about tricks to make the code faster, but I have never heard a programmer talk about tricks to make a code safer. Speed is the opiate of programmers.

The following comment from Les Hatton sets the tone for the “speed freaks:”

It is probably best to ban optimization of any ... code on the grounds that it is responsible for the bulk of the compiler errors reported in most languages and also because it effectively alters the defined characteristics of the program.³

How serious is the speed problem? If the late Seymour Cray was proud of anything, it was the speed of his designs. A lot of that speed was in the arithmetic. In fact, the Linpack benchmarks were the standard fare for the supercomputer-marketing corps for years. Now we have Lapack. According to James Demmel, new releases of Lapack will be for IEEE 754 arithmetic only. A personal communication from Demmel indicates that there are many reasons for going to IEEE-only arithmetic. Good-bye, X-MP.

The main concerns

Looking at the anecdotal evidence, we can find many areas of concern. (For more food for thought, see the related sidebar.)

People. Everyone needs to understand that insight is the reason to model. People can produce high-quality simulations when they are well-trained and well-motivated. CSE means that people have to be trained to work well in the group environment, something for which many people are not trained. It is especially troubling that the software industry continues to have an “any Java programmer will do” attitude.

Design. The evolution of simulations should follow scientific and engineering practice, not marketing practice. Conceptual integrity is often missing, thereby making the current code impossible to understand. Most failures come from what was not correctly specified; in the case of simulations, it is the science. Teams can drift off specification. V&V should not be independent of design. Current testing methodologies ignore numerical analysis, numerical methods, and floating-point computation. Current measures of software engineering merit might not be appropriate.

Food for thought

Below are some conclusions about software quality from what I can only hope will become a widely read paper by Les Hatton¹ and from other sources.²⁻⁵ I've also listed some obvious questions.

- Good practice matters. Unfortunately, academics don't teach it and organizations can't enforce it. (What is good practice? Is good practice decidable? Enforceable? What good practice is prevented by languages? What good practices are available to enhance speed?)
- Software-engineering practice does not address CSE. (What special good practices stem from science, mathematics, and simulations?)
- Speed must become subordinate to correctness. (What practices for speed should be abolished? Which should always be in force?)
- Detailed specifications, quality-assurance procedures, and formal testing are not enough. (Where does proof end and test begin? Why is proof so hard?)
- Double precision does not solve problems of unstable codes or ill-conditioned problems. (What practices enhance numerical stability? What should programmers "tell" programs?)
- Uncertainty caused by using less-well-defined algorithms is several times worse than that from using formal mathematical definitions. (Why don't we understand the translation of mathematics into programs?)
- Paradigm shifts in language or formal methods do not appear to automatically solve the problem. (What paradigm shifts are needed?)
- Safe subsets for languages are very important. How do we identify and enforce these subsets? How do we certify them?
- Comprehensive and objective testing, formal methods, and multiple versions might be helpful. (*Might* is the operative word. Which formal methods? Can we

- make multiple versions legitimately? Cost-effectively?)
- Construction and testing of static code fault finders are needed to find formally undefined behaviors in languages and systems, help enforce known standards, screen out well-defined behaviors we know we should not use, and help assess quality. (What are the undesirable language and design practices? Are these knowable a priori? What is decidable—that is, computable?)
- Documentation is not a panacea. (Why is the "programming" literature such a mess? We are likely documenting the wrong information: The derivation is the thing.)
- Current software-engineering metrics and processes measure nothing of interest except those measures discussed by Norman Fenton and Shari Pfleeger.⁶ I claim that those measures are more of cognitive complexity than any inherent measure of intrinsic complexity.

References

1. L. Hatton, "The T Experiments: Errors in Scientific Software," *IEEE Computational Science & Eng.*, Vol. 4, No. 2, Apr.–Jun. 1997, pp. 27–38.
2. G.V. Wilson, "What Should Computer Scientists Teach to Physical Scientists and Engineers?" *IEEE Computational Science & Eng.*, Vol. 3, No. 2, Summer 1996, pp. 46–55. (See also the responses on pp. 55–65.)
3. D.E. Stevenson, "Science, Computational Science, and Computer Science: At a Crossroads," *Comm. ACM*, Vol. 37, No. 12, Dec. 1994, pp. 85–96.
4. D.E. Stevenson, "Software Engineering Frontiers in Computational Science and Engineering," *Proc. 33rd Ann. Southeast Conf.*, ACM Press, N.Y., 1995, pp. 120–127.
5. D.E. Stevenson, "How Goes CSE? Thoughts on the IEEE CS Workshop at Purdue," *IEEE Computational Science & Eng.*, Vol. 4, No. 2, Apr.–June 1997, pp. 49–54.
6. N.E. Fenton and S. Lawrence Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed., PWS Publishing, Boston, 1997.

Two tasks are especially difficult: judging the appropriateness of the numerical algorithms and the parameter space, and certifying the computational and observed error.

The computing environment. There are three major problems: First, the environment is unstable with respect to tools. Second, machine obsolescence demands constant rewriting of software. Third, the programming model du jour prevents the reuse of either design or software. No end is in sight.

Tools are hard to develop and use. The implementation teams have neither the time nor the

funds to develop their own tools. Computer-algebra systems and theorem-prover support might not be available or used properly. There are no trusted, validated, or verified libraries or compilers.

Technical verification and validation

Our ultimate goal is to develop methods for designing quality into the simulation from the beginning. Thus far in numerical mathematics and computer science, we have not gotten at the root cause of the difficulty of going from pencil-and-

paper mathematics to a validated simulation. Clearly, this is a huge step.

Toward that goal, I've developed a definition of V&V that is reasonably independent of trendy terminology. My view is that Francis Bacon got it (mostly) right in 1620.

Developing the basic terminology

A *system* is a language and a collection of rules. A *model* is something described in a system's language, using the system's rules. We are given an *observational system*, which we can observe and perhaps even alter, but of whose internal functioning we do not have knowledge. Our *observations* are encoded in some *observational language* that need not be numerical. Using the language of a *theoretical* (or *formal*) *system*, we develop a *theoretical model*—that is, a statement of the functioning of the observed system. Using a *calculational system*, we determine that model's *calculational outputs*, and compare them to the observations.

If there is no observable system, there is no validation (whatever validation turns out to mean). You have to have a *standard* to compare against. That standard has to be out of the modeler's control. Nature does not negotiate.

Validation compares the system's observed features, through the observational language, with the theoretical model's outputs. It determines whether or not we are justified in believing that the model's outputs will always predict what would be observed. The validation problem is to set out the conditions under which we agree that the observations and calculations sufficiently agree, and that the theoretical system will produce this agreement in future calculations.

The verification problem is one of formal systems and therefore applies only to the theoretical system.

In the validation context, testing is no more than experimentation in the Baconian paradigm (But in the 85% of the system that is inherently computer science, testing might mean something else.) Although there might be software tools that aid experimentation, such things as “random” or “mutant” testing would seem to not apply.

Developing the definition

We need to pay attention to the three classes of systems: observational, theoretical, and calculational. Rudolf Carnap and Carl Hempel already address the observational-theoretical link.⁷ Crossing the Fetzer boundary (see below) between the theoretical and calculational system leaves the formal world. I believe that the theoretical sys-

tem's main use is to determine properties for consistent calculations: “How do I know it's right?” Complete validation of the observational-theoretical-calculational systems requires that we compute the right numbers for the right reasons.

The following justification attributes are more or less standard: *well-posed problems*, *well-conditioned formulations*, *stable numerical methods*, *convergence*, and *error analysis*. Other attributes should include accuracy, ease of use, maintainability, and validatability. We assume we receive a problem from the scientists that is well-posed and well-conditioned. A well-posed problem has a solution. A well-conditioned problem does not behave badly. Our goal is to use a stable numerical method that converges for the problem and for which we can analyze the numerical errors. We have to guarantee that errors introduced in the solution process are smaller than those introduced during observation.

To understand validation, we must understand the basic method of communication between the scientist and the machine. I want to make some distinctions not normally used in the literature and to merge the computer into the chain. You can think of mathematics as progressing from the ideal to the actual (my thanks to one of the referees who suggested this wording), in seven levels:

1. Classical analysis as practiced in science and engineering. This is the world of idealized science. Here we have infinite processes, an uncountable number of numbers, countably infinite precision, and idealized solution processes. This encoding of the system is the subject of Carnap and Hempel's development.⁷
2. Constructive analysis as practiced by followers of Errett Bishop, theoretical computer science, and certain areas of logic. Here we have rational numbers, so we have a countable number of numbers and so introduce error. Whether or not Platonic analysis and constructive analysis are equivalent is a long-standing point of contention.
3. Numerical analysis as the study of approximation, error, convergence, methods, and stability with rational numbers. You could consider it an extension of constructive analysis. It allows for “indefinitely long but finite” computations.
4. Numerical methods as the study of numerical analysis on finite arithmetics such as

IEEE (abstract) floating point. We have a finite number of numbers, finite precision, and finite processes.

5. Scott domains as a means of understanding semantics. Scott domains let us understand the meaning of computing constructs and programs.
6. The Fetzer boundary.¹² James Fetzer's controversial article "Program Proofs: The Very Idea" advanced the idea that until we actually run a program, everything is a formal system capable of analysis. Once run, the system is no longer formal, and proof is meaningless.
7. Machine codes as the active agents. This now includes real costs and real implementation problems. This produces the calculational output.

Breaking down the modeling in this way is convenient because each level introduces a separate concept of error and the much more dangerous situation of error propagation up and down the chain. Each of Levels 1 to 5 is formal; therefore, *deductive verification* is feasible.

To adequately define validation, I turn to a category-theoretic framework. Categorical language is useful here to eliminate disciplinary thinking. The objects are model representations. Morphisms relate two models: isomorphic ones are abstractly the same, while monomorphisms embed one into another. Each of Levels 1 to 5 can be thought of as a category with functors converting one form to another as we proceed from the ideal to the actual. Validation in the formal systems is determining the properties of the functors and following properties throughout the system.

Saying all this does not make it so. More to the point, what are the properties of these magical morphisms? This is obviously technical work that needs completion.¹³

Intrinsic quality and internal quality

As I discussed in "Quality explained," our view of quality comes by way of epistemology: how good is the insight and knowledge we receive from the simulation, and how well can we justify what we are doing? Whatever warrants we have must be tied to the model itself and hence intrinsic to it. Thus, the knowledge we want has several dimensions. The sum total of our faith in the system of models and machines I call *intrinsic quality*. Each dimension, such as the mathematics or the physics, has its own idea of *internal quality*.

In science, Occam's Razor stands as the measure of internal quality in a relative sense. Each scientific and engineering discipline has its own view of quality. For example, there is the mythical, elusive mathematical elegance in mathematical circles. However, science, engineering, and mathematics all use consensus as the basis of knowledge. In computer science, the concept of quality might be the most elusive of all. I would say that computer science has focused on internal quality: lines of code, test coverage, and so on. Intrinsic quality is more; it addresses all implementations. Intrinsic quality is the realm of the scientist and engineer.

Let's focus on numerical problems with numerical analysis. Numerical mathematics has standards of quality dating back three centuries: rates of convergence, error estimates, condition numbers, and sensitivity, among others. But these standards are for unlimited precision.

The idea of validation is foreign to mathematics. However, I argue that it is an inherent part of numerical programming. Consider zero-finding algorithms. They have a verification requirement (Does the algorithm and code work as required?) and a validation requirement (Is this algorithm appropriate for this particular situation?).

There is much work to do before we have viable, universally acceptable floating-point implementations. The computer industry seems to be dragging its collective heels on implementing IEEE 754. IEEE 754 is not just hardware; it is the entire computing environment. William Kahan regularly updates his Web site with known problems (www.cs.berkeley.edu/~wkahan/). And while we have been worrying about IEEE compliance, the three major chip manufacturers have taken integer overflow interrupts out of the chips or made them hard to intercept. A check of manufacturers' specifications shows that many chips do not generate either integer or IEEE 754 exceptions (see www.cs.clemson.edu/~steve/except.html). Unfortunately, even if all the safeguards were in place, we do not have the design mechanisms to prevent programmers from subverting them or to force programmers to use them at all.

What type of problems might we have? Here is one example. I examined the number of times $\sin^2 x + \cos^2 x \neq 1$. In the interval $[0, \pi/4]$ in steps of 2^{-20} , I found almost 30,000 violations, or approximately 3% of the time. How many of you already knew this? What is the consequence? Nobody seems to know. But we can say that $\sin^2 x + \cos^2 x = 1$ is neither verifi-

able nor a justifiable assumption.

We, as scientists and mathematicians, need to look for the intrinsic attributes of the science and mathematics and how they carry over to the code, perhaps seeking the simplest structures and algorithms (not necessarily the shortest or fastest). Therefore, intrinsic attributes are those in every instance of the simulation. Internal quality refers to those program elements that can be measured or attributed to the abstract program tree. Hatton observes that poor internal quality and dependence on weak linguistic features almost guarantee poor overall quality:

All the [internal] quality that is likely to be built into a code component must be built in before compilation, while the software is soft. ... After compilation, the software becomes brittle and the costs of building in [internal] quality, together with programmer resistance, rise considerably.³

Hatton has proposed these measures of internal quality:

- The number of statically detected faults.
- The number of transgressions of group programming standards.
- The number of uses of nonstandard features or vendor-dependent linguistic features with respect to the programming language standard.
- The number of uses of features outside a validated, safe subset (ANSI Fortran, ISO/IEC 9899 for C, the new standard for C++).
- The number of uses outside standard, validated libraries such as the C standard, Lapack.

All these numbers should be zero. However, experience certainly shows that unless these attributes are uniformly and universally enforced in each and every compile, our best efforts will be for naught.

Likewise, the virtual machine must be validated using such programs as Paranoia and Machar. (Strangely enough, only one reference to Paranoia exists in the literature. My colleagues and I are working to change that.)


Ways to measure the internal quality of programs exist. The compiler literature develops good mathematical models of structure. The hope of measuring the internal quality of every compilation rests on understanding how quality manifests itself. That means that metrics of compiler structures must form the foundation of intrinsic quality: control-flow structure, dataflow

structure, data structure, parallel structure, optimization structure, and semantic structure.

As far as I know, no one has applied such information to the concept of internal quality. These concepts have been used to derive metrics.¹⁴ Some compilers do communicate information about parallelism. However, a better information/feedback mechanism would make for better internal quality. I find the pragma or directive approach dangerous, in the same way Hatton found optimization dangerous.

Here's a homework assignment. The setting is the 1940s. Kelly Johnson designed the Lockheed P38 Lightning. During development and even into early use, its tails twisted off when it flew faster than approximately Mach .60. Johnson had suspected from the beginning that compressibility effects in the transonic region might cause problems. But first, the team had to battle critics who thought it was the Lightning's unique design. Compressibility finally won out as the culprit. The development team eventually overcame the problem, primarily using wind tunnel tests because the plane was too dangerous to fly in those regimes.

Your assignment? Put this into the modern setting. Using only the knowledge available in the early 1940s, your team is to develop a simulation of the P38. Your simulation must work out the compressibility problem and be the basis of the redesign. The simulation must be able to minimize the number of wind tunnel tests. (Modern wind tunnels might cost several millions of dollars per use.)

Hint: NASA researchers I contacted about this task say it is probably impossible. Consider what this means for simulation-based procurement. (This is the doctrine that no prototypes need to be built for something like the B-2.) A discussion for another time, perhaps? 

Acknowledgments

First, thanks to George Cybenko, who thought he saw something worthwhile in the first draft. I'd like to acknowledge the many fine suggestions from the eight

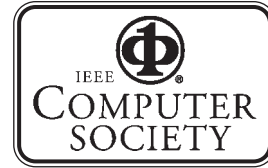
referees; I couldn't have written this article without their help. Several referees clearly did not agree with me, but were generous with their time and insights. Thanks to the CISE staff, who tried valiantly to straighten out my English and the article's organization.

References

1. J. Gustafson, "Computational Verifiability and Feasibility of the ASCII Program," *IEEE Computational Science & Eng.*, Vol. 5, No. 1, Jan.-Mar. 1998, pp. 36-45.
2. A.R. Larzelere II, "Creating Simulation Capabilities," *IEEE Computational Science & Eng.*, Vol. 5, No. 1, Jan.-Mar. 1998, pp. 27-35.
3. L. Hatton, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*, McGraw-Hill, New York, 1995.
4. P.G. Neumann, *Computer-Related Risks*, Addison-Wesley, Reading, Mass., 1995.
5. W. Edwards Deming, *Out of the Crisis*, MIT Press, Cambridge, Mass., 1986.
6. D.E. Stevenson, "Science, Computational Science, and Computer Science: At a Crossroads," *Comm. ACM*, Vol. 37, No. 12, Dec. 1994, pp. 85-96.
7. F. Suppe, "Introduction" and "Afterword," *The Structure of Scientific Theories: The Search for Philosophic Understanding of Scientific Theories*, F. Suppe, ed., Univ. of Illinois Press, Urbana, Ill., 1977, pp. 3-244, 617-730.
8. C. Perrow, *Normal Accidents*, Basic Books, New York, 1984.
9. F.P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, Addison-Wesley, 1995, pp. 207-226.
10. M.M. Lehman, "Models in Software Development and Evolution," *Software Process Modeling in Practice*, Butterworth-Heinemann, London, 1993.
11. G.V. Wilson, "What Should Computer Scientists Teach to Physical Scientists and Engineers?" *IEEE Computational Science & Eng.*, Vol. 3, No. 2, Summer 1996, pp. 46-55. (See also the responses on pp. 55-65.)
12. D.E. Stevenson, "What Is Computational Knowledge and How Do We Acquire It?" tech. report; www.cs.clemson.edu/~steve/resint.html.
13. J.A. Goguen, "Sheaf Semantics for Concurrent Interacting Objects," *Mathematical Structures in Computer Science*, Vol. 2, No. 2, 1992, pp. 159-191.
14. N.E. Fenton and S. Lawrence Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed., PWS Publishing, Boston, 1997.

D.E. Stevenson is an associate professor of computer science at Clemson University. His research interests center on computational science, primarily involving the question of the computational foundations for scientific models. He received his AB in mathematics from Eastern Michigan University, his MS in computer science from Rutgers University, and his PhD in mathematical sciences from Clemson. He is a member of the Shodor Education Foundation. He is also a member of the ACM, the American Mathematical Society, the Association for Symbolic Logic, and SIAM. Contact him at the Dept. of Computer Science, Clemson Univ., Clemson, SC 29634-1906; steve@cs.clemson.edu; www.cs.clemson.edu/~steve/.

PURPOSE The IEEE Computer Society is the world's largest association of computing professionals, and is the leading provider of technical information in the field.



MEMBERSHIP Members receive the monthly magazine **COMPUTER**, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.

EXECUTIVE COMMITTEE

President: LEONARD L. TRIPP
Boeing Commercial Airplane Group
P.O. Box 3707
M/S 19-RF
Seattle, WA 98124

President-Elect:
GUYLAINE M. POLLOCK *
Past President:
DORIS CARVER *
VP, Press Activities:
CARL K. CHANG †
VP, Educational Activities:
JAMES H. CROSS †
VP, Conferences and Tutorials:
WILLIS KING (2ND VP) *
VP, Chapter Activities:
FRANCIS LAU*
VP, Publications:
BENJAMIN W. WAH (1ST VP)*

VP, Standards Activities:
STEVEN L. DIAMOND *
VP, Technical Activities:
JAMES D. ISAAK *
Secretary:
DEBORAH K. SCHERRER*
Treasurer:
MICHEL ISRAEL*
IEEE Division V Director:
MARIO R. BARBACCI †
IEEE Division VIII Director:
BARRY JOHNSON†
Executive Director and Chief Executive Officer:
T. MICHAEL ELLIOTT †

*voting member of the Board of Governors †nonvoting member of the Board of Governors

BOARD OF GOVERNORS

Term Expiring 1999: Steven L. Diamond, Richard A. Eckhouse, Gene F. Hoffnagle, Tadao Ichikawa, James D. Isaak, Karl Reed, Deborah K. Scherrer

Term Expiring 2000: Fiorenza C. Albert-Howard, Paul L. Borrill, Carl K. Chang, Deborah M. Cooper, James H. Cross III, Ming T. Liu, Christina M. Schober

Term Expiring 2001: Kenneth R. Anderson, Wolfgang K. Giloi, Haruhisa Ichikawa, David G. McKendry, Anneliese von Mayrhauser, Thomas W. Williams

Next Board Meeting: June 1999, Richmond, Virginia

COMPUTER SOCIETY OFFICES

Headquarters Office
1730 Massachusetts Ave. NW,
Washington, DC 20036-1992
Phone: (202) 371-0101
Fax: (202) 728-9614
E-mail: hq.ofc@computer.org

Publications Office
10662 Los Vaqueros Cir.,
PO Box 3014
Los Alamitos, CA 90720-1314
General Information:
Phone: (714) 821-8380
membership@computer.org
Membership and
Publication Orders: (800) 272-6657
Fax: (714) 821-4641
E-mail: cs.books@computer.org

European Office
13, Ave. de L'Aquilon
B-1200 Brussels, Belgium
Phone: 32 (2) 7702198
Fax: 32 (2) 7708505
E-mail: euro.ofc@computer.org

Asia/Pacific Office
Watanabe Building
1-4-2 Minami-Aoyama,
Minato-ku, Tokyo 107-0062, Japan
Phone: 81 (3) 3408-3118
Fax: 81 (3) 3408-3553
E-mail: tokyo.ofc@computer.org

EXECUTIVE STAFF

Executive Director and CEO:
T. MICHAEL ELLIOTT

Publisher:
MATTHEW S. LOEB

Director, Volunteer Services:
ANNE MARIE KELLY

CFO: VIOLET S. DOAN

CIO: ROBERT G. CARE

Manager, Research & Planning:
JOHN C. KEATON

IEEE OFFICERS

President: KENNETH R. LAKER
President-Elect: BRUCE A. EISENSTEIN
Executive Director: DANIEL J. SENESE
Secretary: MAURICE PAPO
Treasurer: DAVID CONNOR
VP, Educational Activities: ARTHUR W. WINSTON
VP, Publications: LLOYD "PETE" MORLEY
VP, Regional Activities: DANIEL R. BENIGNI
VP, Standards Activities: DONALD LOUGHRY
VP, Technical Activities: MICHAEL S. ADLER
President, IEEE-USA: PAUL KOSTEK

