A CRITICAL LOOK AT DESIGN, VERIFICATION, AND VALIDATION OF LARGE SCALE SIMULATIONS

D. E. STEVENSON
DEPARTMENT OF COMPUTER SCIENCE
CLEMSON UNIVERSITY
CLEMSON, SC 29634-1906

Note to the Reader. I see six constituencies in CSE: computer, mathematical, and physical scientists; engineers; and technical and non-technical managers. I have adopted a conversational tone to make this article as widely accessible as possible. I have also provided a Bibliography.

1. There Is A Disconnect Between Us?

The January-March, 1998, issue of *IEEE Computational Science and Engineering* presented two papers on the Department of Energy's *Accelerated Scientific Computing Initiative* or *ASCI*. These two papers come from two respected authors. John Gustafson [6] from Ames Lab at Iowa State presented a discussion of the computational problems with ASCI. Alex Larzelere [9], the past Director for Strategic Computing and Modeling for Department of Energy Defense Programs, discussed software engineering and management aspects of ASCI. The two papers could not have been more diametrically opposed in their content. Herein lies the problem: 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: error and convergence on the one hand and lines of code, test coverage, and dollars spent on the other. But the metrics measure different things for different purposes. Discussions about rates of convergence and lines of code tested do not lead to meaningful communications.

The Gustafson-Larzelere discussion is not just an academic discussion. There are real consequences to failure. 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, the weapon 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 the number of dead. Indeed, we may never know.
>
> The entire Arab world is seething in anger. Calls abound for a *jihad* against the United States. ....

2. Is There a Problem?

Can current simulations be held to the high standard that ASCI demands? Can we guarantee the above story from never becomes a reality? Is such an accident possible? Believe it! Some evidence for computer-mediated disasters[1]:

- 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 rises over the horizon and would show on radar. Forgiven. Except twenty years later and twice in one month NORAD threatened to shoot everything due to computer glitches [7].

[1]The major sources only are cited. The references to the many examples in this paper are available from the author.

- The software in *Apollo 11* had the sign wrong on the gravitational constant: some programmer made gravity repulsive instead of attractive [10].
- *Gemini V* was 100 miles off course because a programmer played fast and loose with physics [7, 10].
- The *Patriot* missiles missed a *Scud* over Dhahran, Saudi Arabia, during the Gulf War. One problem among many was that two different binary versions of the number 0.1 were used. This led the *Patriot*s to improperly compute the closing speed.
- On March 13, 1985, a Blackhawk helicopter from Fort Bragg, North Carolina, crashed. Blackhawks are "fly-by-wire," meaning that computers mediate all control actions by the pilot. The final moments of *Chalk Three* were observed from two other helicopters. "... *Chalk Three* suddenly pitches sharply upwards, then pitches sharply downwards, plummeting until it impacts the earth at a near vertical angle, upside down and backwards".

These are just a few of the problems in technical systems that have surfaced in the literature.

Two short examples of how technology has failed us in the past.

- In 1940, the famous *Galloping Gertie*, the Tacoma Narrows Bridge, was destroyed due to undamped oscillations.
- In 1959, wings were falling off Lockheed *Electra*s. The cause was eventually tracked to a sixth harmonic set up by the overly-powerful engines. The harmonic was totally unsuspected; therefore, the problem had escaped exposure by testing.

In the Section 8, I present a problem from World War II that drives home the point that validation is an old problem.

Is it possible that American management practices and software engineering can guarantee that millions of lines of code will be without significant defects? If Hatton is correct that the number of fatal errors is proportional to the log of the number of lines of code [7], then a million line code has around ten fatal errors.

We have some hints from reading Deming's *Out of the Crisis* [2]. We can see that the concept of *quality* he proposes has yet to take hold of American manufacturing. Since software engineering uses these same metaphors, we can expect software engineering to *not* adhere to Deming's thoughts. What is even more disconcerting is that software is nowhere near as well-crafted as an automobile.

So what is the point? There are two:

1. Scientific computing is subject to any number of problems: scientific and engineering judgments subject to modification; numerical techniques subject to change due to new algorithms, environments, or precision requirements; and the vagaries in computers and computer programming leading to an unstable environment.
2. Reliance on simulations represents a huge cultural change for all concerned. We cannot expect this to be a simple transition nor for the world to wait while the transition occurs.

I address two concerns in this paper. The first concern is *people* and the second *technical validation and verification*. My argument is as follows:

- The understanding of scientific and engineering models is why we develop simulations. Simulations help to develop scientific and engineering insights into those models. This begins in Section 3.
- The Pareto distribution is in effect: 15 percent of the code will use 85 percent of the resources. That 15 percent of the code is the scientific software.
- Quality in CSE is based on the "reliability of insight" or what the philosophers call epistemology. But there are (at least) three views of what the epistemology should be.
- A simple model of the development enterprise is that it has four resource elements: People, time, quality, and budget. Management can only control at most two resources and thus the enterprise is under-determined.
- The product has six resources leading to development of verified and validated simulations:

People	Computing Environment
Science	Design and Mathematics
Programming	V&V Methods

- People are the most critical resource. People,not software engineering process, build validated simulations. I cover this Section 5.
- We lack a suitable framework to develop V&V methods. I present a framework for discussion in Section 6.

3. Modeling and Insight

This section takes up two issues. The first is that modeling and insight are the reason we develop simulations. The second point is that the *process of science* is also modified by simulations.

3.1. **Modeling And Insight Are The Focus.** 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. Validation is the process by which we attempt to convince ourselves that the simulations correctly capture the model and have some relation to an observable world. But models are not reality! Validation to a model is not validation to the "real world."

But let us be clear about the process. We are 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 may take years to develop. Insight is what the scientist or engineer counts on to guide her/his research. Nobel prizes are given for insight and not necessarily details.

Technical disciplines are not the only ones who need insight. This story about music illustrates several points and is referenced several times in this paper. First, how valuable is experience and insight?

> Vladimir Horowitz was convinced that one note of the many thousands to be found in Beethoven's *Apassionata* was wrong. Going to the original, he found he was right. The score had been miscopied almost from the beginning [7].

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

These insights are crucial to modeling. Lazerele points this out in ASCI's case: the people with the first hand knowledge of nuclear munitions will most likely be gone by 2010. This is not a problem restricted to ASCI. As the older engineers and scientists retire, their knowledge, insights, and intuitions are lost.

Our present university education 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 there is much hard work in the middle that *might* be usable or *might* not be usable. To the goal-oriented, non-science-trained manager the insights do not count. The consequence for science — no insights, no model improvement, no economic improvement.

3.2. **Simulations and Validations.** Management hopes that simulations will take the place of costly and lengthy development and testing processes. 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 science and engineering has to offer for the problem at hand. Simulations evolve and are subject to long, costly developments. As the understanding of the model becomes better, 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. But each rewrite requires validation. Large portions may not be reusable due to changes in numerical method or computing environment; almost certainly, the detailed scientific codes will change, perhaps drastically.

Modeling is the process of asking and answering questions about a system using a particular paradigm. Validation is about answering 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 the various components of the cycle as problem, conceptual model, and computer model. The various processes are meant to be self-explanatory. The inner arcs represent the development process and the outer arcs the V&V pprocess. There are two separate circles since development and V&V may evolve at different rates. If the V&V oorganization is totally independent of the development organization, these evolutions might be badly out of synchronization.

Balci's and Sargent's Circle fails to capture that V&V plays a vital role in the self-correcting nature of the *process* of science and not just the product. In OR, this may not be a problem, but in science this is exactly the problem (See [13]). Validation should guarantee a model's usefulness and upgrade the process by which science and engineering proceed.

Secondly, the Sargent's Circle is too old (1979) to recognize the modern software development processes. But there are far too many ways to graphically display the various phases of software development and team organization. We leave this to others.

What is quality? Having looked at the end product, modeling and insight, we can now address the question of how the concept of *quality* can be applied to science and mathematics. We can now decide what *quality* means in simulations.

4. Quality Explained

The concept of quality is larger than just simulations or V&V. If you accept my Pareto premise, then simulations may have huge supporting code systems. There is plenty for software engineering to do since computer science defines and implements the vast majority of the system. The ideas of software quality, going back to the early 1980s, may be sufficient for their part. This support system goes to the very heart of software requirements and specification but certainly is not part of the science, *per se*.

But the concept of quality in science and mathematics go back 2,300 years to the *Posterior Analytics* of Aristotle. I discussed the evolution in [13]. Suppe [15] presented a wonderful review of the state of the philosophy of science. There is much to contemplate in Suppe's remarks.

If the purpose of computing is insight, then the quality of the computing is measured by the quality of the insight. Since 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: (1) scientific knowledge is inductive and requires experimentation and observation; (2) mathematical knowledge is classically deductive proof; and (3) it is not clear how computer science will define its epistemology — if they know what the problem is at all. Knowledge requires justification, quality assurance can be taken as knowledge justification. The point is that if we cannot agree as to what constitutes knowledge then we cannot agree what constitutes quality. Nor can we agree on quality assurance until we can agree on justification.

In [13], I did propose three principles for computational science and engineering (CSE) knowledge:

> "
>
> *Physical Exactness.* We must strive to eliminate non-physical (mathematically convenient) assumptions.
>
> 2. *Computability.* We must identify non-computable relationships. Most mathematical relationships turn out to be *approximate*, not exact.

3. *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.

I have tried to establish the following:

1. There is a problem. Software quality is not what it must be.
2. Insight is the reason for modeling and simulations are one of the mechanisms for obtaining such insights.
3. Quality in simulations is hard to define since the technical communities have varying ideas. As a principle, *quality* of simulations refers to the reliability of the model the simulation imparts with each run.

The purpose of the discussion so far is to present some ground rules for the discussion of four categories of active CSE participants: (1) Engineers and physical scientists; (2) software engineering and management; (3) the mathematical sciences; (4) computer engineers and computer scientists. Academia is also implicated since it is a major source of people. Happily, many simulations get written and are adequate, products get designed, built and work just fine. My goal is to point out areas where we can do better. Instead, I make these comments organized on the six resources enumerated in Section 2. I start with an assessment of how people are used in the system. In Section 6, I propose some technical points for V&V. In Section 7, I explore the difference between what we have (termed *internal quality*) and what we want (termed *intrinsic quality*).

5. Assessment of People and the Tools They Use

My purpose is to survey the *whole* problem. Because we have many different players all tightly coupled, there can be no top-down, linear analysis. Therefore, I propose to discuss topics as they naturally arise in the discussion and then summarize.

I see the following as axiomatic: (1) Only well-trained and motivated people can build quality software and (2) the only tools that matter are those that work with the code the compilers actually use to generate code.

I can think of no better way to start any discussion about quality than to look at W. Edward Deming's *Fourteen Points* [2] seen in Figure 2. After all, it has worked well for Japan. Some of these ideas are pretty radical, especially in an organization that has been quite successful over the years doing something else. However, engineering education today stresses these principles because organizations succeed when they adopt these attitudes. Unfortunately, these rules are for manufacturing and construction industries. Scientific simulations are hardly three bedroom homes built from mass produced blueprints; but that is how software engineering sees it.

You, the reader, take a moment to think about a really highly creative, high quality organization (not product!). What is that organization about? How do they do it? For me, that organization is the famous "Skunk Works," the Lockheed-Martin's Lockheed Advanced Development Company. The Skunk Works developed the U-2 and SR-71. A check into how Clarence L. "Kelly" Johnson and Ben Rich ran the Skunk Works reveals an organization that I am sure Deming would have given a stamp of approval[2] .

V. A. Vyssotsky of Bell Telephone Laboratories 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, under-loaded system." This is especially true for people: overloaded, confused people lead to disasters. Charles Perrow, in *Normal Accidents* [11], advances a thesis that there are two causes for accidents such as Three Mile Island: mind-boggling component complexity and mind-boggling interconnectivity of components. 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 is there enough complexity

[2] `http://www.lmsw.external.lmco.com/lmsw/text/body.html`.

and connectivity to harbor another Three Mile Island? We should expect there to be a point at which the complexity of the process overwhelms our cognitive ability to understand. Let me call this whole issue 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?

Quality is hardly a new issue. Historically the cost of quality has been thought to be extremely high in both time and money. *Out of the Crisis* [2] attacks this view much better than I could. But since "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 one can make a case for the resources to insure high quality. But be aware that these costs are high: Leveson reports that $15 million to inspect 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?

While there is no money, there is no dearth of advice (Figure 3); the named documents and organizations show prominently in software engineering literature. Not all of this advice is worth listening to. For example, ISO9126, on quality in simulations, list six attributes:

Efficiency	Functionality	Maintainability
Portability	Reliability	Usability

Notice that correctness, validatability, and verifiability are not among the attributes they think should be in a simulation. One might counter with, "But reliability is the same thing." The Horowitz story above 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 (CMM), one 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 would one rush to ISO/CMM? Consider the following realization about ISO: you can turn out junk as long as you know you are turning out junk.

Standards are taken as the solution to software quality. For example, the UK Defence Department chose the following five rules as guiding the choice of programming language:

1. Formally defined syntax.
2. Means of enforcing subsets.
3. Well-understood semantics and a formal means of relating code to design.
4. Block structured.
5. Strongly typed.

However, the current stable of languages does not stack up well. For example, item 2 requires an *ad hoc* tool, if we knew what the subset was supposed to be. Item 3 is really two items: (3a) the language semantics and (3b) the tracking of code to documents is completely outside the purview of the compilers — another *ad hoc* tool. On semantics: after dealing with programmers and programming for 30 years, I claim that the semantics of any real language are only incompletely understood by even very experienced programmers.

The software engineering literature is filled with magical solutions to development problems. Fred Brooks said it best in "No Silver Bullets" [1]:

> "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."

For ASCI, a major focus is on what are known in the literature as *problem-solving environments* (PSE). Much of the PSE work is being reported out of Purdue under John Rice. It is too early to tell exactly what form these systems will take or what their exact place in the scheme of thing will be. However, one must be wary that management will see PSEs as silver bullets in Brooks' sense. To caution further:

> "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."

But things get worse. In a scathing indictment of academic computer science, Greg Wilson pointed out that academic computer science is flawed in that the students always play with toy projects [16]. It is my observation, both in and out of academia, 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.

Where computer science is practiced for real — in industry — the groups simply are not given time nor 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" [7]. So the hunt continues for quality design disciplines and methods for measuring quality.

There will be those who counter the drive to understand quality in terms of the code as written. Their argument is the "artsy" argument: "Programming is an art and you can't constrain an artist." The tale of Horowitz in Section 3.1 belies that argument. I believe anyone who thinks that great paintings occur without immense periods of training and preparation simply have not read their art history.

The following comment from 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" [7].

It has been my observation, first at Bell Labs and again as moderator of `comp.parallel`, that programmers love to talk about tricks to make the code faster but I have never heard a programmer talk about tricks making a code safer. Speed is the opiate of programmers. Even if PSEs eliminate the human element the underlying virtual machine could do us in.

How serious is the speed problem? If there was anything the late Seymour Cray was proud of, 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 Professor James Demmel, new releases of LAPACK will be for IEEE 754 arithmetic only. A personal communication from Professor Demmel indicates that there are many reasons for going to IEEE only arithmetic [3]. Good bye, X-MP.

Below are some conclusions from what one can only hope will become a widely read paper by Les Hatton[8] and other sources[13, 16]. Some obvious questions are listed too.

1. 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?*
2. Software engineering practice does not address computational science and engineering. *What special good practices stem from science, mathematics, and simulations?*
3. Speed must become subordinate to correctness. *What practices for speed should be abolished? Which should always be in force?*

4. Detailed specifications, quality assurance procedures, and formal testing are not enough. *Where does proof end and test begin? Why is proof so hard?*
5. Double precision does not solve problems of unstable codes or ill-conditioned problems. *What practices enhance numerical stability? What should programmers "tell" programs?*
6. Uncertainty by using less-well-defined algorithms is several times worse than using formal mathematical definitions. *Why don't we understand the translation of mathematics into programs?*
7. Paradigm shifts in language or formal methods do not appear to automatically solve the problem. *What paradigm shifts are needed?*
8. Safe subsets for languages are very important. *How do we identify and enforce these subsets? How do we certify them?*
9. Comprehensive and objective testing, formal methods, and multiple versions may be helpful. May *is the operative word. Which formal methods? Can we make multiple versions legitimately? Cost-effectively?*
10. Construction and testing of static code fault finders are needed to
    (a) Find formally undefined behaviors in languages and systems.
    (b) Help enforce known standards.
    (c) Screen out well-defined behaviors we know we should not use.
    (d) Help assess quality.
    What are the undesirable language and design practices? Are these knowable a priori*? What is decidable (computable)?*
11. 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*
12. Software engineering metrics and processes, as currently practiced, measure nothing of interest except those in Chapter 8 of Fenton and Pfleeger [4]. I claim that those measure are more of *cognitive* complexity than any inherent measure of *intrinsic* complexity.

Looking at the anecdotal evidence, we can find many areas of concern.

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 many people are not trained for. It is especially troubling that the software industry continues to have "any Java programmer will do" attitude.

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

There are two especially difficult tasks: (1) judging the appropriateness of the numerical algorithms and the parameter space and (2) certifying the computational and observed error.

The Computing Environment. There are three major problems: (1) The environment is unstable with respect to tools, (2) machine obsolescence demands constant rewriting of software, and (3) the programming model *du jour* prevents the reuse of either design or software. There is no end in sight.

Tools are hard to develop and use. The implementation teams do not have the time nor funds to develop their own tools. Computer algebra system and theorem prover support may not be available or used properly. There are no *trusted, validated*, or *verified* libraries or compilers.

6. Technical Verification and Validation

The purpose of this section is to develop a definition of validation and verification that is reasonably independent of terminology. My view is that Francis Bacon got it (mostly) right in 1620.

6.1. Developing the Terminology.

Observed System and Models. We are given a system called the *observed system* that we can observe and perhaps even alter but for which we do not have knowledge of the internal functioning. Our *observations* are encoded in some *observational language* which need not be numerical. We develop a statement of the functioning of the observed system in formal systems which we call a *theoretical system*. We are able to calculate values called *calculational outputs* of the model. These calculational outputs can be compared to the observations.

As discussed in Section 4, our view of quality comes by way of epistemology: (1) how good is the insight and knowledge we receive from the simulation and (2) 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.

I want to emphasize that if there is no observable system, there is no validation (whatever *validation* turns out to mean). One has to have a *standard* to compare against. That standard has to be out of the control of the modeler. Nature does not negotiate.

Validation is the process of comparing the observed features of the system through the observational language with the outputs calculated by the theoretical model and determining whether or not we 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 (1) that the observations and calculations are in sufficient agreement and (2) that the theoretical will produce this agreement in future calculations.

The verification problem is a problem of formal systems and therefore only applies to the theoretical system. More about verification below. Testing is no more than experimentation in the Baconian paradigm. In the 85% of the system that is inherently computer science, testing may mean something else; in the validation context, it means experimentation. While there may be software tools to aid in experimentation, such things as "random" or "mutant" testing would seem to not apply.

6.2. Focus on 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.

I can make no claim to being a working scientist. I do pretend to know something about the philosophy of science and something about the philosophy of mathematics as well as having written a program or two using a programming language or two over a 30 years span. That having been said, let me plunge onward.

What I have to say in this section relates to mathematics and computing. I do note the philosophy of science had many very interesting developments on the logic of science [15] and continuing development through the 1990s. I have tried to use these ideas in [14]. The logical rules for the observed system and the theoretical system are beyond the scope of this discussion.

There are three classes of systems to attend to: observational, theoretical or formal, and calculational. Carnap and Hempel [15] already address the observational-theoretical link. Crossing the Fetzer boundary(see below) between the theoretical and calculational system leaves the

formal world. I believe that the main use of the theoretical system is to determine properties for consistent calculations. "How do I know it's right?" Complete validation of the observed-theoretical-calculational system requires requires that we compute the right numbers for the right reasons.

We introduce the following terms which are more or less standard:

Justification Attributes

Well-posed problems.
Well-conditioned formulations.
Stable numerical methods.
Convergence.
Error Analysis.

We assume we receive a problem from the scientists that is well-posed and well-conditioned. A well-posed problem is one that has a unique solution. A well-conditioned problem is one that 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. These ideas can be found in the first chapter of many numerical analysis or numerical methods texts. We have to guarantee that errors introduced in the solution process are smaller than those in the observational process.

In order 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. One can think of mathematics as progressing from the ideal to the actual[3]:

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's and Hempel's development documented in [15].
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 on the rational numbers. If one adopts constructive analysis, it is 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 allow us to understand the meaning of computing constructs and programs.
6. The Fetzer boundary [12]. In Fetzer's controversial (in computer science) 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.

It is convenient to break it down modeling in this way since each introduces a separate concept of *error* and the much more dangerous situation of *error propagation* up and down the chain. Each of the systems 1–5 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.

[3]My thanks to one of the referees who suggested this wording.

Morphisms relate two models: isomorphic ones are abstractly the same while monomorphisms embed one into another. Each of the levels 1–5 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. Some ideas come in [5].

7. Intrinsic Quality versus 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. 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 may be the most elusive of all. I would characterize what computer science has now focused on as *internal quality*: lines of code, test coverage, *etc.* Intrinsic quality is more since I see it as addressing all implementations.

Let me re-iterate that intrinsic quality is the realm of the scientist and engineer.

Let's consider 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. There is a verification requirement: that the algorithm and code work as required. There is a validation concept of *appropriateness*: is this algorithm the appropriate one to use in this particular situation.

There is still work to do on floating point arithmetic. The computer industry seems to be dragging its collective heels on implementing IEEE 754. It is not just hardware; it is the entire computing environment or virtual machine as it is known. William Kahan regularly updates his web site with known problems[4]. 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[5]. Unfortunately, even if all the safeguards were in place, we do not have the *design mechanisms* to prevent the programmers from subverting these safeguards.

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} we find almost 30,000 violations or about 3 percent of the time. How many readers already knew this? What is the consequence? Nobody seems to know. But what we can say is that $\sin^2 x + \cos^2 x = 1$ is neither verifiable nor a justifiable assumption.

We, as scientists and mathematicians, need to look for intrinsic attributes of the science and mathematics and how these carry over to the code, perhaps seeking the simplest structures and algorithms (not necessarily the shortest or fastest). Let me term *internal quality* those elements of programs that can be measured or attributed to the abstract program tree of the program. Hatton observes that poor internal quality and dependence on weak linguistic features almost guarantee poor overall quality [7]:

> ... 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

[4] `http://www.cs.berkeley.edu/ wkahan/`

[5] "... note, though, that integer division by zero is either ignored or generates SIGILL on PowerPC systems, and integer overflow is ignored on all SPARC, Intel, and PowerPC systems. On SPARC and Intel systems, special instructions can cause the delivery of a SIGFPE signal of type FPE_INTOVF, but Sun compilers do not generate these instructions." See `http://docs.sun.com`

becomes brittle and the costs of building in [internal] quality, together with programmer resistance, rise considerably.

The following measures proposed by [7] are measures of internal quality: All the following numbers should be zero:

1. Number of statically-detected faults.
2. Number of transgression of group programming standards.
3. Number of uses of nonstandard features or vendor-dependent linguistic features with respect to the programming language standard.
4. Number of uses of features outside a *validated, safe subset.* (ANSI Fortran, ISO/IEC 9899 for C, New standard for C++).
5. Number of uses outside *standard, validated* libraries such as the `C` standard, `LAPack`.

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`[6] and `machar`.

There are possible measures of internal quality about programs. We need only turn to the compiler literature. The literature develops good mathematical models of structure. The hope of measuring internal quality *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	data flow structure
data structure	parallel structure
optimization structure	semantic structure

These concepts come directly from what we know about compiling. Such information never has, as far as I know, been applied to the concept of internal quality. They have been used to derive metrics [4]. Some compiler 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.

Summary. This section has proposed working definitions of validation and verification. I have tried to indicate why one would want such definitions but there are many others. I have tried to ascribe to the *lingua franca* — mathematics — what I consider worthwhile distinctions. The use of a category-theoretic framework seems natural to me since it gets us out of the terminology dispute.

I also introduced the distinction of intrinsic versus internal quality and indicated various measures we might use.

8. A Homework Assignment

Rather than try to provide a summary of summaries, I have instead decided to give you a homework assignment. The setting is the 1940s.

The Lockheed P38 *Lighting* was designed by "Kelly" Johnson at the *Skunk Works*. During development and even into early use, the tails twisted off when the P38 flew faster than about Mach .60 . Johnson had suspected from the very beginning that compressibility effects in the trans-sonic region might cause problems. But first, the team had to battle critics who thought it was the *Lighting*'s unique design. Compressibility finally won out as the culprit. The development team eventually overcame the problem, primarily using wind tunnel tests as 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

[6] Strangely enough, only one reference to `paranoia` exists in the literature. We are working to change that.

to minimize the number of wind tunnel tests (Modern wind tunnels may cost several millions of dollars per use).

Acknowledgment

First, my thanks go to George Cybenko who thought he saw something worthwhile in the first draft. I would like to acknowledge the many fine suggestions from the eight referees. This paper could not have been written without their help and suggestions. Several referees clearly did not agree with me, but were generous with their time and insights. My thanks to the editors, Molly Davis and Dennis Taylor, who tried valiantly to straighten out my English and the paper's organization.

References

[1] Frederick P. Brooks, Jr. No silver bullet. In H.-J. Kugler, editor, *Information Processing 1986, the Proceedings of the IFIP 10th World Congress*, pages 1069–76, 1986.

[2] W. Edwards Deming. *Out of the Crisis*. MIT Press, 1986.

[3] J. Demmel. Lapack and ieee arithmetic. personal communications, Oct 14 1998.

[4] N. E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, 2 edition, 1997.

[5] J. A. Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, 2:159–191, 1991.

[6] John Gustafson. Computational verifiability and feasibility of the asci program. *IEEE Computational Science & Engineering*, Jan.-Mar. 1998.

[7] Les Hatton. *Safer C: Developing Software for High-Integrity and Safety Critical Systems*. McGraw-Hill, 1995.

[8] Les Hatton. The T experiments: Errors in scientific software. *IEEE Computational Science and Engineering*, pages 27–38, Apr-Jun 1997.

[9] Alex R. Larzelere II. Creating simulation capabilities. *IEEE Computational Science & Engineering*, Jan.-Mar. 1998.

[10] Peter G. Neumann. *Computer Related Risks*. Addison-Wesley, 1995.

[11] Charles Perrow. *Normal Accidents*. Basic Books, 1984.

[12] D. E. Stevenson. What is computational knowledge and how do we acquire it? submitted to *Synthese*.

[13] D. E. Stevenson. Science, computational science, and computer science: At a crossroads. *Comm. ACM*, 37(12):85–96, 1994.

[14] D. E. Stevenson. A foundations of validation: The michelson-morley experiment. In *The Proceedings of the European Simulation Multiconference 1999, Warsaw, Poland, June 1-4, 1999*, pages 269–275, 1999.

[15] Frederick Suppe. Introduction and afterword. In Frederick Suppe, editor, *The Structure of Scientific Theories: The Search for Philosophic Understanding of Scientific Theories*, pages 3–244, 617–730, Urbana, IL, 1977. University of Illinois Press.

[16] Gregory V. Wilson. What should computer scientists teach to physical scientists and engineers. *IEEE Computational Science & Engineering*, pages 46–62, Summer 1996. Responses included.

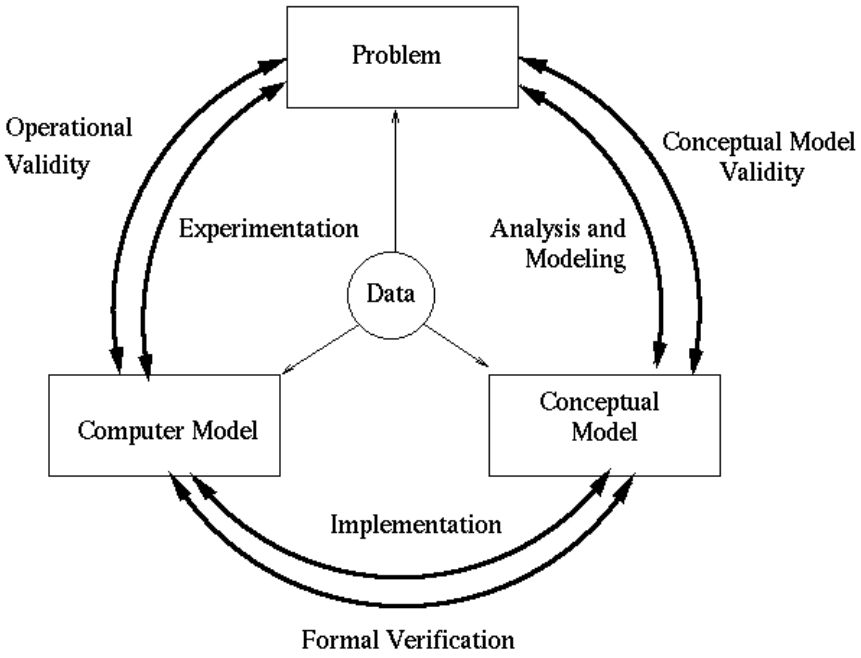

FIGURE 1. Sargent's Circle

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.

FIGURE 2. Deming's 14 Points

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

FIGURE 3. Quality In Software Engineering

Justification Attributes	**Software Attributes**	
Accuracy.	Correctness	Cost
Ease of Use	Efficiency	Portability
Maintainability.	Programmability	Reliability.
Validatibility		Verifiability

FIGURE 4. Major Aspects to Consider: Justification, Software Attributes