Validation and Verification: A CSE Perspective

D. E. "Steve" Stevenson

Department of Computer Science

Clemson University

steve@cs.clemson.edu

January 21, 1998

Abstract

Scientific simulations are quite different from "normal" computer programs. The scientist cannot negotiate with Nature but must describe Nature in mathematical terms as best possible.

In Jurassic times, the scientist then coded and controlled the model. How did they know the model was correct? In the new scheme, many disciplines develop the simulation. Now, correctness is even stickier because the science is buried.

Can we even say what *correct* means?

I argue we're a long time from Francis Bacon, but we've not come very far. It will only come together if science, mathematics, and computer science get together.

Contents

Fair Warning

I'm sort of a philosopher and it is my duty to

- Be the burr under everyone's saddle.
- Find problems with any proposed solution.

Is there a problem?

Stories

- Arithmetic is Unimportant
- Flapping your wings ....
- Endeavour can't get it right
- Repulsive moon
- Marinaer I lost
- Get 'Em, Longhorns
- Armageddon, anyone?
- Scuds over Dhahran
- Four views of rocks
- BoNY Boner
- Philantropic computers
- Harem Scarem
- Damn the torpedos
- *Tempus Fugit*

Arithmetic Is Unimportant

The emphasis in scientific computing over the last twenty years has been on taking advantage of advanced architectures for the sake of speed. Consider the Linpack benchmarks. These were standard fare for the "supercomputer" marketing corps for years. Everyone chased Cray, then NEC, then .... But then LAPack came along. Cray, the machine that started it all, could not compete because their arithmetic was not trustworthy enough to run the new algorithms.

Flapping Your Wings

The story is simple: in 1959, wings were falling off the Lockheed Electras. The problem turned out to be relatively simple: the engines were too powerful and set up a vibration mode in the wings that was not damped out. The vibrations literally tore the wings off the plane. The vibration mode was not accounted for in the models and hence untested.

Endeavour Shuttle Almost Doesn't Rescue Intelsat6

While trying to rescue the *Intelsat6* in early 1990, the computer on the *Endeavour*, the shuttle had much difficulty in approaching the satellite. It was discovered later that the computer had computed very small, but not zero, values and the program did not handle the situation correctly.

Apollo 11 Finds Moon's Gravity Repulsive

The *Apollo 11* software had a bug that made gravity repulsive instead of attractive. One gets the image of Issac Newton in Heaven shaking his head, maybe even shedding a tear.

Mariner I Lost

On July 22, 1962, the *Mariner I* mission was destroyed. The software problem is a beauty. The ground based radar system had a bug that was supposed to compensate for varying echo delays but didn't. The one in question was 43 milliseconds. The bug came from the faulty transcription of a handwritten formula.

Gemini V Lands 100 Miles Off Course

Gemini V landed 100 miles off course because some programmer played fast and loose with physics. The correct elapsed distance should be calculated by using the Sun as a fixed reference point and not a point on the Earth. The programmer instead used elapsed time thinking the reference point on Earth returns every 24 hours. [The ratio of 24 hours/sidereal time is 1.00273790935.]

The time difference (sidereal - 24 hours) in the two calculations put the *Gemini V* off by 100 miles.

Get 'Em, Longhorns

On October 15, 1989, thousands of fans streamed towards Dallas-Fort Worth Airport (DFW) for the Oklahoma-Texas football game. At 9:45 a. m., the antiquated computers at DFW approach control shut down because an operator had entered a routine command but from an unauthorized terminal. The alarm rang and three seconds later the screens in front of the controllers went blank.

Needless to say, but worth saying, all Hell broke loose. Even though the computers came back up relatively quickly, it took hours of human toil to straighten out the mess at airports all over the region.

How Close Was Armageddon?

On October 5, 1960, the North American Defense Command (NORAD) went to 99.9% alert when programmers forgot that the Moon rises and would show on radar.

On June 3, 1980, and again on June 6, 1980, NORAD went to full alert and threatened to launch everything.

The Dhahran Scud

Near the end of the Persian Gulf war, an Iraqi Scud missile evaded American Patriot missiles and hit a U. S. Army barracks in Dhahran, Saudi Arabia, killing 28.

The details are really technical, but the problem is easy to state. The software used two different binary representations of 0.1. Using the different versions of the number in calculations was enough for the launched missiles to completely miss the Scud.

The horrid part of this is that Raytheon knew of the problem and opted for the cheap, human fix. Batteries were to shut down every 8 hours. This battery did not.

Four Programs that Cannot Agree

In [hatton97], Les Hatton reports on his comparisons of various types of software with respect to accuracy and reliability. **This paper should be read by anyone in software management, software engineering, or scientific programming**.

Hatton compared four *commercial grade* software systems for dealing with seismic exploration for oil. The four programs were given the *exact same input data*. The results were shown to geologists with the request that they determine whether or not the company should commit the resources to drill for oil.

Guess what: the same data led to completely different conclusions depending on the program that was used.

BoNY Has $32 Billion Overdraft

In November, 1985, Bank of New York ended up with a $23.6 **billion** overdraft. The cause was an improper integer type. All securities transactions had to be halted. The bank had to get an overnight $24 billion loan from the New York Federal Reserve to cover; it cost BoNY $5 million in interest.

Philanthropic Computer

On the night of February 25, 1988, the Australian Commonwealth Bank doubled all debits and credits on the days transactions. This prompted the manager to make the now famous-in-folk-lore comment, “The effects of software errors are limited only by the imagination...”

He Had No One But Himself To Blame

There is the story of Arab gentleman reported in *Computer Talk*, February 1, 1988. It seems the gentleman was driven to near sexual exhaustion due to a bug in his harem organization program.

Damn the Torpedo ...

Nancy Leveson has led the way in alerting people to safety issues. Here are two:

1. A torpedo was designed to self-destruct if it turned 180°. Unfortunately for the test ship the torpedo stuck in the tube and the captain turned the ship around for port....

2. Another torpedo story. This is the case of the pacifistic torpedo. When launched it immediately dove for the bottom and just laid there.

High Pay, Long Waits....

Finally, my favorite. Nancy Leveson gave a talk at Clemson. She told the story of an aircraft manufacturer who had installed lots of computers that would make pilots a thing of the past. The engineers touted that the computers would "fail once in 10^{16} seconds." (That's once in 3,170,979,198.4 *years* — about 3.2 billion years.)

On the maiden voyage of the prototype, but with a pilot in the left seat, just in case, the plane was making its final approach when — yup — the computers malfunctioned. The pilot looked at his watch, shook his wrist, and said "My, my, my. ten to the sixteenth seconds, already. How time flies."

So much for six sigma engineering.

It seems safe to say that results are mixed at best

There are no consistent causes that run through these examples but all parties from management to computer operator are implicated.

But it looks like the gene pool needs some chlorine.

Modeling

Back to Square 1

The whole enterprise exists to gain insights into the physics, design, development, deployment, storage and maintenance of nuclear weapons.

You can't go around starting wars just to get data to validate the models.

Modeling is about ...

$$\begin{array}{rl} & \text{Putative System Definitions} \\ + & \text{Putative System Behaviors} \\ \hline & \text{Predictions of System Behaviors} \end{array}$$

But, most of all, it's about asking the right questions!

Right Questions Come from Intuition

Vladamir Horawitz was convinced one note in the thousands in von Beethoven's *Apassionata* was wrong. Going to the original, he found he was right. There had been a transcription error that folks dutifully copied over the years. No one ever checked to see if the score was correct.

There is a difference between *reliable* copies and *correct* copies.

Concordat cum originali asservato.

So What's the Problem?

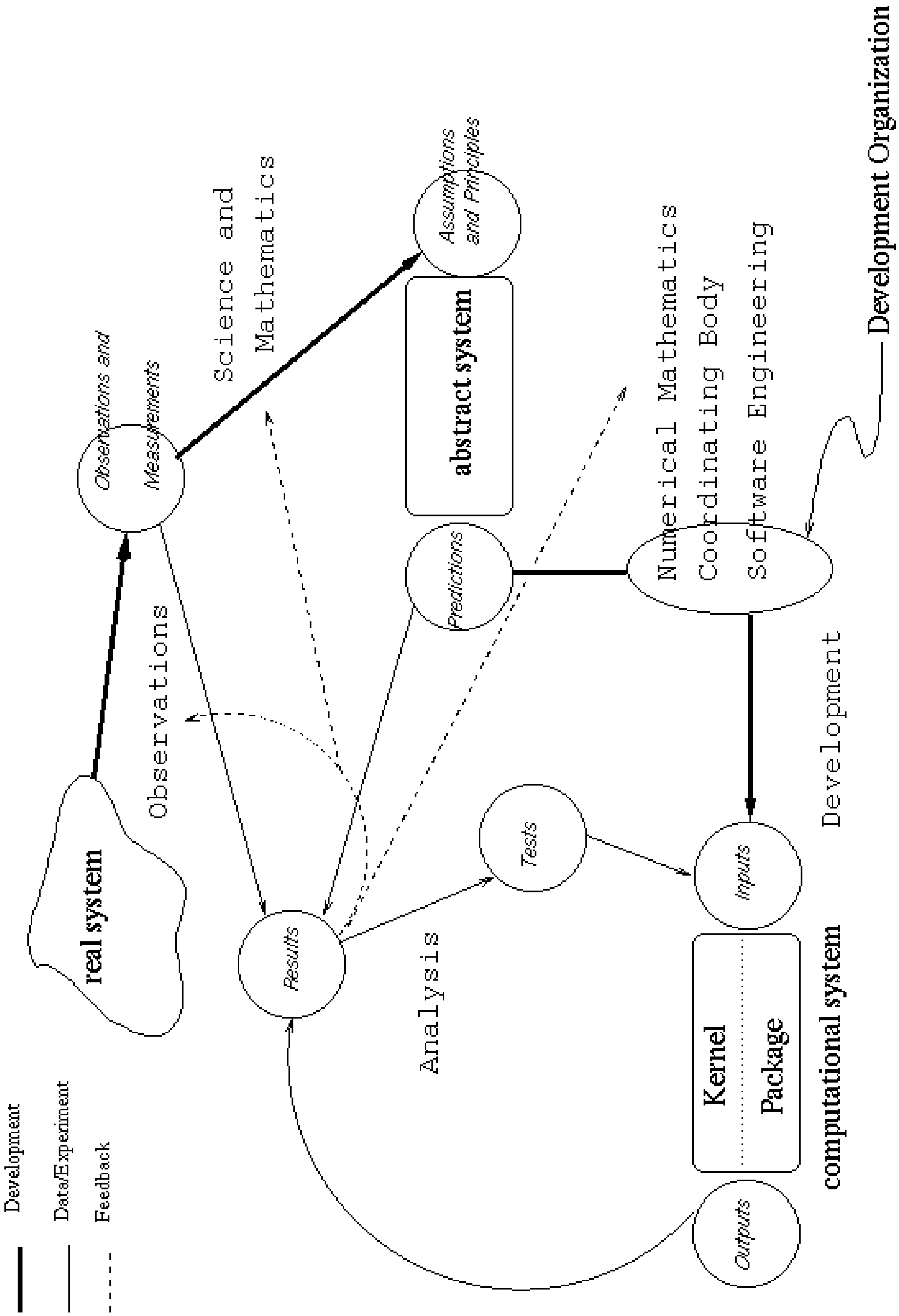
- We're not sure what we've observed.
- We're not sure we were looking at the right things.
- We've played all kinds of math games with the model.
- The programs are huge, complex, and on hardware we have little experience on.
- We're not even sure the boxes work right

And if that doesn't scare you, **we're not even sure we're asking the right questions.**

Finally, ...

Computer models don't produce what we need.

The *linqua franca* of science is *function*; but not *object*, and not numbers on piles of papers.



Development
Data/Experiment
Feedback
real system
Observations
Observations and Measurements
Science and Mathematics
Assumptions and Principles
abstract system
Predictions
Numerical Mathematics
Coordinating Body
Software Engineering
Development Organization
Development
Results
Tests
Inputs
Analysis
Kernel
Package
computational system
Outputs

Meaningfulness and meaningful results.

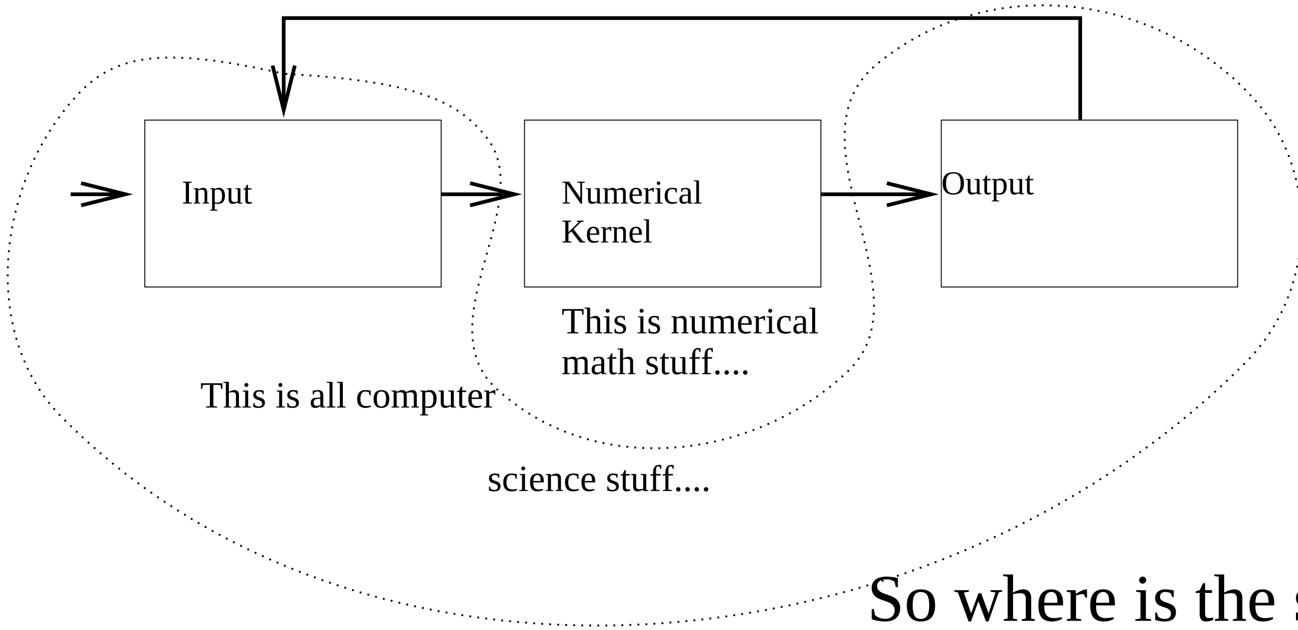
- There is the *physical problem as Mother Nature* sees it. This is the problem that the observer sees and hence the problem the statistician sees. But things are not so simple since there are observational errors and all the mathematical errors described below.

- There is the *physical problem as the scientist* sees it. This runs through the following gauntlet: The physical problem is formulated in pure mathematics (meaning exact everything).

- The pure mathematical problem is recast into a numerical analytic setting.

- The numerical analytic setting is recast into the numerical methods setting.

- Finally, the numerical methods are recast into machine code and run on a concrete machine. Now, what is the relationship between the observations and the computed results on a specific machine? The crux of the matter: each of the *investigators* introduces *errors* whenever a previous formulation is rewritten.



So where is the science?

Quality Explained

The purpose of
computing is insight,
not numbers.

R. H. Hamming (1915 – Jan 8, 1998)

What Does It Mean to Validate or Verify?

The problem is that there are three traditions: science, mathematics, and computer science.

Scientific knowledge requires replication and inductive warrants through experiments.

Mathematical knowledge is deductive and therefore demands deductive proof.

Computer science is constructive and requires a procedure with demonstrable properties

The Clemson CSE Program

proposes that modeling proceed by the following principles:

1. *Physical Exactness.* We strive to identify non-physical (mathematically convenient) assumptions and eliminate them.

2. *Computability.* We must identify non-computable relationships. No mathematical relationship is exact unless it follows directly from the development of an exact model and is computable. In this sense, most mathematical relationships turn out to be *approximate.*

3. *Bounded Errors.* No formulation is acceptable without *a priori* error estimates or *a posteriori* error results.

So I ask again, Do we have a problem?

We all learned that $sin^2(x) + cos^2(x) \equiv 1$

Using SPARC and gcc:

$$
\begin{array}{l}
count = 0; \\
for(x = 0; x < \pi/4; x+ = 2^{-20}) \\
\quad if(sin^2(x) + cos^2(x) > 1) \\
\qquad count + +;
\end{array}
$$

What is $count$'s value?

Quality assurance

is the process by which we convince our employers that we are part of the solution and not part of the problem.

The Face of Quality

Quality — The Dream

So What Are the Dimensions?

1. People;
2. Science;
3. Mathematics, classical and numerical;
4. Hardware architecture;
5. Software virtual machine; and
6. Software implementation.

Hardware, Software, Wetware

Five Fallacies of Ineffective Development

1. People are freely substitutable via the Pregnant Women Theorem.

2. Training is more important than education.

3. Short term success guarantees long term success.

4. There's only one optimization variable: $$$.

5. If it compiles, it works.

The reality

There are four legs

- People
- Time
- Resources other than money
- Budget

Management must choose exactly two!

Deming's Laws of Manufacturing

1. Constancy of purpose.
2. Adopt the new thinking.
3. Cease dependence on mass inspection.
4. Cease doing business on price tag alone.
5. Continual improvement of processes.
6. Institute training on the job.
7. Institute leadership.
8. Drive out fear.
9. Break down barriers between departments.
10. Eliminate slogans, exhortations and targets.
11. Eliminate numerical quotas.
12. Allow pride in workmanship.
13. Institute a program of self-improvement.
14. Do it.

Deming's 14 Points Revisited

2. And just what is the new thinking?

3. And replace it with what?

5. What are we doing now?

9. And replace them with teams.

And all the rest seem to apply *verbatim*.

There is no dearth of advice

1. ISO 9xxx
2. ANSI/ASQC Q 90-4
3. MIL-Q 9858A
4. MIL-I 452058
5. NATO AQAP 1-3
6. Computer Maturity Model (CMU)
7. TQM
8. ISO/IEC SC 65A (65A Secretariat 122)
9. UK Def Stan 00-55
10. IEEE
11. ACM

What are some quality traits in simulations?

Here's the international thinking (ISO9126).

- Functionality.
- Reliability.
- Usability.
- Efficiency.
- Maintainability.
- Portability.

Damn! I guess *correctness* isn't important.
(Reliability $\neq$ Correctness)

Aren't There Any Silver Bullets?

"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." Fred Brooks[brooks86]

"To expect process models of themselves to improve the quality of software or the cost-effectiveness and responsiveness of the software development and evolution process, permitting, for example, total mechanisation [sic] 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.[lehman93].

Even Worse....

- Academic computer science is flawed since it only plays with toys [wilson96].

- Applied computer development can't develop the methods since they're always too busy to do so.

- Tools are mainly *ad hoc* toys (aptly named *shelfware*) and are not systemic.

*PSE*s stand a chance of being silver bullets.

Intrinsic Quality — KISS

“Good internal structure implies good external quality” [fenton97]

“Poor internal quality and dependence on weak linguistic features implies poor quality” [hatton95]

The problem is, software quality is like mathematical elegance: I can’t define it, but I know it when I see it.

“The essence of elegance is simplicity.”

Quality By Fiat ... The European Solution

The Europeans have worried about this standards/quality thing for a long time. The UK Defence Department [def0055] put out a bunch of rules about choosing 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.

For The Current Stable of Languages

1. Okay

2. No compiler I know of without an *ad hoc* tool.

3. Really two things:

   (a) Yeah, right

   (b) None that I know of.

4. Okay, but no silver bullet.

5. Ditto number 4.

Parting Shot....

"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" [hatton95]

Speed without correctness is worse than useless. It is the opiate of computer jocks.

Some ideas on intrinsic quality measurement

1. Zero statically-detectable faults.
2. Zero transgression of internal programming standards.
3. Zero dependence on relevant unspecified features or undefined linguistic features with respect to the programming language standard.
4. Complete reliance on a *validated, safe subset*. (ISO/IEC 9899 for C, ? for C++, NIST)
5. Limitations on component complexity.
6. Validation of standard libraries, including things like LAPack and `C/C++`. [plauger93]
7. Validation of machine via, say, **`paranoia`**.
8. Limitations on *cognitive* complexity.

If it can't be enforced by compilation tools, it won't happen

Such non-compiler based tools are called also *shelfware*.

"The shelfware syndrome is singular, if not unique, to the software world."

I'm calling for a moratorium of new languages until we get the quality paradigm down.

... all the intrinsic 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 intrinsic quality, together with programmer resistance, rise considerably.

[hatton95]

Studies have shown ....

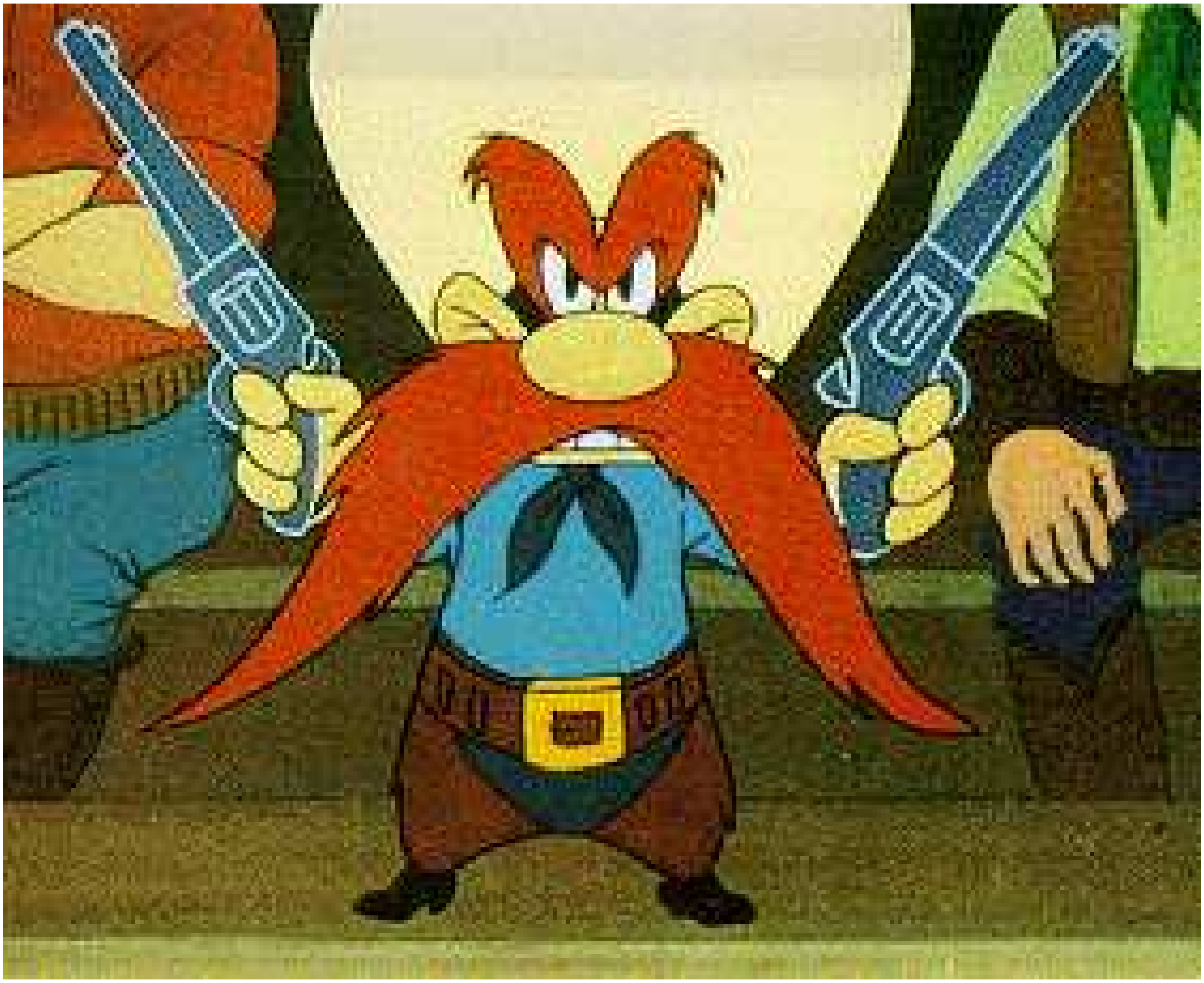
For every hour of maintenance on components:

- 17% bug fixes.
- 18% things that should have been done differently.
- 65% things just flat not done.

The number of bugs in code is approximately

$$c \log(\text{lines of code}).$$

Now What?



Back Off, Varmits

We Have Discovered Some Things

1. Good practice matters. Unfortunately, academics don't teach it and organizations can't enforce it.

2. Software engineering practice does not address computational science and engineering.

3. Speed must become subordinate to correctness.

4. Detailed specifications, quality assurance procedures, and formal testing is not enough.

5. Double precision does not solve problems of unstable codes or ill-conditioned problems.

6. Quality from using less-well-defined algorithms is several times worse than using formal mathematical definitions.

7. Paradigm shifts in language or formal methods does not appear to automatically solve the problem.

8. Safe subsets for languages are very important.

9. Comprehensive and objective testing, formal methods, and multiple versions *may* be helpful.

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.

11. Documentation does not guarantee a high-quality system.

Review

1. What is good practice? Is good practice decidable? Enforceable? What good practice is prevented by languages.

2. What special good practice stems from science and simulations?

3. What practices for "speed" should be abolished? What should always be in force?

4. Where does proof end and test begin? Why is proof so hard?

5. What practices enhance numerical stability? What should programmers "tell" programs?

6. Why don't we understand translation of mathematics into program?

7. What paradigm shifts are needed to solve out problems?

8. How do we identify and enforce these subsets?

9. Why is the "programming" literature such a mess?

10. What are the undesirable language and design practices? Are these knowable *à priori*? What is decidable (computable)?

11. We're probably documenting the wrong stuff. The *derivation* is the thing.

Short Term Plan

1. We have certified parsers, so strategies and programming standards can be enforced.

2. We've already started on a verification suite for the intrinsics.

3. We can get the standards documents, so we can start the "safe subset" exploration.

Intermediate Term

You Can Run But
You Can't Hide

The Feynman Society

The Society for
Scientific Validation
and Verification

Our Motto: Providing Chlorine for the Gene Pool

Long Term

Measures of Internal Quality

- control flow
- data flow
- data structure
- parallel structure
- optimization structure
- intrinsic complexity
- semantic structure

Compiler Writers Have A Lot Of Theory Never Used

But So Far, We've Not Gotten at the Root Cause

The root cause is that we don't understand how *cognitive* complexity inhibits our learning and capability to compute.

Clemson's CS and Psychology departments have started down this path.

In Conclusion

I am here to solicit any and all criticisms, complaints, comments, ideas, ... I'll be here all day and you can get me by email

steve@cs.clemson.edu