Bits don't have error bars

Russ Abbott
Department of Computer Science
California State University
Los Angeles, Ca
01-310-621-3805
Russ.Abbott@GMail.com

ABSTRACT
How engineering enabled abstraction—in computer science.

Categories and Subject Descriptors
K.0 [**Computing Milieux**]: Philosophy of engineering and computer science

General Terms (none)

Keywords
Abstraction, emergence, level of abstraction, philosophy of computer science, philosophy of engineering, philosophy of science, reductionism.

1. TURNING DREAMS INTO REALITY
In 1944, Erwin Schrödinger [5] pondered the nature of life.

> [L]iving matter, while not eluding the 'laws of physics' … is likely to involve 'other laws,' [which] will form just as integral a part of [its] science.

But if biology is not just physics what else is there? Schrödinger's question is a special case of the more general question: can there be independent higher level laws of nature if everything is reducible to the fundamental laws of physics? The computer science notion of *level of abstraction* explains why there can—illustrating how computational thinking can resolve one of philosophy's most vexing problems. (Section 3 explains the essence of the solution.) This paper explores why the solution came from computer science rather than from engineering.

Scientists analyze what exists. Those of us in computer science and engineering build new things. A poetic—if overused—way to put this is to say that we turn our dreams into reality. We transform ideas—which exist only as subjective experience—into phenomena of the material world.

In raising the issue of the relationship between mind and the physical world I am not claiming to explain consciousness. But I am taking mind as a given and claiming that the relationship between ideas and physical reality is at the heart of the difference between engineering and computer science. Both disciplines begin with ideas. Computer science turns ideas into a symbolic reality; engineering turns ideas into a material reality. Although perhaps unremarkably true, the consequences are far-reaching.

2. ENGINERING & COMPUTER SCIENCE
Intellectual leverage: levels of abstraction vs. mathematical modeling and functional decomposition. Computer science gains intellectual leverage by building levels of abstraction, the implementation of new types and operations in terms of existing types and operations. A level of abstraction is always operationally reducible to an underlying domain, but it is characterized independently—what Searle calls causal but not ontological reducibility.[1] Levels of abstraction allow computer scientists to create (symbolic but real) worlds that obey laws that are independent of their underlying platforms.

Engineering gains intellectual leverage through mathematical modeling and functional decomposition. Both are taken only to approximate an underlying reality. Neither yields ontologically independent entities. The National Academy of Engineering points [3] out that "engineering systems often fail … because of [unanticipated low-level interactions (such as acoustic resonance) among well designed components] that could not be identified in isolation from the operation of the full systems."

Symbolic floor vs. no floor. By harnessing electrical signals as bits engineering enabled computer science. Bits are both real (material) and symbolic (no error bars). Computer science builds levels of abstraction on a base of bits—and relies on engineering when faced with issues beyond the bit—such as performance.

But bits prevent computer science from working with the full richness of reality. Every software model has a fixed bottom level—making it impossible to explore phenomena that require dynamically varying lower levels. A good example is a biological arms race. Imagine a plant growing bark to protect itself from an insect. The insect may then develop a way to bore through bark. The plant may develop a toxin—for which the insect develops an anti-toxin. There are no software models in which evolutionary creativity of this sort occurs. The problem is that this sort of creativity is built upon varying levels of physics and biochemistry.

[1] In *Mind* [4] Searle claims that this nicely drawn distinction explains subjective experience. He doesn't say how.

Unless all of biochemistry is built in from the start—i.e., unless the model's bottom level is the lowest level with any possible relevance, which is far beyond our computational means—we cannot build models with this degree of complexity and creativity.

Engineering (like science) does not have this problem. It is both cursed and blessed by its attachment to physicality. It is cursed because one can never be sure of the ground on which one stands—raw nature does not provide a stable base. It is blessed because one can dig as deeply as needed for any particular problem.

Thought externalization: software vs. design documents and material objects. The goal of both the computer scientist and the engineer is to turn ideas into phenomena in the world. (The goal of the scientist is to turn physical phenomena into ideas.) The first step is to externalize thought. By thought externalization I mean not just to act on a thought but to represent the thought outside the realm of subjective experience in a form that allows it to be examined and explored. Computer science and engineering externalize thought in different ways.

An enduring goal [2] of computer science is to develop languages that have two important properties. (a) The language may be used to externalize thought. (b) Expressions in the language can act in the material world—that is, the language is executable. This is remarkably different from anything that has come before. Human beings have always used language to externalize thought. But to have an effect in the world, written expression has always depended on human beings. Words on paper mean nothing unless someone reads them; software acts without human intervention.

Engineers externalize thought either by creating designs or by building objects. Designs—even computer-based designs—have the same limitation as other traditional languages. They require a person to understand them. The engineer who builds an object has indeed turned thought into a material phenomenon. But the thought is gone; all that's left is the physical realization. To recover the thought requires reverse engineering. In computer science externalized thought is executable; in engineering it isn't.

3. THE REDUCTIONIST BLIND SPOT

Here's how levels of abstraction resolve the question posed at the beginning. (See [1] and [2].) In the Game of Life, the rules are analogous to the fundamental laws of physics: they determine everything that happens on a Game of Life grid. Nevertheless there can be higher level laws that are not derivable from them.

Certain Game of Life configurations produce patterns. One can implement arbitrary Turing machines by arranging Game of Life patterns. Computability theory applies to these Turing machines. Thus while not eluding the Game of Life rules, new laws (i.e., computability theory) that are independent of the Game of Life rules apply at the Turing machine level—just as Schrödinger said.

Furthermore, conclusions about Turing machines apply to Game of Life grid cells. Because the halting problem is undecidable, it is undecidable whether an arbitrary Game of Life configuration will reach a stable state. So, not only are there independent higher level laws, those laws have implications for the fundamental elements of the Game of Life. I call this *downward entailment,* a scientifically acceptable alternative to downward causation.

Like all levels of abstraction, Game of Life patterns are epiphenomenal—they have no causal power. It is the elementary Game of Life rules that turn the causal crank. Why not reduce away these epiphenomena? Reducing away a level of abstraction results in a *reductionist blind spot*. No set of equations over the domain of Game of Life grid cells can describe the computations performed by a Game of Life Turing machine—unless the equations themselves model a Turing machine. The laws that characterize regularities at higher levels of abstraction are impossible to express when the abstractions are reduced away.

This perspective applies beyond computer models. Nature—a "blind programmer"—builds levels of abstraction, a phenomenon sometimes referred to as *emergence*. This may be encapsulated as the *principle of emergence*: extant levels of abstraction—naturally occurring or man-made, static (at equilibrium) or dynamic (far from equilibrium)—are those whose implementations have materialized and whose environments support their persistence.

4. SUMMARY

Engineering has no stable base. Engineers must always be concerned about the possibility of lower level physical effects—such as O-rings not functioning as sealants if the temperature is too low.[2] Consequently, engineering has not been in an intellectual position to develop the notion of level-of-abstraction. But with its gift of the bit to computer science, engineering created a world that is both real and symbolic. Computer science developed the level-of-abstraction as a way to gain intellectual leverage over that world—and then applied that concept to solve a long-standing problem in the philosophy of science.

5. REFERENCES

[1] Abbott, Russ, "Emergence explained," *Complexity*, Sep/Oct, 2006, (12, 1) 13-26.

[2] Abbott, Russ, "If a tree casts a shadow is it telling the time?" *Journal of Unconventional Computation*, to appear.

[3] Commission on Engineering and Technical Systems, National Academy of Engineering, *Design in the New Millennium*, National Academy Press, 2000.

[4] Searle, John, *Mind: a brief introduction*, Oxford University Press, 2004.

[5] Schrödinger, Erwin, *What is Life?*, Cambridge University Press, 1944.

[2] Every implementation of a level of abstraction has feasibility conditions. Understanding safety margins as means to ensure that feasibility conditions are met would encourage engineering to operate in terms of levels of abstraction. Engineering designs often have safety factors where software has assertions.