# Bits don’t have error bars: upward conceptualization and downward approximation

**Russ Abbott**

Department of Computer Science, California State University, Los Angeles, California
Russ.Abbott@GMail.com

**Abstract.** How engineering enabled abstraction in computer science, which helped solve a problem in philosophy.

## *1. Introduction*

In 1944, Erwin Schrödinger [7] pondered the science 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? Nearly three decades later Philip Anderson [4] extended Schrödinger’s thought when he argued against what he called the *constructionist hypothesis,* namely, that

> “[the] ability to reduce everything to simple fundamental laws … implies the ability to start from those laws and reconstruct the universe.”

Anderson explained his opposition to the constructionist hypothesis as follows.

> At each level of complexity entirely new properties appear. … [O]ne may array the sciences roughly linearly in [a] hierarchy [in which] the elementary entities of [the science at level n+1] obey the laws of [the science at level n]: elementary particle physics, solid state (or many body) physics, chemistry, molecular biology, cell biology, …, psychology, social sciences. But this hierarchy does not imply that science [n+1] is ‘just applied [science n].’ At each [level] entirely new laws, concepts, and generalization are necessary. … Psychology is not applied biology, nor is biology applied chemistry. … The whole becomes not only more than but very different from the sum of its parts.

Although Anderson agreed with Schrödinger that nothing eludes the laws of physics, he thought that the position he was taking—that the whole is not only more than but very different from the sum of its parts—was radical enough that he should include an explicit reaffirmation of his adherence to reductionism.

[The] workings of all the animate and inanimate matter of which we have any detailed knowledge are all … controlled by the same set of fundamental laws [of physics]. … [W]e must all start with reductionism, which I fully accept.

The questions raised by Schrödinger and Anderson can be put as follows. Are there are independent higher level laws of nature, and if so, is the existence of such laws consistent with reductionism?

The computer science notion of *level of abstraction* explains why there can be independent higher level laws and why those higher level laws do not violate reductionism. Section 7 explains the essence of the argument, which I discuss in greater detail in [1], [2], and [3]. In providing this explanation, computer science illustrates how *computational thinking* [9] applies beyond computer science—and in this case is able to resolve a problem in philosophy. This paper explores the complementary roles that computer science and engineering play in the development of the necessary concepts.

## 2. *Turning dreams into reality*

What has been is what will be, and what has been done is what will be done; there is nothing new under the sun. Is there a thing of which it is said, "See, this is new"?
– Ecclesiastes 1:9-10

Although Ecclesiastes says “No,” engineers and computer scientists say “Yes, there are things that are new under the sun—and we create them.” Where do these new things come from? They start as ideas in our minds. A poetic—if overused—way to put this is that as engineers and computer scientists we turn our dreams into reality. Trite though this phrase may be, I mean to take seriously the relationship between ideas and material reality. Engineers and computer scientists transform ideas—which exist only as subjective experience—into phenomena of the material world.

Although the preceding identifies a bridge between subjective experience and the physical world, this paper is not a theory of mind. I am not claiming to explain about how subjective experience comes into being. Nor am I claiming to explain how we map subjective experience to anything outside the mind.

Nonetheless, I do plan to talk about the relationship between ideas and material reality—and especially about how that relationship is at the heart of the way in which we as computer scientists and engineers do our work. In this article, I will take as given that we each (a) have the subjective experience referred to as having an idea; (b) are aware of ourselves as having ideas; and (c) understand in more or less the same way what it means to say that one has an idea. In other words, I am assuming that even though we can’t explain how it happens, we each experience subjective thought in similar enough ways that our common experience can serve as a starting point for further discussion.

With the preceding in mind, it seems fair to say that to a first approximation[1] engineers and computer scientists turn ideas into material reality. It seems equally fair to say that turning ideas into material reality is the inverse of the work performed by scientists. Science is an attempt to understand nature, i.e., to reverse engineer it. Scientists turn material reality into ideas.[2]

Although engineering and computer science are similar in that they both transform ideas into material reality, there is a difference in the kinds of reality we create. The products that computer scientists create manipulate (physically implemented) symbols.[3] The products that engineers create act in the physical world. Although perhaps unremarkably true, the consequences are far-reaching.

## *3. Thought externalization: engineering is like sculpture; computer science is like music*

If I could say it in words there would be no reason to paint. – Edward Hopper

The first step in turning an idea into reality is to externalize the idea. By externalizing an idea I'm referring to the conversion of the thought from something completely subjective to an external representation—a representation outside the mind in a form that allows the thought to be examined, explored, and communicated. The pervasive example is natural language. Most disciplines use natural language to externalize thought. Other forms of thought externalization include diagrams and equations.

For the most part, externalized thought is intended for human consumption. Expressions in natural language as well as equations and diagrams are meaningless except to other human beings. An externalized thought has the same intentional property as thought itself. Like thought itself, most forms of externalized thought are about something, and for an externalized thought to be about something requires that it be re-internalized, i.e., understood and converted into (intentional) thought.

There are a number of interesting exceptions to this perspective. The first is when the externalized thought is the thing itself. That occurs in the case illustrated by Edward Hopper's remark. Hopper refers to his paintings as "saying" some-

[1] The creative process tends to be iterative: one produces something, examines it, modifies it, examines the result, etc.

[2] Humanists turn reality into dreams. – Debora Shuger (personal communication).

Mathematicians turn coffee into theorems – Paul Erdos (widely quoted but source unknown).

[3] In this article I won't be discussing hybrid systems—physical systems with embedded software, i.e., software that controls physical devices other than input and output devices. Admittedly this is a major area of both engineering and computer science, and it deserves attention. But I'd like to focus on the differences between engineering and computer science. Hybrid systems make it much more difficult to distinguish between the two disciplines.

thing, even though what they say can't be conveyed in words. Presumably his paintings act directly in the world. They don't refer to something else; they are the things themselves. They function in the world by the effect they have on viewers. But even though their effect is on human beings their function is not primarily to refer to something besides themselves. They may evoke thoughts and feelings in their viewers, but they are not intended primarily as messages. They are not about their content. They are the thing itself.

It seems to me that this is the case with most physically externalized thought—including sculpture as well as paintings and most other forms of artistic expression. Engineered objects are in this same category. Engineers turn ideas into physical reality. The dam, bridge, building, rocket, etc. are the engineer's thought externalized as physical object. In discussing externalized thought as "the object itself" I am not saying that these objects are independent of human intention. A bridge is not a bridge unless we use it to cross a divide any more than sculpture is a sculpture unless it is appreciated as such by an observer. Both function in the world with respect to the ways humans interact with and make us of them. But neither is intentional in the sense in which a word or idea is intentional. They don't refer beyond themselves.[4]

Thought externalization in computer science is different. Computer scientists externalize thought as software. Software has the important property that it is *both* intentional and the thing itself. It refers in the same way that expressions in natural language refer: one can read it and understand it as having meaning. Software is also the thing itself—almost. With the help of a computer software acts in the world without need for human understanding.[5] To the best of my knowledge, the only other discipline that is able to externalize thought in a form that both (a) is intentional and (b) can be actualized without further human participation is music. Even before computers, player pianos were able to convert the symbolic expression of musical thought into music.

Music and software can be "played" directly. They aren't quite the things themselves since they must still be played. But in both cases there are non-human devices that are capable of converting symbolic expressions into the thing itself. This is the case with software because we have universal Turing machines, devices that can become any computational process. In music we have something similar. Old time player pianos could play any piano composition. Now computers can play virtually any composition.

Engineering is approaching this capability, but it isn't there yet. Even a fully automated computer aided manufacturing capability would not be quite the same. An engineering design must still be transformed into a physical artifact before it

[4] This is a tricky distinction to make since most human interactions may be understood intentionally to some extent. We cross a bridge to get to the other side; our emotional and aesthetic reactions to artistic works evoke thoughts and feelings beyond the work. I don't know how to draw a clean and simple line between the intentional and the functional..

[5] This feature of software as well as the nature of computation is discussed in [2].

can be used. This is typically a three step process: externalize the thought as a design; convert the design into a physical object; use the object. Computer science and music skip the intermediate step. Until we have the engineering equivalent of a universal Turing machine, a device into which one can insert a design and which then becomes the designed object, engineering won't have thought externalization languages capable of direct actualization.

## 4. *Where thought and matter meet*

The term *bit* is used in three overlapping ways.

1. *Bit* can refer to a binary value, i.e., either **true** and **false** as in Boolean logic. A bit in this sense, i.e., as the notion of true or false, is a thought, an idea in our minds just as **true** and **false** are.
2. *Bit* can refer to a mechanism or means for recording such a value. A bit in this sense is part of the material world, typically a unit of computer storage. It is a physical device that (a) is capable of being in either of exactly two states and (b) can be relied on always to be in one of them.
3. *Bit* can to refer to a unit of information as in information theory. I'm not using *bit* in this sense in this paper—although a *bit* in the second sense can be used to represent a *bit* in this third sense.

The bit is a fundamental example of externalized thought. It is both a Boolean value (i.e., a thought) and a physical device (i.e., a part of the material world). Circuits that when run can be understood as performing Boolean operations provide a way to glide gracefully back and forth between *bit* as thought and *bit* as material device. Bits and the physical machinery that operate on them enable us to externalize Boolean operations and values (thoughts) and to manipulate them in the material world. In other words, the bit is where thought and matter meet, an extraordinary achievement.

Engineering is a physical discipline. Physical disciplines involve physical devices and materials, which are never perfect and never the same from one instance to another. To determine how a physical device behaves, one measures it—often multiple times. The resulting sets of data points typically contain ranges of values. That's the nature of both physical devices and measurement. Such data sets have average values and error bars.

The physical devices used to store and manipulate bits have the same properties as any other physical device. Yet computer scientists, the people who use bits, don't see these aspects of physically implemented bits. The machinery built by engineers hides the reality of averages and error bars from those who use bits. Bit manipulation hardware takes measured values, not always the same, and regularizes them, restoring them to one of two states. It is because of this machinery that as far as computer science is concerned bits don't have error bars.

## 5. *The price*

As externalized thought, the bit is the externalization of the forbidden fruit of the tree of knowledge. Like thought itself it both gives us a way of gaining leverage over reality while at the same time separating us from it.

Because every conceptual model implemented in software is built on bits, every software-based conceptual model has a fixed bottom level, a set of primitives. Whatever the bit—or the lowest level in the model—represents, those primitives serve as the floor of the model. It's not possible within the model to decompose those primitives and model how they work. Whatever those primitives are, they are built into the model.[6]

Because software models have a fixed set of primitives, it is impossible to explore phenomena that require dynamically varying lower levels. A good example of a model that needs a dynamically varying lower level 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 richness occurs. To build software models of such phenomena would require either that the model's bottom level be porous enough that entities within the model are able to develop capabilities that sabotage operations on that lowest level or that the lowest level include all potentially relevant phenomena, from quantum phenomena on up. Neither of these options is within our currently available computational means.

Another example of a model that needs a dynamically varying lower level is the gecko, a macro creature, which uses the quantum van der Waals force to climb vertical surfaces [5]. Imagine what would be required to build a computer model of evolution in which that capability evolved. Again, one would need to model physics and chemistry down to the quantum level.

The inability to develop models with dynamically varying lower levels is not limited to software models. It is a problem with any finite conceptual model. We don't seem to be intellectually capable of building models that at the same time (a) have a lowest level and (b) are able to be extended downward below that lowest level.[7] Yet engineering and science are continually extending models downward. How do they do it?

[6] One might object that when bits are used to represent numeric values that are derived from an equation-based model, they don't represent primitive elements. I wouldn't dispute that. But equation-based models are science and engineering models, not computer science models. See the discussion in Section 6.

[7] This poses a nice challenge to computer science: develop modeling mechanisms in which the lowest level of the model can be varied dynamically as needed.

## *6. Upward conceptualization and downward approximation*

Computer science and engineering both require their practitioners to build sophisticated and complex thoughts, which are generally built on previously established results. Using existing ideas to develop new ideas gives workers in both fields enormous intellectual leverage. Yet engineers and computer scientists build intellectual leverage in significantly different ways.

### *Upward conceptualization*

Computer science gains intellectual leverage through what I'll call upward conceptualization, by building levels of abstraction. A level of abstraction is a collection of conceptual types (categories of entities) and operations on entities of those types. Every level of abstraction (except the bit) has two important properties. (a) It is operationally reducible to pre-existing levels of abstraction. (b) It can be characterized independently of its implementation—i.e., the entity types and operations of a level of abstraction can be specified abstractly.

Consider the non-negative integers as specified by Peano's axioms.[8]

1. Zero is a number.
2. If *A* is a number, the successor of *A* is a number.
3. Zero is not the successor of a number.
4. Two numbers of which the successors are equal are themselves equal.
5. (induction axiom.) If a set S of numbers contains zero and also the successor of every number in S, then every number is in S.

These axioms specify the terms *zero, number, successor*, and *equal*—not by defining them but by establishing relationships among them. If implemented in software the result would be a level of abstraction—or in this case the abstract data type *Number*.

Consider the relationship between specification and implementation. A specification defines relationships; an implementation reifies them. Any time a level of abstraction is implemented, it is possible to examine that implementation and see how the specified relationships are realized. But the mechanisms employed in an implementation are not the point; multiple implementations of the same specification using distinct and different mechanisms produce the same level of abstraction. All that matters is that the specification be honored. Searle [8] invented a nice phrase to describe entities and entity types on a level of abstraction. He called them causally reducible but ontologically real.[9]

[8] As given in Wolfram's MathWorld: http://mathworld.wolfram.com/PeanosAxioms.html.

[9] Searle didn't explain why they are ontologically real. Certainly we treat higher level entities implemented in software as real. Furthermore, higher-level entities exhibit and maintain reduced entropy. When implemented physically, higher-level entities also have distinguishable mass characteristics. Physically implemented levels of abstraction persist only under limited feasibility conditions. For example, $H_2O$ implements the level of abstractions solid, liquid, or gas depending on the temperature. (See [1] and [2].)

Higher-level entities are causally reducible since one can always look at their implementation to see how they work. But higher-level entities are ontologically real when their (formal or informal) specification characterizes how they function. Biological entities, for example, have the properties of being alive or dead, and they perform functions such as eating, sleeping, seeing, moving (as entities), and reproducing. These concepts apply only at the biological level, not at the level of chemistry or physics. The reality of higher-level entities and entity types inheres in their specifications, not their implementations.[10, 11]

As described above, the bit is the lowest level of abstraction. It consists of the values *true* and *false* and the operations (*and*, *or*, *not*, etc.) that can be performed on those values. All other levels of abstraction can be implemented in terms of the bit, which is implemented in hardware. The bit is the interface between engineering and computer science. The bit is the bottommost level of abstraction, and the operations that manipulate it, the ones "exposed" by engineering, are its Application Program(ming) Interface (API). The bit level of abstraction is the foundation upon which all of computer science is built.

Levels of abstraction provide enormous intellectual leverage. Each level can be implemented in terms of the properties and behaviors of lower levels without having to worry about how those lower-level properties and behaviors are implemented. This intellectual leverage has allowed computer scientists to create a wide range of new worlds—worlds consisting of entities that obey laws that are independent of the substrate upon which they are built. But as discussed above, the price is that every software level of abstraction has a fixed floor.

### *Downward approximation*

Engineering models also have floors. But because of the differences between the tasks that engineering and computer science undertake and the differences in the ways engineering and computer science build models, the floors of engineering models are porous whereas the floors of computer science models are opaque.

[10] This is not to say that everything that can be described exists—the unicorn for example. But anything (a) that exists, (b) that can be described independently of its implementation, and (c) whose functioning depends on its specification is ontologically real.

[11] On his website (http://www.cscs.umich.edu/~crshalizi/notebooks/emergent-properties.html) C. Shalizi defines *emergence* of a higher level set of variables from a lower level set as the situation in which (a) the higher level variables are a function of the lower level variables and (b) the higher-level variables can be predicted more efficiently (in a computational complexity sense) than the lower-level variables**.** This is similar to my definition if *variables* are taken to refer to entities and *is a function of* is understood to mean *is implemented by*. Shalizi doesn't discuss why prediction might be more efficient if computed in terms of the higher level variables. In my view it is the fact (a) that the higher level variables/entities have properties that are independently specifiable, (b) that the behavior of the higher level variables/entities can be characterized in terms of those properties, and of course (c) that these higher level behaviors are easier to specify than it would be to specify them in terms of low level behaviors. (See [3] for a brief discussion.)

Unlike computer science, engineering builds material objects—objects whose design must match physical reality well enough that they actually work in the material world. Engineering is both cursed and blessed by this attachment to physicality. It is cursed because in modeling physical reality one can never be sure of the ground on which one stands—raw nature does not provide a stable base. We know a lot about the physical world; but we don't know everything. A consequence of this is that all engineering models are approximations: they have error bars. But engineering is also blessed by its attachment to physicality. For any issue one can decide how deeply to dig for useable physical bedrock.

Engineering models are generally expressed as a collection of interlocking equations. For any engineered system an ideal but unreachable goal is to have a set of equations that connects the most primitives elements in the model to the system characteristics.[12] Even if this were theoretically possible, for a system of any size, this is an unreachable goal because there would be so many equations and the equations would be so complex that the model would be useless. Engineering uses two techniques to deal with this problem: raising the level of physical approximation and functional decomposition.

**Physical approximation.** The deeper in terms of physical primitives one must go in a model, the more complex the model. So one chooses a model that provides the necessary precision on as high a physical level as possible. A civil engineering model, for example, may include the load bearing properties of a steel beam rather than the chemical bonds that produce those properties.

**Functional decomposition.** Most engineered systems may be decomposed into component subsystems. If the subsystems can be modeled independently, then the system as a whole can often be modeled as the collection of subsystems along with a model that ties the subsystems together.

These two heuristics work quite well. But they are not infallible. The National Academy of Engineering [6] explains that when engineering systems fail it is often "because of [unanticipated interactions (such as acoustic resonance) among well designed components] that could not be identified in isolation from the operation of the full systems." That is, the problems that arise often occur at a level below the system primitives (approximation fails), and they manifest only when the system components are joined to make the entire system (functional decomposition fails).

But as I said above, engineering is also blessed by its attachment to physicality. If either the decomposition or depth of a model is inadequate, engineers can revise their models to decompose differently or to model more primitive physical phenomena—or both. It's the physical reality that matters. Engineering is grounded by its allegiance to that physicality, and the physical world reciprocates by its lawfulness—approximate thought it may be. All that is required is to find the right set of physical approximations on which to build one's model. This strategy of revis-

[12] This is the physicist's dream too: a single set of equations that define a theory of everything—what Anderson argued was not possible.

ing a model downward works quite well. But just as computer science pays a price for its intellectual approach there is a price to be paid for this strategy as well.

## *7. The reductionist blind spot*

As the previous section discussed, computer science models are constructive. At each level, there is a commitment to the ontological reality of entities at that level. Engineering (and scientific) models are reductive. When phenomena at one level are reduced to phenomena at a lower level, the entities at the higher level are generally discarded in favor of the entities at the lower level. This strategy works because the higher level entities are in fact causally reducible to the lower level. Yet doing so produces what I have called [2] a reductionist blind spot. Features of the higher level that are independently real are discarded. This reductionist blind spot is the key to answering the questions posed by Schrödinger and Anderson.

I like to use the Game of Life as an illustration. The Game of Life is defined by a set of rules that determine everything that happens on a Game of Life grid.[13] These rules are analogous to the fundamental laws of physics. As Schrödinger and Anderson point out, nothing eludes the laws of physics. Similarly nothing eludes the Game of Life rules; everything that happens on a Game of Life grid can be reduced to these rules. The Game of Life Rules are the bottom level.

Certain Game of Life configurations produce patterns. The most famous is the glider, which moves diagonally across the grid. As far as the Game of Life rules are concerned, there are no such things as gliders, just cells going on and off. This is similar to saying that as far as elementary particle physics is concerned, there are no such things as molecules, biological organisms, or solar systems; just interacting elementary particles, which is just as true. Yet even though they are reducible, gliders and similar patterns form a level of abstraction with its own laws. For example, one can talk about the velocity of a glider and predict when it will reach a certain cell.[14] Gliders and other patterns form a second level of abstraction.

A third level consist of Turing machines, which one can build from gliders and other second level patterns. Again, even though nothing happens on a Game of Life grid that isn't attributable to the Game of Life rules, the Turing machine level can be described and specified independently. Furthermore computability theory applies to Turing machines. Thus while not eluding the Game of Life rules, new laws (computability theory) that are independent of the those rules apply at the Turing machine level of abstraction—just as Schrödinger and Anderson said.

[13] A live cell with exactly two or three live neighbors stays alive; a dead cell with exactly three live neighbors becomes alive; all other cells die.

[14] Appropriately dressed up, these second level rules can even be made to look like downward causation: a glider "causes" a particular cells to go on at particular times. Of course it isn't downward causation; the Game of Life rules are the underlying cause.

As glider laws can be made to look like downward causation, Turing machine properties can too. Because the halting problem is undecidable, it is undecidable whether an arbitrary Game of Life configuration will reach a stable state. Not only are there independent higher level laws, those laws apply to the fundamental elements of the Game of Life. In [1] 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 Game of Life rules that turn the causal crank. Why not reduce away these epiphenomena, throw away gliders and Turing machines? Reducing away a level of abstraction produces a *reductionist blind spot*. If one reduces away Turing machines, one loses computability theory. No equations expressed in terms of Game of Life grid cells can describe the computations performed by a Turing machine unless the equations themselves model a Turing machine. Entities at higher levels are ontologically real and obey their own laws even though they are causally reducible to the level that implements them.

It isn't surprising that levels of abstractions give rise to new laws. To implement a level of abstraction is to impose constraints, to break a symmetry that holds on the constrained layer. By establishing a glider pattern on an otherwise empty Game of Life grid one is limiting the possible ways the Game of Life rules might operate. The glider forces some of them to operate in a particular way.

Once such a constraint is established, the constrained system will almost always obey laws that wouldn't hold otherwise. To express those laws requires reference to the abstractions implemented by the constraints. In Shalizi's terms, the law that predicts when a glider will cause a particular cell will to go on can be made efficiently only when expressed in terms of gliders. A similar law could be expressed directly in terms of Game of Life rules and grid cells, but it would be much less computationally efficient.

Similarly, a prediction that a Game of Life starting configuration (representing the input on a Turing machine tape) will terminate in a particular configuration (representing the output on the tape) could also be made directly in terms of Game of Life rules and grid cells. But if expressed at that level, it will either have to model a Turing machine or be computationally inefficient.

## 8. *Summary*

Since nature is blind, how does it build the wonders that we see around us? Is nature a blind watchmaker or a blind programmer—an engineer or a computer scientist? Just as we build Turing machines from Game of Life elements, nature builds higher level entities bottom-up. Chemistry is based on the binding properties of atoms and molecules, not on quantum mechanics—even though quantum mechanics implements those properties. Similarly biology is the study of  biological organisms, not of collections of atoms and molecules.

Although scientists analyze nature top-down and engineers build systems top-down, nature builds phenomena bottom-up, level of abstraction by level of abstraction. Nature is a computer scientist, not an engineer.

To reprise Anderson [4],

> At each level of complexity entirely new properties appear. … [O]ne may array the sciences roughly linearly in [a] hierarchy [in which] the elementary entities of [the science at level n+1] obey the laws of [the science at level n] … . But this hierarchy does not imply that science [n+1] is 'just applied [science n].' At each [level] entirely new laws, concepts, and generalization are necessary. … The whole becomes not only more than but very different from the sum of its parts.

The computer science notion of level of abstraction shows how this can happen. The laws, behaviors, and properties at one level are implemented by using properties and behaviors at lower levels. Entirely new properties can appear at each level as long as those new properties can be implemented in terms of lower level capabilities. As the example of the gecko illustrated, though, nature's levels are not necessarily strictly and linearly ordered. A level may make use of any lower level, not just the level immediately below.

Engineers must always be looking downward. They must always be concerned about the possibility of lower level physical effects. But engineering's creation of the bit enabled computer science to develop the upward-looking notion of the level-of-abstraction, which can be applied to solve a problem in philosophy.

## References

[1] Abbott, Russ, "Emergence explained," *Complexity*, Sep/Oct, 2006, (12, 1) 13-26. Preprint: http://cs.calstatela.edu/wiki/images/9/95/Emergence_Explained-_Abstractions.pdf.

[2] Abbott, Russ, "If a tree casts a shadow is it telling the time?" *Journal of Unconventional Computation*, Vol. 5, No. 1, pp. 1–28, 2008. Preprint: http://cs.calstatela.edu/wiki/images/6/66/If_a_tree_casts_a_shadow_is_it_telling_the_time.pdf.

[3] Abbott, Russ, "Putting Complex Systems to Work," Complexity, 2007, (13, 2) 30-49. Preprint: http://cs.calstatela.edu/wiki/images/c/cb/Putting_Complex_Systems_to_Work.pdf.

[4] Anderson, P.W., "More is Different**,"** *Science*, 177 393-396, 1972.

[5] Autumn, Kellar, et. al. "Evidence for van der Waals adhesion in gecko setae," *Proceedings of the National Academy of Sciences*, August 27, 2002, 10.1073/pnas.192252799. http://www.pnas.org/cgi/reprint/192252799v1.

[6] Commission on Engineering and Technical Systems, National Academy of Engineering, *Design in the New Millennium*, National Academy Press, 2000. http://books.nap.edu/openbook.php?record_id=9876&page=R1.

[7] Schrödinger, Erwin, *What is Life?*, Cambridge University Press, 1944. http://home.att.net/~p.caimi/Life.doc.

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

[9] Wing, Jeanette, "Computational Thinking," *Communications of the ACM*, March 2006, (49, 3) 33-35. http://www.cs.cmu.edu/afs/cs/usr/wing/www/publications/Wing06.pdf.

*All Internet accesses are as of March 10, 2008.*