# Putting Complex Systems to Work

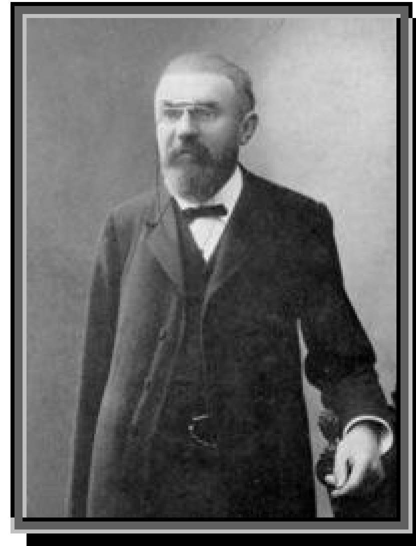
**Russ Abbott**
***The Aerospace Corporation***
***and***
***California State University, Los Angeles***
**Russ.Abbott@GMail.com**

**Abstract.** A primary objective of this paper (as well as of this symposium) is to examine concepts from the field of complex systems that can be applied to systems engineering. In this paper we focus primarily on the notions of emergence and entities and discuss their implications for systems engineering.

## 1 Introduction

**Complex Systems.** The study of Complex Systems (originally called *Complex Adaptive Systems*) as a distinct and recognizable discipline has been ongoing for more than three decades. Most trace its origins to John Holland's work on genetic algorithms (Holland, 1975) in the 1970's. Holland showed that a non-directive random process, which resembles biological evolution, is capable of producing useful designs.

Some trace it back much farther—to Henri Poincaré's invention of chaos theory, approximately a century ago, when he demonstrated that it is mathematically impossible to find a closed form solution to the Newtonian equations for the trajectories of three or more interacting gravitational bodies.[1]



**Henri Poincaré**[2]
Photograph from the frontispiece of the 1913 edition of 'Last thoughts' and therefore published prior to 1923.

[1] The following websites (among others) tell the story.

http://cosmicvariance.com/2006/07/23/n-bodies/, http://www.irit.fr/COSI/training/complexity-tutorial/henri-poincare.htm, and http://www.imcce.fr/Equipes/ASD/preprints/prep.2004/Poincare_Barcelone_2004_en.pdf.

[2] The images used in this paper are all copyright free from Wikipedia Commons: http://commons.wikimedia.org/.

**Systems Engineering.** According to the INCOSE website,[3]

> Systems Engineering is an engineering discipline whose responsibility is creating and executing an interdisciplinary process to ensure that the customer and stakeholder's needs are satisfied in a high quality, trustworthy, cost efficient and schedule compliant manner throughout a system's entire life cycle.

Although not explicit in this definition, a significant aspect of systems engineering is the development of a design for the system being engineered.

Our focus in this paper will be on the relationship between emergence and design. Emergence will lead us to consider the issue of entities, what they are and what it takes for them to persist. We also examine the sorts of environments that support emergence.

*Section 2* discusses the notion of design in complex systems and explores what it means to say that a system is complex.

*Section 3* extends the framework developed in section 2 to define emergence. It also shows how emergence is intimately related to systems engineering.

*Section 4* discusses how emergence is connected to our notion of entities. It discusses the question of whether entities are objectively real. (We conclude that they are.)

*Section 5* discusses the relationship between thoughts and things—and in particular between thoughts, requirements, designs, and things. It discusses computer science's success in developing languages that help us externalize our thoughts. It attributes a significant part of that success to the fact that the languages in which we express our thoughts are also the languages we use to control computers. This section also discusses the notion of a level of abstraction, the technique used in software to create emergence. Systems engineering isn't so fortunate.

*Section 6* discusses multi-sided platforms. A multi-sided platform is a level of abstraction that provides a means, mechanism, or set of conventions to support the interaction of multiple elements.

*Section 7* discusses dissipative systems, a kind of entity intermediate between static and dynamic entities, and the kind of entities engineers tend to build. A major difference between dissipative systems and dynamic entities is that dynamic entities are designed from the core to be self-sustaining, with whatever additional functionality they have built on top of their ability to sustain themselves.

*Section 8* discusses service-oriented designs. It argues that this is not a fad but a fundamental design principle used by nature.

*Section 9* discusses feasibility ranges, that all emergent properties have them, and that it would serve us well to pay more attention to them.

*Section 10* discusses modeling and simulation. Makes the point that we aren't nearly as good at it as we need to be. It also makes the point that even if were much better at it, we would still not necessarily know how to use it to model emergence.

*Section 11* discusses innovative environments. An innovative environment is

[3] http://www.incose.org/practice/fellowsconsensus.aspx

one that may be thought of as emergence-friendly. This section suggests some properties that innovative environments may be expected to have and that seem likely to foster emergence.

# 2 Design in complex systems

Before discussing *design* in the constructive sense as used in systems engineering, we examine what we mean by *design* in general.

A primary goal of science is to compress phenomenological descriptions into concise abstractions. One wants to formulate a statement about nature that (a) captures a wide range of phenomena but (b) is more abstract and concise than a simple enumeration of the phenomena described. If successful, one will have achieved a reduction in the algorithmic complexity[4] of one's description of nature.

Chaitin, one of the developers of the theory of algorithmic complexity, credits Leibniz with being one of the first to formulate the goals of science in this way. (Chaitin, 2003)

> What is a law of nature?
>
> According to Leibniz, a theory must be simpler than the data it explains!
>
> Because if a physical law can be as complicated as the experimental data that it explains, then there is always a law, and the notion of "law" becomes ***meaningless***!
>
> Understanding is compression! A theory as c**omplicated** as the data it explains is NO theory!
>
> All of this is stated very clearly (in French) in 1686 by the mathematical genius and philosopher Leibniz!

**Gottfried Wilhelm Leibniz**
Gottfried Wilhelm Leibniz (Artist: Bernhard Christoph Francke, Braunschweig, Herzog-Anton-Ulrich-Museum, ca 1700)

## *2.1 Upward predictability*

When applied to a relatively well-bounded region of interest (here for convenience called a *system*) one is looking for what we would now call the *design* of the system.

Science often proceeds by examining a system's components and their interrelationships in the hope that in doing so one will be able to formulate a simpler description of the system as a whole.

Newton's law of gravity is a good example. It offers a description of how two

[4] The algorithmic complexity of some information is the smallest computer program required to reproduce that information. A random sequence, a sequence with no internal structure, has maximal algorithmic complexity because it is impossible to find a computer program that will reproduce the sequence that is shorter than the sequence itself.

bodies will move with respect to each other—a description which is much simpler than an enumeration of the relative positions of the bodies in time. Furthermore, given Newton's law of gravity, it is possible to construct closed form equations that characterizes the position of the two bodies when under the influence of mutual gravitational attraction.

This is what one wants: an analysis of the interactions at the component level that yields a simple explanation of the apparent complexity of phenomena observed at the system level. Systems that yield to this paradigm are what might be called *upwardly predictable*.

### *2.2 Complex systems aren't upwardly predictable*

A distinguishing feature of systems that are considered complex is that they tend not to be upward predictable: there is no (known) way to reduce the complexity of a description of the system as a whole by formulating it in terms of descriptions of the system's components. In other words, the properties and behaviors of a complex system are not describable as a simple, closed-form, mathematical function of the properties and behaviors of the system's components. Typically the best one can do is to propagate the descriptions of the component interactions and see what happens at the system level.[5]

An example of such a class of systems are n-body gravitational systems for $n > 2$. As indicated above, Poincaré showed that Newton's laws cannot be used to provide a reduced description of the trajectories of more than two gravitational bodies. There is no simple way to combine descriptions of the system's components to produce a description of the system as a whole that is simpler than propagating the descriptions of the components.[6]

Systems deemed to be complex in this sense are not claimed to violate traditional notions of scientific causation: no additional forces are presumed to contribute to the complex functioning of the system as a whole. The only claim is that there is no compact mathematical equation that will characterize the behavior of the system as a whole as a function of its components.

## 3 Emergence

### *3.1 Emergence and Complexity*

Sometimes higher level systems have properties that seem to be both complex and not complex in the sense just described. Such systems have properties that may be characterized in relatively simple terms—much simpler than the terms required to characterize the elements of which they are composed. Even so, those systems still cannot be usefully understood in terms of a function from their components to their higher level properties.

An example is a computer program that computes a mathematical function. The mathematical function itself is a straightforward characterization of the input-output behavior of such a program. Such a description is far simpler than

[5] This approach is now known as agent based modeling.

[6] Another example is quantum mechanics. There is no simple way to evaluate the Schrödinger equation for systems of more than a very few particles. In other words, nature is complex.

the description of the behavior of the computer on which that program is running. If one were to describe the sequence of states (at virtually any level of abstraction from quantum mechanics to gates to machine language) through which the computer passes as the program runs, that description would be far more complex than the description of the function itself. Furthermore, there is no general way to map the states of the computer (or the computer program that generates them) to the mathematical function the software is computing.[7]

Typically the best one could do would be to trace through the step-by-step operation of the computer and see what happens. So even though our software-plus-computer system is simple to describe at the system level, it is also complex in the sense described earlier.[8]

So here we have an apparent paradox. We have both (a) complexity: there is no natural complexity-reducing transformation of the descriptions at the lower level into descriptions at the higher level and (b) simplification: the higher level may be described in simpler terms than those needed to describe the lower.

We have come to use the term *emergence* to describe situations such as these. In his PhD dissertation Shalizi captures the algorithmic complexity sense of emergence as follows.

> One set of variables, ***A***, emerges from another, ***B*** if (1) ***A*** is a function of ***B***, i.e., at a higher level of abstraction, and (2) the higher-level variables can be predicted more efficiently than the lower-level ones, where "efficiency of prediction" is defined using information theory.[9]

## 3.2 Emergent properties are autonomous

Although Shalizi's definition gets at the algorithmic complexity issues, it misses an important point about emergence. The relationships between the lower and higher level variables are typically much less important than the emergent property itself. In fact, the higher-lower relationships are typically a matter of convenience rather than necessity.

In our example of software that computes a mathematical function one doesn't care what sequence of states the computer traverses. All one cares about is that the function be computed correctly. There may be any number of ways to compute the function. Each may traverse a different sequence of computer states but yield the same eventual output. It doesn't matter which one is

[7] In general it is not decidable what function an arbitrary computer program computes. In using this example, we are not identifying complexity with undecidability. But we are categorizing the undecidable as complex.

[8] Because of this seeming inversion of control—a simpler higher level appears to spring from a more complex lower level—there has been a temptation to look for an explanation in terms of "downward causation"—the simpler higher level *causes* the phenomena observed at the lower level.

Of course there is no downward causation. But because we are so used to thinking in terms of simple abstractions giving rise to complex phenomena it's easy to understand why downward causation may seem like an attractive possibility.

[9] The extract is from his website: http://www.cscs.umich.edu/~crshalizi/notebooks/emergent-properties.html.

used as long as the final result is correct.

To capture this aspect of the issue, I defined (Abbott, 2006b) *emergence* as the situation in which one can describe properties of a system in terms that are *independent* of its components. From this perspective, the property of interest (in this example the computation of a mathematical function) may be *implemented* by lower level components (the program must run on some computer after all), but it is defined independently of those components.[10]

Standard examples of emergence include the hardness of diamonds (static emergence) and the tendency of some species of birds to move in flocks (dynamic emergence). In both cases, the higher-level properties are not only defined *ab initio* at the higher level, they are often meaningless when thought of as properties of the system components.

A diamond is hard because of how its component carbon atoms fit together.[11] But the notion of a collection of carbon atoms fitting together in this way is expressible only at the level of the diamond itself.

Similarly, our notion of a flock is not just a collection of birds; it is a collection of birds that satisfies our intuitive sense of what it means to be a flock.[12] The notion of a flock is no more accessible in the language in which one describes individual birds in isolation than the notion of a diamond is accessible in the language in which one describes individual carbon atoms in isolation.

### *3.3 Emergence and requirements*

The examples discussed above are what might be considered naturally occurring emergence, Systems engineers are familiar with emergence as the requirements that a system must satisfy. Consider what we would now consider a simple system such as an automobile. A primary requirement is that it can be driven from here to there. That property is emergent. It is not meaningfully applied to any of the components of the automobile. Nor is it expressible as a closed form mathematical function of the automobile's components.

---

[10] As developed in (Abbott 2006) this leads to the notion of downward entailment, a phenomena similar to but scientifically more acceptable than downward causation. The specific example developed in the paper shows that because one can simulate a Turing Machine using the Game of Life, the Game of Life is undecidable. The undecidability of Turing Machines downwardly entails the undecidability of the Game of Life.

[11] A diamond provides a nice example of downward entailment. Because the carbon atoms of a diamond implement a rigid lattice, the position and orientation of the diamond as a whole downwardly entail the positions of its components.

This is a good illustration of the significance of multi-scalar phenomena. At one level, the diamond as a whole moves through space. At another the diamond as a rigid lattice structure is maintained. We have two almost independent collections of phenomena that operate on different scales but that are sufficiently intertwined to produce the effect of downward entailment.

[12] The American Heritage® Dictionary (2006) defines *flock* as, "A group of animals that live, travel, or feed together."

Thus systems engineering (in fact, engineering in general) may usefully be understood as the design and development of systems that have desired emergent properties. As Rechtin put it,

> A system is a construct or collection of different elements that together produce results not obtainable by the elements alone.[13]

Virtually any engineered product illustrates emergence in this sense. Consider the fact that computers perform binary arithmetic. Binary arithmetic (or any other kind of arithmetic) has no relevance as a property of any of the components that we use to build computers. There is no sense in which one can say that properties significant to arithmetic apply in any normal sense to electrons. Nor does it make sense to compare binary arithmetic in any information theoretic sense at the electron level to its implementation at the computer instruction level. Binary arithmetic is defined autonomously at the computer level and is not applicable to the components of which the computer is constructed.

Emergence in engineering is standard practice. We tend to be surprised by it when we see it occurring in nature without our help. But it does. One way to think about it is that nature really is a blind watchmaker/engineer—and not only when engineering biological systems.

## 4 Emergence and entities

In all of the preceding examples, emergent properties were defined in terms of a higher level entity—often using language that is not even applicable to the components of the entity. If one thinks about it, this is quite strange. What are these higher level entities we are talking about? On what ontological grounds do we permit ourselves to speak about them? Perhaps more to the point, are such higher level entities objectively real in any way that we can make sense of? Is a diamond or a flock or an automobile (or any other system) a real entity? Or is it simply a collection of its components?

In (Abbott, 2007) we conclude that entities are objectively and recognizably real in that (a) they have either more or less (but not the same) mass as the combined mass of their components considered separately, and (b) they bind their components together in a form that reduces entropy.

It is only because of the entropy reduction contributed by entities that it is possible for Shalizi's definition of emergence to be realized. How is it possible after all for the algorithmic complexity of a system's description to be reduced? A description of a system is a description of a system. If one description is simpler than another, it is only because the simpler description takes advantage of some entropy-reducing structure that the more complex description ignores.

So if a description of a system expressed in terms of "higher level" constructs is simpler than a description expressed in terms of "lower level" constructs, that means that the "higher level" constructs have built into them some structure that the lower level constructs lack. But that raises the question of how the "higher level" constructs embody and maintain that structure.

[13] From the INCOSE website: http://www.incose.org/practice/fellowsconsensus.aspx.

That's what an entity is, a means to embody and preserve structure.

## 4.1 Kinds of entities

There are two kinds of entities: static and dynamic.[14] Static entities (for example, atoms, molecules, and solar systems) maintain structure because they exist in energy wells—and hence have less mass as an aggregate than their components.[15]

Dynamic entities (for example, living organisms, social and political organizations, and, strikingly, hurricanes[16]) maintain structure by using energy they import from outside themselves. Because of the flow of imported energy, they have more mass as an aggregate than the combined mass of their components.[17]

Entities have emergent properties that are defined at the level of the entity itself. That a government is democratic or that a diamond is hard are properties defined at the level of the government or the diamond. They are not properties of the components of a government or a diamond.

## 4.2 The wonder of entities

One must wonder whether this isn't slight of hand. How can one speak of an entity and discuss its properties independently of its components?

Do entities spring into existence fully formed? How is that possible? Because this seems so mysterious, one may be tempted to speak of mechanisms for self-organization. We see this as a distraction.

There is nothing mysterious about how entities form. Static entities form as a result of well understood physical laws: atoms are created from elementary particles; molecules form from atoms; etc. Dynamic entities also form as a result of natural processes. Governments form when people create them—either explicitly or implicitly. Hurricanes form when the atmospheric conditions are right. Self-organization is not the point.[18]

---

[14] Below we will introduce an intermediate third kind, the kind that engineers generally build.

[15] Paul Humphreys (1997) suggested a similar notion, which he called *fusion*. The following is Timothy O'Connor's summary (2006) of Humphreys' position.

> "[Emergent properties] result from an essential interaction [i.e. fusion] between their constituent properties, an interaction that is nomologically necessary for the existence of the emergent property." Fused entities lose certain of their causal powers and cease to exist as separate entities, and the emergents generated by fusion are characterized by novel causal powers. Humphreys emphasizes that fusion is a "real physical operation, not a mathematical or logical operation on predicative representations of properties."

[16] See (Abbott, 2007). We discuss hurricanes as entities below.

[17] Speaking poetically one might refer to the energy flowing through a dynamic entity as its soul or spirit. When the energy stops flowing, the entity dies. From this perspective a soul or spirit has mass.

[18] It is still an open question how one might form a biological cell "from scratch." There is no known mechanism for producing a cell other than through cell division, i.e.,

The marvel of entities is not in some seemingly magical process of self-organization; the marvel is that entities exist at all and that they have properties and behaviors that in some sense may be described autonomously.

The fact that entities seem to spring into existence in some sense fully formed and that they have properties that seem to be defined self-referentially is the basis of the argument from Intelligent Design. How can something that is altogether new and that has new properties—properties like the ability to fly that seem to be defined in terms of the entity itself—appear apparently from nowhere?

One answer—not related to Intelligent Design—is that the "new properties" we attribute to entities are really nothing more than ideas in our minds. Properties as such don't exist in nature. Entities are what they are no matter what properties we attribute to them.

This is not say that an entity's "new properties" are fictitious. Hemoglobin can transport oxygen. But the property of being able to transport oxygen, while true of hemoglobin, is not a label one finds attached to hemoglobin molecules. The conceptualization of the ability of hemoglobin to transport oxygen as a property of hemoglobin is an idea in our minds.[19]

A second answer—also not related to Intelligent Design—is simply to look around and see that it happens. The fact that entities come into existence means that nature provides some way for this to happen. It is our job as curious creatures to understand it, not to deny its possibility.

In attempting to understand how entities form we encounter the real mystery—a mystery deeper than we can explore in this paper. What are the primitive entities—if, indeed there are primitive entities—and how do they interact? In quantum mechanics the primitive elements act as both particles and fields. As fields they interact because they inhabit a common environment of an assumed three-dimensional space—although that doesn't seem to be the complete answer. What is the mechanism, for example, whereby fermions, e.g., electrons, obey the Pauli exclusion principle? What mechanism prevents two fermions (but not two bosons) from occupying the same state?

Of course if there were an answer to the "what mechanism" question, then the elements whose behavior is being explained would not be primitive. The alternative is to say that there is no mechanism and that the Pauli exclusion principle is simply a fact of life. But that seems too arbitrary. Of course any set of primitive elements must be accepted axiomatically. But why these axioms?

Smolin (2006) argues that to find a better answer to these questions we need a background-free theory of fundamental physics. Such a theory would at least provide an explanation in terms of itself and may seem somewhat less arbitrary.

---

from an existing cell. How did the first cell form? We don't yet know.

[19] We discuss below why it is important for systems engineering to distinguish between ideas in our minds and properties of entities.

# 5 Externalizing our thoughts

## 5.1 Emergence and systems engineering

As indicated above, emergence is central to systems engineering. Virtually every important property of an engineered system is emergent. Although we tend to think of human-produced and naturally occurring artifacts as different, emergent properties that result from an engineering effort differ from those that occur in nature only in that they are produced intentionally.

Yet there tends to be a significant difference between these two kinds of systems. Human-built systems are often functionally fragile; their emergent properties don't hold up as robustly as we would like. Naturally occurring systems tend to be more robust, flexible, and adaptable. Why is that?

One reason is that human engineered systems are typically built using a top-down design methodology.[20] We tend to think of our system designs hierarchically. At each level and for each component we conceptualize our designs in terms of the functionality we would like that component to provide. Then we build something that achieves that result by hooking up subcomponents with the needed properties. Then we do the same thing for the subcomponents. Etc.

This is quite different from how nature designs systems. When nature builds a system, existing components (or somewhat random variants of existing components) are put together with no performance or functionality goal in mind. (Nature doesn't have a mind.) The resulting system either survives in its environment or it fails to survive.

This approach doesn't necessarily make nature a brilliant designer. Some of nature's designs are wonderful, and some suck.[21] But significantly nature never has to satisfy a requirement.[22] Systems engineers don't have that luxury. But is there anything we can learn from how nature develops designs that we can apply in our work?

## 5.2 Thoughts and things

A useful way to think about the difference between systems designed to satisfy requirements and naturally occurring systems is that requirements-based systems typically result from an attempt to externalize our thoughts. We think, "I want a system that does this, this, and that—i.e., with these properties and behaviors."

Dreams of this sort, no matter how dressed up and legitimized in terms of formal requirements are still nothing but ideas in our minds. Use of the somewhat deprecatory term *nothing but* is intentional. Ideas by their nature can exist only in the mind of someone who is

[20] Software developers gave up top-down design a quarter century ago. In its place we substituted object-oriented design—the software equivalent of entity-based design. Below we suggest that it's time for systems engineering to do likewise.

[21] Silver (this conference) points to "the panda's thumb, the placement of the windpipe in front of the esophagus (so that food can go down the wrong tube), traversal of the urethra through the prostate gland (so that if the prostate becomes inflamed and swells, it becomes difficult to urinate)" as examples of bad natural design.

[22] Nor does nature have to work within budget and schedule constraints.

thinking them. That's all an idea is and can ever be, a subjective experience in the mind of the idea's thinker. (See Abbott (2006a).)

Yet when our ideas involve systems, we want more than just pretty mental pictures. We want material embodiments of our ideas. We want to have the ideas in our heads converted into physical reality. We want to externalize our ideas and to make them materially concrete.[23] And we often succeed—spectacularly. Much of what we experience in our post-modern 21st century lives is the result of successfully externalized thought.

But let's consider what it means to externalize a thought. There is no *externalize button* on our foreheads which, when pressed, causes our ideas to materialize as physical reality. One cannot simply imagine something and expect a material embodiment of it to spring into existence. Furthermore, even when we do build something that reflects our ideas, it is impossible to create an external replica of a thought. Anything outside our heads is different from something inside our heads. Nothing outside our heads is an idea. The best we can ever do in externalizing a thought is to create something that we can understand as representing—or perhaps better yet embodying—that thought.

### 5.3 Molding reality to resemble thoughts

Consider a word processing computer program. We design word processors to (appear to) operate in terms of characters, words, paragraphs, etc. Characters, words, and paragraphs are ideas. Word processors operate (when described at one reasonable level of abstraction) in terms of character codes, sequences of character codes bounded by white space character codes, and sequences of character codes bound together as what the word processor may internally refer to as a paragraph data structure.

What we do when we attempt to externalize an idea is to mold elements of physical reality into a form onto which we can project the idea we want to externalize. That's all we can ever do. We can never do more than mold existing reality.

But even though we cannot incarnate our ideas as material reality, we can mold physical reality in such a way that it has—or at least appears to have—properties a lot like those of the ideas we want to externalize. [24]

Thus there is always a tension between (a) building something out of real physical substance (even if that substance involves bits) and (b) externalizing one's thoughts about what one wants.

This tension is easiest to describe with respect to software—but it is true of every constructive discipline, including systems engineering. When one writes software, one is writing instructions for how a computer should perform. That's all that one can ever do: tell a computer first to do this and then to do that. The this and that which the software tells the computer to do are the computer's

[23] This, of course, is the engineer's credo.

[24] My wife, an English professor, objected to my claim that word processors don't work with paragraphs. They do such a good job of manipulating text in a way that corresponds to her sense of what a paragraph is, that she wants to credit them with working with actual paragraphs.

primitive instructions. But what we want in the end is for the computer's *this-ing* and *that-ing* to produce a result that resembles some idea in our heads.

Thus in software (as in any engineering discipline) our creations always have two faces: (a) a reality-molding face whereby the software tells the computer what to do and (b) a thought externalizing face which represents our ideas about what we want the result of that molding process to mean. The eternal tension is to make these two faces come together in one artifact.[25]

### 5.4 Thought externalization in computer science

Because software can be about an extraordinarily wide range of possible thoughts, computer science has had to face the reality-vs.-thought confrontation more directly than any other human endeavor. And possibly because software as text seems to be the only example of an artifact that directly embodies both aspects of this tension, computer science has been relatively successful in finding ways to come to grips with this problem.

Computer science has developed languages in which we can both express our thoughts and control the operation of a computer. We invented so-called higher level programming languages (Fortran being one of the earliest) in which one could write something like mathematical expressions which the computer would evaluate. We invented declarative languages (Prolog is a good example) in which one could write statements in something like predicate calculus and have the computer find values that make those statements true. We combined Prolog and Fortran when we invented constraint programming (which has not been as widely appreciated as it deserves) in which one can write mathematical statements of constraints which the computer ensures are met.

We invented relational databases in which one can store information about entity-like elements—along with their attributes and their relationships to each other. We invented languages that allow one to query those databases more or less on the level of that conceptualization.

We invented object-oriented programming languages—which led naturally to agent-based and now service-oriented environments—in which one writes programs that (seem to) consist of interacting entities.

At the application level, virtually *every* computer program—from a payroll program to a word processor to an image processing program—embodies an ontology of the world to which that application applies.

To help us write application programs we invented tools and frameworks that define meta-ontologies within which one can create a desired ontology.

We did all this by writing programs that tell computers first to execute this instruction and then to execute that instruction. The gap between the underly-

[25] Often the two faces of our creations come together as the concrete drives out the conceptual. We now think of *airplane* as meaning a physical airplane and not the idea of a heavier-than-air transport ship. Most likely we will soon think of *paragraph* as meaning whatever MS word produces—although we will continue to distinguish between well-structured and ill-structured paragraphs.

ing computer and the languages in which we write programs is often enormous. But that doesn't mean that we can forget about the computer. No matter what else it is, and no matter how well our programs (seem to) express the thoughts in our heads, a program is nothing unless it tells a computer which instructions to execute and in what order. In the end, that's all a computer program is: a means to tell a computer what to do.

## *5.5 Computer science uses emergence to link ideas to reality*

Computer Science has been called applied philosophy:[26] one can think about virtually anything as long as one can express those thoughts in a form that can be used to control the operation of a computer.

I like to think of the computer as a reification machine: it turns symbolically expressed abstract thought into concrete action in the physical world.[27]

As a reification machine, the computer's interface between thought and action is the computer program. When we write in a programming language we are expressing our thoughts in the programming language—to the extent allowed by the language. When a computer reads what we have written, it takes our writings as instructions about what operations to perform.

We have developed programming languages that allow us to express something close enough our thoughts that the resulting programs, when executed, can be identified with those thoughts.

The primary technique computer scientists use to build programming languages that allow us to externalize our thoughts is emergence. Recall that we defined emergence as a situation in which a property can be described independently of its implementation. That's exactly what a program specification is. Whenever we specify the desired behavior of a computer program independently of the means by which that behavior is implemented, we are asking for the creation of an emergent phenomenon.

Emergence of this sort has a long history. Both axiomatic semantics (Hoare, 1969) and denotational semantics (Tennent, 1976) offer approaches to providing declarative specifications for computer programs—which by intent are independent of the program's implementation.

More generally, workers in the fields of functional and logic programming have created programming languages (e.g., Haskell[28] and Prolog[29]) in which the programs one writes may be understood as a declarative statement of one's intentions rather than as instructions to a computer.[30]

---

[26] Fred Thompson, one of my early mentors, is now Emeritus Professor of Applied Philosophy and Computer Science at Cal Tech.

[27] With virtual reality we complete the cycle: generating real physical signals with the intention of producing particular subjective experiences.

[28] See http://www.haskell.org/.

[29] For example, http://www.swi-prolog.org/.

[30] However practitioners in both paradigms find that most "real" programs written in functional and logic programming languages generally cannot be understood in a fully declarative way. Virtually all real programs—no matter what the language

The ACM's series of conferences on declarative programming[31] explores the even more general question of what one can say about programs that is independent of the steps the program instructs the computer to take.

The prototypical example of emergence in software is the application program interface or API. An API characterizes what some software—generally the elements of a program library—will do when invoked in certain ways. A good API does not describe how the software will accomplish that result, just what the result will be. This is standard good practice about software design and specification.

What is not often mentioned—perhaps because we all take it so much for granted by now—is that an API is generally explicated in terms of an ontology that may have nothing to do with the means by which the API is implemented. As we indicated above, virtually every computer application is intended to implement a conceptual model, i.e., some externalized thought. The same is true for the APIs exposed[32] by a software library, application, or system.

—must be understood operationally. This is understandable. One can never escape the fact that a computer program is necessarily a means for instructing a computer to act.

[31] The International Conferences on Principles and Practice of Declarative Programming (PDPP). See the website, http://pauillac.inria.fr/~fages/PPDP/.

[32] Current usage favors the term *expose* to refer to API operations. The implication is that an API is a window into a secret ontological world (the conceptual model) to which one has access only via the operations made available through the API.

The ontology or conceptual model in terms of which a software system is described is sometime referred to as a *level of abstraction*. In other words a software system creates an emergent ontological domain that can be accessed and manipulated in ways specified by its API.

One of the primary threads in the history of computer science is the development of increasingly powerful ways to create new levels of abstraction. By providing ourselves with the means to create new ontological domains—which can then be used to build other ontological domains, etc.—computer scientists have used the power of emergence to create physical models of an extraordinarily wide range of thoughts.

Although these externalized thoughts are far removed from low-level computers operations, they are nonetheless still grounded by real computers executing one real physical instruction after another. In perhaps more familiar words, software development is both a top-down (thought externalization) and a bottom-up (emergence) endeavor.[33]

## 5.6 Thought externalization in systems engineering

Systems engineering is just beginning to focus on this issue. Model-based development, e.g., SysML, attempts to allow

[33] Of course the "real" "physical" instructions that ground computer software are themselves emergent phenomena built on top of still lower level phenomena. Computer science owes its existence to the ability of electrical engineers to create an emergent digital world—of bits and instructions that manipulate them—that we can use as a platform on which to build our emergent creations.

systems engineers to think in a language that both expresses our thoughts and represents how we mold reality. But systems engineering is at a significant disadvantage. In computer science we write in languages that control real computers.[34] There are no systems engineering languages that generate real physical systems.

When software developers write a computer program, load it into a computer, and press the Start button, the computer *becomes* the program we have written. There is nothing comparable for systems engineers. We don't have a systems engineering language and a device into which descriptions written in that language can be loaded that will *become* the system the language is describing once one presses a Start button.

The closest systems engineering can come to this dream is to write in a language that represents a model of a physical system. But models aren't reality. Programming languages succeed because they are grounded in the reality of an actual computer executing actual instructions. Models, in contrast, are always divorced from reality. One can't ever model all aspects of a system. So one chooses what one considers a system's most important aspects and models those. But that's always dangerous. See the discussion below about the difficulty of looking downwards.

[34] UML is an unfortunate step back from computer science's traditional loyalty to executable languages.

## *5.7 Thought externalization in science*

Science may be understood as a similar process of thought externalization. Science may be understood as a search for an explanation of how nature works. What that amounts to is a search for an explanation of a level of abstraction in terms of implementation mechanisms for the level of abstraction. In other words, scientists observe phenomena, which they describe in some terms that seem to fit the phenomena. Then they look for underlying mechanisms that explain why the phenomena seem to reflect or embody the observed abstractions. Much of early biology and chemistry, for example, fit this pattern quite well. These disciplines organized and catalogued biological and chemical entities into the well known biological taxonomies and the Mendeleev's periodic table of the chemical elements.

As in other forms of thought externalization scientists develop ideas about how to think about nature and then look for ways to make those ideas concrete. In this case those ideas are based on observed phenomena. Then they look for more concrete aspects of nature which can be understood as having brought about those phenomena. The more concrete aspects of nature correspond to what in computer science we have referred to as known operations. The phenomenologically inspired level of abstractions correspond to ideas that one wants to externalize.

It is also the case that as in other forms of thought externalization, as one finds concrete ways of expressing one's thoughts, the thoughts themselves become better defined. As (Scerri, 2006) points out in his review of the development of the periodic table, chemists

originally thought that atomic weight characterized chemical elements. We now know that it is the number of protons that characterize a chemical element. In this way, the intuitive but informal idea of a chemical element that had known properties that differed from other chemical elements was refined and made precise by understanding that atoms are best grouped according to the number of protons they contain.

The work of shaping existing concepts as a way of formalizing and formulating a thought appears to be an important aspect of thought externalization. An important part of thought externalization is the act of grounding a thought in concrete terms, i.e., of expressing it in terms that are—or at least appear to be—grounded in references to known reality.

The challenge of science is often to discover previously unknown reality, e.g., the proton, to ground thoughts that we wish to externalize. Once grounded one often finds that the original thoughts were only approximations of what turn out to be more robust ways of understanding nature.

# 6 Multi-sided platforms

As discussed above, a level of abstraction encapsulates and embodies a specialized ontology (i.e., a conceptual model) of one sort or another. An extraordinarily important kind of level-of-abstraction is the multi-sided platform. Hagui characterizes a multi-sided platform (from the perspective of the platform as a business[35]) as one in which the platform provider must

> get two or more distinct groups of customers who value each other's participation on board the … platform in order to generate any economic value. … Examples are pervasive in today's economy and range from dating clubs ([the two sides are] men and women), financial exchanges [such as a stock market], real estate listings, online intermediaries like eBay (buyers and sellers), ad-supported media (ad sponsors and readers/viewers), computer operating systems (application developers and users), videogame consoles (game developers and geeks), shopping malls (retailers and consumers), digital media platforms (content providers and users), and many others.

When considered more generally, i.e., not necessarily as an business, a multi-sided platform is a level of abstraction that provides a means, mechanism, or set of conventions for structuring and enabling interaction among parties—especially parties that expect to benefit from the interaction.

As Hagiu indicated, men and women in a dating club are able to interact because they belong to the same club. The same is true of merchants and shoppers in a mall and buyers and sellers on eBay.

In general, a multi-sided platform results from the factoring out of an aspect of an interaction. In the first three examples above, what's factored out is (a) the process of finding the other party and (b) the formalism of making contact. In the sending-receiving interaction what's factored out is the actual sending and receiving of mail and packages along

[35] Interview with Andre Hagiu, *Working Knowledge*, Harvard Business School, March 13, 2006. http://hbswk.hbs.edu/item/5237.html. Hagiu is one of the authors of (Evans, 2006).

with a formalized way of making use of that service once it's reified as a service on its own.

By factoring out an aspect of an interaction and providing it more efficiently, the platform makes the interaction more efficient for the parties and at the same time generally makes money for itself.

Multi-sided platforms are extraordinarily pervasive. Any file extension (such as .doc, .exe, .pdf, etc.) that is associated with a file that is passed from developer to user (the two sides) is associated with a multi-sided platform. When people use such a platform to work collaboratively, both sides assume both roles.

The platform itself consists of the software that takes the file as input and "brings it to life"—allowing the user to use it as the platform enables.

Many but not all programming languages define multi-sided platforms. Java does because both the user and the developer rely on Java to make use of software developed in Java. C++ isn't because only the developer uses it. Software developed in C++ is created to run on the operating system itself as a platform.

Browsers are multi-sided platforms. Software plug-ins to browsers such as Flash are platforms built on top of the platform provided by browsers.

Google defines two multi-sided platforms. The first is the platform that brings together web sites and web surfers; the second is the platform that brings together web advertisers and web surfers. Google created the second platform—the one on which it makes it money—by giving away the first platform. The second platform consists of the words people use in queries to the first platform.

## 6.1 Multi-sided platforms create new interaction opportunities

As exemplified by Google, multi-sided platforms often create interaction opportunities where they didn't exist before. Other good examples are online mailing lists (such as YahooGroups) and bulletin boards. It is common wisdom that mailing lists and bulletin boards have created communities—and hence interactions among members of those communities—that never would have come into existence otherwise. The same is true of multi-person games such as World of Warfare and Second Life, which are primarily communities rather than competitions. Systems such as MySpace and FaceBook provide another sort of community building platform. The interactions that occur on these platforms would almost certainly never had occurred were it not for the existence of these platforms.

## 6.2 Our multi-sided platforms define our infrastructure

Once a commercial platform becomes established, conflicts may arise when the interests of the platform owner differ from those of the platform users. Pressure may develop among platform users to de-commercialize the platform and to move its governance out of the commercial realm and to bring it under the control of the users.

Our regulated utilities—such as power and telephone services—illustrate a successful combination of user governance and private ownership. Other platforms, e.g., our road and highway system, are owned and operated directly by

the government. We find these platforms so essential that we want to ensure that the interest of the platform users take precedence over the interest of the platform providers.

Platforms such as these, along with the rest of our community-wide platforms (such as our transportation, package delivery, and mail systems[36] and others), define what we refer to generically as a community's infrastructure.

### 6.3 Standards and open source platforms

Organizations that are able to establish a multi-sided platform as a widely used standard (explicit or *de facto*) are often able to profit from it. Familiar examples are Microsoft Windows and eBay. This has led to the notion of what has been referred to as a network effect, namely that the value of a network increases more than linearly with an increase in the size of the network.[37]

The literature on network effects seems not to identify platforms as the source of the value: networks hogs the spotlight. Nonetheless, it is the establishment and ownership of platforms that has economic value. Thus platforms become very important to commercial organizations, who will fight to establish the dominance of their platform in a certain realm.

There are (at least) three countervailing forces. The first, as we mentioned above, is the regulation and in some cases government control of platforms.

The second is the adoption of neutral standards, i.e., standards that are neither controlled by nor tailored to the interests of any particular vendor. When a vendor-neutral standard is defined for a platform, the platform functionality is defined independently of any specific implementation of that functionality. This is of to the benefit of platform users because vendors must then compete to provide better implementations of a platform with a well-defined specification.

Thus the most ephemeral of multi-sided platforms is the standard. Users of systems/components that adhere to a standard are able to interact with each other only because they both conform to the standard.

The third force that mitigates the commercialization of a platform is open source software. Many commercial software products depend on platforms for their operation. The most widespread case is the dependence of software application programs on operating systems. Consider the position of the developer of such a software product. He is essentially at the mercy of the platform owner. Should the platform owner decide to enter the same market, that owner has an enormous advantage. The most widely known case is the way in which Microsoft destroyed the Netscape browser. No company wants to be that vulnerable. The developer of a software application will be motivated to support alternative platforms for his product. The most attractive alternative platform is

---

[36] The platform that facilitates the electronic version of such interchanges is the collection of Internet email standards. See the Internet Email Consortium website (http://www.imc.org/mail-standards.html) for a list of email standards. We discuss below the importance of standards as platforms.

[37] See (Briscoe, 2006) for an argument that the rate of growth is typically *n log(n)* rather than $n^2$.

one that is neither controlled by a commercial entity not regulated by the government. Hence it is not at all surprising that many commercial companies provide significant support to the development of open source platforms.

In a draft article, Iansiti and Richards (2007) analyse open source systems. They find that one can group open source software (OSS) into two categories: the "money driven cluster," which receives 99% of corporate funding of OSS and the "community driven cluster," which receives very little corporate funding.

The big four in the money driven cluster are Linux, Firefox, OpenOffice, and MySQL. All four are platforms that are central to how computers are used. The first three compete with platforms that are owned and controlled by a single for-profit company. No corporation wants to see the platforms on which its products depend subject to the profit calculations of some other commercial entity. It's no wonder that corporations are willing to spend money to strengthen publicly and openly controlled alternative platforms.

## 6.4 Platforms as environments

The preceding examples were all specialized platforms that support important but limited kinds of interaction. The more significant community-level multi-sided platforms are those that structure economic interaction itself. The two most important are (a) the monetary and banking system and (b) the laws of commerce.

By factoring out the economic notion of value, the monetary system is the multi-sided platform that allows economic value to be abstracted, stored, exchanged, and transformed.

Similarly, the laws of commerce (and its associated judicial and police system) is the multi-sided platform that enables economic agreements to be made and transactions to occur—both with an increased level of confidence and security. This latter multi-sided platform has factored out what would otherwise be the need on the part of the participants to establish their own enforcement mechanisms.

As a society we clearly believe that these platforms should be controlled by the government and not by commercial organizations.

## 6.5 Natural language as a platform

An even more pervasive platform is language itself. Our natural languages provide us means to interact. Language as a platform is different from most of the other platforms we have discussed in that it is implemented by each of us individually.

Natural language is like a standard in that no one entity provides an implementation. It is like a standard in that we have dictionaries and grammar books and “standard English” reference implantations. But clearly it is not a fixed standard; nor is it a standard whose precepts change only with the approval of the standardization committee.

Natural language is also like open source in that its evolution depends on a large community of contributors. But it is even less structured than open source, most of which is controlled by a small group of top-level developers.

As indicated above, the natural language platform is implemented by each

of us individually. We each spend the first six years of our lives learning how to do that. Even though we are certainly not completely consistent about our private implementations, it is really quite amazing that we all do it as consistently as we do and that it is as successful a platform as it is.

## 6.6 Multi-sided platforms and systems engineering

Like levels of abstraction in general, multi-sided platforms should be central to systems engineering. Unfortunately, they tend not to be. In systems engineering we tend to focus on pair-wise communication among systems components. Often that pair-wise communication is hierarchical; sometimes it is horizontal. But in either case, we don't think about factoring out any of the functionality built into that communication.

When working on interaction we often write what are called interface control documents (ICDs). But like the interfaces they specify, these documents are defined on a pair-wise basis.

Recently, the notion of net-centric operation has gained significant traction. That notion is built on the idea of the network as a platform. This, of course, is a very powerful idea—one that was inspired by the Internet—and one that we applaud. But the network as a platform is just one example. In software platforms abound. As we said above, most file types correspond to platforms. It's time for systems engineering to begin to conceptualize systems in terms of (a) levels of abstraction in general and (b) platforms in particular. We return to this theme below when we discuss service oriented architecture.

## 6.7 Multi-sided platforms as dynamic entities

All multi-sided platforms are dynamic entities. As such they must extract energy from their environment to persist. Platforms such as Microsoft Windows, eBay, and regulated monopolies extract that energy by making a profit on their operations. Platforms provided by the government are supported by taxes and usage fees. At the other end of the spectrum the (much smaller amount of) energy needed for the persistence of standards making bodies is contributed by the individuals and corporations who see it as in their interest that the standard continue to exist as a non-commercial enterprise.

## 6.8 Platforms as both entities and environments

As a level of abstraction, a platform is an entity. However, as our discussion makes clear, platforms are also environments—or at least parts of environments. They are environments (or elements of an environment) for the parties that interact by making use of the platform.

In some cases, a platform is a complete environment. The platform consisting of the instruction and interrupt set of a computer establishes the complete environment for software that runs in that computer—although the instructions and interrupts provide access to the larger world within which the computer itself exists.

Similarly, the platform defined by a programming language establishes a complete environment for programs written in that language—with the same caveat as above.

More familiar complete environments are the platforms established by

(a) simulation and modeling environments and (b) online multi-player games.[38]

Because a level of abstraction, and in particular a platform, serves as both an environment and an entity, the level of abstraction (as a conceptual construct) is the fundamental notion in complex systems.

### *6.9 Platforms and science*

Much of science can be understood as an attempt to explain phenomena in terms of previously established platforms. The traditional hierarchy of the sciences is a simplified version of that perspective.

Earlier we referred to the search for background-free models of fundamental physics. Such a background free model will almost certainly be formulated as an entity that serves as its own platform/environment. What such a self-referential entity/platform will look like is still uncertain. Many programming languages, for example Lisp,[39] have been defined in terms of themselves. It's not clear to what extent that sort of self-definition can serve as a model for a self-referential entity/platform.

[38] There are well-known “leakages” from multi-player game environments. Participants trade virtual possessions for real money. They also meet in person outside the game environment.

[39] See John McCarthy's website documenting the history of Lisp: http://www-formal.stanford.edu/jmc/history/lisp/lisp.html.

## 7 Dissipative systems and dynamic entities

Systems engineers tend to build special kinds of entities which are intermediate between static and dynamic entities. Prigogene coined the term *dissipative system* (see, for example, 1997) for a static entity that exhibits regularities when energy is pumped through it.[40] Most of the widely cited examples of dissipative systems consist of relatively unstructured static entities that exhibit somewhat surprising structures—e.g., Rayleigh-Benard convection patterns—when they are forced to respond to energy inputs.

But virtually any static entity will exhibit some response to an energy flow—especially when that energy flow is both sufficient to have some noticeable effect on the entity and moderate enough not to destroy it. Much of what engineers build, e.g., automobiles and computers, are static entities whose (necessarily dissipative) responses to energy flows are in some way useful to us.

### *7.1 Dissipative systems vs. dynamic entities*

A dissipative system is intermediate between a static entity and a dynamic entity in that it consists of a static entity skeleton (which is more or less stable without an energy flow) through which one pumps energy. Dynamic entities do not have such stable static skeletons. Dynamic entities depend on their own ongoing processes to maintain their structures.[41] A living organism, a hurri-

[40] That's my summary of what a dissipative system (also known as a dissipative structure) amounts to.

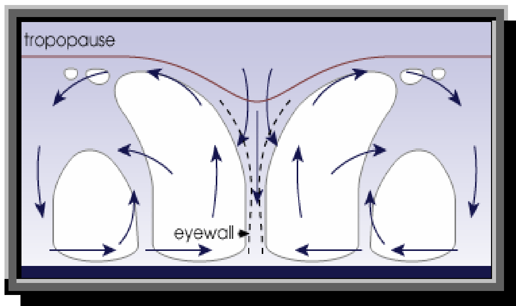
[41] See (Abbott, 2007) for a more detailed discussion.

cane, or a government would not persist even as a skeletal structure without a continual flow of externally supplied energy.

By working primarily with dissipative static entities engineers save themselves from having to build systems that are continually rebuilding and repairing themselves. But the price for that convenience is that the systems we build are not self-persistent.

## *7.2 Self-persistent dynamic entities*

To date, we don't know how to build systems that persist on their own. To the extent that we try it at all, our approach tends to be backwards: design a dissipative static entity and then add features to it that might allow it to repair itself. That isn't how naturally occurring dynamic entities work. Most naturally occurring dynamic entities are built to be self-persistent—although not eternal—from their core.




**A hurricane**
Original image from NASA

A hurricane, for example, maintains its structure simply because of how it is organized. The same is true for a government and a living cell.

Much of what goes on in a cell is cell maintenance—which is not the same as self-organization, which we said was a distraction. Here's how Frank Harold (2001) describes this aspect of a cell's functioning.

> Is the cell as a whole a self-assembling structure? ... Would a mixture of cellular molecules, gently warmed in some buffer, reconstitute cells? Surely not, and it is worthwhile to spell out why not. One reason is [that] assembly is never fully autonomous, but involves [pre-existing] enzymes or regulatory molecules that link [developing elements] to the larger whole. But there are three more fundamental reasons … First, some cellular components are not fashioned by self-assembly, particularly the … cell wall which resembles a woven fabric and must be enlarged by cutting and splicing. Second, many membrane proteins are oriented with respect to the membrane and catalyze vectorial reactions; this vector is not specified in the primary amino acid sequence, but is supplied by the cell. Third, certain processes occur at particular times and places, most notably the formation of a septum at the time of division. Localization on the cellular place is not in the genes but in the larger system. Cells do assemble themselves, but in quite another sense of the word: they grow.[42]

Compared to a cell a hurricane is somewhat ethereal or insubstantial—strange as that term may seem when applied to hurricanes. Hurricanes have no material skeletal framework that holds them together. Cells clearly do—although unlike static entities a cell's structural framework requires fairly frequent maintenance. Because a hurricane has no skeletal framework it would appear not to be a good candidate as a starting point for additional functionality: there

[42] Recall our previous discussion of self-referential entities. This passage illustrates how self-referential cells are.

are no attachment points for adding anything on.

Perhaps an even more significant difference between hurricanes and cells is that a hurricane has no DNA. Contrary to the popular image, DNA is not an internal representation of a cell's (or any biological organism's) design. A more appropriate way to think about DNA—along with the protein-manufacturing machinery that accompanies it—is as an internal "factory" that converts raw materials into the physical stuff out of which a cell's (or more generally a biological organism's) material framework is composed.

As Harold points out, DNA does not instruct a biological organism how to use the proteins that are generated; DNA is not a master plan for the cell. DNA and its associated mechanisms simply generate proteins—the disposition of which is taken care of outside the DNA mechanism.[43] Since hurricanes have no such skeleton they have no need for mechanisms that keep that skeleton in good repair.

It would seem that if we are to build self-persistent systems, the first step is to learn how to build minimal dynamic entities that have as their core (a) a means for converting available raw materials into the substances needed to create and maintain their physical structures and (b) mechanisms for using those generated materials for self-persistence.

This is quite a trick. A cell manufactures its own building blocks and then uses those building blocks to keep itself in good repair. A hurricane doesn't manufacture anything, but it does use the raw materials at hand (water vapor, rain drops, air, etc.) to maintain its structure. Since it doesn't manufacture anything, and since the materials at hand are fairly insubstantial as building blocks, a hurricane's structural framework is itself insubstantial.

After we learn how to build dynamic entities that have the ability to convert available materials into structural building blocks, then we can move on to adding additional functionality. Two examples of additional functionality oxygen transport and mobility. The DNA mechanism is used to produce hemoglobin, which is not a building block for cell structures. But once a mechanism exists for building proteins like hemoglobin and once a mechanism exists for transporting material throughout an organism, oxygen transport becomes possible. Similarly, cells have features that allow them to move themselves about in their environments—another bit of functionality that was added on to a cell's basic structure.

The lesson is that once the mechanism for producing building-block materials is in place, it then makes sense to exploit that mechanism to create new functionality.[44] But until we learn how to build basic self-provisioning dynamic entities

[43] Of course DNA and its associated mechanism are responsible for determining when particular proteins are generated—which also is a primary factor in determining how the proteins are used.

[44] In the section to follow we will see that the technique of building new capabilities on top of existing capabilities is one of nature's standard tricks—and one that we would do well to emulate.

we will be stuck with dissipative static entities.[45]

### 7.3 Wikipedia as a case study of a complex system

Wikipedia—the social entity consisting of the software and the people who keep it going—is a very public example of what may not at first seem like a traditional complex system. Yet it has all the properties of a complex system. It is multi-scalar. It includes (at a very broad level) the MediaWiki platform on which it runs, the Wikipedia conventions and Templates that make give it some overall consistency, the users who contribute content, and the users who make use of that content.

Furthermore Wikipedia, like all dynamic entities is both deployed and under development simultaneously. Like a biological organism, Wikipedia exists and functions in the world at the same time that it is undergoing development and self-repair. The MediaWiki software is open source software that continually being modified and extended. Similarly, Wikipedia content is also continually being extended at the same time that it is used by others.

Like most dynamic entities Wikipedia also has very effective mechanisms for self-repair. Editors and other users continually monitor pages for damage, which they repair very quickly whenever it occurs.

[45] Of course, like Theseus' ship, most of the large systems we build are embedded within social dynamic entities that provide for their maintenance—although we too rarely conceptualize our systems that broadly.

Wikipedia is also a nice example of a multi-sided platform. The two most obvious sides are the readers and the content providers. (At the MediaWiki level, the two other sides: the software developers and the web site as a software product.) Often a single person interacts with Wikipedia in different roles at different times. As a multi-sided platform, Wikipedia is also a system of systems. It may also be seen as a system of systems from the perspective of the various systems that keep it operational. These include the source code maintenance system, the mechanisms it uses to "serve" itself as web pages, and the Wikipedia Foundation, which plays a major role in governance, operation, and now fund-raising.

## 8 Service-oriented design

Much of what succeeds in nature consists of processes that build on other processes. Food web analysis illustrates how species depend on other species. Ecologies are built on seasonal cycles and resources flows (energy from the sun being the most basic but ocean and river currents being other examples). A species, a seasonal cycle, and a resource flow can all be understood as emergent phenomena.[46] In other words, nature builds new emergent phenomena on existing emergent phenomena.

When this happens in an ecological system, we call it *succession*[47]—a territory proceeds though a series of relatively stable stages.[48] At each relatively stable

[46] See (Abbott, 2006) and (Abbott, 2007).

[47] See, for example, http://www.mansfield.ohio-state.edu/ ~sabedon/campbl53.htm.

[48] This resembles what on an evolutionary scale we refer to as punctuated equilib-

stage, the species that populate that stage depend on each other and on the other aspects of the environment.

Progression occurs either because something disturbs the *status quo* and destroys some of the structures on which some of the participants depend or because there is an inefficiency in the system that can be exploited by some new mechanism.

This is pretty much the same picture one sees in a market-based economy. A collection of products, services, and community-supplied infrastructures (such as a monetary system, a postal system, a judicial system, etc.) develops into an ecology of mutual dependencies. Such a system remains stable until either a disturbance destroys something on which part of the system depends or a new way is found to use some of the available energy.

Natural ecologies and market economies are both examples of what we call innovative environments—which we discuss below. In this section we focus on how such environments work and how the principles underlying how they work may be applied to system design.

The fundamental principle of innovative environments is that new things are built on top of existing things. Because we have a well-developed transportation system, for example, we can produce products in one location and move them to other locations to be consumed—or otherwise used. One doesn't have to develop a transportation system from scratch in order to establish an off-shore production facility.

This web-of-interrelationships perspective has implications for systems engineering from two perspectives.

## 8.1 Products and services evolve

Even though most marketed products and services tend to originate as externalized thought, well-managed companies are always looking for new applications of their products—even applications that have little to do with the originally conceived market. In other words products and services evolve to fit their environments.[49]

Products and services that survive over the long term are not stuck attempting forever to implement the original vision of what they were intended to be. A product or service may have been born of externalized thought, but the original externalized thought is not considered a constraint on the evolution of the product or service. It's the environment that determines how a product or service will evolve.

In order for a product or service to evolve, its design must support change. If a system is designed in such a way that modification of that design is not feasible, it will die. Thus any system that is expected to survive over the long term must have evolvability as a primary design consideration.

rium. The difference is that in succession outside species replace existing species in a habitat. But the outside species are not generally created as new species.

[49] We have all encountered the now familiar version progression among software products. Version 5.0 is frequently quite different from version 1.0. It might even serve a significantly different customer base.

Unfortunately, we tend not to build systems this way. Customers often want a set of functional requirements satisfied as inexpensively as possible. Normally that entails sacrificing design flexibility and evolvability for a rigid focus on specific functionality.[50]

Furthermore, for a product or service to survive it must be robust. We don't build products for robustness except to the extent that the requirements specify a specific degree and form of robustness. Of course one aspect of robustness is to survive the unanticipated. So in some sense requiring robustness is a self-contradiction.

## *8.2 Products and services are built on top of an established base of other products and services*

The second and perhaps more significant lesson to be learned from the web-of-interrelationships perspective is that when building something new it's a good idea (actually more than just a good idea) to make use of existing products and services. This is quite different from how most of our systems work.

As we said earlier, we tend to build systems hierarchically. We formulate a top-level design that meets top level requirements and then determine what components we need to implement it. We then decide how to build the components in terms of sub-components, etc. This approach doesn't take advantage of existing products and services except when we use standard parts—and we do that too rarely.

A hierarchical design approach has (at least) two disadvantages. Firstly, it tends to result in what have been called stove-piped systems—systems that may work successfully on their own but that are very difficult to use in conjunction with other systems. That such a consequence will occur is quite understandable. When a system is built from the top-down without regard to what else exists, it is likely to be incompatible with other systems.

Secondly, the internal design of such systems tend to be rigid in the same way. Just as a hierarchically designed system isolates itself from other systems, the system components of a hierarchically designed system isolate themselves from other system components. Hierarchical design results in stove-piping both inside and out.

The alternative is to take advantage of what exists and build on top of it. In software there are now innumerable tools, frameworks, components, and libraries (both open source and commercial) that serve as the basis for further development.[51]

[50] This is one reason why it is a bad idea for a customer ever to buy a major system. If the system developer has a financial interest in seeing the system flourish over the long term, that developer will (presumably) design it to allow it to evolve. In contrast, a customer, who has no idea about these sorts of things, cannot define evolvability as a requirement. (We don't know how to do that in any case.) Even if the customer could require evolvability as a product property, he or she would probably not be in a position to exploit it. After all it is the developer who is on the lookout for new uses of the system, not the individual customer.

[51] The term *level of abstraction* is sometimes used to characterize the use of an abstract specification of a service as part of a design.

The prototypical example of building on top of existing products and services is a service-oriented architecture (SOA).[52] Service oriented architecture is a nice example because it illustrates how systems can be built on top of existing elements at both the system-to-system and internal design levels.

Through an SOA, systems can provide services for other systems.[53] Similarly system components can provide services for other system components through an SOA. In both cases, one is building on two foundations: (a) the network itself as a service (e.g., a level of abstraction) that all elements that reside on it use and (b) the design principle whereby elements provides service for each other.

It's a positive development that service-oriented and net-centric architectures are becoming desirable attributes within the systems engineering world.

It's important to remember that SOA and net-centricity are examples not principles. The principles are (a) build on top of existing capabilities and (b) conceptualize whatever one builds as a service that others will use, not as an end in itself.

[52] This is a design fad that has staying power. It's useful to think of our entire economic system as a service oriented architecture: every economic transaction is essentially a service transaction. The SOA nature of our economic system is one of the reasons it is both strong and agile.

[53] Hence SOA provides a natural framework for exploring system-of-systems issues.

## *8.3 Dynamic entities need energy to persist*

The second principle captures one difference between most systems engineered systems and systems that appear either in nature or in a market-based economy.

The systems we are talking about are almost all dynamic entities. They are not just static objects, they generally do something as a result of energy flows. Even static objects, such as a bridge, require maintenance. The real system is not just the static bridge. The real system is the bridge along with the maintenance process that keeps the bridge in good repair.[54] When understood from that broader perspective, it's clear that most of the systems that systems engineers build are dynamic entities.

Dynamic entities persist only as long as the energy that flows through them continues to flow. For a business, which is also a dynamic entity, money is a proxy for energy. A business exists only while the money flowing into it is at least as large as the money flowing out of it.

Unfortunately, it very rare that we ask ourselves about the energy flows needed for the persistence of the systems that we are asked to build. Long

[54] This perspective explains the Theseus' ship paradox. Is a ship maintained in port so long that all its parts have been replaced "the same ship" as the original? The answer is that the ship maintenance process is the same entity (even if it involves numerous people cycling though it—a property of entities). The physical ship is just a component of that social entity in much the same way as our (replaceable) bones are a component of ourselves as entities.

term energy flow considerations (i.e., funding) should be fundamental to any system development project. But it generally isn't. Because it isn't we don't think about systems in terms of what it would take to make them self-persistent.

This is not to say that every system must be profitable in the traditional sense of profitable. Many of our system, e.g., our judicial and monetary systems are both so central to the functioning of our society and so ill suited to be funded by their direct customers that we properly treat them as a commons. Commons too must be funded, but they are funded in different ways from most entities.[55]

But whether the system we are building is expected to be a commons or self-sustaining, we must understand from the start how the energy flow required to allow it to sustain itself will be provided.

In business the answer to this sort of question would be recorded in a business plan. In systems engineering we don't have a name for where we record answers to these questions because we rarely ask these questions.

# 9 Feasibility ranges

Emergence occurs within feasibility ranges. A visible and tragic illustration of this is the Challenger disaster in which the O-rings lost their (emergent) sealant property because the temperature was too low.

Since there are *always* feasibility ranges for emergent properties[56] we should make it standard practice to identify and determine the feasibility ranges of each emergent property we expect our system and system components to display. For each emergent property we should explain why its feasibility range won't be violated—and what happens if it is. Had this been done for the Challenger, we would not have lost our astronauts.

For computer and software systems, feasibility range concerns typically involve such issues as data rates, access rates (for quality of service issues), data storage demands, assumed data (and other input) ranges and limits, computational demands, accuracy assumptions, and precision needs. In software these are often lumped together as performance (as distinguished from functionality) issues.

Although many of these issues are not new, it is useful to see them as instances as the more general category of emergence feasibility ranges and to be aware that feasibility range issues arise throughout our systems.

Feasibility range issues are often orthogonal to other design considerations. The term *cross-cutting* is typically applied to such situations. In software, aspect-oriented programming (and in some cases creative application of constraint programming) may be used to handle cross-cutting issues. I am not aware of a standard approach for handling cross cutting issues in systems engineering.

# 10 Modeling and Simulation

## *10.1 For want of a nail …*

An important characteristic of most complex systems is that they are multiscalar. Every system that exhibits emergence exists on at least two scales, the scale at which the emergent property

[55] Elinor Ostrom (1990) began the modern era of understanding how successful commons function.

[56] See (Abbott, 2006).

appears and the scale at which the emergent property is implemented. Often there are many more scales. This is especially true when emergence is built upon emergence, as it often is. The poem telling the story of how a missing horseshoe nail led to the loss of a kingdom illustrates the potential significance of multiscale phenomena.

Much of the work in systems engineering relies on the results of simulations. We build models of possible system designs, and we run them, watching what happens as we vary the parameters.

Even with our advanced modeling and simulation capabilities, however, it would be virtually impossible for us to model all the nails in all the horseshoes on all feet of all the horses ridden by all the men in King Richard's army. Certainly we can't do anything remotely like that if we were to model today's massively larger systems.

Since we depend profoundly on simulations, and since we are unable to simulate our intended systems at the many scales at which we build them, and since multi-scale phenomena pose a potential threat to successful systems engineering, what are we to do?

This is a major research issue and one for which I have no answer. In (Abbott, 2006b) I called this the difficulty of looking downward. The first step, though, is to recognize that we have a serious problem.

## *10.2 For want of imagination …*

Imagine (unrealistically) that we were able to simulate our air transportation system and everything else relevant to how airplanes are used and maintained in this country. Would that capability have helped prevent 9/11?

My answer is "No." The problem is that we have no idea how to build simulations that can identify emergent phenomena—or even more difficult, how to identify the possibility of emergent phenomena.

Earlier we urged that new systems be built on top of existing capabilities. That's exactly what the 9/11 terrorists did. They used the capability provided by the airlines of carrying and delivering large amounts of explosive material to virtually any location within the country. All the terrorists had to do was to take over the planes at the critical times—a brilliant example of using an existing capability to produce a new capability.

We know how to write simulations in which emergence occurs. Any agent-based model is capable of fostering emergence. But we don't know how to write simulations that will recognize that emergence has occurred and issue a report about it. In (Abbott, 2006b) I called this the difficulty of looking upward.

This is a nice illustration of the difficulty of upwardly predicting emergence. Let's return to our fully accurate simulation of our air transportation system. Suppose it included (a) airplanes accidentally crashing into buildings and (b) air hijackings. Perhaps in such a virtual world, a hijacked airplane accidentally crashed into a building, destroying it. Even so, it is difficult to imagine that the simulation would be able to predict the intentional hijacking of an airplane for the purpose of crashing it into a building.

A system might be able to make such a prediction if it were (a) programmed to look for instances of significant destruc-

tion (and categorize the accidental crash as such an instance), (b) able to conclude that the accidental crash could also be caused intentionally, and (c) aware of the possibility of suicide actions.

The ability to make those observations, draw those inferences, and predict a 9/11 type of attack goes far beyond what one would normally find in an air transportation simulation—and probably far beyond any system so far yet developed.

# 11 Software that generates new ideas?

In this section we explore what might be required to build a system that could generate the idea of attacking the World Trade Center with hijacked airplanes.

I know of no system that is capable of generating new ideas. For all the advances we have made in externalizing particular modes of thought and conceptual models, we do not yet know how to externalize the idea of an idea in anything like its full richness.

In saying this we are assuming that (a) ideas themselves exist only in the minds of their thinkers, i.e., only as subjective experience and that (b) computers don't have subjective experience. An immediate consequence of this is that computers don't have ideas as we understand them. So the best we can possibly hope for with current technology (and with any technology that we can currently envision) is that we might be able (a) to externalize the process of generating new ideas and (b) to execute that externalized process as software.

To do this we would have to find a way (a) to represent the idea of an idea and (b) to generate new ones artificially. To use the example from the previous section, we would have to develop a computer systems that could come up with an idea such as, "let's use a commercial airplane as a weapon to be wielded by a suicide hijack crew."

To accomplish this, four technologies would have to be brought together: knowledge representation, ontology generation, modeling and simulation, and exploratory search.

The question of how to represent ideas in general has long been a subject of study within computer science. Brachman (2004) provides a survey of the current the state of the art of knowledge representation. Most of the material in that book is well known. It's surprising how disappointing and stale it seems. As well as we have done in building computer systems that externalize particular realms of thought, we have done surprisingly poorly at externalizing thinking as such. There is nothing in Brachman (or anywhere else) that suggests that we have any new ideas about how to write a computer program that can represent ideas in general.

If we think of knowledge representation as a structure for representing ideas, we need a way to populate such a knowledge representation database. That's the subject matter of ontology. Ontology, of course, is as old as philosophy. What is needed here is an ontology that is both rich enough to have the potential to be the source of new ideas and flexible enough to be able to incorporate new ideas as they are generated. The two most active projects in this area are the various Semantic Web projects and

Cyc[57]. These show promise, but none seem mature enough to be put to work in generating new ideas.

The third area is modeling and simulation. Once one has an ontology captured by some knowledge representation formalism, to make real use of it, one needs more than static information.[58] But executing a generic ontology is far beyond our current state-of-the-art.

To take a simple example, we are not currently able to simulate a multi-level model of a diamond—a simple static entity. On one level the simulation would illustrate how the diamond is held together as a lattice by atomic forces. On a second level the simulation would illustrate how the diamond as a whole moves through space. A third and even more difficult combination of these levels would show how a diamond can be used to cut glass. Since a diamond is a relatively simple static entity, imagine how far we are from being able to build adequate multi-level simulations that involve dynamic entities. Imagine, for example, building a simulation of an evolutionary arms race in which insects and plants compete with each other by growing bark, evolving a bark boring capability, or evolving toxic compounds.

[57] See http://www.w3.org/2001/sw/ and http://cyc.com/ respectively.

[58] Cyc contains lots of static information about its subject matters. But it has no way to execute operations that the elements of its database are able to perform. Perhaps for that reason Cyc seems particularly weak in its catalog of verbs.

Finally, exploratory search, e.g., genetic algorithms and genetic programming,[59] is needed to allow our system to explore various possibilities and come up with ones that achieve its objectives.

Work in all four of these areas is ongoing, but I am not aware of any current projects that attempt to integrate these area in a system that would be powerful enough to generate an idea such as the one that resulted in the destruction of the World Trade Center. To do so would achieve the original grand dream of artificial intelligence. We are still far from that goal.

## 12 Innovative environments

We end this survey on a positive note. As we have seen, emergence occurs in a wide range of situations. Four environments that are justifiably celebrated for an outpouring of emergent phenomena are the Internet (in particular the World Wide Web), the U.S. (and now the global) market-oriented economic system, our system of scientific research, and biological evolution.

Although quite diverse in their underlying domains, all four have been extraordinarily fruitful and have fostered an ever-broadening flow of innovative products, services, and other elements.[60]

[59] See, for example, http://www.aaai.org/AITopics/html/genalg.html.

[60] Transformation in the Defense Department—including capability-based acquisition, net-centric operations, and service oriented architectures—has been motivated at least in part by a desire to produce similar benefits within the DoD.

Although it is widely believed that environments that enable and facilitate emergence share some common characteristics, we have no universally accepted list of exactly what those characteristics are or why they matter. Whichever characteristics appear on a final list—if there is a final, definitive list—the following (or variants thereof) are likely to be candidates.

1. **Access to a supply of externally provided energy and means for exchanging it.** All environment that foster emergence are what is commonly known as *far from equilibrium*: externally supplied energy continually flows through them. The overall creative process can be summarized as consisting of finding increasingly innovative ways of using the available energy. To facilitate this process, mechanisms must be available to support the fungibility of energy and its proxies such as money, power, and attention.
2. **Standards.** New products, services, and other items are almost always created from existing products, services, and other items. Composition is greatly facilitated when the elements to be composed adhere to widely accepted standards. Standards facilitate the composition of products and services to produce new products and services.
3. **Communication and transportation infrastructures.** Communication and transportation infrastructures facilitate the exchange/transfer/flow of (a) information throughout the environment and (b.1) energy (in one direction) and (b.2) products and services (in the other) among trading partners.
4. **A reasonable level of confidence in the stability and continuity of the products and services installed in the environment.** Mechanisms must be available to allow agreements to be made and for installed products and services to be relied upon.
5. **Minimum overhead.** Cultural or other mechanisms must exist to discourage corruption along with enforcement mechanisms to make it harder to siphon off energy flows for non-productive uses. More generally, the environment must incorporate mechanisms that minimize the overhead of participating in the environment.
6. Both (a) **centralized but quasi-democratic and transparent governance** of the overall system, its infrastructure, and the standards making process and (b) **decentralized overall control** ("power to the edge") in which as much autonomy as possible is ceded to environment participants.
7. **Mechanisms that ensure that a certain amount of the available energy is devoted to the exploration of the space of possible new elements.** There must be some means to encourage the exploration of new possibilities.
8. **Mechanisms that allow new products and services to be developed and installed in the environment and then made known** to other participants in the environment.
9. **A primarily bottom-up means for allocating energy** (or its proxies) according to use: the more (less) useful a product or service is found to be (according to actual usage),

the more (fewer) resources it will have at its disposal. This implies a market-like means for allocating most of the resources available in the environment. All of the participants in the environment must be self-sustaining in terms of their overall energy transactions. Since the environment itself is predicated on an external source of "free" energy, this should be possible.

10. **An ability to form communities of interest** (formal, informal, voluntary, and fee-based) to facilitate the sharing of information, experience, and expertise. The value of shared information is typically enhanced when it is shared in groups.

11. Both (a) **sufficient stability of the overall environment** that participants can establish regularized modes of participation and (b) (generally collaborative) **means to allow the environment to evolve** as conditions change. This implies treating the environment as a commons and finding a successful way to govern it as such.[61]

Innovative environments are important to systems engineering for at least three reasons.

1. We want the systems we build (or at least many of them) to be innovative environments. Look at how the example of the internet has inspired transformation in the DoD. We want the other systems we build to further enable that vision and to provide additional innovative environments for our customers.
2. We want our own processes to be innovative. As we build systems, we want to encourage innovation among our analysts and developers.[62]
3. We want our own intellectual environment to be innovative. Systems engineering is constantly innovating; it has never stood still. This symposium is an example of continued vigor. We want to encourage innovation in our systems engineering community.

As we understand more about how to make environments innovative, we will become more and more successful in achieving these goals.

# 13 Summary

In this paper we have taken a brief tour of the landscape of emergence and explored how it may be useful to systems engineering. We hope that the ideas presented here will be useful to the systems engineering community.

**References**


Abbott, Russ, (2006a), "If a Tree Casts a Shadow is it Telling the Time," *International Conference on Unconventional Computation*.

Abbott, Russ, (2006b) "Emergence Explained: Abstractions," *Complexity*, 12/1 (September-October).

Abbott, Russ, (2007) "Emergence Explained: Entities," in preparation.

Briscoe, Bob, Andrew Odlyzko, and Benjamin Tilly, (2006) "Metcalfe's Law is Wrong," *IEEE Spectrum*, July 2006. (Available online at:


[61] Ostrom's work on commons (1990 and more recent work, not cited) is directly relevant here.

[62] See Horowitz (this conference) for an example.

http://www.spectrum.ieee.org/jul06/4109.)

Brachman, Ronald and Hector Levesque (2004) *Knowledge Representation and Reasoning*, Morgan Kaufmann.

Chaitin, Gregory J. (2003) *From Philosophy to Program Size*, Institute of Cybernetics, Tallinn, Estonia.

Evans, David S., Andrei Hagiu, and Richard Schmalensee (2006) *Invisible Engines: How Software Platforms Drive Innovation and Transform Industries*, MIT Press.

Harold, Franklyn M. (2001) *The Way of the Cell: Molecules, Organisms, and the Order of Life*, Oxford University Press.

Hoare, C. A. R. (1969) "An axiomatic basis for computer programming". *Communications of the ACM*, 12(10):576–585, October 1969. (Available online at: http://www.spatial.maine.edu/~worboys/processes/hoare%20axiomatic.pdf.)

Holland, John H., (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor.

Horowitz, Barry (2007) “Self-Evaluating Agile Large-Scale Systems: SEALS,” Symposium on Complex Systems Engineering.

Humphreys, Paul (1997) “Emergence, Not Supervenience”, *Philosophy of Science* 64, pp. S337-S345.

Iansiti, Marco and Gregory, (2007), “The Business of Free Software: Enterprise Incentives, Investment, and Motivation in the Open Source Community,” (draft) [Available online at: http://www.hbs.edu/research/pdf/07-028.pdf.]

O'Connor, Timothy, Wong, Hong Yu "Emergent Properties", *The Stanford Encyclopedia of Philosophy* (Winter 2006 Edition), Edward N. Zalta (ed.), forthcoming URL = <http://plato.stanford.edu/archives/win2006/entries/properties-emergent/>.

Ostrom, Elinor, (1990) *Governing the Commons: The Evolution of Institutions for Collective Action*, Cambridge University Press.

Prigogine, Ilya and Dilip Kondepudi, *Modern Thermodynamics: from Heat Engines to Dissipative Structures*, John Wiley & Sons, N.Y., 1997.

Scerri, Eric (2006), *The Periodic Table: Its Story and Its Significance*, Oxford University Press.

Shalizi, Cosma R. (2001), *Causal Architecture, Complexity, and Self-Organization in Time Series and Cellular Automata,* Ph.D. dissertation, Physics Department, University of Wisconsin-Madison.

Silver, Pamela A., (2007) “Why we need systems biology,” Symposium on Complex Systems Engineering.

Smolin, Lee (2006) *The Trouble with Physics*, Houghton Mifflin Company.

Tennent, R.D. (1976), “The Denotational Semantics of Programming Languages,” Communications of the ACM, 19(8):437-453, August 1976. (Available online at http://www.csc.liv.ac.uk/~grant/Teaching/COMP317/densem.pdf.)

All trademarks, service marks, and trade names are the property of their respective owners.