Challenging the Computational Metaphor: Implications for How We Think

Lynn Andrea Stein

Massachusetts Institute of Technology and the Bunting Institute, Radcliffe College

Abstract

This paper explores the role of the traditional computational metaphor in our thinking as computer scientists, its influence on epistemological styles, and its implications for our understanding of cognition. It proposes to replace the conventional metaphor—a sequence of steps—with the notion of a community of interacting entities, and examines the ramifications of such a shift on these various ways in which we think.

1 Computation's Central Metaphor

In every endeavor that we undertake, we rely on a set of implicit or explicit principles that guide our understanding and shape our course of action. In computer science, one such idea is what I will call the computational metaphor. The computational metaphor is an image of how computing works—or what computing is made of—that serves as the foundation for our understanding of all things computational. Perhaps because computation plays a central role in modern culture, the computational metaphor extends beyond computer science and plays a role both in other disciplines and in our everyday understanding of the world around us.

This paper addresses the need for a fundamental shift in the computational metaphor. This shift is motivated by changes in the nature of computation as practiced and the inefficacy of the traditional computational metaphor in describing current practice. We are standing at the cusp of what Thomas S. Kuhn called a paradigm shift, in which the very foundations of our field are being reconceived. This has profound implications for all aspects of our understanding of computational science and, given its central role in modern thought, for our broader understanding as well.

What is the computational metaphor? I think that it goes something like this:

> Computation is a function from its inputs to its output. It is made up of a sequence of functional steps that produce—at its end—some result that is its goal.

This is what I was taught when I was trained as a computer scientist. It is a model that computer scientists by and large take for granted. It is something the members of the field share. Sometimes we refer to it as Turing's or von Neumann's model; both men were influential in elucidating this particular way of thinking about computation. Although Turing's machine was abstract and von Neumann's concrete, each outlined a mechanism of execution that was strikingly centralized, sequential and result-oriented (Turing 1936; von Neumann 1945).

Figure 1 depicts this image of computation iconically, highlighting several of its major features. Computation is composed of steps. These steps are combined by temporal sequencing. The computational process is evaluated by looking at its result (and, to some extent, its resource utilization). To a first approximation, the computation's result characterizes the computation.[1] Throughout this paper, I shall refer to this as the calculation model of computation.

This computational metaphor is an image by which we understand our field. It is a common reference model that encodes how computer scientists see what we do and how we think about our work. Computer scientists frequently use this model even when talking about systems that are not, strictly speaking, sequential.[2]

The traditional computational metaphor preaches that—for almost all purposes—there is a single thread of control and the programmer owns it. The programmer’s problem is to figure out what happens next. The metaphor encourages us to ignore the fact that computers are actually built out of analog components. It obscures the fact that each component is fallible. It hides the ways in which the computer is physically coupled into the world, and largely ignores any systems—social or mechanical or otherwise—within which the computation is embedded.

As seen through this sequentialist metaphor, computation is a sort of glorified calculation. A computer is, in this view, an extremely sophisticated (symbol-processing) abacus. Historically, this is entirely appropriate. The first “computors” were people assigned to perform mathematical calculations; mechanical computers were machines that simulated the mathematical processing of these human calculators. Although some early computers were used in actual physically coupled control systems, more frequently they were used to provide data to human decision makers.

The calculation model of computation goes hand-in-hand with the idea of black box (or procedural) abstraction. This is the equation of a computation with the functional result that it computes over its input. Black-box abstraction is a powerful technique that permits reasoning about systems at a fairly high level, e.g., combining functional pieces without considering the details of their implementations. Without black-box abstraction, it is difficult to imagine that much of the history of modern software development would have been possible.

[1] This is, of course, an exaggeration. Both resource utilization and side effects play a significant role in our evaluation of a computation. Nonetheless, the functional behavior of a computation is generally taken to be its most salient identifying characteristic. Further, taking a suitably broad reading of the term "result"—subsuming both resource utilization and side effects—does not materially affect the point I wish to make here.

[2] In an otherwise forward-thinking book, Carriero and Gelernter explicitly equate programming and problem-solving at the beginning of their *How to Write Parallel Programs* (1990). This is in spite of the fact that the book discusses techniques that the authors believe to be at least as well suited to interactive, distributed, and endpoint-less computation.

A corollary of this approach is modular-functionalism. Since a computation is defined in terms of the functional result that it computes over its input, each piece of calculation can be identified with its associated function. Modular-functionalism is a method for constructing systems in which the problem is decomposed into its constituent functions and a solution is constructed by sequencing calculations associated with each of the constituent functions. That is, the structural decomposition of the program is identical to the functional decomposition of the problem that it solves. Within the context of computation-as-calculation, modular-functionalism seems almost tautological. Later in this paper, we will see an alternate view of computation in which modular-functionalism is only one possible approach.

This approach is also consonant with Turing's abstract machine, with its step-at-a-time processing, or von Neumann's architecture for manipulating memory-based data. Hendriks-Jansen (1996) has observed that Turing worked at the time of, and was quite possibly influenced by, the advent of assembly line manufacturing. Von Neumann maintained a clean separation between instruction and data while placing both squarely within the machine's memory. This prefigures the 1960s and 70s notions of pure "thought" (process) operating on memory-based data, and stands in stark contrast to the messy intertwining more typical of 1940s and 50s cybernetics and early control systems.

2 The Power of the Paradigm

The tremendous simplifications afforded by the traditional computational metaphor were historically crucial. From the digital abstraction to procedural abstraction, from high level languages to abstract program design, the conceptual vocabulary of computer science today was facilitated by the computational metaphor. As a consequence, computer technology has revolutionized much of the world.

Certainly, the computational metaphor enabled computer science to focus on the organization of sequences of steps into larger functional units without worrying about transient voltage levels or multiple simultaneous transitions within the hardware. This way of thinking about computation also let us ignore the occasional power fault, the mechanical misfire, delays in operator feedback, or other human activities. By hiding the details of hardware's actual behavior behind artifices such as the digital abstraction, tremendous advances in computational science were made possible.

If we had to directly manage the variations in voltage across each individual component in our computer's motherboard, we would be hard pressed to write even small programs. The digital abstraction—looking at those voltages only when in stable configurations and even then regarding them as ones and zeros—allows us to treat the computer as a discrete logical artifact. The von Neuman machine architecture and, in turn, higher level programming languages, were subsequent advances that removed us still further from the actual analog machine. Memory mapping is a technique that allows us to access complex peripheral devices—often containing their own processors—as though they were simple storage cells.

Or consider parallel programming. Parallel processors today are benchmarked on how well they emulate the ideal of a sequential machines (while improving its performance).[3] A particularly extreme example of this approach is automatic program parallelization. This is an attempt to make multiple processors look, to the programmer, like a single sequential machine of much the sort that the computational metaphor prescribes. That is, it is a way to enable a programmer to harness the power of multiple processors without ever knowing that there are many things going on at once.

The structures contributed by Turing's and von Neumann's machines were tremendously empowering during computation's first half-century, and not just for computer science itself. The computational metaphor also plays a major role in shaping how we, our students, and colleagues in other disciplines, see computation. As some of today's most advanced technology, computation has become a significant model system for interpreting many complex phenomena. It influences how we—all of us—see the world. It is a filter through which we view everything from cognition to economics to ecology. Whether or not we impute explicit computation to these phenomena, we often conceptualize them in computational terms. In certain disciplines—from the cognitive sciences to fields as disparate as organizational science and molecular biology—computation has become a central metaphor for organizing work in the field. It dictates the questions we ask and the answers we are able to imagine.

In a variety of disciplines, the computational metaphor has had equally striking impact. For example, in molecular biology, Keller (1995) points to an analogy between what in biology is called the central dogma and what I here call the computational metaphor. The central dogma (Crick) refers to the idea that DNA is the blueprint for RNA, which in turn produces cytoplasmic proteins, in an unremittingly unidirectional process. She argues that the central dogma has been modeled as a theory of information much like computation and that biology's notion of nuclear regulation of the cell owes much to computational notions. Traditional organizational science, with its notions of centralized, hierarchical control, is similar.

This way of thinking—this computational metaphor—has serious implications for how our own students learn to think. Beginning programmers have historically been taught to decompose problems logically into sequences of steps. Primary school mathematics instruction emulates programming, including the teaching of "algorithmic thinking" in elementary schools. Goal-directed, endpoint-driven planning is seen as preferable to a more fluidly serendipitous exploration (Lawler 1985; Papert 1980). This approach privileges certain epistemological styles over others and leads successful students to fluency with one particular set of techniques, discouraging others.

The traditional computational metaphor affects our models of thinking as well. In "Good Old-Fashioned Artificial Intelligence" (GOFAI), the single-minded result-oriented model predominates. Newell and Simon's (1972) "Physical Symbol System Hypothesis" gave

[3] But see (Hillis 1989) for a competing view.

credence to the idea that the brain was appropriately modeled as a computer or, more specifically, as a symbolic calculator. The emphasis in much of the early days of AI was on problem solving, game playing, and other forms of “puzzle-mode” intelligence. That is, early AI systems were concerned with well-defined problem domains in which the task of an ostensibly intelligent system was to deduce the correct answer from the problem as presented: cognition as computation as calculation.

During the 1970s, nearly the whole of cognitive science followed, adopting this computational model as a prototype for human cognition. In its most extreme form, this view revolves around the notion of "thinking" as a separately modularized system communicating with perceptual and motor "peripherals". (Dennett and Kinsbourne 1992) refer to this approach somewhat disparagingly as the "Cartesian Theater", the place where it all comes together. This is the cognitive analog of the computer's central processing unit distinct from von Neumann’s memory module.

3 Computation In Crisis

The computational metaphor was tremendously valuable in the first half-century of computation’s history, fostering emulation across a wide range of fields. But while it was both empowering and arguably essential for the early progress of the field, it was never completely true. Initially, it was an extremely useful way of making sense of the many constituents of a dynamically interacting community of hardware. By focusing on activity within the CPU, insisting on the purity of the digital abstraction, and ignoring the vicissitudes of erratic hardware components and I/O devices, we were able to progress from electrical engineering to a discipline of discrete and controllable algorithmic computation. Innovations like timesharing made it possible for multiple users each to operate under the assumptions of the central metaphor (while in fact the virtual machines on which these users worked were artifacts established by a more complex underlying program). The few overt exceptions to the central metaphor—hardware interrupts, networking, and eventually parallelism—were assiduously hidden from the end user.

Increasingly, however, the traditional computational metaphor limits rather than empowers us today. It prevents us from confronting and working effectively with computation as it actually occurs. This is true both within computer science, which still clings fervently to the metaphor, and in other disciplines where dissatisfaction with the computational metaphor has in some cases caused an anti-computationalist backlash.

With the advent in turn of timesharing systems, of smart peripherals, of increasingly networked computers, and of computational boxes containing more than one central processing unit, the single-minded myopia of the traditional metaphor has become less and less viable. It has become increasingly difficult to argue that things outside the program itself "don't matter". The introduction of the activity of another user on the same timesharing system did impact computation, and the virtual machine model went to great lengths to minimize this interaction and so sustain the traditional metaphor for another two decades or so. But rigid adherence to the computational metaphor can impede the progress of software engineering. More and more, computations involve multiple virtually or actually simultaneous activities.

Today, computations that involve coordinated concurrent activity are poorly explained in terms of the traditional computational metaphor. Those computations that necessarily take place across multiple computers, such as the world-wide web, are only the most visible examples. If we construe a computation as a sequence of steps designed to produce a result, what is it that the world-wide web calculates? What are its constituent functions? While these questions are a clear mismatch for the behavior of the world-wide web, they are hardly more appropriate for virtual reality, an autopilot system, or almost any other computation of interest today. Even word processing now involves the logically concurrent execution of formatting, layout, spelling and grammar checking, and—coming soon to a PC near you—bibliographic suggestions or other “agent based” assistance.

A concrete example of the limitations of the calculation metaphor was provided by a senior developer at a major software company. Although his company is able to hire some of the best computer science graduates produced, he complained of difficulty finding students who can write programs in which many things are happening simultaneously. I originally assumed that he meant that their new hires had difficulty with some of the finer points of synchronization and concurrency control. He corrected this impression, explaining that his problem was “journeymen programmers” who didn’t know how to *think* concurrently.[4] Our students are learning to decompose and solve problems in a way that is problematic even for today's software market.

Today, the line between hardware and software is blurred beyond recognition. We can now construct almost any computation either in silicon or in software; we choose based on the needs of the particular application. Custom silicon makes any program realizable in hardware. Field-programmable gate arrays promise software-like flexibility in hardware itself. (Waingold et al. 1997) have even suggested dynamically reconfigurable silicon. The von Neumann architecture is no longer the clear choice, though any report of its precipitous demise would surely be exaggeration. Any new metaphor for computation must give equal precedence to hardware and software implementations, to traditional architectures and novel ones as well.

If the hardware-software line is fading, the line between the computer and its environment is following rapidly. In ritualized or regimented transactions, we are increasingly replacing people with computers. Computers answer our telephones, supply our cash, pay our bills, sell us merchandise. Computers control our cars and our appliances. They cooperate and collaborate with one another and with the world around us. The traditional metaphor, with its “what happens next?” mentality, leaves little room for the users or environmental partners of a computation. A new theory of computing must accommodate this fluidity between computer and user and physical environment.

[4] Bob Atkinson, personal communication.

The traditional computational metaphor is also problematic as a guiding epistemology. In our classrooms, certain styles of thinking and understanding are discouraged for deviating from the unrelenting sequentialism of the computational metaphor. Turkle (1984; Turkle and Papert 1990) studied how programming is presented as a rigidly linear, sequential, and logical process. For some students—those she identifies as bricoleurs or tinkerers—this way of decomposing problems is uncomfortable. These students prefer to experiment with partial programs, piecing them together to build larger structures only as they become comfortable with how they interact. Frustrated by black-boxing and modular-functional linear design, many of Turkle's tinkerers abandoned computer science. Those who did remain succeeded by suppressing, or at least hiding, their epistemological style. Turkle notes that this style is disproportionately observed among female students, giving rise to one possible explanation of the differential representation of women within the field. Further, componential tinkering may be precisely what is needed in today’s toolkit- and library-rich programming environment.

Early historical attempts to capture computation and computation-like processes were not exclusively sequentialist (nor were they exclusively digital). Work in fields such as cybernetics was contemporaneous with the early days of computation, but has not become a part of computer science's legacy. (See Weiner 1948; Ashby 1954, 1956.) Cybernetics took seriously the idea of a computation embedded in and coupled to its environment. These were precisely the issues suppressed by the computationalist approaches. In the intellectual battles of mid-century, cybernetics failed to provide the necessary empowerment for the emerging science of computation and so was lost, dominated by the computational metaphor. The nascent field of computational science was set on a steady path, but its connections to the world around it were weakened.

It was this disembodied information-processing approach that gained prominence both within computer science and as a reference model for neighboring disciplines. Research in cognitive science today is still defined either within it or in opposition to it. But its influence is declining. Recent perspectives on artificial intelligence and cognitive science have accepted that GOFAI’s puzzle-mode intelligence is only one, and probably not the central, form of intelligent behavior.[5]

Increasingly, a more communal, contextual, interactive approach to cognitive science is coming into its own, particularly among those whose research is informed by neuroscience. These new-school cognitive scientists reject the traditional metaphor’s centralized architecture, in some cases rejecting computationalism as a result. Scientists like Smithers (1992), Port and van Gelder (1995) and Beer (1995) have even begun to rediscover—and sometimes reinvent—the work of the cyberneticists. They argue that dynamical systems provide a promising route to understanding and building intelligent systems. Any new computationalist theory of intelligence must provide the infrastructure to reconcile their advances with more traditionally computational theories of intelligence.

[5] For a clear articulation of this argument, see (Agre 1988), (Brooks 1991), or (Clark 1997).

Molecular biologists now accept that their equivalent to our computational metaphor, Crick's central dogma, DNA to RNA to cytoplasmic protein, is only an approximation. In fact, there are many examples of feedback along this path. Keller (1983) argues that one early recognition of this feedback—McClintock's work on transposition, involving cytoplasmic influence in the production of RNA—was made relatively inaccessible to molecular biologists as long as they insisted on rigid adherence to the central dogma. Schuman (1998) describes how the dogmatic assumption of centralized—nuclear—control obscured existing evidence for cytoplasmic protein synthesis in hippocampal learning. Keller argues that biology's reliance on the sequential information processing metaphor has limited its disciplinary vision. I would argue that some of computation's "central dogma" similarly blinds us to some of the truths of modern computer science.

4 Changing the Metaphor

Today's computations are embedded in physical and virtual environments. They interact with people, hardware, networks, software. These computations do not necessarily have ends, let alone results to evaluate at those ends. They are called agents, servers, processors, entities. As an evocative example, consider a robot. For a robot, stopping is failure. A robot is not evaluated by the final result it produces; instead, it is judged by its ongoing behavior. Robots are interactive, ongoing, partners in their environments.

But this observation is not limited to such obviously animate computations as a robot. Consider a video game, a spreadsheet, an automobile's cruise control system, a cellular telephone network. Like robots, these computations are interactive. What we care about is their ongoing behavior. We do not wait for some hypothetical endpoint to decide whether they have done the right thing, past tense. Instead, expect them to work with us (or with each other, or with our automobile or toaster oven). When we sit down at the computer, we may well have goals. What we expect from a computer is not that it fulfill this goal independently (i.e., compute a "result") but that it cooperate and collaborate with us in our tasks.

If the traditional computational metaphor (as depicted in Figure 1) is computation as calculation, I would argue instead for something one might call computation as interaction. This is illustrated in Figure 2. Time again runs vertically, but in this illustration, an additional spatial dimension has been added. The bars in the figure are intended as spread out over space, with arrows representing communication from one entity to another. Again, the figure is a schematic illustration highlighting some of the main features of this model.

The pieces of this model are persistent entities coupled together by their ongoing interactive behavior. It is the behavior of the whole system that constitutes grounds for evaluation of that system. Beginning and end, when present, are special cases that can often be ignored. The focus is on interactions among these entities. The computation cannot be said to reside in any one of the entities; instead, it is the result of the interactions among them. In the calculation model, inputs come at the beginning; outputs are produced at the end. In the interactive model, inputs are things you monitor; outputs are things that you do. The computational system is open: it may influence, or be influenced by, things outside of the system as depicted.

If today's computations are interactive, we need a way to think about how to build such systems that corresponds to the traditional story but encompasses this richer metaphor. Such a metaphor will be the starting point for thinking about a wide range of disciplines. I wish to suggest that an appropriate metaphor is that of a community of interacting entities. Computation is an entity with ongoing interactive behavior (i.e., providing ongoing services). This entity may itself be made up of entities interacting to produce its behavior. The problem of programming is the problem of designing this community: Who are its members? How do they interact? What goes inside each one?

Many computational systems already incorporate some aspects of this decomposition. Examples include robots, user interfaces, embedded systems, video games, web browsers, control systems, information access utilities, and operating systems. In some research communities, these entities are called servers; in others they are agents, or behaviors, or actors. I have chosen "entities" as a relatively neutral term, although it too has its baggage.[6] Nor is the community that is the computation restricted to software. Hardware, physical artifacts, even human beings can be participants in this computational community.

This is the picture of computation that I think best characterizes today's computational system. (By this I mean to include everything from spreadsheets and video games to the control systems for automobiles and nuclear power plants.) In fact, this model characterizes a very wide range of systems, including many without apparent computers involved.

[6] Note, however, that an entity in my terminology is not the same as an object in the sense of object-oriented programming. First, not all objects are autonomous and self-animating. It is customary to explicitly distinguish such animate objects by means of labels such as “research on *concurrent* objects.” (See, e.g., Yonezawa and Tokoro (1987) or Agha (1990).) Second, an entity in my terminology need not have the explicit data-plus-invocable-methods structural interface generally associated with object-oriented methodology, i.e., animate entities need not be objects at all. Agha and Hewitt’s (1988; Agha 1986) actors or other concurrent object approaches are good representative examples of what I mean by entities, but so are Brooks's (1990) augmented-finite-state behaviors. Entities capitalize on, rather than hide, inherent concurrency.
Just as entities are not objects, what I am advocating here is not a shift to object-oriented thinking, though it bears some resemblance to what (Kay 1997) claims originally to have intended by that term. The current practice of object-oriented programming has largely been coopted by the traditional metaphor. This is why the phrase “concurrent objects” is not redundant, but a necessary further specification. Object-oriented programming encapsulates data with function; but this function is generally sequential and largely contextually oblivious. Because objects are generally seen as passive, act-only-when-invoked kinds of creatures, they have become a part of the traditional metaphor’s mainstream.

Changing the computational metaphor—moving from computation as calculation to computation as interaction—has far-reaching and fundamental effects on the way that we think. I mean this in three senses:

- It changes how computer scientists view computer science. This has particular implications for how and what we teach.
- It changes how we all approach problem solving and software design as well as the domains from which we draw our models and solutions. This has implications for the kinds of epistemological styles that we consider acceptable and, as a result, on the kinds of answers that future generations will produce.
- It changes how scientists—especially but not exclusively cognitive scientists—use computation as a reference model. This affects our understanding of thinking itself.

This argument—that computation-as-interaction is an important motivating metaphor—is one that I wish to make in the remainder of this paper.

5 How We Think

The remainder of this paper explores the ramifications of this shift in the computational metaphor, from traditional computation-as-calculation to today's computation-as-interaction. Throughout this journey, I will use the single motivating example of a navigation-based mobile robot. It will be developed from a basic community of interacting entities suitable for classroom presentation to a more complex artifact that gives us insight into how biological systems might think. Along the way, I will make several detours to explore related issues.

This example has been personally motivating, as it reflects my route into these issues. It also unifies many of the issues that I wish to highlight, including the urgency and feasibility of changing our approaches to introductory pedagogy, the epistemological styles necessitated by this alternate computational paradigm, and the ways in which this shift both reflects and is reflected by the newest approaches to cognitive science.[7]

My research into cognitive architectures led directly to significant frustrations with the inapplicability of my training as a computer scientist for those problems of cognitive science. As I have described above, I found the sequentialist, centralized, disembodied nature of the traditional metaphor—and its corollary modular-functionalism—inappropriate for the artifacts—robots—with which I worked. My empirical work in robotics pushed me to explore alternate approaches to cognitive architectures At the same time, I realized that the problems that I saw in cognitive robotics mirrored architectural difficulties my research group was having in such

[7] Three threads of my own history converge in this work. The first is an ongoing investigation of cognitive architectures: How might intelligent systems be put together? (Stein 1994 1997) The second thread is an investigation of the semantics of sharing in object-oriented systems, including the tradeoff between flexibility and behavioral guarantees. (Stein 1987; Stein et al. 1988; Stein and Zdonik 1999) The final thread involves the use of simple, inexpensive robots to enhance the classroom experience of undergraduates in computer science (Hendler and Stein; Stein 1996 1999).

different arenas as software agents (Coen 1997) and information retrieval (Karger and Stein 1997).

These problems in turn sounded like what my colleagues across computer science increasingly described: the importance of interface, the inevitability of implicit or explicit concurrency, the valuation of behavior by ongoing invariants rather than end-products. The watchwords of cognitive robotics—embedded, embodied, situated—applied equally to what most of my colleagues were doing. Though they used terms like “server” rather than “agent”, they, too, were building communities of interacting entities. Their systems, like mine, were not well-described by computation’s central dogma.

The world in which cognitive robotics resides is the world of interactive systems. This domain, which includes real-time systems and user interfaces, computational hardware and distributed systems, software agents and servers, is largely outside the traditional computational paradigm. In this world, the time- rules of modular construction do not always make it apparent how to combine such functions to produce desired behavior in a principled way. Others have noticed the interconnectedness of software systems as well. (See, e.g., the recent Workshop on the Interactive Foundations of Computation at Washington University St. Louis, Wegner’s recent article in the *Communications of the ACM* or any of the recent work in component architectures.) New approaches to computation are needed at many levels, from theoretical foundations to design methodologies.

In the next three sections, I will use the idea of a navigation-based mobile robot to talk about the three kinds of thinking that conclude the previous section. First, I will look at the content of our conceptualizations: What problems are the rightful domain of computational thinking? A robot is evocative of the community-based conceptualizations that I believe provide today’s answer to this question. Second, I will turn to the ways in which we approach these phenomena: What questions we ask about these problems, and what techniques we bring to bear on them? Third, I will ask what this alternate conceptualization says about cognitive architectures, the mechanisms by which thinking is accomplished.

6 Implication: Thinking Like a Computer Scientist

Introductory computer science education is the place where we as a community articulate the principles that underlie our field. It is in this course that we lay out the foundations of computation and teach students to think in computational terms. The traditional computational metaphor has—literally and figuratively—been a central part of this course. If computation today is more appropriately construed as a community, we must rethink the story that we tell our community's newest members.

Although robots are not common in the introductory programming curriculum, they prove a wonderfully effective vehicle for illustrating important principles, both rhetorically and in actual classroom practice. (See, e.g., Resnick 1988; Martin 1994; Stein and Hendler.) This section begins with a simple example of the interactive approach: a robot that wanders without bumping into walls. The particular example is a common one in reactive robotics, but also derives from Braitenberg (1984) and from my own experiences using simple robots in undergraduate education (Stein 1996).

6.1 Constituting a Community

This problem —like every problem of interactive computation —is specified in terms of ongoing behavior. The robot—depicted in figure 3—has two distance sensors, one angled to the right of forward and one to the left. It has two motors, one controlling its left rear wheel and one its right. By driving both wheels forward, the robot moves in a straight line; by driving only one wheel, the robot turns. The job of this robot's control program is to keep moving without running into obstacles.[8]

Like a traditional mathematical program, this robot's behavior is amenable to recursive decomposition—breaking it down into like-styled parts. In a traditional functional decomposition, the constituent pieces are the steps to be sequenced. In this interactive environment, the programmer's question is instead to identify the entities whose ongoing interactions constitute the behavior of this robot. The programmer's questions are: Who are these entities? How do they interact? And how is each of these constituent entities in turn implemented?

A first approximation to such a decomposition—sufficient for our pedagogic purposes—involves one entity to control each motor and one to monitor each sensor. The sensor-monitoring entities are tasked with reporting danger to the contralateral motor whenever an obstacle looms near. Each motor-monitoring entites are responsible for driving its motor in reverse while it is being warned. (This situation is depicted in Figure 4.) The result is that when an obstacle is visible ahead to the left, the left-sensor-monitoring entity reports this to the right motor monitor, which in turn stops the motion of the right motor. This causes the robot to turn toward the right, away from the obstacle on the left. When the robot moves far enough that the obstacle clears, the left sensor monitor ceases its report and the right motor monitor resumes the forward motion of the robot. We can further constrain the robot's behavior, e.g., by relating the latency of the notification and response to the robot's speed and turning radius.

There are many ways to implement a variety of increasingly sophisticated navigation behaviors. For example, there might be intervening entities between the sensor-monitors and the effector-monitors, allowing a more complex decision-making process. Alternately, the processing of a single sensor might be accomplished by an entity that is actually itself a community. (If this seems overkill for a simple distance sensor, consider instead the processing of a camera image.) Of course, there are many variants implementing increasingly sophisticated behaviors. We will revisit this example in section 8, below.

[8] This description elides certain issues, such as the necessary relationship between the sensitivity of the sensor and the turning radius of the robot, for the sake of clarity and conciseness of exposition.

As described, this example is not very complicated. Significantly, the simplest behavior satisfying the specification is quite straightforward. The purpose of this example is not to illustrate the complexities of reactive robotics; rather, it is to show that computations of this sort—interactive, embedded, ongoing—can be largely straightforward and accessible to the beginning student. At the same time, this example highlights the ways in which the questions of this new paradigm differ from the traditional questions of result-oriented step-sequencing programming.

6.2 Beyond Robots: Interactive Programming in the Curriculum

The robotic example described here serves as a great motivator for introductory students. Robots are hands-on. They make much of their internal state manifest, misbehaving in ways that are often readily apparent. They encourage experimentation and observation. And interesting issues arise with much less complexity than in an operating system, the one example of a concurrent system found in the traditional undergraduate curriculum.

Of course, the idea of interactive systems in the introductory classroom does not depend on robots. In (Stein 1999), I describe a new curriculum for the introductory programming course, i.e., for students with no prior programming experience. This course differs from the traditional one both in the questions that are asked and in the territory that is covered as a consequence. Every program that students encounter in this class is inherently concurrent and embedded in a context. Functionality to be implemented is always specified in terms of interactions and ongoing behavior.

In this single semester course, students progress from simple expressions and statements to client/server chat programs and networked video games. Although this sounds like extremely advanced material, these topics proceed naturally and straightforwardly from the interactive computational metaphor. Because the programmer's questions concern the relationships between components, topics like push vs. pull, event-driven vs. message passing, and local vs. networked communication are integral aspects of this course. The curriculum exploits this shift in the fundamental story of programming to restructure what is basic and what is advanced curricular material. In other words, this course does not go deeper into the curriculum than a traditional introductory course; rather, it stands the traditional curriculum on its end.

The introductory course is where we make our metaphors explicit, where we lay out what computation is all about. By recasting the course in terms of a new metaphor for computation, I was able to teach beginning students about ideas traditionally considered too complex and inaccessible for that level. This changes every subsequent course, without actually changing the course sequence. Everything that we teach our students takes on new meaning. For example, this approach makes it easier to contextualize traditionally hard-to-fit-in topics such as user interfaces. If computation is about what to do next, what role could a user possibly play? But if computation is about designing the coordinated activity of a community, a user is simply another member of the community within which the software system is embedded. Rethinking the computational metaphor turns the discipline on its side, giving us new ways to understand a wide range of phenomena.

6.3 Shifting the Vocabulary

In the simple robotic example at the beginning of this section, the behaviors of the system as a whole and of its constituent entities were described in terms of ongoing contracts or promises. The individual routines executed by each entity are trivial: "If the sensor reports an object, inform the opposite motor-monitor", for example. It is not any one entity that performs the navigation of this robot; rather it is an emergent property of the coupled interactions among the constituent entities—the members of the community—as well as the interactions between this computational community and the surrounding world in which it is embedded.

The kinds of questions to which this example lends itself typify the issues of modern software design. How reliable does communication between the entities need to be? (In this case, not every signal need reach the motor-monitor; lossy communication is generally adequate.) Whose responsibility is transmission of the signal: push or pull? (In this example, I have allocated that task to the sensor-monitor, a signal "push".) What kinds of latencies can be tolerated? (This depends on the mechanical properties of the robot within its environment.) Under what circumstances can this robot reasonably be expected to perform correctly?

These questions are difficult to ask within the traditional paradigm. Recasting the problem as the coordination of a community of interacting entities brings them to the fore. The traditional metaphor dictates questions of asymptotic complexity. Computation-as-interaction asks about throughput and latency. Tradition dictates procedural abstraction. Interaction calls for component architectures. Tradition suggests pre- and post-conditions. Interaction demands protocol design and analyses in terms of system dynamics.

The need for new kinds of architectural tools becomes readily apparent. Some recent attempts to address this need, and to provide new languages for describing the coupling of interactive systems, include catalogue-based approaches such as (Gamma et al. 1995)'s design patterns or (Shaw and Garlan 1996)'s software architectures; metalinguistic strategies such as (Dellarocas 1996) work on coordination protocols or (Kiczales et al. 1997)'s Aspect-Oriented Programming; new formalisms such as (Lynch et al. 1996)'s IO Automata; and component architectures such as CORBA or COM. Each of these pieces of work is difficult to motivate from within the result-oriented sequential approach to programming; the inspirations for each come from the desire to integrate interaction among distributed concurrent service-providing systems. Interaction-based computation demands that computer science invent new ways to think.

7 Implication: Epistemology of Software Engineering

The previous section explored the ways in which the paradigm shift from computation-as-calculation to computation-as-interaction changes the material presented in an introductory course and the tools and languages that we use to describe it. This section will look at how this shift plays out in terms of our relationships with and expectations of computation per se.

7.1 Thinking Concurrently

One of the most profound implications of this metaphoric shift is to bring all of the finely honed intuition that we have developed in the course of everyday life to bear on computational problems. Every three-year-old knows that you need to distract your parent before sneaking a cookie. Schoolchildren organize their fellows to carry out group activities all of the time.

And yet in the traditional view of computation, we go to great lengths to hide the fact that there might be more than one thing happening at a time. Programmers are not to know that their processor is issuing multiple instructions simultaneously; rather, whole pipelines are stalled or unrolled if necessary. Software engineers should not even have to think about the idea that multiple processors might be at work within a single so-called computer. And access to storage on remote machines carefully masquerades as local memory in many systems, often to the program developer's detriment (Waldo et al. 1994). Of course, this picture is an exaggeration, but like every straw man its kernel is true.

A robot is not this kind of beast. The left hand cannot wait for the right to conclude its computation; like a group of schoolchildren or a massive corporation, coordinated activity is its only viable option. In order to successfully program a robot, one must learn to think in terms of coordination frameworks, protocols, interfaces. This is the stuff of software engineering. Indeed, a brief experience programming a robot is a software lifecycle in an afternoon.

One of the most interesting things about physical robots is that the world provides a sufficiently dynamic environment as to make precise behavior almost non-repeatable. Lighting conditions change; initial positions vary; wheel slippage is unpredictable. These things change as the robot is executing. A robot does not wait for the world to complete its computation before acting; instead, the robot operates concurrently and interactively with the world in which it is embedded. As a result, running a real robot in the unconstrained real world invariably provides new challenges and new test conditions in a way almost entirely lacking in current computer science classrooms.

7.2 Interacting with Computation

Consider, for example, what happens if the robot of the previous section approaches a corner. Now both sensors fire. Each sensor-monitoring entity signals a reverse to the opposite motor monitor. Each motor-monitoring entity, being warned, stops moving.

The robot stops. Each sensor continues to fire. Each sensor monitor continues to issue a warning. Each driver continues to do nothing. And so on, forever.

This interaction may have been unanticipated by students, but it was not unanticipated by the instructor. In fact, my laboratory assignments are designed to create just such circumstances. Before my students go to the laboratory, they are required to design their programs and predict the behavior that will result. In laboratory, they build their code and run experiments. In testing their programs, students are expected to report on their observations, including the ways in which these observations did not match their predictions. Then, after the laboratory portion is complete, students write about how they could change the observed behavior of their programs. I do not expect them to resolve every issue; I do, however, expect them to develop intuitions for anticipating behavior and a range of options for addressing them.

This process mimics the real-world experience of the software lifecycle. As we all know, the vast majority of software development takes place after the initial program is officially complete. Just because it passes the test regimen you design doesn't mean it won't need significant modification based on later testing or the shifting of requirements. Students of the observational/experimental approach learn to anticipate the aspects of their program that cause particular behavior and to work with existing programs to modify them. This is software engineering as we have been trying to teach it for the last several decades; it is a classroom technique that steps outside the run-once format so common in introductory programming laboratories today.

7.3 Validating New Ways of Thinking

This example highlights the importance of experimentation as an engineering technique. This is common technique in scientific laboratories and is crucially important in building programmer intuition. It is often an essential tool for testing the kinds of interactive, concurrent, service-based computations that typify this style of software system. A further benefit is that it reaches out to those whose natural epistemological styles may not accord well with the purely hierarchical, functional, black-box-based approaches common in the traditional paradigm.

When we interact with our code, we are performing exactly the sort of experimental tinkering that Turkle and Papert (1990) report as a style disenfranchised by traditional approaches to computer programming. This style of experimentation is not a part of the traditional means-end goal-oriented problem-solving representative of the sequential metaphor. Nonetheless, this experimental style is precisely what is needed to understand programs that are communities of interacting entities. It necessitates a new generation of software engineering and design tools, such as (Carrierro and Gelernter 1990)’s concurrent workspace visualization tools for Linda; (Brooks et al. 1994)’s software oscilloscopes for visualization of discrete software signals; or (Kölling 1998)’s Blue, which allows students to interact with individual objects prior to embedding them in larger systems.

It is no coincidence that this metaphor of computation—which brings computation into line with our real-world experiences, which treats computational entities as artifacts to be interacted with—adapts the scientist's laboratory style and gives voice to Turkle's silenced tinkerers. Nonetheless, observational techniques commonly play a far smaller role in computational training or even computational practice than the underlying metaphor dictates. As the nature of computation changes, we need to find—or rediscover—different styles of thinking.

8 Implication: Cognitive Science and the Mechanisms of Thinking

In this final exploratory section, I look at the implications of the shift from calculation to interaction on a field traditionally indebted to the computational metaphor: cognitive science. Cognitive science attempts to explain how thinking might actually work. Historically, it has done so by relying heavily on the computational metaphor (especially as articulated by Marr (1982)). Cognitive problems are described as abstract result-oriented functions; cognitive circuitry is simply a particular implementation of these calculational processes.

Earlier, I described how artificial intelligence (a constituent field of the cognitive sciences) began with a focus on puzzle-mode problem solving. Cognitive psychologists (e.g. (Fodor 1983), (Pylyshyn 1984)) and linguists (especially Chomsky and his followers) made heavy use of a similar information-processing metaphor. Even in the construction of robots—physically interactive, albeit computationally based, systems—the calculation model prevailed (Fikes and Nilsson 1971).

More recently, some cognitive scientists have expressed frustration with the computational metaphor. Some have rebelled against the idea that the brain is like a computer. One form of this argument resorts to artificial neural networks as an alternate implementation (Rumelhart and McClelland 1986).[9] Others have turned to the embedded ("connected to the world") and embodied ("physically realized") nature of natural cognition as an alternative basis for understanding and replicating mental phenomena. In so doing, these cognitive scientists have also rediscovered the work of the cyberneticists. (See, for example, (Varela and Borgine 1992), (Port and van Gelder 1995), (Steels and Brooks 1995).)

This argument has been most extreme—and the transition away from the traditional computational metaphor most clear cut—in the field of robotics. As articulated by Brooks (1986), the idea is that physical control of a robot is best achieved through a "horizontal" decomposition in which each entity (or behavior) bridges from sensors to actuators. The resulting communities of interacting entities have produced significant advances in the state of the art in robotics and led to radically new ways in which robots solve a range of physical problems.

[9] This is both ironic and apt. Artificial neural networks are almost invariably implemented on traditional digital computers, forcing them into the sequential calculation model. Nonetheless, they are as plausibly implemented on analog and massively parallel architectures, making them community- rather than calculation-based.

8.1 Navigation as a Community

The robot described in the preceding sections is very much concerned with physical tasks. In this section, I will describe a similar robot and its extension into a more cognitive domain. In the process, I hope to illustrate how the traditional computational metaphor fails here as well. While the robot I describe begins to give us some insight into how cognitive functionality might be bootstrapped off of physically interactive behavior, the means by which this is accomplished is difficult to describe in traditional modular-functional terms.

Mataric (1992) describes a variation on the robot that we have seen before. Like our classroom robot, Mataric's Toto uses a community-based approach to wander, avoiding obstacles. Additional community members add biases towards wall-following and a primitive sort of experiential memory. For example, as Toto's sensor- and motor-monitoring entities keep it wandering down the hall, a corridor-classificatory entity is activated (by the sustained perception of left- and right-side obstacles, or walls) and a memory entity records certain salient aspects of this experience.[10] By means of a crude button-based interface, a person can direct Toto to return to a previously experienced landmark. The various entities that constitute the robot then uses a spreading activation algorithm to return to the appropriately salient place.

This robot, by itself, is one that is difficult to describe in von Neumann terms. Its cognitive architecture is a community of interacting communities. There are no central portions of code where the behavior all comes together. Instead, every decision is made locally on the basis of particular patterns of input: the robot moves away from a wall that may be too close, a landmark-detector creates itself when consistent sensory readings exceed a threshold, and random wandering is biased in the direction of a goal location when that goal location pulls more strongly than the competition. Each constituent is an ongoing interactive entity that continually senses and acts. The collective behavior of the community is goal-directed navigation, although no particular entity performs this task. Toto has no single, central "self." This interactive, community-based approach is typical of the new cognitive science.

8.2 Cognition and Non-Modular-Functionalism

Much of my own research work has been in the domain of cognitive robotics (e.g., Yanco and Stein 1992; Brooks and Stein 1994; Stein 1994 1997). Cognitive robotics is an attempt to scale these community-based approaches so popular in robotics into the traditional domains of artificial intelligence: reasoning and problem-solving.[11] The new computational metaphor—in which behavior emerges from interactions, rather than a composition of independent constituents—has a crucial role to play here.

[10] The robot distinguishes single-wall, corridor, and other environmental categories; it also uses a compass to determine when it has made significant rotations, so that (for example), navigating a corner may produce two consecutive but distinct left-wall landmarks.
[11] This is in contrast to the traditional artificial intelligence (GOFAI) approaches, which maintain a distinction between physical processes and purely symbolic cognitive ones.

In the early 1990s, I extended Mataric's work to include a "module" that allowed her robot to read maps. Mataric's robot could previously only go to places of which it had previously accumulated experience. It had no way of understanding the notion of a place it had not visited. (Stein 1994) describes an extended system that allows the robot to build the same kind of representation of unvisited space that it has of visited space. The robot is given a floor plan and explores that plan, creating an experiential memory that subsequently allows it to navigate as though it had explored the physical environment.

One might imagine that this new system is built out of two distinct components sequenced in an entirely conventional way. First, the floor plan processing module would study the map and create a representation. Subsequently, Mataric's robot would use that representation to navigate to the desired location. This would certainly be the traditional—GOFAI—approach. The cognitive robotics story is not that simple.

Instead, I exploited the robot's existing interactive properties. Rather than using an independent "map-processing" component, the robot interacts with the map as a virtual sensory environment, "imagining" that it is inside that environment. There is no separate map-to-internal-representation functional module. Instead, Mataric's existing robot-cum-community is coupled to (i.e., embedded in a virtual environment consisting of) a very simple piece of code that keeps track of x, y, and heading coordinates within the floor plan. This interactive map-entity processes "move" requests (generated by Mataric's original code) by updating its internal coordinates and returning simulated sensory readings as though the robot were actually standing at the corresponding point in physical space[12]. This[13] is the entire extent of the code that I added to Mataric's.

8.3 Approaches to Cognitive Architectures

This cognitive system—the ability to read maps and act based on information therein—is an emergent property of the interactions between a very simple spatial calculation on the floor plan and a very complex system for choosing actions including archiving and acting upon remembered experience. In reporting these results in the technical literature, I found that the traditional computational metaphor did not provide an adequate vocabulary in terms of which to explain this robot. This system is not fundamentally constituted out of steps to achieve a goal; instead, it is a concurrent collection of interacting behavior-modules.

By replacing the sequentialist model with a community of interacting entities, I was suddenly and strikingly empowered to give an adequate reconstitution of this system. Mataric's robot achieves its behavior by means of continual interaction with an environment. My system achieves an alternate behavior by temporarily replacing the

[12] These sensory readings consisted of ray projections from the robot's position, giving extremely rough approximations to the robot's actual sonar. The robot itself was approximated as a point. The sonar, which are in reality highly non-linear, were approximated as linear and non-disbursive.
[13] Plus a simple mode-switch to allow the robot to move between "imagine" mode and physical, real-world interactions.

physical environment with the virtual one represented by the floor plan. The community that constitutes Mataric's robot's internal controllers interacts with the simulated world of the floor plan in a way that is equivalent to the interactions that would occur in the real, physical world. This equivalence is partial; the simulation is incredibly simplistic and unrealistic. Nonetheless, with respect to the interactional invariances observed by Mataric's system, the two environments are equivalent.

This story describes one particular example of a cognitive behavior that was achieved by bootstrapping directly from a more physical interaction. There is significant evidence for this idea of building more cognitive constituents by reusing systems of visceral interaction. For example, Kosslyn (1980 1994) has long argued that mental imagery tasks critically engage human visual cortex. Damasio (1994) contends that rational decision making is crucially dependent on the limbic emotional system. And Clark (1997) reviews the inseparability of body-based manipulations and cognitive strategies employed by artificial and natural organisms. In each of these cases, the cognitive and the physical are not sub-functions—sequential or otherwise—in a result-oriented computation. Instead, it is the interactions themselves constitute the more cognitive aspects of computation.

The recent emphasis on social cognition only adds fuel to this fire. If thinking in a single brain is communally interactive, how much more so the distributed “intelligence” of a community! Hutchins (1996) goes so far as to suggest that cognition—in his case of a naval navigation team guiding a warship—is necessarily distributed not just within a single brain but across a community of people, instruments, and culture. Computation as traditionally construed—the calculational metaphor—provides little leverage for these new theories of thinking. Shifting the computational metaphor opens up the possibility of reuniting computation with cognition. Like the electronic computer, a human brain is a community of interacting entities. This represents a fundamental change in our understanding of how we think.

9 Summary

We live in a time of transition. Computer science is undergoing a Kuhnian revolution. The traditional foundations of our field are shifting, making way for an alternate conceptualization that better explains the phenomena we see. The previous metaphor—computation as calculation, sequencing steps to produce a result—was crucially empowering in computation’s early history. Today, that metaphor creates more puzzles than it solves. We cannot even explain our field’s best-known artifact—the world-wide web—in traditional terms.

This paper is about changing the ways in which computer scientists think about computation. Many subdisciplines of computer science have their own language for describing computation-as-interaction. In artificial intelligence, the recent attention to embodiment, to agents, to behaviors, is indicative of this shift. The computer systems community uses terms like server, transaction, thread. Other research communities that rely on similar notions—by still other names—are those that study networking, distributed systems, information management, human-computer interaction and computer-supported collaboration, web computing, and embedded systems. Each of these research communities has its own terminology for describing the interactive community metaphor, impeding the opportunities for cross-field discourse and collaborative problem solving.

Recasting all of computational science in terms of the interactive community shifts the center of the field. Efforts to make multiple CPUs look like a single processor—as in automatic program parallelization—now seem peripheral. Research on user interfaces, or on component architectures such as CORBA or COM, take on new centrality given their focus on coupling subsystems together. The heart of current computational thinking is in agents, servers, distributed systems, and protocols.

This way of approaching computation also has profound implications for the kinds of thinking we do. For our students, it means that we harness their native intuition about how to survive in an inherently concurrent and asynchronous world. We never put on the blinders of calculational sequentialism. Students and professionals alike are encouraged to interact with computational artifacts, to experiment, to tinker. And we no longer silence those students whose problem-solving skills derive from experiential rather than mathematical and logical approaches.

In other disciplines, we find that the new metaphors we are using are more appropriate for bi-directional cross-disciplinary communication. Just as computation is a reference model for understanding cognitive and biological science, so what we learn about the robustness of biological systems inspires us in the construction of “survivable” computational systems (e.g., Forrest) or programmable cells (Abelson et al.). Both natural and artificial computations produce behavior by virtue of the interactions of a community.

Many disciplines study systems of interaction. We have historically claimed a distinct role for computation. Now, as computational science itself shifts to embrace interaction, our field can become reunited with its surrounding disciplines. The cognitive sciences looks at how natural intelligence works. Organizational science analyzes the ways in which corporations and other large administrative entities function. Engineering provides vocabularies and techniques for coordinating complex systems. Each of these fields has the potential to contribute to, and to benefit from, a computational science of interaction.

Changing the fundamental metaphor underlying computation shifts the very questions that we as a discipline ask. It affects the appropriate matter of a first course and of a research project. It has ramifications for the styles and techniques that we use in our work. And it bears on our understanding of our own cognitive processes, both those that affect all people and those that are stylized by our disciplinary culture. The current computational revolution has profound implications for how we think.

10 Acknowledgements

This paper began as the W. Ross Ashby Plenary Address of the International Federation for Systems Research at the European Meeting on Cybernetics and Systems Research in Vienna in April 1998. I am grateful to the IFSR and to Robert Trappl, as well as to Gerda Helscher, Erich Prem, and Paulo Petta for their gracious hospitality. In addition, Gerda did yeoman's work transcribing that first version of the paper.

The research that turned into this paper has been supported by the National Science Foundation under Young Investigator Grant IRI-9357761, and by the Office of Naval Research under the Science Scholars Program at the Mary Ingraham Bunting Institute of Radcliffe College, where I spent a year on sabbatical leave from MIT.

In a paper of this scope, there are far too many people to be thanked than could fit into the confines of a journal article. Those who are not mentioned by name will have to take on faith the depths of my appreciation. The various members of the AP lab at MIT, the 1998 Bunting Fellows and staff, the participants in WUSTL's IFOC Symposium, and many colleagues at MIT and elsewhere have shaped this paper in innumerable ways. Special thanks go to Hal Abelson, John Chapin, Philip Resnik, and H. David Stein, who read and commented on earlier drafts.

11 References

Abelson, Hal, Tom Knight, and Gerry Sussman. 1995. Amorphous Computing. *White paper***.**

Agha, Gul. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, Massachusetts: The MIT Press.

Agha, Gul. 1990. Concurrent Object-Oriented Programming. *Communications of the ACM* **33** (9):125-141.

Agha, Gul, and Carl Hewitt. 1988. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In Bruce Shriver and Peter Wegner, editors. *Research Directions in Object-Oriented Programming*. Cambridge, Massachusetts: The MIT Press, pp. 49—74.

Agre, Philip E. 1988. *The Dynamic Structure of Everyday Life*. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Agre, Philip E. and Stanley J. Rosenschein, editors. 1996. *Computational Theories of Interaction and Agency*. Cambridge, Massachusetts: The MIT Press.

Ashby, W. Ross. 1954. *Design for a Brain*. London: Chapman and Hall.

Ashby, W. Ross. 1956. *An Introduction to Cybernetics*. London: Chapman and Hall.

Beer, R. 1995. A Dynamical Systems Perspective on Agent-Environment Interaction. *Artificial Intelligence* **72**:173-215.

Braitenberg, Valentino. 1984. *Vehicles: Experiments in Synthetic Psychology .* The MIT Press. Cambridge, MA.

Brooks, Rodney A. 1986. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation* **2** (1):14-23.

Brooks, Rodney A. 1990. The Behavior Language User's Guide. Memo 1227. Massachusetts Institute of Technology Artificial Intelligence Laboratory. Cambridge, Massachusetts.

Brooks, Rodney A. 1991. Intelligence without Reason. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Sydney, Australia, pp. 569-595.

Brooks, Rodney A., and Lynn Andrea Stein. Building Brains for Bodies, *Autonomous Robotics* **1** (1), 7—25 1994.

Carriero, N. and D. Gelernter. 1990. *How to Write Parallel Programs: A First Course*. Cambridge, Massachusetts: The MIT Press.

Chomsky, Noam. 1980. *Rules and Representation*. Columbia University Press: New York.

Chomsky, Noam. 1993. *Language and Thought*. Moyer Bell: Wakefield, Rhode Island.

Clark, Andy. 1997. *Being There: Putting Brain, Body, and World Together Again*. Cambridge, Massachusetts: The MIT Press.

Coen, Michael H. 1994. *SodaBot: A Software Agent Environment and Construction System*, Sc.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Coen, Michael H. 1997. Building Brains for Rooms: Designing Distributed Software Agents. In *Proceedings of Ninth Conference on Innovative Applications of Artificial Intelligence*. Providence, Rhode Island.

Cypher, R., A. Ho,, S. Konstantinidou, and P. Messina. 1993. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *IEEE Proceedings of the 20th International Symposium on Computer Architecture*. San Diego, California, pp.2-13.

Damasio, Antonio R. 1994. *Descartes' Error: Emotion, Reason, and the Human Brain*. New York: G.P. Putnam's Sons.

Dellarocas, Chrysanthos N. 1996. *A coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components*. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Center for Coordination Science Working Paper 198.

Dennett, Daniel C. and Marcel Kinsbourne. 1992. Time and the Observer: The Where and When of Consciousness in the Brain. *Brain and Behavioral Sciences* **15**, 183-247.

Dourish, Paul, and Graham Button. 1996. Technomethodology: Paradoxes and Possibilities. In *Proceedings of the ACM Conference on Human Factors in Computing Systems CHI'96* (Vancouver, Canada). New York: ACM Press.

Fikes, R. and N. Nilsson. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* **2** (3-4):189-208.

Fodor, J. A. 1983. *The Modularity of Mind*. Cambridge, Massachusetts: The MIT Press.

Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley.

Hendriks-Jansen, Horst. 1996. *Catching Ourselves in the Act*. Cambridge, Massachusetts: The MIT Press.

Hillis, W. Daniel. 1989. *The Connection Machine*. Cambridge, Massachusetts: The MIT Press.

Hutchins, Edwin. 1996. *Cognition in the Wild.* Cambridge, Massachusetts: The MIT Press.

Karger, David., and Lynn Andrea Stein. 1997. Haystack: Per-User Information Environments. *White Paper*.

Kay, Alan. 1997. *The Computer Revolution Hasn't Happened Yet*. Keynote address at the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications.

Keller, Evelyn Fox. 1983. *A Feeling for the Organism: The Life and Work of Barbara McClintock.* San Francisco: W. H. Freeman.

Keller, Evelyn Fox. 1995. *Refiguring Life*. New York: Columbia University Press.

Kölling, Michael I. 1998. *The Blue programming environment - Reference manual - version 1.0*. Technical report 98/19. School of Computer Science and Software Engineering, Monash University, Melbourne.

Kosslyn, Stephen M. 1980. *Image and Mind*. Cambridge, Massachusetts: Harvard University Press.

Kosslyn, Stephen M. 1982. *Ghosts in the Mind's Machine*. New York: Norton.

Kosslyn, Stephen M. 1994. *Image and Brain: The Resolution of the Imagery Debate*. Cambridge, Massachusetts: The MIT Press.

Kuhn, Thomas S. 1962.*The Structure of Scientific Revolutions*. University of Chicago Press.

Lawler, Robert W. 1985. *Computer Experience and Cognitive Development: A Child's Learning in a Computer Culutre*. John Wiley and Sons.

Lynch, Nancy, Roberto Segala, Frits Vaandrager, and H. B. Weinberg. 1996. Hybrid I/O Automata. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III: Verification and Control* (DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, October 1995), volume 1066 of Lecture Notes in Computer Science, pages 496-510. Springer-Verlag.

Martin, Fred. 1994. *Circuits to Control: Learning Engineering by Designing LEGO Robots.* Ph.D. Dissertation. Media Laboratory. Massachusetts Institute of Technology. Cambridge, Massachusetts.

Marr, David. 1982. *Vision*. W. H. Freeman: San Francisco, California.

Mataric, Maja. 1992. Integration of Representation Into Goal-Driven Behavior-Based Robots. *IEEE Transactions on Robotics and Automation* **8** (3).

Mindell, David A. 1996. *Datum for its Own Annhilation: Feedback, Control, and Computing 1916-1945*. Ph.D.Thesis. Program in Science, Technology, and Society. Massachusetts Institute of Technology. Cambridge, Massachusetts.

Minsky, Marvin. 1987. *The Society of Mind*. New York: Simon and Schuster.

Newell, A. and H. A. Simon. 1963. GPS: A Program that Simulates Human Thought. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pp. 279-293.

Newell, A. and H. A. Simon. 1972. *Human Problem Solving*, Englewood Cliffs, New Jersey: Prentice Hall.

Papert, Seymour. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books.

Port, Robert F., and Timothy van Gelder. 1995. *Mind as Motion: Explorations in the Dynamics of Cognition*. Cambridge, Massachusetts: The MIT Press.

Pylyshyn, Zenon. 1984. *Computation and Cognition*. Cambridge, Massachusetts: The MIT Press.

Resnick, Mitchel. 1988. *MultiLogo: A Study of Children and Concurrent Programming*. Sc.M. Thesis. Department of Electrical Engineering and Computer Science. Massachusetts Institute of Technology. Cambridge, Massachusetts.

Resnick, Mitchel. 1994. *Turtles, Termintes, and Traffic Jams: Explorations in Massively Parallel Microworlds*. Cambridge, Massachusetts: The MIT Press.

Rumelhart, D. E. and J. L. McClelland, editors. 1986. *Parallel Distributed Processing*. Cambridge, Massachusetts: The MIT Press.

Schuman, Erin. 1998. Invited presentation on *Learning and Memory*. Tenth Annual Symposium on Frontiers of Science.

Shaw, Mary and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

Smith, Brian Cantwell. 1997. *On the Origin of Objects*. Cambridge, Massachusetts: The MIT Press.

Smithers, Tim. 1995. What the Dynamics of Adaptive Behavior and Cognition Might Look Like in Agent-Environment Interaction Systems. In *Practice and Future of Autonomous Agents*. Monte Verita, Ticino, Switzerland, pp. 1-27.

Steels, Luc and Rodney Brooks, editors. 1995. *The Artificial Life Route to Artificial Intelligence: Building Embodied, Situated Agents*. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Inc.

Stein, Lynn Andrea. 1987. Delegation Is Inheritance. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, Orlando, Florida, pp. 138-146.

Stein, Lynn Andrea. 1994. Imagination and Situated Cognition. *Journal of Experimental and Theoretical Artificial Intelligence* **6**:393-407.

Stein, Lynn Andrea. 1996. Interactive Programming: Revolutionizing Introductory Computer Science. *Computing Surveys* **28A** (4).

Stein, Lynn Andrea. 1997. PostModular Systems: Architectural Principles for Cognitive Robotics. *Cybernetics and Systems* **28** (6):471-487.

Stein, Lynn Andrea. 1999. What We've Swept Under the Rug: Radically Rethinking CS1. *Computer Science Education* **9.**

Stein, Lynn Andrea. Forthcoming. *Introduction to Interactive Programming*. . San Francisco, California: Morgan Kaufmann Publishers, Inc.

Stein, Lynn Andrea. Rethinking CS101: Or, How Robots Revolutionize Introductory Computer Programming. Accepted for publication in *Computer Science Education*.

Stein, Lynn Andrea and James A. Hendler. Robotics-based Undergraduate Computer Programming Courses.

Stein, Lynn Andrea, Henry Lieberman, and David Ungar. 1989. A Shared View of Sharing: The Treaty of Orlando. In Won Kim and Fred Lochovsky, editors, In *Object-Oriented Concepts, Databases, and Applications*, ACM Press, pp. 31-48.

Stein, Lynn Andrea and Stanley B. Zdonik. 1998. Clovers: The Dynamic Behavior of Types and Instances. *International Journal of Computer Science and Information Management* **1** (3).

Turing, Alan M. 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society* **2** (42): 230-267.

Turkle, Sherry. 1984. *The Second Self: Computers and the Human Spirit*. New York: Simon and Schuster.

Turkle, Sherry, and Seymour Papert. 1990. Epistemological Pluralism: Styles and Voices within the Computer Culture. *Signs: Journal of Women in Culture and Society* 16(1): 128-157.

Varela, Francisco J. and Paul Borgine, editors. 1992. *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. Cambridge, Massachusetts: The MIT Press.

Von Neumann, J. First draft of a report on the EDVAC. Contract No. W-670-ORD-4926. Philadelphia, Pennsylvania: Moore School of Electrical Engineering, University of Pennsylvania, 30 June 1945.

Waingold, Elliot, Michael Taylor, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Srikrishna Devabhaktuni, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. 1997. *Baring it all to Software: The Raw Machine*. MIT/LCS Technical Report TR-709.

Waldo, Jim, Geoff Wyant, Ann Wollrath, and Sam Kendall. 1994. *A Note on Distributed Computing*. Technical Report SMLI TR-94-29. Sun Microsystems Laboratories, Inc.

Wegner, Peter. 1997. Why Interaction Is More Powerful Than Algorithms*, Communications of the ACM*.

Weiner, Norbert. 1948. *Cybernetics*. New York: John Wiley & Sons.

Yonezawa, Akinori and Mario Tokoro. 1987. *Object-Oriented Concurrent Programming*. Cambridge, Massachusetts: The MIT Press.

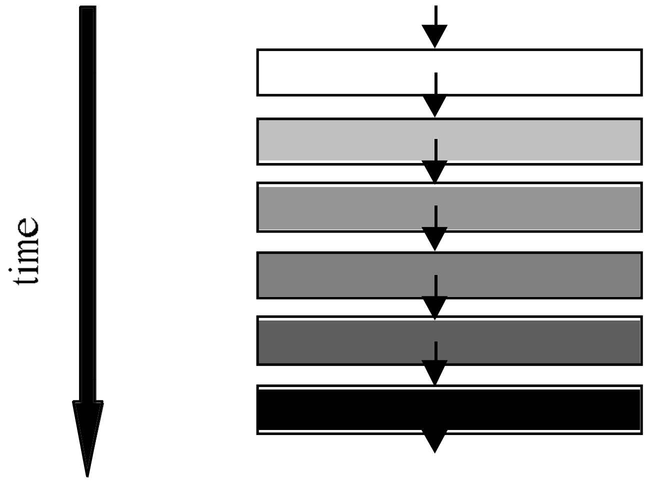

Figure 1. Sequential computation: Beginning with some input, execute a sequence of steps that terminates, producing a result.

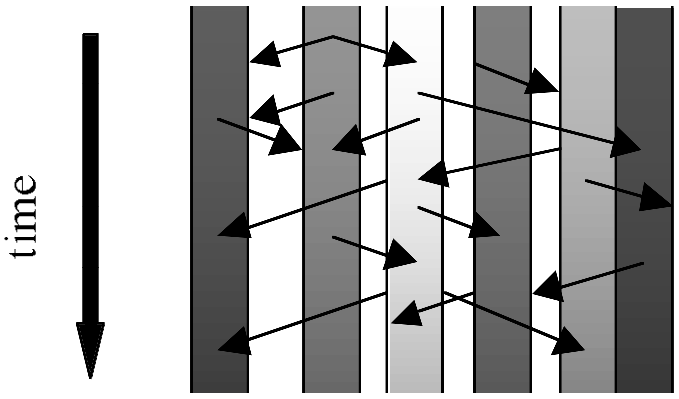

Figure 2. Computation as interaction: Many persistent entities communicate and coordinate over time.

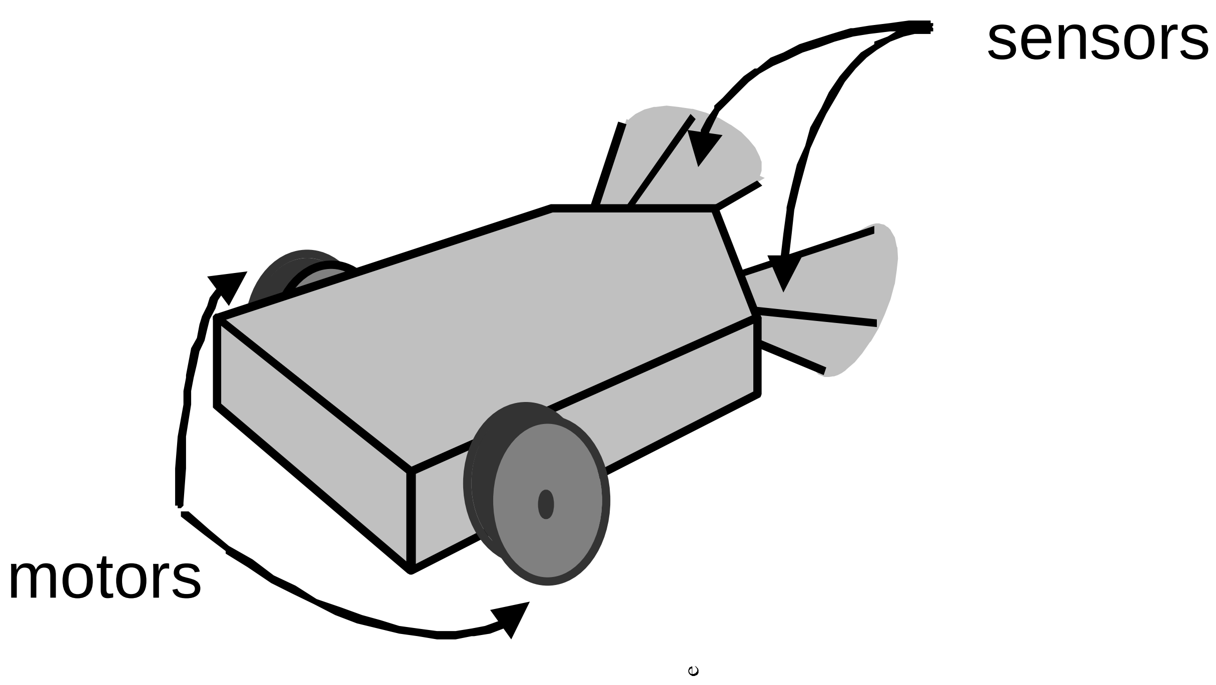

Figure 3. A simple robot whose task is to navigate without running into obstacles.

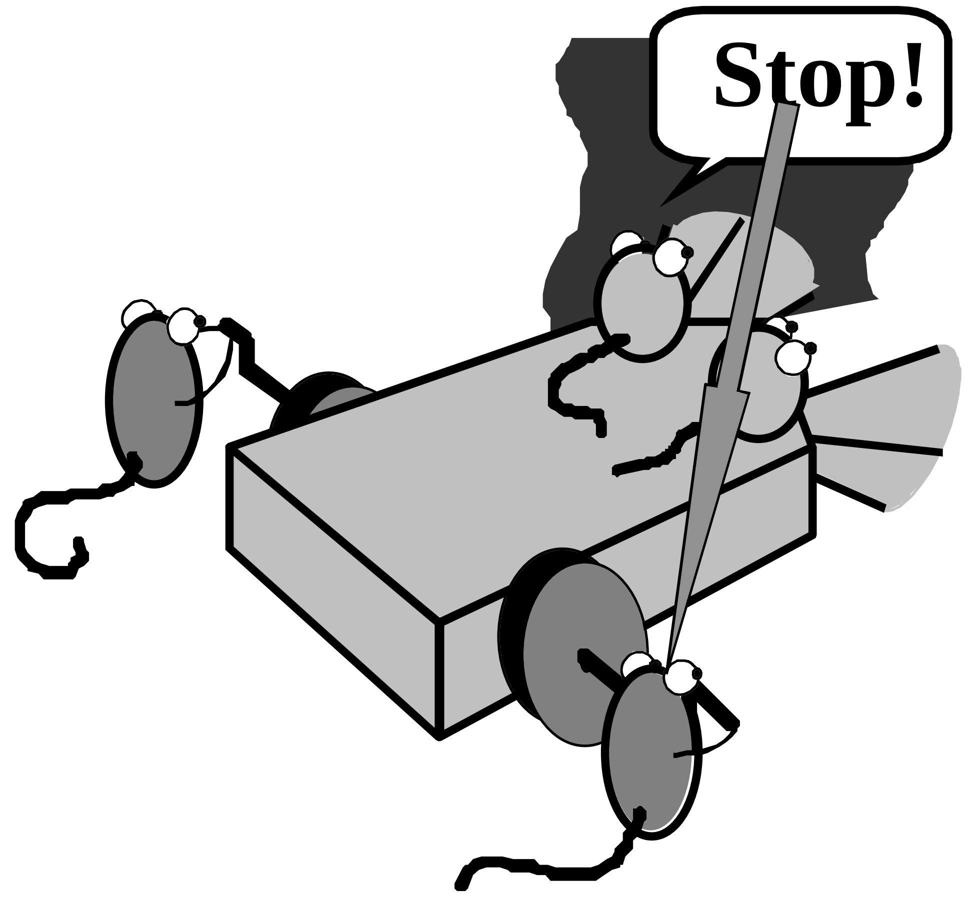

Figure 4. The robot's behavior results from the emergent interactions of a community.