Natural Computing (2006) 5: 321–362
DOI: 10.1007/s11047-005-5869-3

Dynamic self-assembly in living systems as computation

ANN M. BOUCHARD[1,*] and GORDON C. OSBOURN[2]
[1]*Physical and Chemical Sciences, Sandia National Laboratories, Albuquerque, NM, 871 85-1423, USA;* [2]*Complex Systems Science, Sandia National Laboratories, Albuquerque, NM, 87185-1423, USA (*Author for correspondence, e-mail: bouchar@sandia.gov)*

Abstract. Biochemical reactions taking place in living systems that map different inputs to specific outputs are intuitively recognized as performing information processing. Conventional wisdom distinguishes such proteins, whose primary function is to transfer and process information, from proteins that perform the vast majority of the construction, maintenance, and actuation tasks of the cell (assembling and disassembling macromolecular structures, producing movement, and synthesizing and degrading molecules). In this paper, we examine the computing capabilities of biological processes in the context of the formal model of computing known as the random access machine (RAM) [Dewdney AK (1993) The New Turing Omnibus. Computer Science Press, New York], which is equivalent to a Turing machine [Minsky ML (1967) Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, NJ]. When viewed from the RAM perspective, we observe that many of these dynamic self-assembly processes – synthesis, degradation, assembly, movement – do carry out computational operations. We also show that the same computing model is applicable at other hierarchical levels of biological systems (e.g., cellular or organism networks as well as molecular networks). We present stochastic simulations of idealized protein networks designed explicitly to carry out a numeric calculation. We explore the reliability of such computations and discuss error-correction strategies (algorithms) employed by living systems. Finally, we discuss some real examples of dynamic self-assembly processes that occur in living systems, and describe the RAM computer programs they implement. Thus, by viewing the processes of living systems from the RAM perspective, a far greater fraction of these processes can be understood as computing than has been previously recognized.

Key words: algorithm, biological information processing, computation, computing, information, microtubule, motor protein, protein network, random access machine, self-assembly, stochastic, unary number

Abbreviations: ATP – adenosine triphosphate; DNA – deoxyribonucleic acid; GDP – guanosine diphosphate; GTP – guanosine triphosphate; MCP – methyl-accepting chemotaxis protein; MT – microtubule; RAM – random access machine; RNA – ribonucleic acid; rRNA – ribosomal ribonucleic acid; SSR – simple sequence repeat

1. Introduction

Biochemical reactions taking place in living systems that map different inputs to specific outputs are intuitively recognized as performing information processing. This information processing occurs at many levels. At the molecular level, information transfer and processing can take place by shape recognition and conformation changes. Inputs of specific molecular species bind selectively to molecules with complementary shape and chemistry. These can self-assemble into a more complicated structure, which can then be read out as an output, again by shape recognition (Conrad and Zauner, 1998; Conrad, 1999). Other information processing can occur at the level of molecular networks. Protein networks can perform several computational functions, as noted by Bray (1995), and references therein. For example, they can map one input to multiple divergent paths; integrate multiple inputs to produce a single output; amplify a faint or ephemeral signal into substantive biochemical changes; adapt to signal intensity to remain sensitive over many orders of magnitude of impinging signal; and act as a memory store. In gene regulatory networks, specifically, the process of gene expression has been demonstrated by Ben-Hur and Seigelmann (2004) to simulate memory bounded Turing machines. Thus gene expression can be considered as true computation.

Communication between different levels of information processing has also been noted. Signals external to an organism are sensed by specific cells designed to transduce this type of signal. These cells process the signal, then can output communication to other cells, which can result in gene expression in those cells. The outputs (newly synthesized proteins, hormone secretion, etc.) then percolate from the molecular level back up the cellular level to the organism level to drive the new state or behavior of the organism (Conrad, 1995).

Other efforts have shown how selected chemical reactions can be mapped to a digital computing model. Magnasco (1997) demonstrated theoretically that digital logic can be implemented with standard chemical kinetics. He explicitly constructed logic gates AND, OR, XOR, and NAND from a small number of coupled chemical reactions. Arkin and Ross (1994) showed that specific enzyme cycles can produce AND, OR, XOR and NOT gates. In contrast, Agutter and Wheatley (1997) argued that such protein networks behave as analog rather than digital devices. Assuming that the actual computational units are *populations* of proteins, rather than individual molecules, they pointed out that different parts of the population do not receive

simultaneous identical inputs (because of variations in chemical composition of the environment). Therefore the output is a weighted sum of responses over the population, and will provide an analog rather than digital output. In any event, regardless of whether the computing is thought to be digital or analog, or whether the processes occur at the molecular, cellular, or organism level, the focus of these efforts has been on processes that are readily recognized as information processing, mapping inputs to outputs.

Living systems, however, and protein molecular machines in particular, perform a large range of other functions besides obvious input–output "circuits." They perform the vast majority of the construction, maintenance, and actuation tasks of the cell: building macromolecular structures, producing movement, synthesizing specific chemical species, degrading molecules, and disassembling structures. Conventional wisdom distinguishes such proteins from those whose primary function is to transfer and process information. For example, Bray (1995, p. 310) stated, "To liken such proteins to computational devices is inappropriate," except to the extent that they are regulated by the information processing mechanisms. This sentiment was echoed by Agutter and Wheatley (1997, p. 14), who wrote, "Apart from the numerous other functions of cell proteins, including synthesis, degradation, structural assembly, motor activity and secretion," a subset of proteins are specifically involved in information transfer and processing. Conrad (1995, p. 157) also suggested that there are two distinct types of functional processes in biological systems, those that "involve information processing in a direct way, as in control, measurement, or problem solving" and those such as energy metabolism, that only involve information processing in an indirect way as a control mechanism.

In this paper, we take a different approach to examining the computing capabilities of biological processes. We show that the protein properties that enable dynamic self-assembly map very naturally to a different formal model of computing, the program machine (Minsky, 1967), also called the random access machine (RAM) (Dewdney, 1993), which is equivalent to a Turing machine (Minsky, 1967). When viewed from the RAM perspective, we observe that many of these other biological processes – synthesis, degradation, assembly, movement – do carry out computational operations. We also show that the same computing model is applicable at other hierarchy levels of biological systems (e.g., cellular or organism networks as well as molecular networks), suggesting that RAM algorithms can scale up to

greater spatial and temporal scales. Thus, by viewing the processes of living systems from the RAM perspective, a far greater fraction of these processes can be understood as computing than has been previously recognized. We have two motivations for establishing the mapping between biological dynamic self-assembly processes and RAM computing. First, if biological systems are computing via dynamic self-assembly, then these self-assembly processes must be programmable. We wish to learn the "programming language," and then use it to program dynamic self-assembly, to build novel structures and materials. Second, perhaps understanding the algorithms of biological systems will provide some insight into their ability to develop, persist, reproduce, and evolve.

First, we describe the RAM model and its components: registers (number storage), operations on those registers, sequencing the operations, and branching under different conditions. We then identify the dynamic (energy-driven) self-assembly processes of proteins that can implement the required machinery of RAM computers. We discuss a variety of alternative protein structures/mechanisms (e.g., populations of molecules vs. polymers; transport vs. polymerization) as different embodiments of the same RAM model. That is, there are many different types of "hardware" that can carry out the same "software." It is noteworthy that in living systems, registers are commonly embodied in a unary, rather than binary, representation. Thus, RAM computing in living systems does look very different from the digital, binary computing that has become virtually ubiquitous in modern society.

After discussing the different individual components of a RAM computer, we put these components together to demonstrate protein RAM computing in two different ways. The first is to explicitly demonstrate how the dynamic self-assembly processes of proteins can perform computing. We present idealized (artificial) protein networks designed explicitly to carry out a numeric computation, $g=(a*b)+(c*d)+(e*f)$, where a–g are numeric values of specific registers. Although this numeric computation is not a natural algorithm (indeed, we do not expect that living systems implement *numeric* computation to any great extent), the objective is to explicitly demonstrate that the RAM computing properties of proteins we discuss are sufficient to support true computing in an easily recognizable form. We present stochastic simulations and yield statistics of these idealized networks and show that, because the computation is stochastic, errors do occur, even with an adequate energy supply. We discuss a number

of strategies used by biological systems to achieve robust functionality from these "unreliable" stochastic processes.

The second way we demonstrate protein RAM computing is to examine some real biological dynamic self-assembly processes, in computational terms, and describe the algorithms they are implementing in the RAM model. Specifically, we present pseudocode programs for the dispersal and condensation of pigment granules in melanophores. In this example, transport of the pigment granules by motor proteins plays a key role. We also discuss how the condensation algorithm scales to populations of cells and organisms. Then we present a flow diagram of a program performed by microtubules (MTs), using dynamic instability to search for stabilizing proteins located around the cell. Here the RAM program is implemented by the dynamic assembly and disassembly of physical structures, MTs. This demonstrates how some of the processes specifically identified by others as non-computational (assembly, disassembly, and transport) really do carry out computation from the RAM perspective.

Although our examples are far from exhaustive, they motivate a broader perspective on natural computing. A living system does not only process information when sensing and responding to its environment. It is our view that much, if not all, of the business of a living system's building and maintaining itself is also a physical form of stochastic computing.

2. The RAM computing model

Protein interactions map well onto a non-deterministic random access machine (RAM) model of computing. The RAM model is perhaps less well known than the Turing machine model of computing. It has been proved to be equivalent to a Turing machine. That is, any program implemented by a RAM machine could be implemented by a Turing machine and vice versa (Minsky, 1967; Dewdney, 1993). It is important to note, however, that the protein computing we discuss here is *not* equivalent to a *universal* Turing machine – a Turing machine that can run any computable program on a single set of hardware. Rather, it is equivalent to a Turing machine designed to run one program with a table of inputs and states that map to outputs and states. Furthermore, it is a *non-deterministic* Turing machine, in which there is more than one output/state that could result from the same input/state.

What features or components are necessary for computing with the RAM model? We review the features here, from Minsky (1967).

(i) First, it requires data registers, a means of storing and accessing numbers. Registers must be able to hold any size number. Modern digital computers rely on digital logic and binary numbers. In living systems, in contrast, registers are commonly embodied in a unary, rather than binary, representation. It is interesting to note that Minsky often analyzed registers in proofs as unary rather than binary encoded.

(ii) Computing with the RAM model also requires a small number of fundamental operations on those registers. There are many combinations of those fundamental operations out of which all computations can be built. They are all equivalent. That is, it has been proved that one set of fundamental operations can be computed from any of the others. The basic set is {zero, increment, decrement/jump, halt}. ''Zero'' means set the value of the register to 0, then go on to the next instruction. ''Increment'' means add 1 to the value of the register, then go on to the next instruction. ''Decrement/jump'' means if the value of the register is not zero, then decrease it by 1 and go on to the next instruction; otherwise jump to some other instruction, which must be specified. ''Halt'' means stop the program. If one register is already zero at the beginning of the program, then only increment, decrement/jump, and halt are needed.

The decrement/jump operation is noteworthy because it provides a mechanism for branching, i.e., decision making. Under one set of conditions (the register value is non-zero) the program does one thing; under different conditions (register value is zero) it does something else. This is an essential part of computing (and living systems must be able to make decisions in order to branch to different behaviors under different conditions).

Minsky (1967) also proved that other operations can be made up of this fundamental set: copy from one register to another; jump unless equal (another branching operation); and repeat, or loop. It is a trivial exercise to show that the branching operations greater than, greater than or equal to, less than, and less than or equal to can also be programmed from the fundamental operations. Thus, we can use these operations as well in our RAM programs, as if they were subroutines. For convenience in comparing to proteins, we shall consider increment and decrement as operations, and discuss branching, or decision making, separately.

(iii) The other requirement for RAM computing is to sequence the operations. Operation sequencing is taken for granted by programmers of modern digital computers, because the operations of the program are executed deterministically in the order in which the programmer wrote them. As will be explained shortly, however, when computing with proteins, the sequencing is where much of the nondeterministic aspect of the computing occurs, because it is implemented by stochastic dynamic self-assembly processes.

3. Proteins as elements of a RAM computer

3.1. *Registers and their operations*

We want to show how proteins (or indeed, other types of molecules as well) can be used to implement a RAM computer. We discuss data registers and their operations together. Proteins readily embody data registers in a unary representation: the current value of a data register (a number) is represented by a simple count of objects. The fact that proteins embody numbers in a *unary* representation is highly significant. If we look for *binary* numbers (to which modem computer users are accustomed) in biological systems, few if any examples are found, and the ubiquitous unary numbers are completely overlooked. Furthermore, although binary registers require complex encoding machinery to do even the simple increment and decrement operations (because of having to carry, borrow, and reset digits), with unary registers, these same operations are easy to achieve physically. The unary representation also appears to lend itself well to evolving many different ways to store numbers and operate on them. Figure 1 illustrates several examples of unary registers implemented with proteins or other molecules.

The simplest example (Figure 1a) is a count of a specific molecular species. This type of register is incremented by the synthesis of another copy of this species and decremented by the degradation of one of these molecules. For example, if the molecular species representing the register is a protein, then the molecular machinery associated with the expression of the protein (transcription factors, RNA polymerase, ribosome, etc.) collectively implement the increment operation. The decrement operation is implemented by mechanisms that can degrade the protein (e.g., ubiquitination and a proteasome or a protease specific to the register protein). We question whether, for living systems,

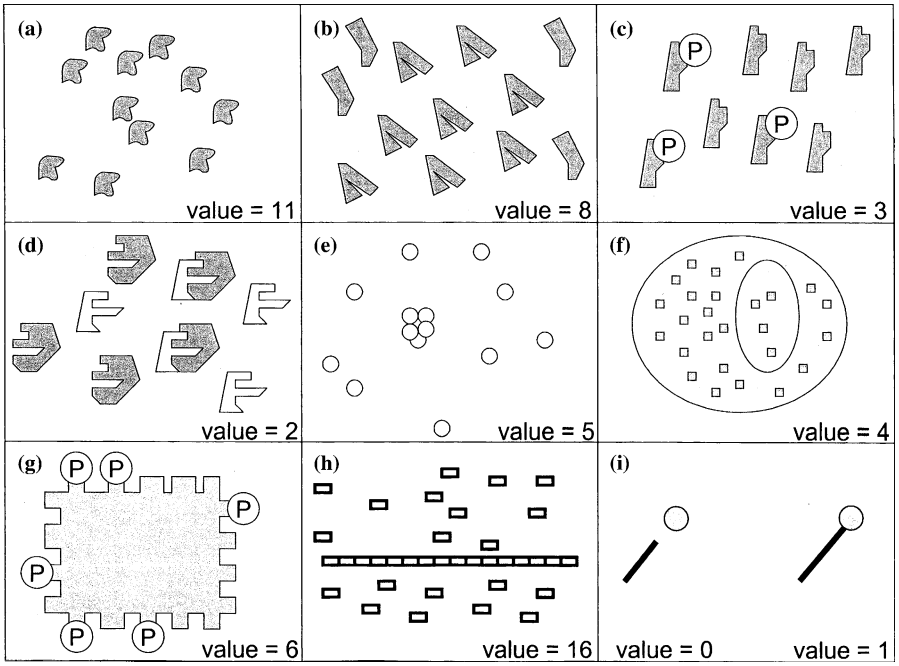


Figure 1. Illustration of different protein implementations of unary registers and their values. (a) Count of proteins. (b) Count of proteins in the folded conformation. (c) Count of phosphorylated proteins. (d) Count of assembled heterodimers. (e) Count of objects in the pile. (f) Count of proteins within the inner membrane. (g) Count of phorphorylated sites. (h) Length of polymer. (i) Boolean. whether or not the MT (line) is bound to the stabilizing protein (circle).

the halt operation is needed, because one could view every program as a subroutine instead, which leads to another subroutine. Even death of an organism is not really the end of the program. The molecules of the organism are degraded and processed in other ways, and are used as inputs to other programs, as those molecules are taken up into other organisms. Even if a halt operation were needed, it does not require an explicit implementation. The program would just have no next instruction to go on to.

A second example implementation of the register plus operations relies on a molecular species being able to remain stable in two alternative conformations, say, conformation A or B (Figure 1b), with the value of the register given by the number of molecules in conformation B. Assume that the protein is folded into the default conformation A when it comes out of the ribosome. Then, a protein that can

change this protein from conformation A to B can serve as an increment operator. A different protein that can change B conformation back to A acts as a decrement operator. Additionally, if B is a metastable state, and will spontaneously (although perhaps slowly) revert to conformation A, this can be a decrement operation in itself. In this case, then, the number of molecules in conformation A is irrelevant (as long as there is always one available to convert to B when needed). New proteins in conformation A can be synthesized and degraded, and this does not affect the value of the register at all.

A third example is to represent the register by a protein that can be phosphorylated (Figure 1c). A kinase (an enzyme that phorphorylates a specific substrate) associated with the register protein can act as an increment, while the associated phosphatase (an enzyme that dephosphorylates the substrate) can serve as a decrement. In the same way, other chemical groups that can be bound to proteins, such as methyl, nucleotidyl, or fatty acyl groups, can also be used to represent a register.

There are other variations on this theme. If two molecular species, say, C and D, that can be stably bound together (e.g., via covalent or very strong non-covalent bonding) into a heterodimer CD, then the number of CD heterodimers can represent a register (Figure 1d). A protein driving the dimerization increments the register; one that breaks the dimers into their constituents C and D decrements the register. This same scheme works even if the two molecules are the same, e.g., CC dimers could represent a register. Alternatively, the assembled complex could be more complicated than a dimer. For example, the number of ribosomes in a cell could represent a register, with all the 80-some proteins and half a dozen or so ribosomal ribonucleic acid (rRNA) molecules having to be assembled in order to increment the register. Degradation of the ribosome, effectively taking it ''out of service,'' represents a decrement.

Rather, than representing a register by the overall number of objects anywhere in the cell, it could be represented by a localized population – a ''pile'' of objects (Figure 1e). Any mechanism that transports objects to the specific location, to add them to the pile, can increment the register. Removing from the pile, transporting away from the location, is a decrement. An example is motor proteins moving cargo to/from a specific location. An abacus, with the physical location of beads on each wire representing numbers, is reminiscent of this type of register. An alternative implementation is the number of molecules of a particular species within a specific membrane-bound

compartment (Figure 1f). The transfer of a register molecule across a membrane from one compartment (e.g., the cytoplasm of a cell) into another compartment (e.g., into an organelle) effectively decrements one register and increments the other.

Some molecules or molecular complexes have a large number of phosphorylation or methylation sites. The number of these sites that are phosphorylated or methylated can represent a register, with the appropriate proteins acting as increment and decrement operators (Figure 1g). Examples of this are methyl-accepting chemotaxis proteins (MCPs), which are transmembrane attractant receptors used in bacterial chemotaxis. MCPs have multiple methylation sites, and the number of these sites that are methylated controls signal transduction, allowing the sensitivity of the signaling network to adapt to changing concentrations of attractant or repellant (Ideker et al., 2001)

Yet another representation is the size or magnitude of an assembled structure, for example, the length of a polymer, either in units of monomers or in units of length (e.g., nm). (Figure 1h) Obvious examples of this are the cytoskeletal filaments MTs, actin filaments, and intermediate filaments. Each of these polymers is made up of repeating units of the same molecular building block, self-assembled in a stable manner. The length of one of these filaments can represent a register. Processes that grow (shrink) the filament serve as increment (decrement) operations. We mentioned at the outset of this section that molecules other than proteins can also act as RAM computing elements. Deoxyribonucleic acid (DNA) provides two examples of polymer registers. At the ends of chromosomes in eukaryotes, telomeres contain repeated nucleotide sequences. ''The idea that telomere length acts as a 'measuring stick' to count cell divisions and thereby regulate the cell's lifetime'' is supported by experiment (Alberts et al., 2002, p. 265). Thus, telomeres can be viewed as registers that are decremented by cell division. King et al. (1997) proposed the intriguing possibility that the number of simple sequence repeats (SSRs) in microsatellite DNA acts as a ''tuning knob'' influencing gene activity. If this is correct, then SSRs can be viewed as unary registers whose values are important parameters for the algorithm associated with gene expression.

Figure 1i illustrates another embodiment of a register that serves well as representing a Boolean value. Analogous to the examples shown in Figures 1c, d and g, it relies on the binding of one type of molecule to another. However, in this case of Figure 1i, there is only one MT (line) and one stabilizing protein (circle), so there are only

two possible values to the register, 0 (unbound) or 1 (bound). In bacterial chemotaxis, the direction of the flagellar motor (clockwise or counter-clockwise) can also be viewed as a register with only two values.

Any of these representations of registers and their operations are equivalent. That is, any physical embodiment of these numerical and Boolean registers can be used to compute the same algorithm. Thus, in living systems, many alternative sets of "hardware" can carry out the same "software." This list of representations is not exhaustive, but even so, it is interesting to note how many assembly, degradation, transport, synthesis, etc. activities (i.e., dynamic self-assembly activities) do map onto these essential properties of the RAM model of computing.

3.2. *Sequencing operations on registers*

The third ingredient of RAM computing is sequencing the operations. Just as there are a number of different physical manifestations of registers and operations, there are multiple ways that sequencing can be implemented. They all rely on a few crucial properties of proteins and their interactions that we have identified as important for dynamic self-assembly and RAM computation. (1) Proteins have tremendous selectivity of their binding sites, analogous to a lock and key. (2) Binding or unbinding a ligand at one of these sites can result in a conformational change of another part of the protein. This conformational change can perform some sort of actuation, such as moving (e.g., in motor proteins) or driving an assembly or disassembly reaction. (The reader should notice that these two properties are also important for all of the embodiments of registers and operations described above.) (3) A conformational change can also expose (or hide) additional binding sites, which in turn can bind and cause a conformational change resulting in actuation, or exposing or hiding yet another binding site. See Figure 2.

Signaling cascades are easily recognized as a set of interactions that proceeds in a controlled sequence. A specific sequence of interactions is "wired" together by mixing together a set of molecules with binding sites that drive them to execute sequentially. For example, suppose each protein has a "trigger" site that activates it (e.g., the phosphorylation site of a kinase that must be phosphorylated itself in order to be active) so that other sites are exposed, such as a catalysis

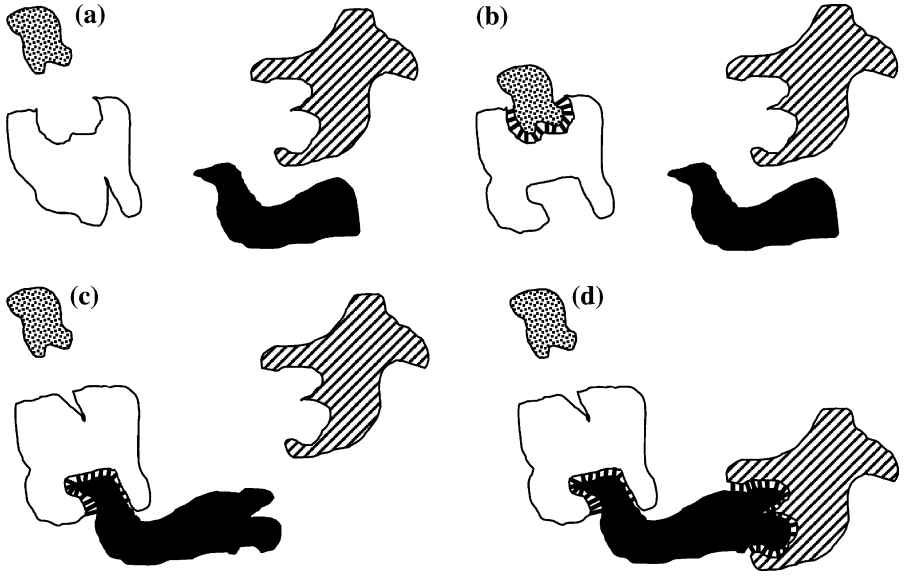


Figure 2. Illustration of protein dynamic self-assembly properties important for sequencing RAM computations. Of the four proteins shown in (a), the textured and white proteins on the left can form non-covalent bonds through selective binding sites (b). This results in a conformation change of the white protein, exposing another binding site, to bind to the black protein (c). This binding again changes the conformation of the white protein to release the textured one to which it was bound. It also changes the conformation of the black protein, exposing another binding site, so it can bind to the striped protein (d). The order in which these proteins interact is driven by the order in which matching binding sites are exposed.

(kinase) site and an ATPase site (to bind and hydrolyze ATP). Suppose kinase A's catalysis site binds to kinase B and phosphorylates its trigger site, kinase B's catalysis site phosphorylates kinase C's trigger site, and kinase C's catalysis site phosphorylates kinase D's trigger site. Once A is triggered, then B will execute, followed by C, followed by D.

The kinase cascade described above can be viewed as either a sequence of operations strung together, or as a single sequencing signal. If each activated kinase species represents a register as in Figure 1c, then one kinase acts as an increment operator on its substrate. Therefore, the A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D pathway would be viewed as sequencing three increment operations together (A increments B, B increments C, and C increments D). Alternatively, if the entire

cascade is sandwiched between other register operations (e.g., the expression of more copies of one protein and the degradation of another), and the count of molecular species A, B, C, and D is not important, then the entire cascade may be viewed as a single jump from one operation to the other. These cascades can thus be viewed either by the input–output circuit model used by other authors or by the RAM model, whichever is most convenient for the current purposes.

This dynamic, stochastic sequencing mechanism, provided by the binding-driven conformation changes of proteins (property (3) above, and shown in Figure 2), enables Nature to wire together RAM algorithms in a modular, evolvable (Kirschner and Gerhart, 1998) way. The notion that natural algorithms are modular and evolvable is supported by the observation that some protein network motifs are seen over and over in Nature (kinase cascades, for example). These networks may be carrying out the same algorithm, but wiring together the operations on different registers. We can also provide a plausibility argument for how this sequencing mechanism can facilitate the evolution of new sequences from existing ones. Returning to our hypothetical kinase cascade, suppose the effect of phosphorylating (incrementing) D is to initiate the expression of protein X (because D is a critical transcription factor for X, for example). Then suppose the gene for D is duplicated and mutated so that a protein E is produced that has one protein domain that is identical to D (the one that binds to C), but another one that is different to stimulate the expression of Y instead of X). Then the network A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ X is rewired to A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ E $\rightarrow$ Y. This dynamic, stochastic sequencing mechanism enables a potentially major rewiring of the algorithm sequence, resulting in the expression of Y instead of (or in addition to) X, from the relatively small change of a single protein domain.

Another mechanism for enforcing sequence is with scaffold proteins. These proteins enforce a particular sequence of operations by physically tethering or activating specific proteins (operators) in a particular order. In the budding yeast *Saccharomyces cerevisiae*, scaffold proteins Ste5 and Pbs2 are essential to the mating and high-osmolarity response pathways, respectively (Park et al., 2003 and references therein). Figure 3 illustrates the role of the scaffold proteins in directing the appropriate output from the specific input. Although both mating and osmoresponse pathways involve the protein Ste11, they exhibit no cross-signaling under normal conditions. Ste5 has binding

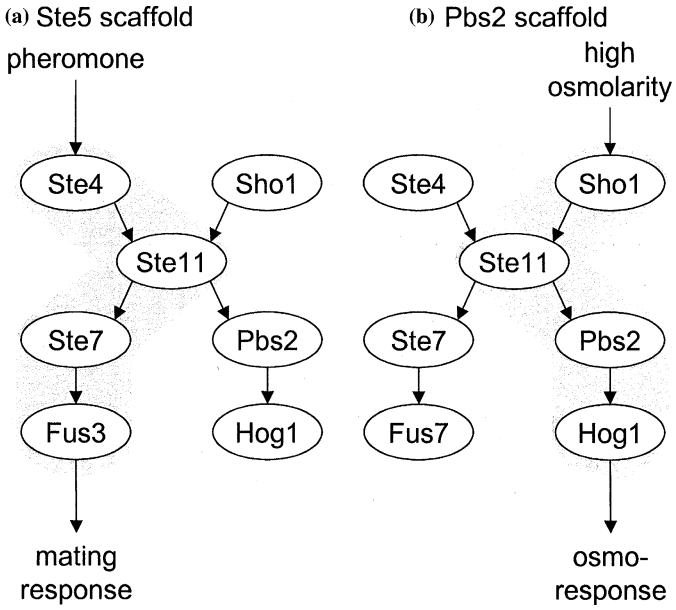


Figure 3. After Park et al. (2003), illustration of scaffold proteins that control sequencing of mitogen-activated protein kinase (MAPK) cascades, The gray background indicates which proteins are bound by the scaffold protein. Although the MAPK kinase kinase (MAPKKK) Ste11 is involved in both cascades, the scaffold Ste5 (a) ensures that the mating response is the' outcome of pheromone detection, whereas Pbs2 (b) directs the osmoresponse to high osmolarity.

sites for the input Ste4, as well as Ste11, Ste7, and Fus3, which results in the mating response. In contrast, Pbs2 binds a portion of the osmosensor Sho1 as input, and also binds Ste11 and Hog1. Pbs2 itself also plays a catalytic role in this pathway, leading to the osmoresponse (Park et al., 2003)

Interesting new work suggests that pathways directed by scaffold proteins are also modular and evolvable. Park et al. (2003) have shown that it is possible to wire together new scaffold proteins, to redirect the signaling pathway. Specifically, they engineered a diverter scaffold that enforced a non-natural pathway, in which pheromones selectively triggered the osmoresponse in *S. cerevisiae*. They concluded that "scaffolds are highly flexible organizing factors that can facilitate pathway evolution and engineering." (Park et al., 2003, p. 1061)

It should be noted that in both types of sequencing mechanisms described here, the binding rates between protein components are stochastic. Thus, the execution time of a particular sequence of operations

is not fixed. Rather, it follows some distribution. Additionally, during the time between the exposure of a binding site and the event of it binding to a ligand, the algorithm can be re-wired "on-the-fly" by the intervention of a different compatible ligand. For example, the D-binding site of the C kinase in the pathway A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ X may be exposed, but before it binds to a D molecule, an E molecule is synthesized and interacts with the C, thus dynamically rewiring the pathway to A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ E $\rightarrow$ Y. This effect is one factor that makes computing via protein dynamic self-assembly processes *stochastic* and *nondeterministic* RAM computing.

3.3. *Branching (decision-making)*

The rewiring of a pathway just described is one form of branching, or decision-making. Another type of branching results from a competition between two populations, e.g., kinases and phosphatases that act on the same type of protein. Assume that when the target protein is phosphorylated it is activated as an enzyme for some reaction. In a deterministic sense, if there are more kinases than phosphatases for the target proteins, then the kinases will "win," and some target proteins will remain phosphorylated, so that they can catalyze their reaction. If there are more phosphatases, then they "win," and no target proteins stay activated long enough to perform their catalytic function. This represents an algorithm like "if $a > b$, then do x", where a represents the population of kinases, b the population of phosphatases, and "do x" is the reaction catalyzed by the target protein. However, because the kinase and phosphatase molecules diffuse around in the "soup" of the cytosol, reacting at random with the target proteins, there is a "race" between the two species to react with the target protein. Sometimes when there are fewer kinases than phosphatases, a phosphorylated target protein will have the opportunity to perform its catalytic function before it becomes dephosphorylated. Similarly, occasionally when there are more kinases than phosphatases, the target protein will not get a chance to catalyze a reaction before being dephosphorylated. Statistically, when $a \gg b$, "do x" will almost always occur, and when $a \ll b$, "do x" will almost never occur, but when $a \sim b$, there is a smooth sigmoidal transition where the probability of "doing x" goes from 0 to 1. Thus, the stochastic nature of such races leads to "errors" in computation, particularly when the race is close.

Decision making or branching can also be accomplished with only one or two molecules. The only requirement is two possible outcomes. For example, a new tubulin dimer could be added to the end of a MT (increment), or the last tubulin dimer might dissociate (decrement). A motor protein may take another step along the cytoskeletal fiber to which it is currently bound, or it might fall off. In both of these cases, there is a "race" between two stochastic events. There are two possible outcomes; the decision is made by whichever one happens first. In general, the decision-making/branching we observe in Nature involves some sort of race or competition. The decision is made to follow the "winner" – the faster, stronger, or more numerous.

3.4. *Hierarchical RAM computing in living systems*

The RAM computing model is applicable to living systems not only at the level of protein networks, but at other hierarchy levels as well. By encapsulating proteins within membranes (the nuclear membrane, organelle membranes, or the cell's plasma membrane), RAM algorithms can be carried out at the level of cellular networks. Then, by encapsulating cells within another "membrane" (e.g., a skin or exoskeleton), algorithms can again be carried out at the level of organism networks. In this way, RAM programs can be scaled up to greater spatial and temporal scales.

Since cells and organisms are readily observed to make decisions and change their behavior, the branching and sequencing aspects of the RAM model are obvious at these levels, so we will not discuss them further. What may not be obvious is how the registers and their operations are represented at the cellular or organism levels. Here, again, living systems embody registers in a unary representation, which can perhaps account for their lack of previous recognition. Figure 4 shows some examples of similar registers and operations at the levels of proteins, cells, and multi-cellular organisms. In Figure 4a, the number of proteins is the register, and synthesis and degradation of the proteins are the increment and decrement operations, respectively. Compare this to Figure 4b, in which the number of cells is a register. Cell division increments the register; apoptosis (or other mechanisms of cell death) decrements the register. Figure 4c shows a population of organisms as a register. Birth is an increment; death is a decrement.

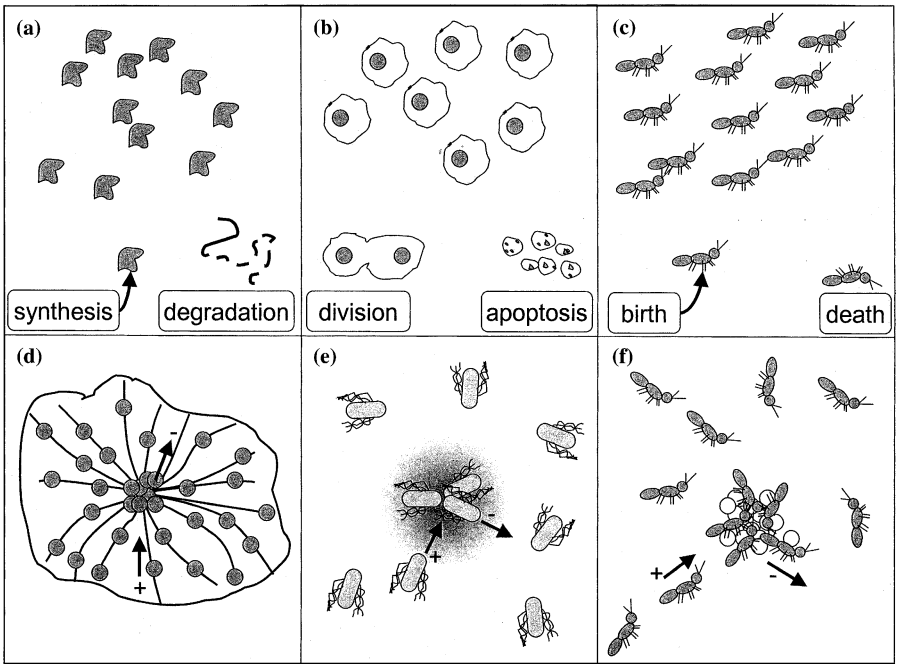


Figure 4. Illustration of registers and their operators at different hierarchy levels. In (a)–(c), the register is represented by the number of objects, (a) molecules, (b) cells, and (c) organisms. The increment (decrement) operations are shown in the lower left (lower right) corners of each panel. In (d)–(f), the register is represented by a localization of objects in the center, (d) intracellular cargoes, (e) bacteria, and (f) ants. Increments (decrements), illustrated by + (−) arrows, are implemented by (d) motor proteins adding (removing) cargoes from the center, (e) bacteria moving by chemotaxis to (from) the food source, and (f) ants walking to (from) the food pile.

We mentioned above (and illustrated in Figure 1e and f) that a localized collection of objects can represent a register. Figure 4d illustrates a cell in which motor proteins carry cargo by "walking" on MTs (lines radiating outward from the center). MTs are cytoskeletal filaments that nucleate on a protein complex, the centrosome, usually located near the nucleus of a eukaryotic cell. The number of cargoes at the centrosome represents a register. Motor proteins carrying cargo to the centrosome increment the register; those carrying cargo away decrement it. This is analogous to Figure 4e, which illustrates bacterial chemotaxis. The number of bacteria that have reached the attractant (e.g., food source) in the center represents a register. Each individual bacterium's arrival at the food source increments the

register. A bacterium that leaves the attractant serves as a decrement. Figure 4f shows ants arriving at a food source, incrementing the register (number of organisms at the source). When an ant leaves the source, it decrements the register.

Although these examples are far from exhaustive, they illustrate a key point. Just as, at the molecular level, the same algorithms can be implemented by different hardware, the same algorithms can even be implemented at different hierarchy levels, again with different hardware. We hypothesize that the algorithms that life does implement are the ones that (i) result in the persistence of life, and (ii) can be reproduced at the next level, so that life can scale up and evolve more complex forms. Indeed, there may be survival advantages for algorithms that can be implemented in many different hardware forms. That is, the algorithms that survive evolution may be the very ones that can be re-used robustly in many different contexts.

4. Numeric computation with idealized protein networks

So far we have discussed how protein dynamic self-assembly (or cells or organisms) can be used to implement all the individual ingredients of the RAM model of computing. Now we demonstrate how these ingredients can be put together to create a specific algorithm. We present stochastic simulations of protein networks explicitly designed to implement a numeric computation. Although this numeric computation is not a natural algorithm, this demonstrates how the RAM computing properties of proteins we have been discussing are sufficient to support true computing in an easily recognizable form. Indeed, we believe it is possible that with today's biotechnology capabilities, these artificial proteins could be synthesized to carry out numeric computation if desired.

4.1. *Stochastic simulations*

Computer simulations are applied widely to simulate cellular processes and predict future behavior. See, for example, Endy and Brent (2001) and references therein. For example, the chemical networks controlling bacterial chemotaxis (Bray et al., 1998; Alon et al., 1999), developmental patterning in *Drosophila* (Burstein, 1995), and infection of *E. coli* by lambda phage (McAdams and

Shapiro, 1995) have been modeled. Stochastic methods (Gillespie, 1976) are particularly valuable in representing chemical reactions involving relatively small numbers of reactant molecules, when the continuous-variation approximation of differential equation methods breaks down.

To demonstrate numeric computation using the RAM computing protein properties described above, we developed stochastic simulations similar to those employed by the biological community. Each molecule is represented by an "agent," a collection of binding sites that can bind selectively to sites of other molecules. To initialize the simulation, pairs of molecules with matching binding sites are selected for reaction. A time for each reaction is randomly selected from an exponential arrival-time distribution, $P(t) \sim \exp(-t/\tau)$, where $P(t)$ is the probability of the reaction occurring at time t, and the parameter τ is characteristic of the reaction type. The reactions are scheduled on a priority queue with the smallest reaction time in root position. Then, the first reaction is removed from the priority queue and executed. In the process of executing the reaction, the protein can (via a conformation change that is not explicitly modeled) actuate, such as taking a step, and/or expose, hide, or change the reaction rates of other binding sites. The events on the priority queue are updated accordingly. Then the next reaction in the priority queue is removed and executed. This process is repeated until there are no more reactions on the queue or the user terminates the simulation. This approach is similar to the Gibson-modified Gillespie algorithm (Gibson and Bruck, 2000); a detailed description of the simulation infrastructure is provided elsewhere (Bouchard and Osbourn, 2004).

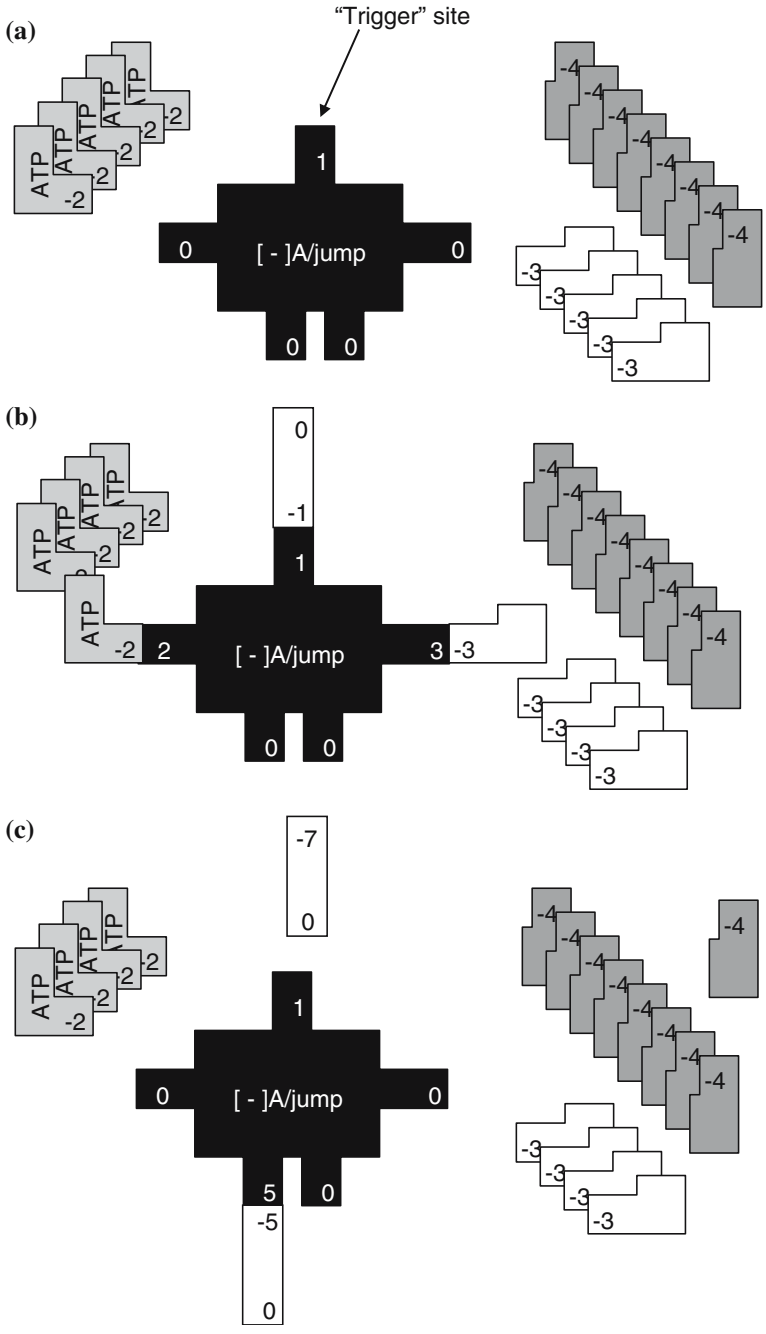
4.2. *Idealized proteins as RAM components*

For simplicity, we implemented register proteins of the variety illustrated in Figure 1b. A register protein can be in one of two conformations, A or A′, for register A, for example. The number of proteins in conformation A′ represents the value of the register. Increment and decrement/jump operators change a register protein from one conformation to the other. We could have implemented any other embodiment of registers and their corresponding increment and decrement/jump operators, and the results would be identical (with stochastic variations).

Protein dynamic self-assembly processes are non-equilibrium, energy-driven processes. Adenosine triphosphate (ATP) is a common energy currency used by Nature. We have designed the increment and decrement/jump operators to harness the hydrolysis of ATP to drive the conformation change of the register proteins. Note that, since these idealized proteins are to perform a numeric computation, we have designed them explicitly to only increment or decrement the register by 1 each time it is triggered. This is in contrast to the more common behavior of real enzymes, which can catalyze numerous reactions after initial activation. This design is justified, however, by the fact that some proteins do have a mechanism for repeated triggering by consuming free energy. For example, it has been shown that kinesin, a motor protein, hydrolyzes one ATP molecule per step (Schnitzer and Block, 1997). This could be viewed as having to be triggered again (by the arrival of a new ATP molecule) after each step.

In order to be able to distinguish between the increment and decrement/jump operators of different registers, we adopt the notation [+]*reg* and [−]*reg*/jump, where *reg* can be any register A, B, C, etc. We refer to any molecule – protein, ATP, etc. – as an "agent." Signaling proteins, which can toggle between two conformations, exposing only one of two binding sites at a time, are used to wire together sequences of [+]*reg* and [−]*reg*/jump operators. Figure 5 illustrates the interactions of the [−]A/jump agent with the signaling proteins, register proteins, and ATP. Agents are represented by polygon shapes. The binding sites are shown as tabs at the perimeter of the agent. Complementary sites are indicated with numerical keys. (e.g., 4 and −4 are complements). When the sites of two agents are bound, they are shown as touching. The [−]A/jump agent is labeled, as are the ATP agents. The two collections of agents to the right represent a single register. The alternate conformation (A′) of the register protein is shown in white. Initially, in panel (a), the value of the register is five, and the [−]A/jump agent has a single "trigger" binding site exposed,

Figure 5. Illustration of the decrement operation. (a) The [−]A/jump and ATP agents are labeled. The two collections of agents to the right represent a single register with a value of 5 (A′ agents are white). (b) When the [−]A/jump agent is triggered, it binds to an ATP and an A′ register protein. (c) Then it changes the conformation of the register protein, thereby decrementing the register, releases the ATP and register protein, and signals success.


(a)
"Trigger" site
ATP
-2
1
0
[-]A/jump
0
0
0
-4
-3
(b)
0
-1
1
2
3
-3
(c)
-7
0
5
-5

with a key of 1. It also has four other sites that are hidden (they have an invalid key, 0). In panel (b), when a signaling agent with a complementary key of −1 binds with the trigger site of the [−]A/jump agent, two additional sites are exposed, with key values of 2 and 3. When the trigger site unbinding event occurs, if (as in panel (b)) both the ATP and register proteins are bound to these two sites, then in panel (c), the hydrolysis of ATP drives the [−]A/jump agent to change the conformation back to the default (note that in panel (c), there are only four A′ register proteins, and an additional A version), release it and the "spent" ATP, and expose the "done" site with a key of 5. When the trigger signaling protein breaks from the [−]A/jump agent, it changes conformation (exposing a −7 key) so that it cannot immediately re-trigger the operation. Some other agent will have to bind to the exposed −7 binding site and toggle the trigger back to a key of −1 in order to re-trigger the [−]A/jump agent. In panel (c), a signaling protein with a key of −5 is bound to the done site of the [−]A/jump agent. When released, it will toggle to a key of −8, enabling it to act as a trigger signaling protein for the next [+]*reg* or [−]*reg*/jump operator (not shown) in the sequence (whatever operator has an exposed trigger site with a key of 8).

If there had been *no* register protein bound when the trigger site unbinding event occurred, then the "jump" site (lower right site of the [−]A/jump agent in Figure 5) would have been exposed with a key of 6, rather than the done site with a key of 5. As a result, a different signaling protein would become bound to the jump site, and a different execution path would follow. Certainly, if there are no A′ versions of the register protein (i.e., the register value is zero), then the jump pathway will be taken. However, there is a "race" between the arrivals of the register protein and ATP on the one hand and the unbinding of the trigger site on the other. The stochastic nature of this race will produce incorrect jumps (when the register is non-zero) with some probability that depends on the relative rates involved.

The increment agent, [+]*reg*, is similar to, but slightly simpler than, the decrement agent. The binding of the trigger site exposes the ATP- and register-protein-binding sites. The ATP key is the same, 2, but in this case the register-protein-binding site's key is 4, to bind to the default (A) version of the register protein. To increment the register, it changes the conformation of the register protein to A′ (i.e., changes its key to −3), then exposes a done site with a key of 9. There is no "jump" associated with the increment operation.

4.3. *Numeric computation*

We have implemented simulated protein networks for elementary operations such as zeroing a register, register copying, adding contents of one register to another, using a register to control the number of loops through a repeated sequence of agent operations, multiplying two register contents into a third register, and computing a modulus of a register value.

To illustrate how this simple set of agents can accomplish such computations, Figure 6 shows a schematic diagram of the network of proteins required to multiply two registers, A and B, into a third register G. Only the increment and decrement agents are shown. Each of these agents in the actual simulation interacts with the register proteins and ATP, as shown in Figure 5, but these are omitted from Figure 6 for clearer viewing of the execution sequence itself. Recall that a signaling protein toggles between two binding sites, so that it can wire together a sequence by binding first to the "done" or "jump" site of one agent, and then to the trigger site of the next agent. A *solid* arrow represents a pathway that a signaling protein makes from the *done* site of one agent (tail of the arrow) to the trigger site of the next agent (head of the arrow). A *dashed* arrow represents a signaling protein's pathway from the *jump* site of one agent to the trigger site of the next agent.

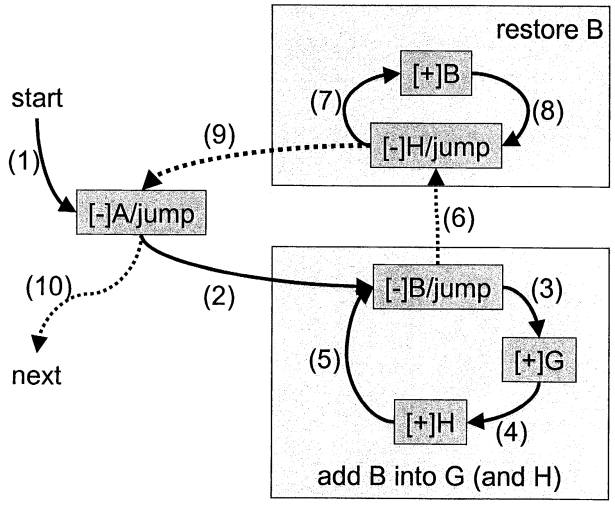


Figure 6. Schematic diagram of protein network to multiply registers A and B into register G (see text for discussion).

The order in which the signals are propagated is indicated by a number in parentheses along the signaling pathway. We will describe the sequence using an example in which registers A and B are initially set to 2 and 3, respectively, and G and H are both 0. A start signal (1) triggers the decrement of register A, so that we now have A = 1. We then (2) enter a loop contained in the lower box in the figure. In this loop, B is decremented, (3) G is incremented, and (4) H is incremented. (It will become apparent shortly why we must increment H.) The loop is repeated (5), beginning with the decrement of B. After three passes through the loop, B = 0, and G = H = 3. This loop has the effect of adding the value of B into registers G and H. The next attempt to decrement B will find a zero-valued B register and therefore jump (6) to the next loop to restore B from H (upper box). In this loop, (7) and (8), H is decremented and B incremented until H = 0 and B = 3. When we attempt to decrement H again, it jumps (9) to decrementing A (A = 0), and then the entire outer loop, (2)–(9) is repeated, so that G = 6(= 2*3, the original values of A*B). On the next attempt to decrement A, it jumps (10) to whatever the next operation might be in a more extensive calculation.

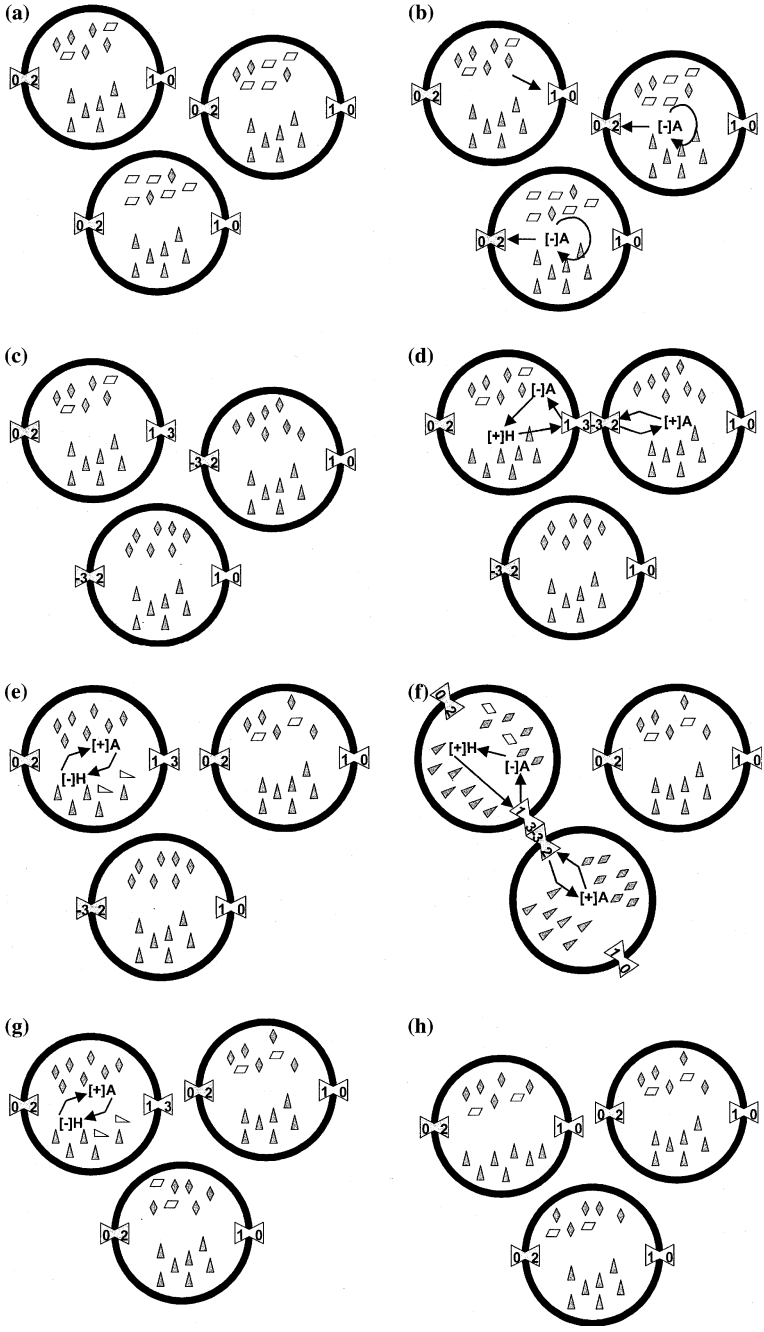
For this illustration, we have described the ideal, ''correct'' behavior of the network. However, any time a decrement occurs, it could jump even though the register is non-zero, due to the stochastic nature of this agent. So, in fact, there are numerous opportunities for errors in even this simple computation.

4.4. *Simulation results and discussion: stochastic computing, errors, and entropy*

We present results of stochastic simulations of encapsulants computing $g = (a * b) + (c * d) + (e * f)$, where a through f are initial values of registers A–F, and g is the answer stored in register G. An encapsulant is analogous to a cell's plasma membrane, in that it isolates its internal population of agents from binding externally. Thus, multiple encapsulants can have identical agent populations carrying out the same calculation in parallel with no interference. We simulate a small population of encapsulants with identical internal component populations and examine the error rates and configurational entropy (S_{config}) of this system as a function of time. For this analysis, we consider two encapsulants to be in the same configuration if all of the [+]*reg*

and [−]*reg*/jump agents and signaling proteins are in the same binding state and all of the register populations have the same associated integer value. S_{config} of these small populations can be zero when all encapsulants are in the same configuration, so that these encapsulants are far from equilibrium. The stochastic nature of the jump operations and stochastic rates of the other operations means that such a set of identically configured encapsulants with $S_{\text{config}} = 0$ will not remain so, and S_{config} will tend to increase with time (but not monotonically, as we show below). The maximum S_{config} condition is for each encapsulant to be in a unique state.

The simulation begins with a population of 10 duplicate encapsulants, but with randomly selected initial register values. The first phase of the simulation is to copy all register values from a single ''starter'' encapsulant to the other nine encapsulants, so that they all begin the calculation with the same values in registers A–F. Figure 7 illustrates the process of copying register A from the starter encapsulant to two other encapsulants. Initially (Figure 7a), the encapsulants have different values in register A. The starter encapsulant signals one ''copy-signal'' transmembrane protein per register (Figure 7b), which exposes sites externally (Figure 7c). In the other encapsulants, an algorithm is launched to zero out all of the registers (Figure 7b). When this is complete, they signal to transmembrane ''copy-receptor'' proteins, one for each register, so that they expose sites external to the encapsulant (Figure 7c). The copy-signal-A from the starter encapsulant binds to the copy-receptor-A of any of the other encapsulants (Figure 7d). The copy-signal/copy-receptor complex provides inter-encapsulant (analogous to intercellular) communication. Note that this is analogous to individual proteins' exposing binding sites and binding together, only it is on the next higher hierarchy level. The starter encapsulant decrements A, increments H, and then signals the copy-signal protein. The copy-receptor receives the signal and increments that encapsulant's A register, then returns the ''done'' signal to the receptor/signal complex, which stimulates another decrement in the starter encapsulant. This loop repeats until the starter encapsulant's register A is zero. In this way, the value of register A gets copied from one encapsulant to another. In the process, the starter encapsulant also copies A to register H, so that when the copy to the other encapsulant is complete, it can restore the value of A (Figure 7e). Note that this copy algorithm is identical to the two boxed loops in Figure 6, except that the G register is replaced by the A register of a different encapsulant.


(a)
(b)
[-]A
[-]A
(c)
(d)
[-]A
[+]H
[+]A
(e)
[+]A
[-]H
(f)
[+]H
[-]A
[+]A
(g)
[+]A
[-]H
(h)

The starter encapsulant then breaks the bond to the current encapsulant and can bind to another encapsulant's exposed copy-receptor and copy A again to that encapsulant (Figure 7f), restoring A from H again each time (Figure 7g). It keeps looping through encapsulants until all of its registers have been copied to all other encapsulants. After some waiting time during which no other encapsulant binds to the starter encapsulant's copy-signal, it decays to the inactive state, so that it is no longer exposed externally. If the copying occurred without error, the system is left in a zero-entropy configuration (Figure 7g). However, there is a non-zero probability that one or more register copy operations will produce an incorrect register value, so this process occurs with some "yield." When the copying is completed, a synchronizing encapsulant is used, to trigger the calculation (Figure 8). (Again, protein signals are used to communicate between encapsulants at the "cellular network" level.) The calculation process then proceeds to completion, also with some "yield" of correct register values.

The averaged yields of final results were obtained from 220 simulations. Figure 9a shows the average yield for the computation as a function of ATP concentration. These results make clear that the dynamic, non-equilibrium behavior of these encapsulated protein networks is driven by the free energy of the ATP population. If the system does not have sufficient energy (ATP), it cannot perform the computation correctly. Even at high ATP concentrations, the yield is not perfect, due to the stochastic race of the decrement/jump operations.

Figure 9b shows a scatter plot of final normalized entropy (S_{config} divided by its maximum) as a function of errors in the final answers.

Figure 7. Illustration of copying register A from starter encapsulant to the other two encapsulants. Register A is represented by parallelograms (A′) and diamonds (A). Register H is represented by triangles. (a) Initially, the values in the encapsulants are $a = 2$, 4, and 5; $h = 0$. (b) Starter encapsulant (upper left) signals to copy-signal; other encapsulants zero register A, then signal to copy-receptor. (c) Starter encapsulant's copy-signal exposes 3 externally; other encapsulants' copy-receptors expose -3 externally. (d) Starter encapsulant and second encapsulant bind via copy-signal/copy-receptor complex, and copy the starter encapsulant's register A to its register H and the other encapsulant's register A. (e) Starter encapsulant restores A from H; second encapsulant's copy-receptor no longer externally exposed. (f) Starter encapsulant and third encapsulant bind and copy. (g) Starter encapsulant restores A from H; third encapsulant's copy-receptor no longer externally exposed. (h) After copying to all encapsulants, starter encapsulant's copysignal no longer externally exposed; encapsulant population in zero-entropy state.

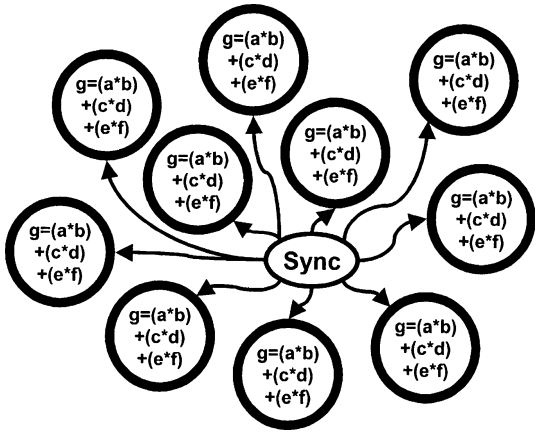


Figure 8. When all registers A–F have been copied from the starter encapsulant, the sync encapsulant releases signals to all 10 of the identical encapsulants to begin computing $g = (a*b) + (c*d) + (e*f)$ at nearly the same time.

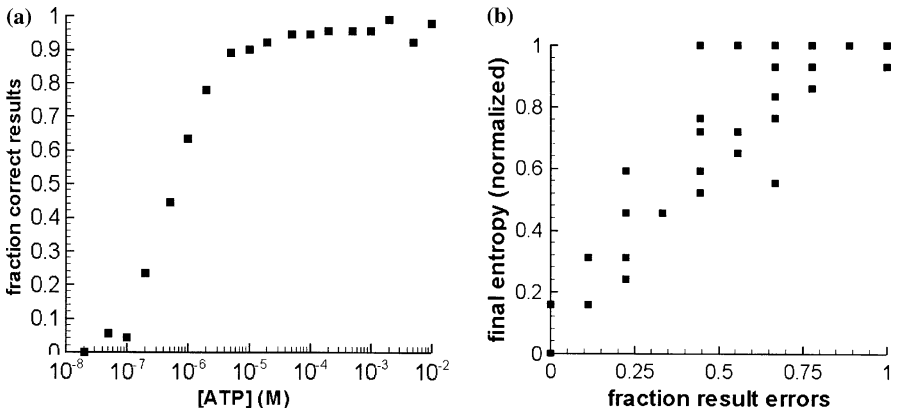


Figure 9. Results of 220 stochastic simulations of 10 encapsulated idealized protein networks, each computing $g = (a*b) + (c*d) + (e*f)$. (a) The fraction of encapsulants with the correct final results as a function of ATP concentration, (b) Normalized final entropy vs. fraction of encapsulants with errors in their final results.

These results show that ending in a more highly ordered state (low entropy) is clearly correlated with high yields of correct computational results (low errors), so that maintaining far-from-equilibrium configurations is the desired outcome for these protein networks. The entropy captures all configurational differences, including those that do not disrupt the final register values, and this produces the scatter in the plot.

The S_{config} as a function of time for a single computational run is shown in Figure 10. We have chosen a case where all of the encapsulants are correctly copied, and all but one of the encapsulants achieved the correct result. S_{config} begins at a large value due to the initial randomized values of the registers in each encapsulant. The register-copying phase is completed at $t \sim 5000$, in a totally ordered configuration of encapsulants ($S_{\text{config}} = 0$). The calculation is initiated at $t \sim 22000$, and while each encapsulant is performing its calculation independent of the others, their configurations again diverge ($S_{\text{config}} = 1$). Finally, all of the encapsulants reach a finished state, with all but one encapsulant reaching the same final state (low, but non-zero S_{config}). Thus, this non-equilibrium process is cyclic in the S_{config}.

4.5. *Yielding robust outcomes from "unreliable" stochastic processes*

Figure 10 illustrates the tendency of these stochastic computational processes to increase their S_{config} after a computational cycle. After the completion of the computation, the entropy is not exactly zero,

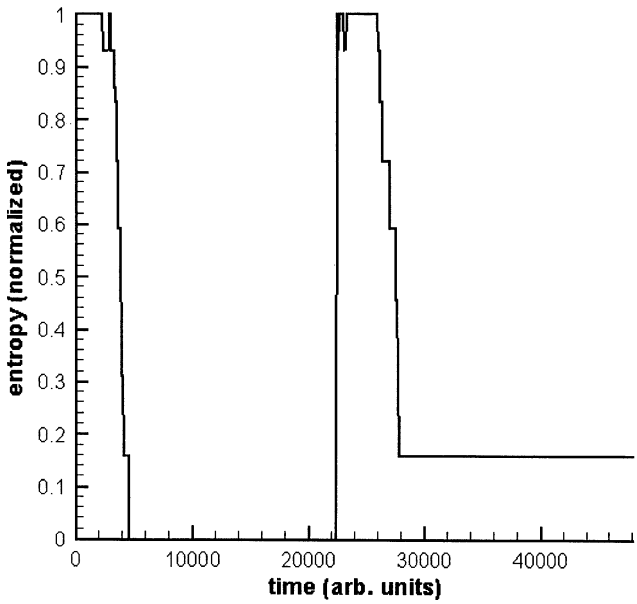


Figure 10. Normalized entropy as a function of time for a single simulation from Figure 9 in which all of the encapsulants are correctly copied, and all but one of the computations achieved the correct result.

but slightly non-zero. This is a manifestation of the slow equilibration of the configurational degrees of freedom. This clearly prevents arbitrarily long computations from being performed in the simple manner described above. That is, if a number of such computations were strung together in a sequence, and if no error correction mechanism is applied to restore the entropy to zero occasionally, then after several computations, the system would have equilibrated to the point where the correct answer could not be recovered. The imperfect yield in the computational processes described here has some similarities to the classic problem of communicating through a noisy channel (Shannon, 1948). Here we have a more general process of noisy computing processes (state transitions) in addition to noisy information transfer.

It may seem objectionable that Nature could rely on stochastic processes, if they give incorrect answers under even favorable conditions, as shown in these simulations. How can a reliable outcome be achieved, if the system is slowly equilibrating? First, we do not expect that living systems perform *numeric* computation to any great extent. Although they do compute, they are computing something other than numbers. We speculate that the algorithms implemented by living systems are really a way of programming dynamic self-assembly. (This notion will be illustrated in the later section entitled ''RAM Programs in Living Systems.'') So, there may not actually be a ''right answer'' in the numeric sense. Second, we know that Nature relies on ''unreliable'' stochastic processes, and yet, is extremely robust. We examine here some of Nature's strategies in dealing with stochastic processes, and discuss how they can indeed achieve reliable outcomes.

One of Nature's strategies is to value function over form, so that many imperfect forms (made by imperfect processes) can successfully fulfill the required function. For example, every bird's nest is unique and imperfect, with different numbers of twigs, different asymmetries, etc., but when completed, imperfect or not, it serves the function of holding eggs. A human's vascular system does not consist of perfectly straight or perfectly positioned blood vessels. In every individual, they are uniquely placed, with bumps and twists; they grow at different rates during development. However, as long as the two vascular systems carrying blood to and from the heart join up, they serve the function of allowing blood to flow in a complete circuit. Whether or not the structure performs its intended function is the criterion for acceptance. A bird's nest with too large a hole will not hold eggs, nor will a vascular system with a large hole or closure.

A second strategy is to cycle imperfect stochastic processes repeatedly until success is achieved. The MT search-and-stabilize algorithm discussed in a later section of this paper is a perfect example of this strategy. Each MT repeatedly cycles through growth and collapse, growing in an arbitrary new direction each time, until it locks onto a stabilizing protein and becomes fixed. Scout ants search for new food sources, but may return to the colony empty-handed. Multiple scouts continue going out in different directions and coming back, collectively covering as much territory as possible until food is found. Then they "lock on" to that food source, recruiting large numbers of workers to gather the food to the nest.

A third strategy is to wait for one stage of growth or development to be fully completed, passing a "checkpoint," before initiating the next step. The cell-cycle control system has checkpoints to ensure that cell division occurs only when conditions are right, and that it proceeds successfully. A cell should only divide if it is large enough and the environmental conditions are favorable for cell proliferation. The G_1 checkpoint halts the cell cycle until the situation is right, before committing itself to DNA replication. The G_2 checkpoint gives the system another opportunity to halt, to ensure that DNA replication is completed, before triggering mitosis (Alberts et al., 1998) Another way to look at this is as a mechanism for restoring order (i.e., configurational entropy to zero) at different stages in the algorithm, with each restoration occurring before the distribution equilibrates too far. We are currently developing simulations to apply this strategy to equilibration of the numeric computation via idealized protein networks discussed above and shown in Figure 10. The simulation implements a hierarchical algorithm (i.e., in which the encapsulants act as agents) to restore low entropy in order to correct computational errors.

In earlier sections of this paper, we identified the properties of proteins that map directly onto the RAM model of computing. The point of this section was to explicitly demonstrate how proteins could carry out a numeric calculation, $g = (a * b) + (c * d) + (e * f)$, that we clearly recognize as computing. We devised artificial proteins (but with all the important dynamic self-assembly properties of real proteins!) to carry out this computation explicitly in the RAM model. We also demonstrated the importance of maintaining far-from-equilibrium conditions in order to achieve a robust (low-error) outcome from the computations, and suggested strategies (algorithms) that biological systems employ for driving down the entropy in order to realize correct outcomes.

5. RAM programs in living systems

Now we move on to real proteins in real living systems and examine what algorithms they actually carry out. That is, by scrutinizing a specific biological process, we identify unary registers and the mechanisms of incrementing and decrementing those registers. Then we track the sequencing of those operations in order to capture explicitly (in pseudocode or a flow diagram) the RAM program corresponding to that biological process. We have two motivations for wishing to understand the algorithms that living systems implement. First, if biological systems are computing via dynamic self-assembly, then these self-assembly processes must be programmable. We wish to learn the "programming language," and then use it to program dynamic self-assembly, to build novel structures and materials. Second, perhaps understanding the algorithms of biological systems will provide some insight into their ability to develop, persist, reproduce, and evolve. In this section, we present two examples of protein behaviors in living systems, and discuss how these behaviors can be viewed as RAM programs. These examples are relatively simple, but they do demonstrate how RAM programs can be implemented in real systems and give us the first steps toward understanding this different kind of programming language.

5.1. *Melanophore programs using active transport*

In fish skin cells (melanophores), pigment granules (cargoes) with both inward- and outward-walking MT-associated motor proteins enable the cell to change color. One cellular signal drives the inward motors to carry the pigment to the centrosome, leaving the body of the cell free of pigment, thereby making the cell transparent. Different cellular signals stimulate the outward motors to disperse the pigment, making the cell opaque (Sköld et al., 2002).

We identify a collection of pigment particles at the centrosome (as in Figure 4d) as register A. An accumulation of pigment particles anywhere else within the cell represents registers B, C, D,..., as many registers as necessary to account for all of the collections of pigment particles at any time. Anywhere a single pigment granule sits by itself is a register of value 1. A motor protein that carries a pigment granule to a pile increments that register, while one that carries a pigment granule away from a pile decrements the register. So, with these definitions, if a pigment granule is sitting by itself ($B = 1$) and a motor

protein moves it somewhere else, this is equivalent to (decrement B, increment C) so that (B = 0; C = 1).

There are two algorithms we want to examine. One is to disperse the pigment; the other is to condense it at the centrosome. We write the algorithms as if they were deterministic and carried out in a single thread, because such notation is easier to make clear. We use italics for variables, normal text for constants and operations.

The dispersal algorithm could be viewed as

```
loop:
    for reg = A, B, C, D, ...
        for i = n(reg) down to 1
            decrement reg
        next i
    next reg
until n(reg) = 1 for all reg
```

where *n(reg)* is the value in register *reg*. Although this is written as if there are three nested loops, nested in a particular order, in reality, there is no real ordering. All of the registers can be decremented simultaneously, and multiple decrements can occur on a single register at the same time. Because when a register is decremented, it might inadvertently increment another register, we add the outer loop to split up piles that accidentally got piled together. In a deterministic sense, the process continues until every pigment granule sits alone. That would be perfect dispersal. However, because of the stochastic nature of the system, there may be places (including at the centrosome) where a small number of pigment granules stay together, so that the registers are not exactly one. Nevertheless, the end result is close enough to have the physical effect of making the cell opaque.

The condensation algorithm can be written as

```
for reg = B, C, D, ...
        for i = n(reg) down to 1
            decrement reg
            increment A
        next i
next reg
```

Again, the loops need not be performed in order. The important part is "decrement *reg*; increment A." All pigment particles can be moved to the centrosome simultaneously. The outcome of a deterministic version of this algorithm would be every granule residing at the centrosome (register A). However, again because of the stochastic nature of the system, perhaps not every particle will be moved to the centrosome, but it will be close enough that the cell is transparent.

These programs are so simple that no branching is required. Since all registers can run in parallel (i.e., the innermost loop of both programs is like a subroutine that can be run on each register simultaneously), there is very little sequencing to enforce either. It just needs a signal to start either the dispersal program or the condensation program on each register. Then the only sequencing within the program is the (decrement *reg*, increment A) sequence of the condensation program. This is implemented easily by the motor protein not stopping until it reaches the centrosome. This behavior is natural to a motor protein as long as the conditions in the cell are such that the inward-walking motor is more active than the outward-walking motor.

5.2. *Scaling algorithms to higher hierarchy levels*

As mentioned previously, the same RAM algorithms can be implemented by hardware at different levels of hierarchy (and therefore at different scales), such as populations of molecules, cells, or organisms. We just described the condensation algorithm for populations of molecules in a melanophore. This same algorithm is applicable also to bacteria in chemotaxis. The motility of each bacterium itself acts as a combination of decrementing another register and incrementing register A, the number of bacteria at the focus of the attractant. Similarly, one could view the number of ants (or food particles) in an ant colony as register A. The ants gather food from other locations (other registers) to the colony, decrementing the other registers and incrementing register A. Hence, this one condensation, or "gathering," algorithm is implemented in Nature on at least three different scales or levels of hierarchy.

5.3. *Search and stabilize program using structural assembly*

The second example program we describe has much more complicated sequencing, with numerous branches and loops. In contrast to the

previous example, where active transport was used to operate on registers, in this example registers and their operations are implemented by the structural assembly of MTs. In order to put this example in context, we briefly review a current well-accepted model of MT growth and dynamics (Alberts et al., 1998, 2002). In Nature, α- and β-tubulin dinners stack together by non-covalent bonding to form the wall of a cylindrical microbule. The cylinder is made from 13 parallel protofilaments, each a linear chain of alternating α- and β-tubulin. In a living animal cell, the concentration of αβ-tubulin dimers is too low to drive the nucleation of the first ring of a MT. Instead, nucleating sites of γ-tubulin are provided by the centrosome, and αβ-tubulin can bind to and grow a MT from each of these nucleating sites. Each free αβ-tubulin is tightly bound to a guanosine triphosphate (GTP) molecule that is hydrolyzed to guanosine diphosphate (GDP) shortly after the tubulin is added to the growing MT. Whereas the GTP-bound tubulin packs efficiently together, GDP-tubulin molecules have a different conformation and bind less strongly to each other. If GTP-tubulin is added faster than the GTP in the MT is hydrolyzed, a cap of GTP-tubulin holds together the growing end, and the MT continues to grow for some time. However, if, due to the randomness of chemical processes, the GTP is hydrolyzed all the way to the end of the MT before new GTP-tubulin is added, the weakly interacting GDP-tubulin at the end will unravel the MT, often in a catastrophic manner. This alternating MT growth and collapse, due to the stochastic nature of the race between adding GTP-tubulin and hydrolyzing the GTP to GDP, is termed *dynamic instability*. If a MT collapses completely, a new one is quickly nucleated in its place, and its growth (and dynamic instability) is, in general, in a different direction from the previous MT. In living cells, dynamic instability is a natural and common behavior of MTs. Dynamic instability can be suppressed, however, by stabilizing proteins that bind to the ends of MTs (or along their length), stabilizing them against disassembly. Such stabilized MTs serve as tracks for the transport of intracellular cargo by motor proteins, and help position organelles where needed by the cell. Therefore, dynamic instability, and specifically growth in a new direction after complete MT collapse, provides a mechanism for searching the space of the cell until the stabilizers are located and bound in a long-term physical structure in support of the cell's activities.

Now let us examine this search-and-stabilize behavior as a RAM computer program. Each MT in the cell is executing the same program, so here we discuss the algorithm for a single MT. We have iden-

tified two registers embodied by the MT, a illustrated in Figure 11. Register A consists of the GTP-tubulin capping the MT; register B is made up of the GDP-tubulin closer to the centrosome. When a new GTP-tubulin dimer is added, register A is incremented. When a GTP-tubulin is converted to GDP-tubulin by GTP hydrolysis, register A is decremented and register B is incremented. If there is no GTP cap, GDP-tubulin dimers can dissociate and decrement register B.

Figure 12 shows a flow diagram of the search-and-stabilize program. STABILIZED may be viewed as a register with a Boolean value. This register is implemented by a bond between the end of the MT and a stabilizer, as in Figure 1i. If a bond exists, STABILIZED is true, otherwise false. At the beginning of the program, STABILIZED is false. DIRECTION can also be viewed as a register that represents the direction of MT growth. A DIRECTION is selected at random (rand) by physical processes.

Then the program enters a network of operations that can branch and loop, all running in parallel, with each operation occurring at a stochastic time, depending on diffusion of molecules, collisions, and binding rates. Adding a GTP-tubulin to the MT increments register A (operation [+]A). After this operation, two operations follow. Note that this is the only fork in the flow diagram where *both* paths are executed; all others are *conditional* branches. On the left branch, if the MT has reached a stabilizer, it binds to it, setting the STABILIZED register to true. If not, it loops back to adding another tubulin dimer ([+]A).

The right branch from [+]A implements GTP-hydrolysis. When this occurs, at some stochastic time later, this decrements register A

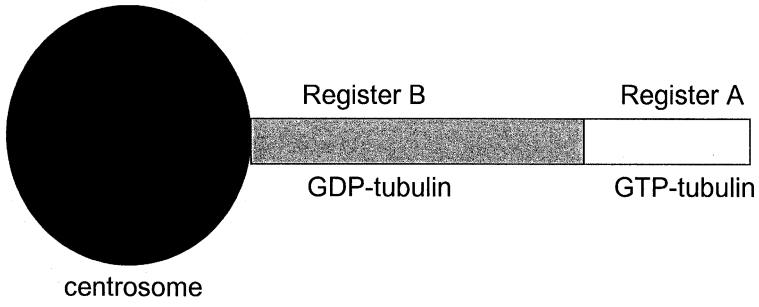


Figure 11. Schematic illustration of registers represented by a MT. When a new GTP-tubulin dimer is added, register A is incremented. When a GTP-tubulin is converted to GDP-tubulin by GTP hydrolysis, register A is decremented and register B is incremented.

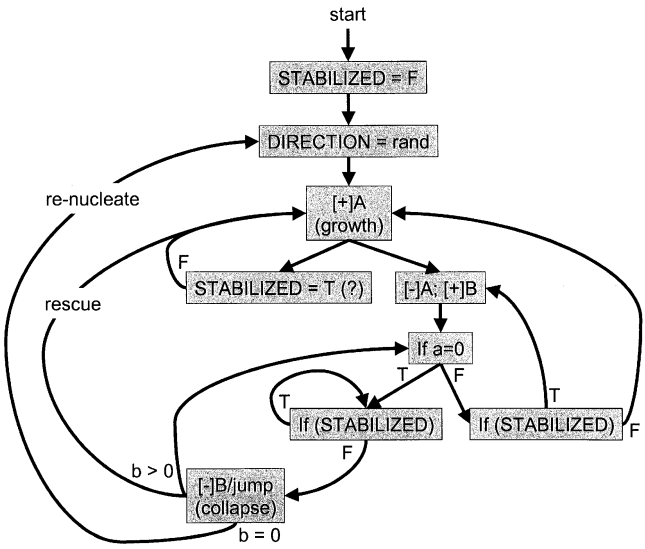


Figure 12. Flow diagram of the MT search-and-stabilize program. Each box represents a RAM computer operation. A and B are registers as shown in Figure 11. STABILIZED and DIRECTION are registers that hold a Boolean value and a direction, respectively; *a* and *b* are the values of registers A (number of GTP-tubulin dimers) and B (number of GDP-tubulin dimers), respectively. Arrows show the sequencing of operations, with labels indicating branch conditions; there are many parallel sequences in this program. Growth of the MT occurs through the [+]A operation; collapse occurs via the [−]B/jump operation; paths that lead to rescue and renucleation are labeled.

and increments register B. Both occur simultaneously, so we have shown it as one operation ([−]A; [+]B). Note that we used [−]A rather than [−]A/jump here. The jump occurs when the value of the register is already zero when the decrement is attempted. In this program, there is no way to reach this instruction if the value of register A is already zero, therefore no jump can occur.

After GTP-hydrolysis of one tubulin dimer, what happens next depends on whether all of the GTP has been hydrolyzed. If there are still GTP-tubulin dimers left in the MT, i.e., the value of register A, *a*, is not zero, and the MT is not stabilized, then it loops back to [+]A, adding another GTP-tubulin. Thus there are two loops in this program (so far) that represent MT growth, both originating and returning to [+]A. The left loop implements the situation where multiple tubulin dimers are added without any GTP-hydrolysis occurring in between. The right loop includes GTP-hydrolysis, but at a slow

enough rate that the GTP-tubulin cap remains ($a \neq 0$) and growth continues.

If, after the GTP hydrolysis, there are no GTP-tubulin dimers left ($a = 0$) and the MT is not stabilized, GDP-tubulin can dissociate from the MT, decrementing register B ([−]B/jump). As mentioned before, branching (decision making) is one of the mechanisms that make this type of computing stochastic and non-deterministic. The GDP-tubulin could begin dissociating before a is *exactly* zero. There may be a few dimers of GTP-tubulin left, but not enough to effectively cap the MT. This is the same sort of "error" that we saw in the numeric computations described above. However, it is irrelevant from the standpoint of the physical behavior of the MT, just as the stochastic errors in the dispersal and condensation programs of the melanophore were irrelevant.

The [−]B/jump operation branches depending on whether the value of register B, b, is zero or non-zero, i.e., whether, or not the MT has completely collapsed. If $b = 0$ (or, with stochasticity, reasonably close to it), the program loops all the way back to picking a new random direction and nucleating a new MT. If the MT has not completely collapsed ($b > 0$) then the program loops back to two different places in the program. Another tubulin can be added ([+]A), or as long as a is still 0 and the MT is still not stabilized, another GDP-tubulin can dissociate ([−]B/jump). The next operation in both of these loops occurs at some stochastic time later, so essentially they are in a race. If GDP-tubulin dissociates first ([−]B/jump), then the MT continues to collapse. If GTP-tubulin is added first ([+]A), the MT can be "rescued" and the growth mode resumed.

So far, we have examined all of the paths in the program involved in the "search" part of "search-and-stabilize." This fully describes the algorithm for MT dynamic instability, in which paths that lead to the growth of the MT ([+]A) race against the paths leading to GTP-hydrolysis ([−]A; [+]B) and GDP-tubulin dissociation ([−]B/jump). It also includes nucleating a MT in a new growth direction after the complete collapse of a previous one.

When the MT encounters a stabilizer and is stabilized (STABILIZED = T), that is the end of the left branch, but not the end of the program. Recall that different paths through the program are running in parallel, so there are many paths down the right branch that have already been "launched." Specifically, GTP-tubulin can still be hydrolyzed ([−]A; [+]B). As long as there are still GTP-tubulin dimers in the MT ($a \neq 0$), the program keeps looping back to hydrolyzing it. Finally, when all the GTP is hydrolyzed ($a = 0$), the program goes

into a "sleep" mode in which it simply keeps "checking" if the MT is still stabilized. If this loop were programmed on a modern computer, it would be an infinite loop, and just checking the flag over and over would be a waste of processing resources. However, in reality, the MT does not have to do anything active to check. It waits passively until something occurs to release the stabilizer from the MT. For example, suppose a signal molecule reaches the stabilizer, stimulating it to unbind from the MT. At that point STABILIZED becomes false, and the program immediately switches over to [–]B/jump, relaunching the race between collapse and growth, and the MT automatically resumes the dynamic instability, "search" part of the program.

6. Conclusion

In this paper, we have presented the notion that dynamic self-assembly processes of biological systems – even processes like assembly, movement, synthesis, disassembly, and degradation – can be viewed as carrying out RAM computing. We described many different ways that proteins can implement the features of RAM computing: registers and their fundamental increment and decrement operations, branching, and sequencing. We discussed how the RAM computing model applies at multiple hierarchy levels. Then we demonstrated how proteins with these properties could carry out a numeric computation $g = (a * b) + (c * d) + (e * f)$. We discussed how the stochastic nature of the protein interactions leads to slow equilibration of the system, resulting in computational errors, and described strategies that Nature employs to drive down entropy and correct errors. Finally, we presented two examples of real (although relatively simple) biological processes, and the RAM programs they implement.

These results are a significant step in two avenues of research we are pursuing, one technological, the other scientific. From a technological point of view, if biological systems are carrying out computing via dynamic self-assembly, then these self-assembly processes are programmable. In principle, we should be able to turn this around, and figure out how to *program* dynamic self-assembly, to build novel structures and materials. We have made some progress in this direction. We have developed stochastic simulations of MTs and motor proteins, which demonstrate how they can be programmed to build nanostructures (Bouchard and Osbourn, submitted). We hope to

report experimental verification of these simulations in the near future. In addition, if we can solve the problem of manufacturing small machines that have all the properties of protein dynamic self-assembly, then we envision being able to program life-like systems that can self-assemble, self-heal, and self-reconfigure, from totally non-biological components.

From a scientific perspective, this RAM computing view of self-assembly processes may enable us to identify larger, more complicated algorithms biological systems are carrying out. What algorithms enable living systems to develop, persist, reproduce, and evolve? What enabled higher levels of hierarchy, such as multi-cellular organisms and colonies of organisms, to emerge? Indeed, is the ability to execute algorithms central to the emergence of life itself?

We have already identified a few common themes. (1) Living systems represent quantities directly, using unary registers. This representation can be implemented physically in a wide variety of ways, and the fundamental operations on these registers are easy to achieve physically. (2) Living systems can implement the same software with different hardware, and (3) can scale algorithms up by applying the same algorithms at different levels of hierarchy (molecules, cells, organisms). (4) Sequencing mechanisms wire together fundamental operations in a modular, evolvable manner. (5) Decisions are often made by races or competitions; the decision follows the winner – the faster, stronger, or more numerous. (6) There are a few common strategies for yielding robust outcomes from ''unreliable'' stochastic processes: focusing on function over form, cycling repeatedly until success is achieved, and imposing a checkpoint or error-correction mechanism to restore order.

As we continue to examine biological systems in the RAM computing framework, we will continue to test and refine these themes, and search for additional themes. Over time, being able to tease out common algorithms from diverse and seemingly unrelated biophysical processes may enable us to make progress toward some aspects of ambitious questions regarding the algorithms of life.

Acknowledgements

We thank Matt Glickman for useful discussion and suggestions. Sandia is a multiprogram laboratory operated by Sandia Corporation, a

Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration. under contract DE-AC04-94AL85000.

References

Alberts B, Bray D, Johnson A, Lewis J, Raff M, Roberts K and Walter P (1998) Essential Cell Biology: An Introduction to the Molecular Biology of the Cell. Garland Publishing, New York

Alberts B, Johnson A, Lewis J, Raff M, Roberts K and Walter P (2002) Molecular Biology of the Cell. Garland Science, New York

Alon U, Surette MG, Barkai N and Leibler S (1999) Robustness in bacterial chemotaxis. Nature 397: 168–471

Agutter PS and Wheatley DN (1997) Information processing and intracellular 'neural' (protein) networks: considerations regarding the diffusion-based hypothesis of Bray. Biology of the Cell 89: 13–18

Arkin AP and Ross J (1994) Computational functions in biochemical reaction networks. Biophysical Journal 67: 560–578

Ben-hur A and Siegelmann ET (2004) Computation in gene networks. Chaos 14(1): 145–151

Bouchard AM and Osbourn GC (2004) Dynamic self-assembly and computation: From biological to information systems. Biologically Inspired Approaches to Advanced Information Technology. Springer-Verlag, Berlin. 95–110

Bray D (1995) Protein molecules as computational elements in living cells. Nature 376: 307–312

Bray D, Levin MD and Morton-Firth CJ (1998) Receptor clustering as a cellular mechanism to control sensitivity. Nature 393: 85–88

Burstein Z (1995) A network model of developmental gene hierarchy. Journal of Theoretical Biology 174: 1–11

Conrad M (1995) Cross-scale interactions in biomolecular information processing. BioSystems 35: 157–160

Conrad M (1999) Molecular and evolutionary computation: the tug of war between context freedom and context sensitivity. BioSystems 52: 99–110

Conrad M and Zauner K-P (1998) Conformation-driven computing: a comparison of designs based on DNA, RNA, and protein. Supramolecular Science 5: 787–790

Dewdney AK (1993) The New Turing Omnibus. Computer Science Press, New York

Endy D and Brent R (2001) Modelling cellular behaviour. Nature 409: 391–395

Gibson MA and Bruck J (2000) Efficient exact stochastic simulation of chemical systems with many species and many channels. Journal of Physical Chemistry A 104: 1876–1889

Gillespie DT (1976) A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. Journal of Computational Physics 22: 403–434

Ideker T, Galitski T and Hood L (2001) A new approach to decoding life: systems biology. Annual Review of Genomics and Human Genetics 2: 343–372

King DG, Soller M and Kashi Y (1997) Evolutionary tuning knobs. Endeavour 21: 36–40

Kirschner M and Gerhart J (1998) Evolvability. Proceedings of the National Academy of Sciences USA 95: 8420–8427

Magnasco MO (1997) Chemical kinetics is Turing Universal. Physical Review Letters 78(6): 1190–1193

McAdams HH and Shapiro L (1995) Circuit simulation of genetic networks. Science 269 650–656

Minsky ML (1967) Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, NJ

Park S-H, Zarrinpar A and Lim WA (2003) Rewiring MAP kinase pathways using alternative scaffold assembly mechanisms. Science 299: 1061–1064

Schnitzer MJ and Block SM (1997) Kinesin hydrolyses one ATP per 8-nm step. Nature 388: 386–390

Shannon CE (1948) A mathematical theory of communication. The Bell System Technical Journal 27: 379–423

Sköld HN, Aspengren S and Wallin M (2002) The cytoskeleton in fish melanophore melanosome positioning. Microscopy Research and Technique 58: 464–469