Dynamic Self-Assembly and Computation: From Biological to Information Systems

Sandia National Laboratories, Physical and Chemical Sciences Center,
P.O. Box 5800, MS 1423,
Albuquerque, NM 87185-1423, U.S.A.
{bouchar, gcosbou}@sandia.gov

Abstract. We present two ways in which dynamic self-assembly can be used to perform computation, via stochastic protein networks and self-assembling software. We describe our protein-emulating agent-based simulation infrastructure, which is used for both types of computations, and the few agent properties sufficient for dynamic self-assembly. Examples of protein-network-based computation and self-assembling software are presented. We describe some novel capabilities that are enabled by the inherently *dynamic* nature of the self-assembling executable code.

1 Introduction

Dynamic self-assembly (consisting of the energy-dissipating processes of building up, tearing down, and dynamically modifying structures or patterns) is a ubiquitous process in non-equilibrium physical and biological systems. [1] Protein networks carry out much of the molecular-scale directed transport, assembly, communication, and decision-making activity within and across cells, and do so via dynamic self-assembly. It has been suggested that protein networks play a computational role in single cells analogous to that of neural networks in multi-cellular organisms. [2] Biomolecular systems provide models for guiding the development of molecular-based computing and self-assembly technologies. For example, chemical systems [3-5] and DNA-based systems for computing [6,7] have been discussed. In this paper, we explore how dynamic self-assembly, such as is carried out by protein networks, can be used to perform computation, and ask: Can any arbitrary Turing machine be implemented? If so, what are the key properties required of the proteins? How reliable would such Turing machinery be, and how would errors be corrected?

A major motivation for exploring computation via protein networks is that biological systems are robust, dynamic, adaptable, and self-healing—all properties that are highly desirable for information technologies. By abstracting the key properties that allow protein networks to implement computation via dynamic self-assembly, we hope to achieve a new "self-assembling software" approach that provides robust, dynamic, adaptable, self-healing software.

A.J. Ijspeert et al. (Eds.): BioADIT 2004, LNCS 3141, pp. 95–110, 2004.

We have identified a few crucial properties of proteins and their interactions that are sufficient to enable the processes of dynamic self-assembly and computation. (1) Proteins have tremendous selectivity of their binding sites, operating much like 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 catalyzing an assembly or disassembly reaction (e.g., in enzymes). (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.

We abstract these important self-assembly and computational properties of proteins into an "agent," the fundamental building block of our self-assembling software. An agent can store data, perform some simple or complex computation, or both. Each agent has one or more binding sites (each labeled with a numeric key) that can bind only to complementary sites (property (1)). Sites are said to be complementary when they have keys with the same absolute value but opposite sign. Once bound, property (2) enables the agent to actuate (perform its computation). Property (3) enables it to then bind to another agent, to trigger it to execute next.

A specific execution sequence or biological signaling pathway can be "wired" together by including a set of agents with binding sites that drive them to execute sequentially. For example, suppose each agent has a "trigger" site that activates it (causes it to execute some code) and a "done" site that is exposed when its task is complete. Suppose agent A's done site is complementary to agent B's trigger site, agent B's done site is complementary to agent C's trigger site, and agent C's done site is complementary with agent D's trigger site. Once A is triggered, then B will execute, followed by C, followed by D.

It is important to note that such an execution sequence or pathway is not hard-coded, but *self-assembled*. That is, the agents are just "dumped" into the simulation environment, and the execution order occurs as a natural consequence of the order in which binding and unbinding events occur. As a result, the self-assembling executable code is inherently *dynamic* in nature. The structure of the executing code is assembling and disassembling all the while it is executing, with execution pathways that are driven dynamically by matching keys between agents. All that is required to change the execution pathway—"re-wire" what the code does, or turn code on or off—is a change of keys.

This feature leads to innovative and powerful capabilities in software developed by this approach. For example, changes to existing self-assembled code (due to changing user requirements, or to reuse an existing self-assembled software package for another application) can also be self-assembled, without having to modify the original source code or shut down the running program. This is achieved by adding new agents with keys such that they rewire the existing code, even while it is running. Another example is "situation detection," a mechanism for "sensing" whenever certain conditions or events occur by providing passive agents with empty binding sites. These binding sites correspond to the conditions of interest, and when all sites are bound, the sensing agent is activated to report or trigger a desired response. Situation detection is asynchronous. It is also passive, in that no repeated active polling by the agent itself is required to detect the events. This capability can also be used to inter-

rogate or monitor the code itself during runtime. For example, if a program is using up a tremendous amount of CPU without producing any output, a query (a special case of situation detection) can be constructed to determine what the program (which was perhaps written by another programmer) is doing right now. These and other unusual capabilities are a consequence of the protein-network-emulating approach of self-assembling software. Although we are very early in the development of this self-assembling software technology, we have already demonstrated that the approach is dynamic and adaptable. Current and future work will focus on the issues of robustness, evolutionary (autonomous) adaptability, and self-healing.

2 Simulation Infrastructure

Stochastic simulations are an effective tool for modeling the dynamics of small protein networks, and have been used, for example, to understand protein network properties responsible for the robust adaptivity of chemotaxis. [8] Because the properties of the proteins and our self-assembling software agents are, by design, so similar, we have built a common agent-based simulation infrastructure for use both in the stochastic simulations of protein networks and for the self-assembling software. In the protein network simulations, an agent represents a single protein or protein complex. In the self-assembling software, an agent stores data and/or performs an atomic computational operation (such as adding two numbers, solving an equation, or writing some output to a file).

An agent is constructed from a sequence of parts. These parts are roughly analogous to protein domains, except that only those domains with binding sites are included. The detailed physics and chemistry of conformational changes is not modeled. Instead, we directly model the properties of the agent that matter for self-assembly and computation—the actuation and exposing/hiding of other binding sites. Each part has a binding site that can be bound to at most one other site at any time. Each site has a numeric key that can either be invalid (hiding the site, preventing it from binding), or that only allows binding with complementary sites. Thus, this binding is a selective process as in biological systems (property (1) of the Introduction). Matching binding sites can be thought of as having a virtual attraction, since binding will readily occur between them when they become available (by becoming exposed or unbound from an existing ligand).

Each binding site can have two types of events, binding and unbinding, and has an "event handler" associated with each event type. These event handlers are executable code, and implement properties (2) and/or (3) of the Introduction. For example, in a self-assembling software system, a binding event at site A could trigger the summation of two numbers (property (2)) and also expose site B for binding (property (3)). A comparable example from the protein network simulations would be a kinase that, when bound to a substrate, phosphorylates the substrate, releases it, and then hides its own substrate-binding site until the kinase is activated again. All of the "action" of the agent, then, is coded in the event handlers. In other words, the stochastic binding or unbinding of these sites triggers the deterministic execution of code, whether that

code represents a physical process (for protein network simulations) or an information process (for self-assembling software).

Initially, a population of agents is included in the simulation environment. The simulation infrastructure locates any exposed sites with complementary keys, and schedules binding events for these sites on an event queue, ordered by the scheduled event time. Any unmatched sites are placed on a free-site list to wait passively until a complementary site becomes available. The simulation proceeds by pulling the first event from the event queue, binding the designated sites to each other (essentially, setting the two sites' pointers pointing to each other), and executes the two sites' binding event handlers. During the execution of the event handlers, a number of things could happen. (a) Some physical actuation or a calculation could be performed. (b) A binding site could be exposed. If a site with a complementary key is found on the free-site list, a binding event is scheduled. If no complement is found, the site is placed on the free-site list. (c) A site could be hidden. If that site is associated with a scheduled event, that event is canceled. If the site was on the free-site list, it is removed. (d) The key of a site could be changed. Corrections are made to a scheduled event, and/or a correction is made to the free-site list to reflect the new key. (e) An unbinding event could be scheduled.

The simulation proceeds by pulling the next event from the queue, binding or unbinding the designated sites, according to the event type, and then executing the event handlers. (The same possibilities (a)-(e) could occur during the execution of an unbinding event handler.) This process continues until there are no more events on the event queue, or the maximum desired time is reached. The implementation details have been described elsewhere. [9]

A specific execution sequence or biological signaling pathway can be "wired" together by including a set of agents with keys that drive them to execute sequentially. The A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D pathway described in the Introduction was wired together by assigning complementary keys to the done site of one agent and the trigger site of the next. We reemphasize here that such an execution sequence or pathway is not hard-coded, but self-assembled. The agents are just "dumped" into the simulation environment, and the execution order occurs as a natural consequence of the binding and unbinding events that are pulled from the event queue.

A natural property of this approach is the self-assembly of concurrent non-deterministic execution pathways in parallel, or multi-threading execution paths. For example, the A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D pathway can be executed in parallel with a completely different pathway Q $\rightarrow$ R $\rightarrow$ S $\rightarrow$ T, as long as the keys from one pathway do not match those of the other. To synchronize multiple threads (for example, if agent U can execute only after both D and T have completed execution), no special synchronizing code is required. The synchronizing agent (U) simply waits passively with its keys on the free-site list until triggered (when D's and T's done sites bind to U).

"Encapsulants" effectively create local environments in which collections of free binding sites can interact. Encapsulants in our approach are meant to resemble biological cell membranes that isolate their internal contents from interactions with external structures. Thus, identical A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D pathways could be executing in parallel in different encapsulants, without any interference, even though they have matching keys. Encapsulants can contain agents as well as other encapsulants (for

hierarchical organization). They also contain "surface" agents that act as signals or receptors for interaction with other encapsulants, or gates to move agents and other encapsulants into and out of the encapsulant. These surface agents, analogous to transmembrane proteins in biological cells, manage all external interactions of the encapsulant, and allow it to act as an "agent" building block for structures and execution pathways at another (higher) hierarchy level.

3 Computing with Protein Networks

3.1 RAM Machine Computing with Proteins

We wish to examine how the self-assembly processes of protein networks can be harnessed to perform computation. Instead of dealing with Turing machines directly, we will discuss RAM machines. [10] A RAM machine is more directly realizable using proteins. Turing and RAM machines are equivalent, i.e., any Turing machine can be assembled from a suitable set of RAM machines, and vice-versa. [10] RAM machine computing requires an ordered sequence of operations that are carried out on a small set of idealized integer registers (each of unlimited capacity). Any computation can be programmed using only two types of operations: those that increment a particular register by 1 ([+]reg); and those that either decrement a particular register by 1 (if the register is nonzero) or else jump to some other part of the program sequence ([-]reg/jump). Thus, to construct a RAM machine from the protein-emulating agents described in Section 2, we need agents that represent registers, agents that perform the increment operation on each register, and agents that perform the decrement/jump operation on each register.

A unary representation [10] for integers allows the size of any clone population of assembled molecules to serve as a register. The register molecules can be free-floating or can be assembled into polymers. We use the phosphorylated state (pA') of a model protein (pA) as an individual count of a register (called A). To be more concrete, if five of the pA proteins are phosphorylated, then the value of register A is five. A kinase that can phosphorylate protein pA can act as the increment agent [+]A, if it can be activated and can signal as described below. Similarly, a phosphatase that can dephosphorylate pA' can decrement register A if it is nonzero. Different registers are made from different types of proteins.

Ordered sequences of [+]reg and [-]reg/jump are dynamically self-assembled by switching on the appropriate agent at the appropriate step in the computation sequence. The system produces an ordered sequence of computational operations by temporal activation, rather than through spatial wiring. To implement this, we consider protein complexes that must be triggered by another selective signaling protein to become active. Similarly, these protein complexes must release another signaling protein to activate the next protein agent. Allosteric proteins with unique binding site selectivity and switchable binding site dynamics are ideal for creating the unique sequences of protein activity needed for computation. Signal cascades can also be implemented, so that parallel execution pathways can be triggered. The timings of the

sequence depend on bonding rates that in turn depend on molecular arrival time statistics. Thus, the computation execution times are stochastic.

Decisions/branchings are carried out by the exit pathways of the [-]reg/jump agents. This means that these agents must be able to release two alternate signaling molecules—one if a dephosphorylation actually occurred, and a different jump signal molecule after a "waiting" time in which no register protein binds to this agent. The "jump" molecule clearly must be released with a rate that is, on average, slow compared to the arrival time of a register protein (when one is present). Also, the arrival of the register protein must prevent the jump molecule from being released. These properties are designed into the event handlers of the agent's binding sites, and are of similar complexity to those of a conventional kinesin protein that "walks" along a microtubule in a eucaryotic cell. [11] The hydrolysis of ATP drives cyclic irreversible behavior.

Fig. 1 illustrates the interactions of the [-]reg/jump agent with the signaling proteins, register proteins, and ATP. Agents are represented by polygon shapes. The binding sites and key values are shown as tabs at the perimeter of the agent. When the sites of two agents are bound, they are shown as touching. The [-]reg/jump agent is labeled, as are the ATP agents. The two collections of agents to the right represent a single register. The phosphorylated version of the register protein is shown in a lighter gray. Initially, in panel (a), the value of the register is five, and the [-]reg/jump agent has a single "trigger" binding site exposed, 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 [-]reg/jump agent, two additional sites are exposed, with key values of 2 and 3. When the trigger site unbinding event is handled, if both the ATP and register proteins are bound to these two sites (as in panel (b)), then in panel (c), the hydrolysis of ATP drives the [-]reg/jump agent to dephosphorylate the register protein (note that in panel (c), there are only four phosphorylated register proteins, and an additional unphosphorylated version), release it and the "spent" ATP, and expose the "done" site with a key of 5. A signaling protein with a key of -5 binds to the done site. When released, it will trigger the next operation in the execution sequence.

If there had been *no* register protein bound when the trigger site unbinding event was handled, then the "jump" site (lower right site of the [-]reg/jump agent in Fig. 1) 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 phosphorylated versions of the register protein (i.e., the register value is zero), then the jump pathway will be taken. However, due to the stochastic nature of the "race" between the binding event of the register site and the unbinding event of the trigger site, the stochastic jump process will produce incorrect jumps (when the register is nonzero) 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 unphosphorylated version of the register protein. To increment the register, it phosphorylates the register protein (i.e., changes its key to -3),

then exposes a done site with a key of 8. There is no "jump" associated with the increment operation.

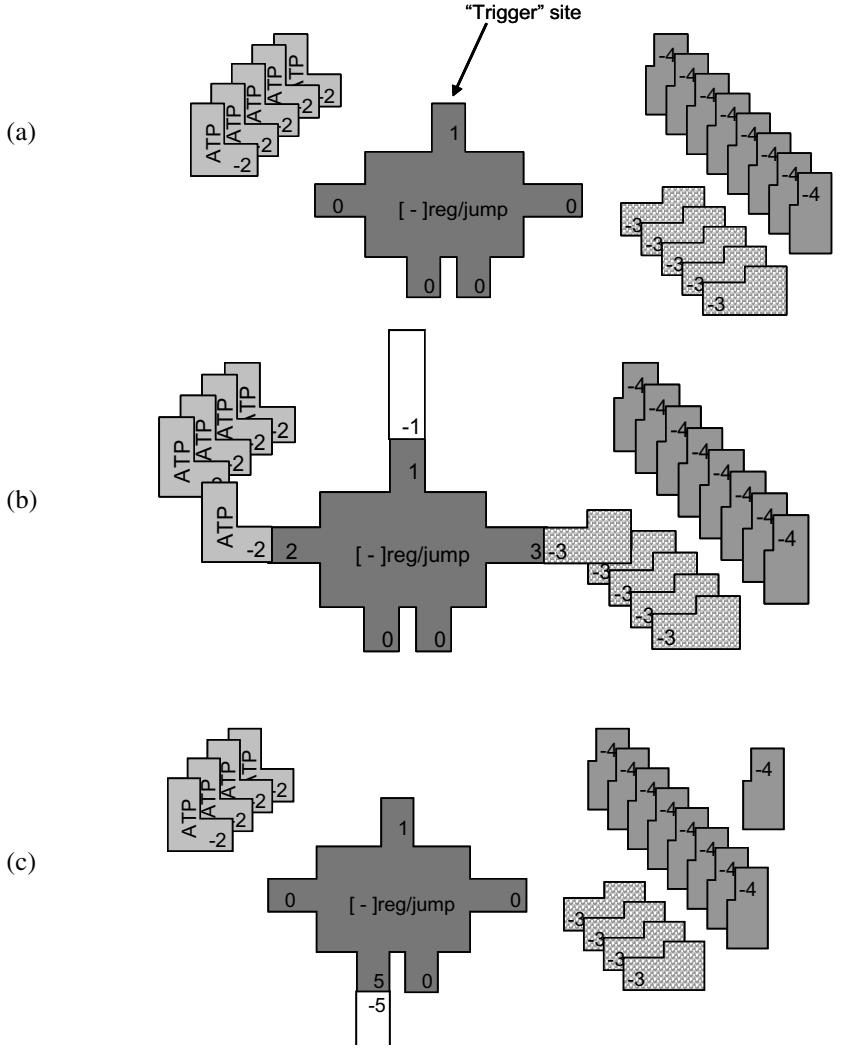

Fig. 1. Illustration of the decrement operation. (a) The [-]reg/jump and ATP agents are labeled. The two collections of agents to the right represent a single register with a value of 5 (phosphorylated proteins are lighter gray). (b) When the [-]reg/jump agent is triggered, it binds to an ATP and a phosphorylated register protein. (c) Then it dephosphorylates the register protein, thereby decrementing the register, releases the ATP and register protein, and signals success

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. S*tochastic* versions of any deterministic Turing machine can in principle be obtained using dynamic self-assembly of proteins that exhibit commonly available properties.

To illustrate how this simple set of agents can accomplish such computations, Fig. 2 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 Fig. 1, but these are omitted from Fig. 2 for clearer viewing of the execution sequence itself. 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.

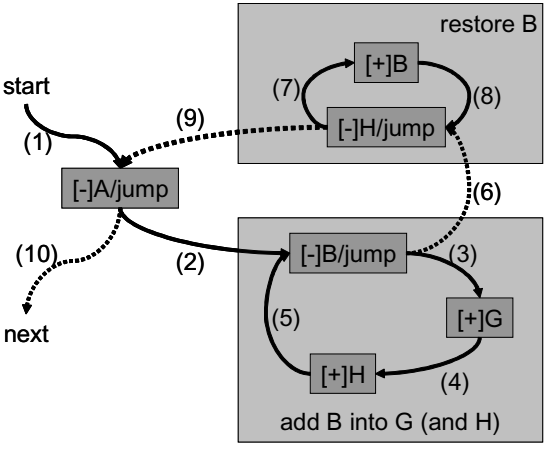

Fig. 2. Schematic diagram of protein network to multiply registers A and B into register G (see text for discussion)

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 a 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. 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 nonzero, due to the stochastic nature of this agent. So, in fact, there are numerous opportunities for errors in even this simple computation.

3.2 Stochastic Computing, Errors, and Entropy

We present results of stochastic simulations of encapsulants computing (A*B)+(C*D)+(E*F), where A, B, C, D, E, and F are initial register values. 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 means that such a set of identically configured encapsulants with S_{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 ten 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 through F. This process occurs with some "yield," i.e., there is a nonzero probability that one or more register copy operations will produce an incorrect register value. When the copying is completed, a synchronizing encapsulant is used to trigger the calculation. 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. Fig. 3 (left panel) 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. Fig. 3 (right panel) shows a scatter plot of final normalized entropy (S_{config} divided by its maximum) as a function of errors in the final answers. 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.

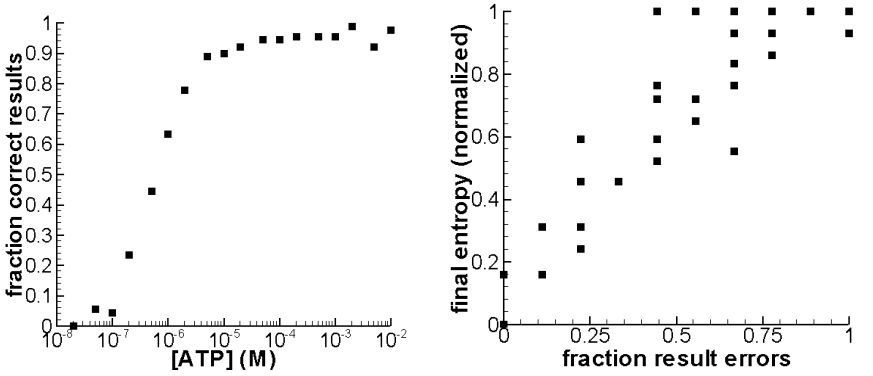

Fig. 3. (*left*) The fraction of encapsulants with the correct final results as a function of ATP concentration. (*right*) Normalized final entropy vs. fraction of encapsulants with errors in their final results

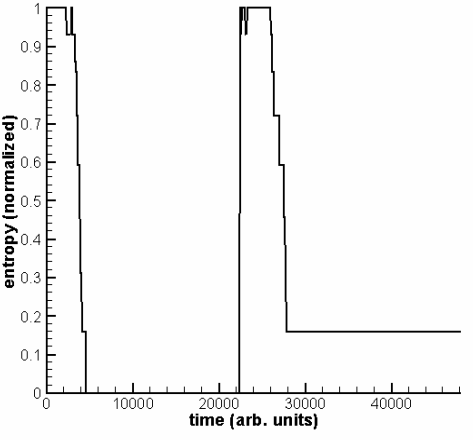

Fig. 4. Normalized entropy as a function of time for a single computation where all of the encapsulants are correctly copied, and all but one of the computations achieved the correct result

The S_{config} as a function of time for a single computational run is shown in Fig. 4. 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 ~ 5000, in a totally ordered configuration of encapsulants ($S_{config} = 0$). The calculation is initiated at t ~ 22000, and while each encapsulant is performing its calculation independent of the others, their configurations again diverge ($S_{config} = 1$). Finally, all of the encapsulants reach a finished state, with

all but one encapsulant reaching the same final state (low, but nonzero S_{config}). Thus, this non-equilibrium process is cyclic in the S_{config}.

The tendency of these stochastic computational processes to increase their Sconfig after a computational cycle is simply the slow equilibration of the configurational degrees of freedom. This clearly prevents arbitrarily long computations from being performed in the simple manner described above. The imperfect yield in the computational processes described above has some similarities to the classic problem of communicating through a noisy channel. [12] Here we have a more general process of noisy computing processes (state transitions) in addition to noisy information transfer. Correct computing in general requires a mechanism for restoring Sconfig to zero periodically, with each restoration occurring before the distribution equilibrates too far. We are currently developing simulations of a hierarchical algorithm (i.e., in which the encapsulants act as agents) to restore low entropy in order to correct computational errors.

4 Self-Assembling Software

4.1 From Protein Networks to Self-Assembling Software

Our goal is to abstract the relevant properties of the proteins and their networks to devise a novel self-assembling software technology. The work on protein networks has provided valuable intuition about the important protein properties, agent design, interactions, etc. that enable self-assembly of physical structures and execution sequences. It has also provided insight as to what properties of real physical systems should be *omitted* for efficient software technology.

The issues of stochastics, equilibration, and error-correction discussed in the previous section are real issues for any molecular computing with protein networks that might be attempted experimentally. However, to develop self-assembling software, we conveniently side-step these issues by choosing not to model the non-equilibrium, dissipative aspects of the protein interactions, and by using deterministic binding and unbinding event times. In addition, although it is instructive to demonstrate that a RAM machine can be constructed and a computation carried out using protein machinery, building software with the fundamental increment and decrement operations would be inefficient and wasteful. Instead, each agent in the self-assembling software is designed to do anything from an arithmetic operation like adding two numbers, to reading or writing to a file, to implementing an entire algorithm.

4.2 Example: Bank Transaction

We have described the essential properties of our fundamental building blocks (agents) and infrastructure (Section 2), and we have described how protein networks can self-assemble a computation using those agents and infrastructure (Section 3). We now present an example of self-assembling software, chosen for its simplicity, to demonstrate how the protein-emulating agents can self-assemble to handle savings account withdrawals.

Initially (Fig. 5), three agents are present, the Balance, the Withdrawal Process, and the Primer. The Balance stores the current balance for the account ($100.00).

The Withdrawal Process has the task of subtracting the withdrawal amount from the current balance and updating the balance. The Primer acts as the "head" of a "polymer" of completed transactions, which can be walked later by a Monthly Account Report agent. Their binding sites are posted on the free-site list.

When a Withdrawal occurs, binding events are scheduled between its free sites and the complementary sites of the Withdrawal Process. When the binding event handlers are executed, the Withdrawal Process exposes a site with a key of –102. This results in a binding event with the 102 site of the Balance (Fig. 6). When the Withdrawal Process is bound to both the Withdrawal and the Balance simultaneously, it subtracts the withdrawal amount (stored in the Withdrawal agent) from the current balance (stored in the Balance agent), and saves the result back to the Balance agent. The Withdrawal Process then changes the keys of the Withdrawal (Fig. 7), so that (1) it will not bind again to the Withdrawal Process (which would result in subtracting the same withdrawal again) and (2) it will bind to the Primer and leave a 103 site available for the next Withdrawal to bind to. Lastly, the Withdrawal Process hides its own –102 site and resets to its original state. Now it is ready for another Withdrawal (Fig. 7). Note that the Withdrawal Process exposes a site to bind to the Balance only temporarily. This leaves the Balance free to bind to other agents (such as a Deposit Process or Interest Compounder) when needed.

This very simple example illustrates all of the properties of the agent described in the Introduction. The agents have selective binding sites. When binding occurs, they actuate and/or expose, hide, or change the keys of other sites. This results in the self-assembly of an execution sequence (the withdrawal of funds from a bank account) and of a data structure (in this case, a linked list of completed withdrawals).

Finally, when the Savings Account software module is completed, it is encapsulated (recall that, as discussed in Section 2, an encapsulant is analogous to a cell's plasma membrane, isolating its contents from the external environment). Other banking modules, such as Auto Loans and Credit Cards, are also encapsulated. Each encapsulant has a Gate agent embedded in its surface, which selectively allows agents to enter, based on matching keys. In the overall banking system, when a Withdrawal occurs, its key matches only the Gate of the Savings Account module, so it enters and undergoes the same process described above. Similarly, credit card payments enter and undergo processing in the Credit Card encapsulant, etc. In our computational experiments, we have implemented all of the behaviors described here. In addition we have implemented agent and encapsulant transport into and out of encapsulants executing concurrently with the above example.

4.3 Novel Capabilities of Self-Assembling Software

External Override. The fact that an execution sequence is self-assembled, rather than hard-coded, leads to innovative and powerful capabilities in software developed by this approach. One example is the "external override." This self-assembling software construct *overrides* the behavior of the existing code, and it is imposed *externally*. I.e., the original source code "inside" the executable is not modified; instead, additional agents are added from the outside to effect the override. Although there are many similarities between our agent-based approach and object-oriented methods, the external override provides functionality that is distinct from object-oriented in-

heritance, as it allows removal of unwanted functionality (corresponding to part of a method) from the "outside" of the existing (compiled) software structure during runtime. This additional flexibility may be useful for enhancing software reuse.

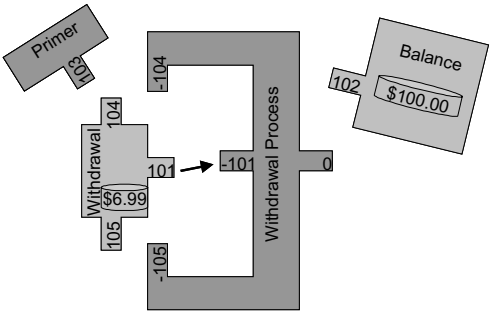

Fig. 5. Initially, the Balance, Withdrawal Process, and Primer agents are available for binding. When a withdrawal occurs, the Withdrawal agent promptly binds to the matching sites of the Withdrawal Process

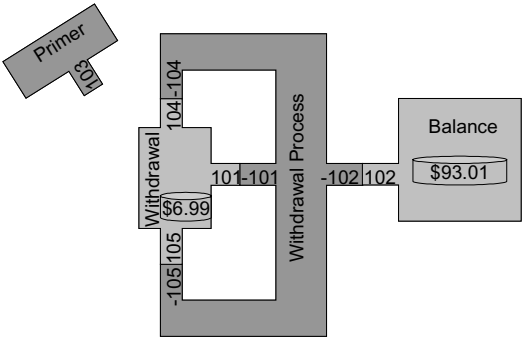

Fig. 6. When the Withdrawal binds to the Withdrawal Process, the Withdrawal Process changes one of its keys from 0 to –102. The Balance then binds with the Withdrawal Process. The Withdrawal Process subtracts the withdrawal amount from the balance, and updates the balance

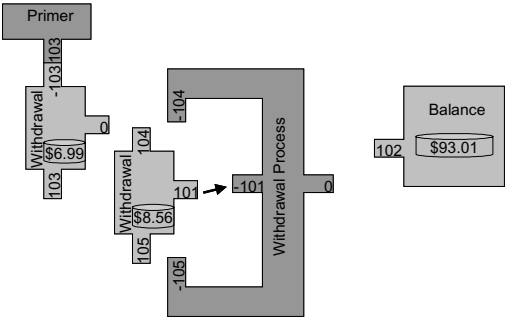

Fig. 7. After completing the withdrawal transaction, the Withdrawal Process changes the keys of the Withdrawal, so that (1) it will not bind again to the Withdrawal Process and (2) it binds with the Primer, exposing a site with a key of 103 for binding later to other completed Withdrawals. Then the Withdrawal Process is ready to handle a new Withdrawal

As an example, suppose that, after executing the withdrawal code described in the previous section, we "realize" that a requirement was omitted: the system shall prevent the withdrawal of an amount exceeding the current balance. To accommodate this requirement, we implement an external override (not shown in the figures). We add a Change Keys machine into the simulation, which binds to the –101 key of the Withdrawal Process and modifies it to –106. In addition, a Verify Balance agent is added, with keys of –101, –104, and –105. Now, when a Withdrawal (which has a 101 key) occurs, it binds with the Verify Balance agent instead of the Withdrawal Process. The Verify Balance agent compares the withdrawal amount to the account balance. If there are sufficient funds for the withdrawal, the Verify Balance agent changes the 101 key of the Withdrawal to 106 and releases it, enabling binding with the Withdrawal Process, and the transaction proceeds as before (Figs. 5-7). If there are insufficient funds, the Verify Balance agent changes the keys of the Withdrawal to some other values, resulting in binding with an Insufficient Funds agent instead.

Note that with our dynamic self-assembly approach, this new function was inserted into the existing program *without* (a) rewriting the original source code, (b) compiling an entire new program, or (c) shutting down the already running software.

Internal "Re-wiring" and Optimization. The external override just described illustrates the inherently *dynamic* nature of the self-assembling executable code. The structure of the executing code is assembling and disassembling all the while it is executing, with execution pathways that are driven dynamically by matching keys between agents. All that is required to change the execution pathway—"re-wire" what the code does, or turn code on or off—is a change of keys.

Not only can the executable be re-wired from the outside with an external override, it can also re-wire itself from the inside, both what it does and when it does it. The code itself could be designed to detect its own properties, such as memory usage, speed, etc., and modify its own code and/or data structures in order to optimize in a particular way, such as using more memory in order to speed up a large calculation. Similarly, runtime priority can be modified for multiple concurrent self-assembly processes. Processor allocation is often implemented at the operating system level. It is easy to allocate different amounts of processing time to concurrent processes here by varying the future (virtual) event times associated with each process. Those with short times will repeatedly activate more frequently.

Situation Detection. Another aspect of the dynamic nature of the executable code is the fact that the execution sequences are self-assembled whenever binding sites "find" each other. An agent can wait passively with its available sites on the free-site list until complementary sites are available. They could become available immediately, or they might not become available until a million events have been handled. We harness this "uncertainty" to implement a self-assembling software construct called a "situation." Situations provide a mechanism for "sensing" whenever certain conditions or events occur by providing passive agents with empty binding sites. These binding sites correspond to the conditions of interest, and when all sites are bound, the sensing agent is activated to report or trigger a desired response. Situation detection is asynchronous. It is also passive, in that no repeated active polling by the agent itself is required to detect the events. It simply waits with its sites on the free-site list.

Situations can be added at runtime to compiled code to monitor the code structure itself. For example, the activity of other agents, their status (number of bound and unbound sites, active or dormant), their functionality, and the numbers and types of agents present in an encapsulant can all be determined automatically. Our agent binding events have some similarities to asynchronous message passing between concurrent servers, e.g., as in the JOULE language. In contrast to message passing, our events provide bi-directional communication in which both agents "know" that they both have been triggered, and both execute code based on this knowledge.

"Monitoring" and "querying" are special cases of the override and situation processes, and are used to inspect the code or the status of its agents. They are like the external override in that they are implemented by inserting agents into the execution pathway during runtime. They are like the situation in that they can sense sought-after conditions of the running code and report on activity or on the data that are being manipulated. The functionality of the agents being monitored is not affected during monitoring. Monitoring and querying only differ in their usage. Monitoring is used to "keep an eye on" some aspect of the code. For example, a goal such as "report every time this credit card is used in two different cities on the same day" would be implemented as a monitor. Once the monitoring agent has detected the situation and reported it, it passively waits for the situation to arise again. In contrast, a query is used to determine something immediate, and then self-destructs and is removed from the simulation. For example, if a program is using up a tremendous amount of CPU without producing any output, a query can be constructed to determine what the program (which was perhaps written by another programmer) is doing right now.

5 Summary

In this paper, we have shown two ways in which dynamic self-assembly can be used to perform computation, via stochastic protein networks and self-assembling software. We described our protein-emulating agent-based simulation infrastructure, which is used for both types of computations. The agents have a few properties sufficient for dynamic self-assembly: they have selective binding sites, and when binding occurs, they actuate and/or expose, hide, or change the keys of other sites. Examples of protein-network-based computation and self-assembling software were presented. We described some novel programming constructs that are enabled by the inherently *dynamic* nature of the self-assembling executable code: the "situation", the "external override" for software reuse, and the ability to monitor or query preexisting code as it executes. These novel capabilities demonstrate that the self-assembling software approach is dynamic and adaptable. Current and future work will focus on the issues of robustness, evolutionary (autonomous) adaptability, and self-healing, as well as code generation from user-specified goals.

We thank Gerry Hays, Wil Gauster, and Julie Phillips for their support of this research effort. 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

1. Whitesides, G., Grzybowski, B.: Self-Assembly at All Scales. Science 295 (2002) 2418-2421
2. Bray, D.: Protein molecules as computation elements in living cells. Nature 376 (1995) 307-312
3. Magnasco, M.: Chemical Kinetics is Turing Universal. Phys. Rev. Lett. 78 (1997) 1190-1193
4. Steinbock, O., Ketturan, P., Showalter, K.: Chemical Wave Logic Gates. J. Phys. Chem. 100 (1996) 18970-18975
5. Berry, G., Boudal, G.: The chemical abstract machine. Theoretical Computer Science 96 (1992) 217-248
6. Winfree, E., Liu, F., Wenzler, L.A., Seeman, N.C.: Design and self-assembly of two-dimensional DNA crystals. Nature 394 (1998) 539-544
7. Yokomori, Takashi: Molecular computing paradigm—toward freedom from Turing's charm. Natural Computing 1 (2002) 333-390
8. Ideker, T., Galitski, T., Hood, L.: A New Approach to Decoding Life: Systems Biology. Annu. Rev. Genomics Hum. Genet. 2 (2001) 343-372
9. Osbourn, G. C., Bouchard, A. M.: Dynamic Self-Assembly of Hierarchical Software Structures/Systems. AAAI Spring Symposium "Computational Synthesis: From Basic Building Blocks to High Level Functionality" (2003) 181-188
10. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, N. J. (1967).
11. Howard, J.: Mechanics of Motor Proteins and the Cytoskeleton. Sinauer Associates, Sunderland, MA (2001)
12. Shannon, C. E.: A Mathematical Theory of Communication. The Bell System Technical Journal 27 (1948) 379-423