SANDIA REPORT

SAND2003-4724
Unlimited Release
Printed January 2004

Self Organizing Software Research: LDRD Final Report

Gordon C. Osbourn

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

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.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2003-4724
Unlimited Release
Printed January 2004

Self Organizing Software Research:

LDRD Final Report

Gordon C. Osbourn
Complex Systems Science
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1423

Abstract

We have made progress in developing a new statistical mechanics approach to designing self organizing systems that is unique to SNL. The primary application target for this ongoing research has been the development of new kinds of nanoscale components and hardware systems. However, this research also enables an out of the box connection to the field of software development. With appropriate modification, the collective behavior physics ideas for enabling simple hardware components to self organize may also provide design methods for a new class of software modules. Our current physics simulations suggest that populations of these special software components would be able to self assemble into a variety of much larger and more complex software systems. If successful, this would provide a radical (disruptive technology) path to developing complex, high reliability software unlike any known today. This high risk, high payoff opportunity does not fit well into existing SNL funding categories, as it is well outside of the mainstreams of both conventional software development practices and the nanoscience research area that spawned it. This LDRD effort was aimed at developing and extending the capabilities of self organizing/assembling software systems, and to demonstrate the unique capabilities and advantages of this radical new approach for software development.

Project Background

Biological self-organizing systems are all capable of building their own highly complex organic "hardware" and "software". The methods by which biological systems self-organize the reliable processing of large amounts of complex information (e.g. directing the construction and real-time operation of a living organism) are quite different from the software engineering techniques developed by humans. Besides the inability of our software systems to self-construct, human software development techniques seem generally less effective and less reliable than those of self-organizing systems. For example, software users have become painfully aware that errors are never fully eliminated from today's large commercial software packages. Surveys from the software industry indicate that: 5 out of 6 large software systems fail to perform as required; 1 out of 3 large development projects are abandoned without completion. Large, abandoned software projects are typically terminated when the true scaling of development cost/time with system size/functionality becomes clear only late in the project. Some analyses of complex software projects (e.g. the IBM mainframe OS/360) have shown that the software functionality can grow as slowly as the square root or even the cube root of the manpower effort invested in software development. These scaling behaviors have the very unfortunate property that progress always appears to be rapid at the beginning of the project, but then it stagnates after a large effort/expense has already been invested. Nature "knows" much about developing large and effective information processing systems that eludes us.

Computer scientists have long sought to improve the process of software development. Many useful techniques for improvement, including structured design, CASE tools, software patterns, code reuse and object oriented design have been championed in the last several decades. All have provided definite benefits, but none of these has been the desired "silver bullet", i.e. none have reduced the development costs by orders of magnitude. A key problem in software development is that there is no room for uncertainty in the software commands. Every minute detail of every task must be explicitly, completely and perfectly described somewhere in the software code. Any and all special cases that arise must be explicitly addressed. All possible situations, user input combinations, ranges of variables and so on should be anticipated in the design to avoid "crashes". These requirements are essentially impossible to meet while developing complex software codes using existing techniques. We call this the "minutia specification" problem. Generating vast amounts of perfect minutia is a task that clashes with the way that human minds deal with complex tasks. Humans tend to make mistakes when specifying large amounts of minutia, and many of these mistakes go undiscovered. Even when errors are discovered, it has been estimated that 1/5 to 1/3 of software corrections that fix a known mistake inadvertently introduce a new (but now undiscovered) mistake.

We believe that the effectiveness of biological information systems is a consequence of the hierarchical self-organization processes that occur in such systems, and one goal of this project was to provide scientific support for this idea. Hierarchical self-organization is the process by which biological systems build themselves through separate self-organization processes at different length scales. Understanding this process is one of the major scientific goals of the nation's nanoscience initiative. We have

been developing a general non-equilibrium statistical physics model for understanding the collective behaviors that underlie a variety of physical self-organization processes at different length scales. As noted above, living systems self-organize not only organic physical "hardware", but complex "software" as well.

Further, we expect that self-organized software will exhibit other novel properties in common with self-organized living systems. To illustrate one example, consider the common situation of added-on software requirements that are requested after code has been developed. Such requirements are typically inconsistent with the original set of requirements. All existing software engineering techniques require a human to internally modify the minutia of the existing code to satisfy the new requirements without breaking any of the remaining functionality. This would be difficult to do correctly even for the original development team, but such modifications are often implemented later by someone unfamiliar with the complex interdependencies of this minutia with the rest of the software system. Such code modifications can break complex software systems in unanticipated ways that may not be realized at the time. In contrast, we expect that self-organized software will self-adapt to inconsistent changes in design specification without internal intervention by a software engineer. In effect, we expect to have the general ability to correctly override and modify existing functionality through *external* interactions with the self-organized software. This would be a powerful ability that no existing software engineering system exhibits today.

Project Results

In this section we summarize some of the key accomplishments of the LDRD project. The full details of the methods we developed and the results we obtained are described in two conference proceedings papers that are included as Appendices in this report.

We have identified a few crucial properties of proteins and their interactions that are sufficient to enable the processes of 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 abstracted 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 binding sites that can bind only to matching sites (property (1)). Once bound, property (2) enables it to actuate (perform its computation). Property (3) enables it to then bind to another agent, to trigger it to execute next, so that a sequence of computations may be carried out in a specific order. We have developed the infrastructure to allow software self-assembly processes to occur, and have implemented a simple example of the use of this approach to self-assemble and modify software modules.

We demonstrated the synthesis of some simple software systems as a test our biophysics-emulating, dynamic self-assembly scheme. Sets of software building blocks actively participate in the construction and subsequent modification of the larger-scale programs of which they are a part. Self-assembly generates hierarchical modules (including both data and executables); creates software execution pathways; and concurrently executes code via the formation and release of activity-triggering bonds. Hierarchical structuring is enabled through encapsulants that isolate populations of building block binding sites. The encapsulated populations act as larger-scale building blocks for the next hierarchy level. Encapsulant populations are dynamic, as their contents can move in and out. Such movement changes the populations of interacting sites and also modifies the software execution.

We showed that our new approach offers novel constructs for constructing large hierarchical software systems and reusing parts of them. For example, we implemented 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. We also implemented an "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. External overrides, inspired by the biological roles of protein phosphorylation, temporarily (or permanently) switch off undesired subsets of behaviors (code execution, data access/modification) of other agents. "Monitoring" is a special case of the override and situation processes, and was implemented to inspect the code or the status of its agents. Monitors 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. Monitoring and querying only differ in their usage. Monitoring is used to "keep an eye on" some aspect of the code.

Conclusions

At the end of this short-term, out-of-the-box project, we expected to have in hand the beginnings of a system that can hierarchically self-organize functional software, and an initial set of tools that will allow us to interact with and steer the self-organization process toward desired specifications. Given the limited time frame and funding level of this ambitious project, we could only expect to develop, test and characterize our approach using example software tasks of modest size. We succeeded at these objectives, and demonstrated some of the advantages we can expect from a more fully developed self-assembling software development system. We have made considerable progress towards our long-term goal of developing a self organizing/assembling software capability that is modeled on biological processes.

Acknowledgements

This work was sponsored by the U.S. Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corp., a Lockheed Martin Co., for the U.S. Department of Energy.

Appendix A

(Conference proceedings paper)

Dynamic Self-Assembly of Hierarchical Software Structures/Systems

Gordon C. Osbourn and Ann M. Bouchard

Sandia National Laboratories
P.O. Box 5800 MS-1423
Albuquerque, NM 87185-1423
gcosbou@sandia.gov, bouchar@sandia.gov

Abstract

We present initial results on achieving synthesis of complex software systems via a biophysics-emulating, dynamic self-assembly scheme. This approach offers novel constructs for constructing large hierarchical software systems and reusing parts of them. Sets of software building blocks actively participate in the construction and subsequent modification of the larger-scale programs of which they are a part. The building blocks interact through a software analog of selective protein-protein bonding. Self-assembly generates hierarchical modules (including both data and executables); creates software execution pathways; and concurrently executes code via the formation and release of activity-triggering bonds. Hierarchical structuring is enabled through encapsulants that isolate populations of building block binding sites. The encapsulated populations act as larger-scale building blocks for the next hierarchy level. Encapsulant populations are dynamic, as their contents can move in and out. Such movement changes the populations of interacting sites and also modifies the software execution. "External overrides", analogous to protein phosphorylation, temporarily switch off undesired subsets of behaviors (code execution, data access/modification) of other structures. This provides a novel abstraction mechanism for code reuse. We present an implemented example of dynamic self-assembly and present several alternative strategies for specifying goals and guiding the self-assembly process.

Self-assembling Software Background

Dynamic self-assembly is a ubiquitous process in non-equilibrium physical and biological systems (Whitesides and Grzybowski, 2002). We are developing an approach to create artificial systems that dynamically self-assemble into hierarchical structures. We are interested more broadly in physical realizations of such processes and how computational capability emerges in biological systems.

As a first step, we are developing dynamically-self-assembling software systems that are modeled after physical systems and physical self-assembly processes. This paper is our first report on this research direction. We have developed the infrastructure to allow software self-assembly processes to occur, and provide an example of the use of this approach to self-assemble and modify software modules.

A central result here is that a variety of software self-assembly processes become available by emulating physical self assembly. As we describe below, physics-emulating self-assembly can generate data structures, multiple kinds of executable code structures, dynamic execution pathways, hierarchies of software modules, movement of modules within the hierarchy and triggers that execute or inhibit certain code structures. These processes can also dismantle any structure that has been assembled.

The concept of bonding is a central part of our approach. We translate the physical notions of bonding, as they occur in biology (i.e. strong covalent bonds and weak protein-protein bonds), into software. Our "strong" software bonding mechanism directly builds long-lived software structures. These lead to software structures with parts that execute sequentially and deterministically. "Weak" bonding is a more active process that not only assembles executable software structures but also triggers their execution. The weakly-bonded structures and the code execution pathways associated with them are transient. Further, weak bonds can be used to interfere with the action of other bonding processes on the same structure. This type of override is analogous to protein phosphorylation. This provides functionality that is distinct from object-oriented inheritance as it allows removal of unwanted functionality from the "outside" of the existing software structure. This additional flexibility may be useful for enhancing software reuse. The detailed implementation of these ideas is described in a later section.

Weak bonding occurs at bonding sites. Each site allows at most one bond with another individual site at any time. These sites have numerical keys that only allow bonding with complementary sites. Thus, this bonding is a selective process as in biological and physical systems. The selectivity of bonding sites provides certain error-prevention capability intrinsically and provides a general mechanism for self-assembly of desired structures and execution pathways. Matching bond sites can be thought of as having a virtual attraction, as weak bonds will readily form between them when they become available (by breaking existing bonds).

A natural property of this physics-emulating approach is the availability of concurrent non-deterministic execution pathways that can self-assemble. Here, populations of individual software structures self-assemble individual execution steps in single execution pathways or complex execution networks over time by making and breaking

weak bonds with each other. It is possible to completely "wire" together modules into an execution software process using only these flexible (but relatively slow) stochastic processes. Deterministic (faster but inflexible) execution can also be assembled, using structures that are strongly bonded, in which the order of the components in memory determine the execution sequence. The ability to readily mix and modify both sequential deterministic execution processes and dynamic stochastic execution processes provides a novel flexibility to the software self-assembly processes. In fact, the executing self-assembling software alternates between these two mechanisms. Stochastic weak bonding and unbonding events trigger a set of deterministic actions within the associated structures, which in turn lead to more stochastic bond formation and release events.

Newly freed bonding sites become available for bonding with other free sites that have complementary key matches. If no matching sites are available, such sites passively "wait" until matching sites do become available for bond formation. The new bonds may activate dormant structures that contain these sites. In this way, execution pathways become alternately active and dormant, so that the physical order of such software components in memory becomes irrelevant to the execution behavior of the system. Software structures with free sites can act as passive (i.e. non-polling) sensors for detecting complex situations that generate matching bonding sites. This is unlike the conventional conditional branching constructs such as IF and CASE, and is a software analog of hardware interrupts. We discuss this construct (called the "situation") further below.

The hierarchical structure of the self-assembling software is enabled through an encapsulant structure. This is analogous to a cell wall. Encapsulants allow bonds to form only for pairs of sites that are within the same encapsulant. By limiting the population size of machines in any encapsulant, we prevent an $O(N^2)$ escalation of possible site-site interactions and help enforce scalability of the approach to large software systems. The encapsulants manage external interactions with other encapsulants through surface sites. These surface sites enable "transport" in and out of the encapsulant. Encapsulants can contain other encapsulants, allowing a hierarchical structure. Movement of machines and encapsulants in and out of other encapsulants changes the populations of sites that can form bonds within these encapsulants, and so directly modifies the internal software execution.

Our system intentionally resembles a stochastic physics or biology simulation (Ideker, Galitski, and Hood, 2001), in that the stochastic bonding and unbonding events are posted to a priority queue and assigned a future (virtual, not processor) "time" for execution that is used simply to provide an ordering to event execution. Despite the non-physical nature of software modules, we subject them to several physics-emulating processes. Modules can be moved through the encapsulant hierarchy, machine parts can be assembled and eliminated dynamically, and machines and encapsulants can stick together and come apart dynamically. Machine proximity is used here as well, albeit in a graph-theory sense. The bonds between machines form graph edges, and we can use these graph edges to directly locate "nearby" machines. We use this in some cases to deterministically search for multiple matching sites between machines that have just formed a new weak bond. This allows groups of matching bonding sites on two machines to bond at essentially the same time, so as to behave like a single effective pair of larger scale bonding sites.

Multiple, concurrent threads of self-assembly and associated computation are automatically available in this approach. We note that the virtual event times can be used to provide execution priority to concurrent processes without the involvement of the operating system. Further, additional code for monitoring and querying the existing code can be introduced during execution.

This approach exhibits features that may prove useful for generating large software systems. First, self-assembly reduces the amount of minutia that must be provided by the software developer. The self-assembly processes take over some of the details that must be designed and coded. This may save development time. It may also reduce coding errors. The interactions between modules are self-assembling, and are enforced to generate hierarchical structuring. Second, this approach enables novel programming constructs, e.g. the "situation", the "external override" for software reuse, concurrent "stochastic" reconfigurable execution pathways, and the ability to modify and add monitoring capability to a preexisting machine as it executes. Third, the bonding selectivity enforces correct interactions between modules and data structures that may allow greater surety of the implementation.

The downside is that this system will pay an execution speed penalty. The impact on code size, compared to compiled code from a conventional language like C++, is unclear at present.

System Infrastructure

Overview

We call the low-level constructs of our approach "machines". High-level "language" commands are used to clone populations of these machines (rather than be parsed and compiled into machine code). The machines are constructed from sequences of machine parts. High-level commands can also be used to combine a sequence of certain generic machine parts into a single (new) machine. The machine parts, in turn, have sites for bonding and optional executable code attached to them.

There are two part types: controls and actuators. Controls have only one bonding site. The controls are further categorized as activating or non-activating. An activating control must have its single site bonded in order for the machine it is in to become active (i.e. execute the

code in the actuators). A non-activating control has a bonding site that does not activate the machine but is useful for other machines that must dock to or manipulate the machine. Controls may also be associated with data in a type of control called a "data store." A data store has all the features of a simple control and also points to a block of memory that is used for general-purpose data storage. The data-associated site keys of data stores can be used to enforce correct matching and usage, and give a form of unit checking (for example, it would enforce that meters are only added to other meters, and never added to, say, seconds). It can also enforce the correct transfer and usage of complex data structures that are self-assembled.

Actuator parts each contain a "small" piece of execution code and execute sequentially (in the order that they exist in their machine). Actuators can have multiple bonding sites. Each actuator part in a machine may also be active or inactive depending on the bonding status of the sites in the part. An inactive actuator will halt execution of a machine, and this execution can resume when the actuator site forms the necessary bond (and all activating control bonds are still in place). Actuators can also be internal to a machine, typically, to manipulate the data stores of its own machine. It that case, it has no sites exposed to other machines. Instead, it checks that its associated data stores are bonded, in order to activate data manipulation.

Both control and actuator parts are described by generic design data and execution code (analogous to a class definition in object oriented programming). One aspect of this design is whether the part makes bonds stochastically (by finding a match on the free-site list) or deterministically (through proximity). Individual versions of these parts are instantiated into particular machines when these machines are created.

Encapsulants effectively create local environments in which collections of free bond sites can interact to form new bonds. Encapsulants in our approach are meant to resemble biological cell walls that isolate their internal contents from bonding interactions with external structures. Encapsulants can contain machines as well as other encapsulants (for hierarchical organization). They also contain "surface" machines that act as gates to move machines and other encapsulants in and out of the gate's encapsulant. These surface machines manage all external interactions of the encapsulant, and allow it to act as a "machine" building block for structures and execution pathways at another (higher) hierarchy level. The encapsulant gates are analogous to membrane proteins in biological cells. Our encapsulants play some of the roles that "modules" or objects play in modern computer languages (McConnel, 1993; Watt, 1990). That is, they provide modularity and information hiding. In contrast to object modules, the contents of encapsulants are dynamic, with machines (containing data and executables) and other encapsulants being moved in and out during self-assembly and software execution.

The overall action of the system is to execute make-bond and break-bond events, and these then trigger the activation or deactivation of associated machines that can carry out deterministic behaviors. This system is thus event-driven, with the events consisting of stochastic bond formation and bond breaking. An event queue is maintained to efficiently post future events and to execute the events in chronological order. Free bonds are generally posted to a data structure, with sites arranged according to their site keys so that matching site pairs can be efficiently found. Bond formation triggers the execution of the machine(s) that contain the sites. Machines do not become active unless all of their activating controls have bonds. Machine actuators then can execute their code in the sequence that they occur in the machine if their sites are in the necessary bonding configuration. Execution stops at an actuator site that is not "ready" to execute. Each machine maintains its own "instruction pointer" to enable restart of the machine execution at the proper part when bonding conditions change externally to allow restart. We do not allow deterministic machines to execute arbitrary numbers of loops as this would prevent the stochastic actions from taking place. Instead, the number of deterministic repeats is constrained, and then the machine must relinquish control by posting a future activation event for itself on the priority queue.

The code executed by the actuator parts is typically the lowest level functionality that a language would provide. The complexity of the overall software comes from: the assembly of parts into machines; the stochastic assembly of machine execution sequences within encapsulants; and the hierarchical assembly and interaction of encapsulant execution structures.

Novel Software Constructs

Situations are a generalization of the IF branching construct. Situations provide a mechanism for "sensing" whenever certain conditions or events occur by providing passive machines with empty bonds. These bonds correspond to the conditions of interest, and when all bonds are satisfied, the sensing machine 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 machine itself is required to detect the events. Situations can monitor the code structure itself. For example, the activity of other machines, their status (number of bonded and unbonded sites, active or dormant), their functionality, and the numbers and types of machines present in an encapsulant can all be determined automatically.

External overrides are a useful and novel construct that is enabled in our approach. The term "external" indicates that the code designer does not alter or remove the original source software that is being overridden. There are a variety of ways that the self-assembling software system can carry out external overrides, and they can be carried out at the encapsulant level or at the machine level. In all cases, *additional* generic override machines are introduced into the system (even to *remove* existing functionality). At the encapsulant level, existing machines can be skipped,

made to wait for new conditions (not present in the original design), or to take part in alternative stochastic execution pathways not present originally. At the machine level, modified clones of the original machines can be self-assembled. In this work we describe only the encapsulant-level override process. These external overrides can be introduced into existing self-assembling software in "real-time" while the existing software is being executed.

Monitoring and querying of self-assembling and executing software during runtime are special cases of the override and situation processes. These processes can be developed long after the software of interest has self-assembled. Monitoring can be accomplished by inserting sensors into the stochastic execution pathway during execution and having them report on activity or on the data that are being manipulated. The functionality of the monitored machines is not affected during monitoring. However, the total execution time will clearly be altered by this monitoring process.

Runtime priority can be modified for various 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.

Implementation Details

We chose FORTH to implement our self-assembling software system. FORTH finds use both for developing embedded software applications (Napier, 1999) and Windows applications (Conklin and Rather, 2000). FORTH essentially lacks conventional language syntax. Our self-assembled software can execute without concern for syntax errors or keyword use restriction. FORTH permits the entry of executable code directly and allows code definitions to be deferred and redefined later. This allows the software to directly modify itself while running without the offline compilation step that would be required by a compiled language.

We implement the two types of software bonds as follows. Weak bonds (corresponding to protein-protein binding) are implemented by setting pointers of the bonding sites of two machines pointing to each other. Strong bonds are formed by placing items in contiguous memory locations and result in arrays of executable parts. This type of bonding is used to implement the machine structures with ordered parts that execute sequentially and deterministically. Machines are "born" when they are instantiated. Multiple copies of a machine are readily cloned if needed.

Figure 1 illustrates the layout of machines, controls, and actuators in computer memory. The machine is the left column: a set of consecutive memory cells, with eight controls (gray) and four actuators (white). Each cell of the machine has the address of its control or actuator, which can be anywhere in memory. The essential parts of the controls are shown: the key for its bonding site, and the

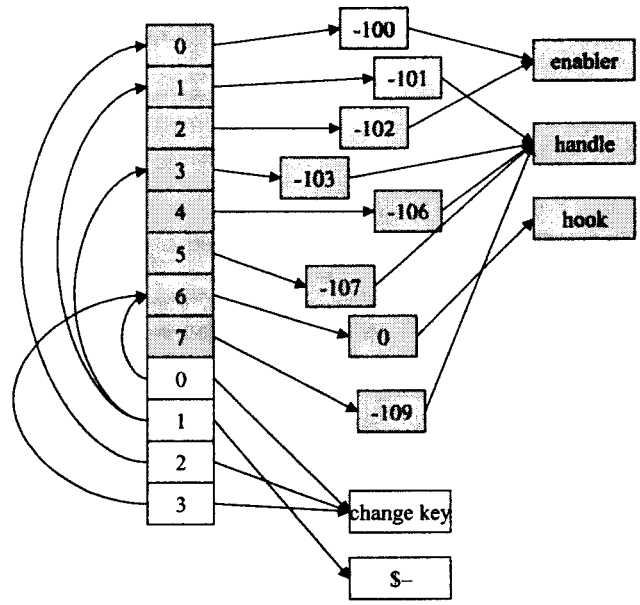

Figure 1. A schematic illustration of the data structure associating machines, controls, and actuators. Refer to the text for details.

type of control. The control type contains code that executes when the control's site makes or breaks a bond. The actuators shown are internal actuators, so rather than having keys, they have a pointer to their associated data stores. They also have a pointer to the actuator type. The actuator type contains code that handles not only make- and break-bond events, but also the actuator's activation. Any exterior actuators would look schematically like the controls of Figure 1.

We chose the calendar queue as the data structure for implementing our event priority queue (Brown, 1988) and also for the free-site data structures in each encapsulant.

An overview of the software execution is as follows: The next event (a make- or break-bond event) is pulled from the priority queue. If it is a make-bond event, a weak bond is made between the two specified sites (that is, their "site-bonded-to" pointers are set pointing to each other). Each site's make-bond event handler is executed. These event handlers typically update the active state of the part, and any deterministically bonding parts on the two machines make additional bonds if their keys match. Then the machine logic for each machine is executed. This checks if all activating controls are active, and if so, executes the actuators in sequence until either an actuator is not ready, or the end of the actuators is reached. When a machine's actuation is complete, it is "reset." It breaks all of its bonds. The sites of stochastically bonding parts are matched against the free-site list. If a matching site is found, a future make-bond event is posted to the priority queue. If no match is found, the free site is put on the free-site list to wait for a free matching site.

Actuators are the parts that perform software functions most programmers expect, such as reading or writing data,

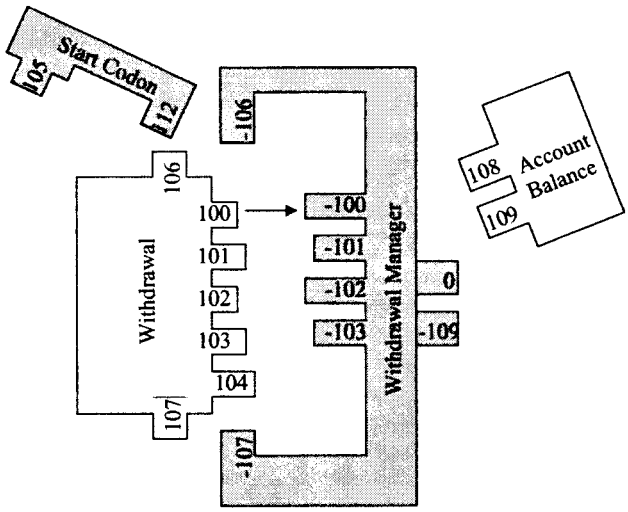

Figure 2. Initially, the Account Balance, Withdrawal Manager, and Start Codon machines are available for making bonds. A Withdrawal occurs, and many bonds with the Withdrawal Manager promptly form.

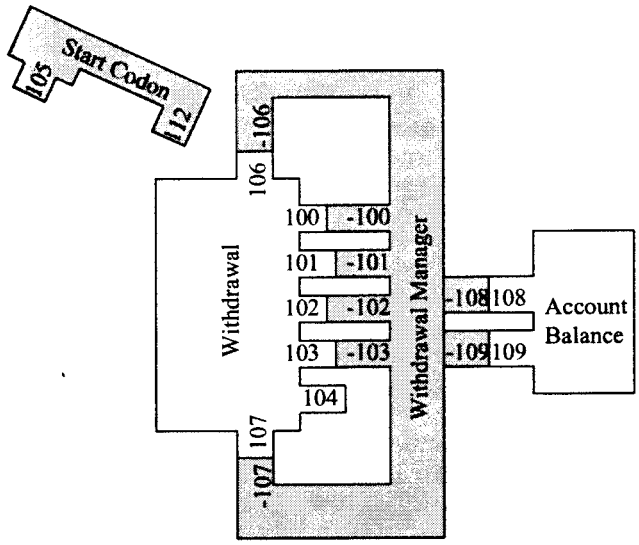

Figure 3. When the Withdrawal bonds to the Withdrawal Manager, the Withdrawal Manager changes one of its keys from 0 to −108. A bond then forms between the Account Balance and the Withdrawal Manager. The Withdrawal Manager subtracts the withdrawal amount from the balance, and updates the balance.

or performing calculations or otherwise manipulating data. An actuator may also change the keys of its own machine's sites, or those of the machine it is bonded to. As described in the previous paragraph, when a machine is reset, its stochastically bonding sites are matched against the free sites. If the machine's actuators changed some of its site's keys in one way, it will bond to a different machine, resulting in the execution of a different software function than if the actuators had changed the site's keys in some other way. In this way, actuators can influence the execution pathway of the self-assembled software.

If the event pulled from the priority queue was a break event, the bond between the two specified sites is broken (i.e., their "site-bonded-to" pointers are set to 0). Each site's break-bond event handler is executed. These event handlers typically update the active state of the part (to set it inactive). Since breaking a bond cannot make a machine go from inactive to active, there is no need to execute the machine logic for the two machines.

When the make- or break-bond processing is completed, the next event is pulled from the priority queue, and the process is repeated until there are no more events on the priority queue. Alternatively, a "pause" event can be placed on the priority queue to temporarily pause execution. Such an event may be used, for example, to update a Windows display or output to a file at regular intervals.

We implemented a simple but general mechanism for overrides via machines that modify the keys of other machines. Altering a key to an unusual or "invalid" value prevents the associated site from forming any bonds. This allows bonding to be turned on and off externally. Altering keys also allows stochastic execution pathways to be altered. Machines can be added or removed from an execution path through the generation of "glue" machines that manage the key alterations. The appropriate sites for modification can be found by the machines themselves, so that human designer intervention can be at a high level.

Sequential stochastic execution pathways can be implemented among machines in multiple ways. One method is to introduce a signal machine that bonds to a corresponding control site on the machines of interest. A sequencing machine can alter the key of this signal machine so that it triggers a series of machines to act in the desired order. Multiple pathways can be spawned by generating multiple signal machines at the same time.

A more direct method is to have an "output" site on one machine match an enabling control site on a second machine that is to execute after the first machine. The first machine site can hide its output site (the site key made an invalid value) until it is finished executing, then it can restore the necessary output site key.

Steering the Self-Assembly Process

The ultimate goal is to cause self-assembling software to create data structures and behaviors that conform to the software designer's requirements. There are a variety of potential mechanisms for accomplishing this. The simplest is to start with initial conditions – i.e. initial sets of machines – that are already known to self-assemble in ways that lead to desired types of results. One can design and verify that particular populations of machines will carry out frequently needed behaviors, and then create machine clone populations in an encapsulant with a single high level command word. Further, machines can be designed that implement common types of overriding modifications in the self-assembly process, and these override machine populations can similarly be introduced into existing encapsulants by high level words. By combining these high

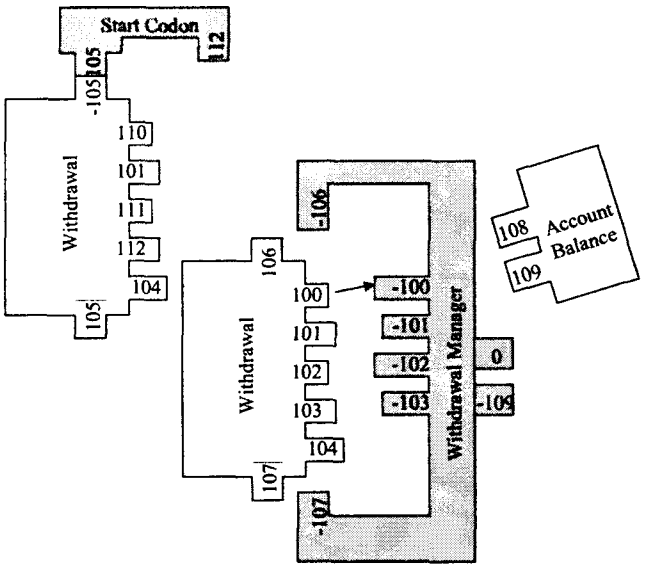

Figure 4. After completing the withdrawal transaction, the Withdrawal Manager changes the keys of the Withdrawal, so that it bonds with the Start Codon. Then the Withdrawal Manager is ready to handle a new Withdrawal.

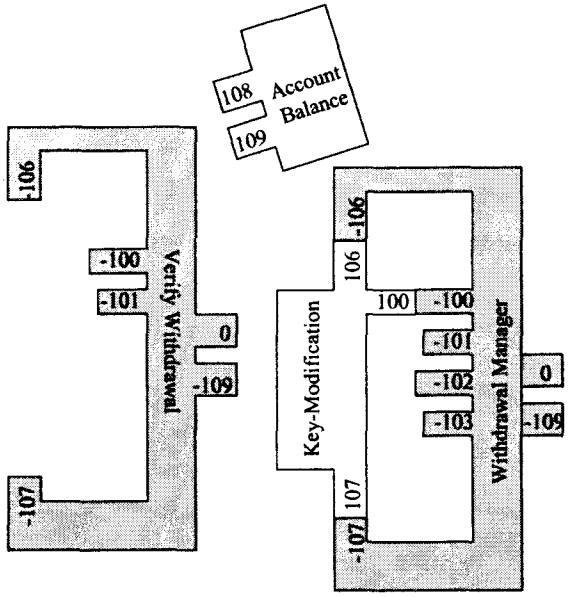

Figure 5. The Key-Modification machine bonds to the Withdrawal Manager to modify its keys. Then the Verify Withdrawal machine inserts itself.

level constructs, more complex behaviors can be assembled. Further, hierarchical structuring can be enforced by limiting the population size in any encapsulant, and automatically triggering the creation of additional encapsulants as machine population sizes exceed selected limits.

Another approach is to provide time-dependent steering by adding or taking away machines or by suppressing or overriding existing machines (again using high level words) at various times as self-assembly progresses. This breaks up the development into well defined stages.

Another category of steering involves evolutionary modification of machine properties and machine designs in populations of machines. This will be a subject of future work.

Example: Bank Transaction

We present an example of the handling of savings account withdrawals, chosen for its simplicity to demonstrate our concepts and infrastructure. We represent machines graphically by polygon shapes. For example, the Withdrawal Manager in Figure 2 represents the machine data structure shown in Figure 1. The bonding sites and key values are tabs at the perimeter of the machine. The internal parts are omitted for clarity. When the sites of two machines touch, this represents a weak bond.

Initially (Figure 2), three machines are present, the Account Balance, the Withdrawal Manager, and the Start Codon. The Account Balance holds the current balance for the account in a data store, the Withdrawal Manager subtracts the withdrawal amount from the current balance and updates the current balance. The Start Codon acts as the "head" of a "polymer" of completed transactions, which can be walked later by a Monthly Account Report machine. All of their stochastically bonding sites are posted on the free-site list.

When a Withdrawal occurs, its free sites post make-bond events with the Withdrawal Manager. When these make-bond events are handled, the Withdrawal Manager's actuators activate, changing its site with a key of 0 to −108. The −108 site now bonds with the Account Balance (Figure 3). Additional actuators in the Withdrawal Manager then activate, subtracting the withdrawal amount (in a data store of the Withdrawal machine) from the current balance (in a data store of the Account Balance machine), and saving the result back to the Account Balance machine. The Withdrawal Manager then changes several keys of the Withdrawal (Figure 4), so that (1) it will not bond again to the Withdrawal Manager (which would result in subtracting the same withdrawal again) and (2) it will bond to the Start Codon and leave a 105 site available for the next Withdrawal to bond to. Lastly, the Withdrawal Manager sets its −108 key back to 0 and resets. Now it is ready for another Withdrawal (Figure 4). Note that the Withdrawal Manager changes its site keys to bond to the Account Balance only temporarily. This leaves the Account Balance free to bond to other machines (such as a Deposit Manager or Interest Compounder) when needed.

After executing this code, 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. In essence, a machine inserts itself into the execution pathway before the Withdrawal Manager to check whether there are sufficient funds to complete the transaction.

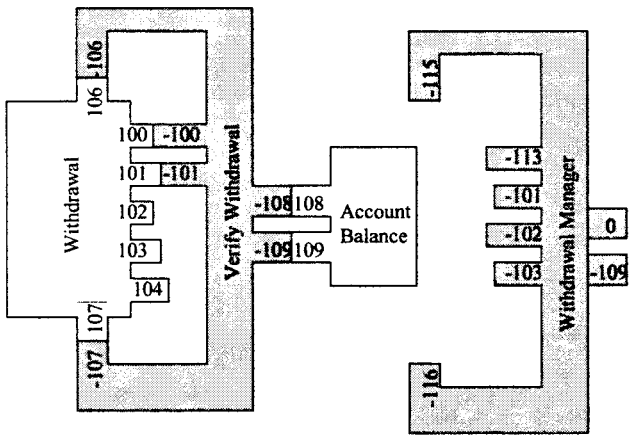

Figure 6. Now, when a Withdrawal occurs, the Verify Withdrawal machine bonds to it first.

A Key-Modification machine (Figure 5) modifies some of the keys of the Withdrawal Manager so that Withdrawals will no longer automatically bond with it. Only the 100, 106, and 107 sites of the Withdrawal make bonds stochastically. Therefore only –100, –106, and –107 of the Withdrawal Manager need to be changed. Once the Key-Modification machine has done its job, it sets all of its own keys to 0.

When the Verify Withdrawal machine inserts itself, its keys are posted to the free-site list. Now, when a Withdrawal occurs, it bonds with the Verify Withdrawal machine instead of the Withdrawal Manager (Figure 6).

The Verify Withdrawal actuators compare the withdrawal amount to the account balance. If there are sufficient funds for the withdrawal, the Verify Withdrawal machine changes the keys of the Withdrawal (Figure 7) enabling bonding with the Withdrawal Manager, and the transaction proceeds as illustrated in Figures 2-4. If there are insufficient funds, the Verify Withdrawal machine changes the keys of the Withdrawal to some other values, resulting in bonding with an Insufficient Funds machine instead (not shown).

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.

Finally, when the Savings Account software module is completed, it is encapsulated. Other banking functions are also encapsulated (Figure 8). Each encapsulant has a Gate machine embedded in its surface, which selectively allows machines 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.

In our computational experiments, we have implemented all of the behaviors described here. In addition we have implemented machine and encapsulant transport into and out of encapsulants executing concurrently with the above example.

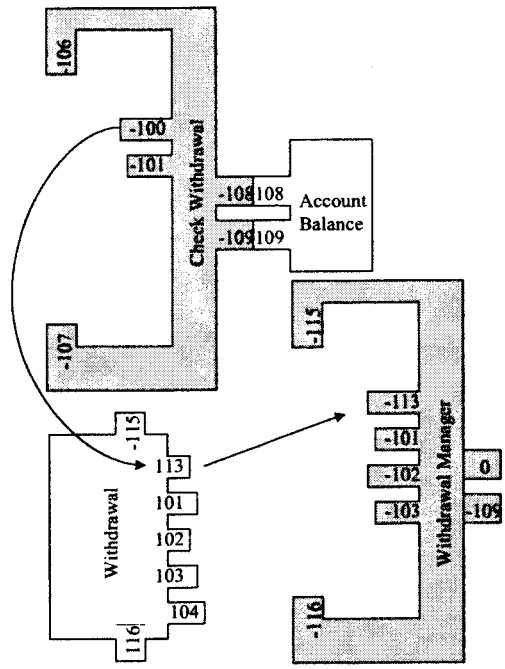

Figure 7. If there are sufficient funds, the Verify Withdrawal machine changes the keys of the Withdrawal so that it bonds with the Withdrawal Manager, and the transaction proceeds as before. If there are insufficient funds, the Verify Withdrawal machine changes the keys of the Withdrawal so that it bonds with an Insufficient Funds machine (not shown).

Future Directions

We are in the process of developing a language for steering self-assembling software for general-purpose applications. The language words will generate populations of machines and encapsulants that carry out the intent behind the high level words. Our infrastructure is designed enable the autonomous generation of encapsulants, machines, and keys that implement the desired execution paths, so that the software designer will not be required to specify detail at that level. For example, the external override described above would be programmed as "VERIFY balance > withdrawal BEFORE ALLOWING withdrawal." The novel constructs of passive situation monitoring, external overrides for reuse, and correctness enforcement through selective bonding site keys will enable programming with a reduced burden of minutia specification.

An interesting extension of our approach is to add evolutionary processes into the self-assembly process. The ability to selectively override actions of an existing machine or module externally provides novel opportunities for "mutating" existing software structure in ways that are more likely to remain functional than random changes or recombinations of existing code. In particular, the stochastic execution pathways provide a mechanism for introducing new kinds of machines into an execution pathway that is very robust, and may require only the

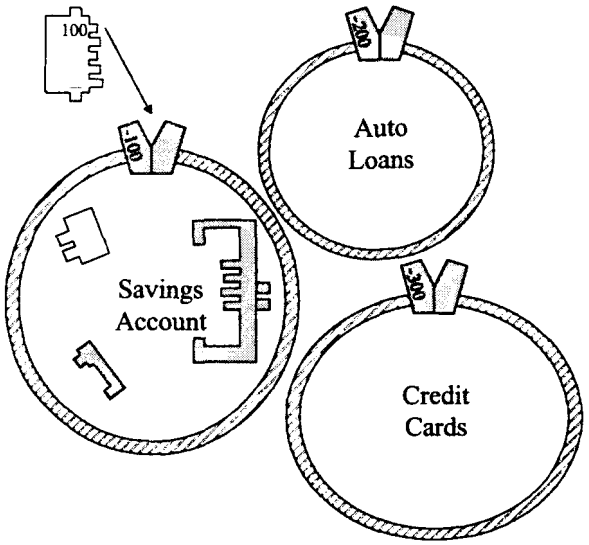

Figure 8. A schematic illustration of different encapsulants that might be in an overall banking application. Each encapsulant has a Gate machine on its surface with a specific key. The Withdrawal (upper left) will only enter the Savings Withdrawals module, because they have matching keys. Similarly, auto loan payments would only pass into the Auto Loan module and Credit Card charges would only enter the Credit Cards module.

automated reassignment of a few bonding site keys in the original machinery. Further, the ability of the machines to self-monitor their performance means that ineffective modifications can be "backed out of" without necessarily destroying the machine. Populations of competing machines can readily be maintained, with winners achieving more access to processor time as described above. Apoptosis (programmed death) of machines within encapsulant populations provides for a finer tuning of evolved performance. Machines can monitor the activity of machines or machine parts within an encapsulant, and identify unused structures for elimination. It will be of interest to see if this eliminates the accumulation of "introns" in the software developed via such evolutionary processes (Banzhaf, Nordin, Keller, and Fancone, 1998). Evolving, self-assembling software promises to be a rich research topic that we will explore in considerable depth.

Another future direction for this research is to develop self-optimizing behaviors for the self-assembling software performance. One approach is to convert stochastic execution pathways directly into deterministic machinery. This will speed up the code execution at the cost of interfering with future external overrides at the module level. Such changes are reversible, so that any relative ratio of stochastic and deterministic pathways is achievable at any time.

In addition, we envision alternative data structures or algorithms, all appropriate for the same task (but each more effective for a different size of data set or a different distribution of data values) that can replace each other based on their monitoring of the overall execution and the available data set. On a larger scale, populations of similar machines/encapsulants with a distribution of operating parameters could be deployed for evolving an optimal population mix.

Again, we note that such optimizations can occur in "real-time" as the original code is executing. This could be convenient for high-consequence applications that cannot be frequently taken off-line.

Acknowledgement

We would like to thank Gerry Hays and Julia Phillips for their support of this research effort. This work was carried out at Sandia National Laboratories. 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

Whitesides, G., and Grzybowski, B. 2002. Self-Assembly at All Scales. *Science* 295: 2418-2421.

Ideker, T., Galitski, T., and Hood, L. 2001. A New Approach to Decoding Life: Systems Biology. *Annu. Rev. Genomics Hum. Genet.* 2: 3413-3472.

McConnell, S. 1993. *Code Complete.* Redmond CA.: Microsoft Press.

Watt, D. 1990. *Programming Language Concepts and Paradigms.* New York, NY.: Prentice Hall.

Napier, T.,1999. Forth Still Suits Embedded Applications. *Electronic Design* 47(24) :97-106.

Conklin, E. and Rather, E. 2000. *Forth Programmer's Handbook.* Manhattan Beach, CA.: Forth, Inc.

Brown, R. 1988. Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Commun. ACM* 31: 1220-1227.

Banzhaf, W., Nordin, P., Keller, R., and Francone, F. 1998. *Genetic Programming.* San Francisco, CA.: Morgan Kauffman Publishers, Inc.

Appendix B

(Conference proceedings paper)

Computation Via Dynamic Self-Assembly of Idealized Protein Networks

Ann M. Bouchard* and Gordon C. Osbourn
Sandia National Laboratories
P.O. Box 5800 MS 1423
Albuquerque, NM 87185-1423

We describe stochastic agent-based simulations of protein-emulating agents to perform computation via dynamic self-assembly. The binding and actuation properties of the types of agents required to construct a RAM machine (equivalent to a Turing machine) are described. We present an example computation and describe the molecular biology and non-equilibrium statistical mechanics, and information science properties of this system.

1 Introduction

Dynamic self-assembly is a ubiquitous process in non-equilibrium physical and biological systems.[1] It is our view that in such systems, dynamic self-assembly and computation are intimately related: that dynamic self-assembly can be used to perform computation, and that if the computational language can be understood, it can be used to program self-assembly. In this paper, we focus on the first of these relationships, that dynamic self-assembly can be used to perform computation. We examine how the self-assembly processes of protein networks can be harnessed to perform computation at the molecular scale. We present results of stochastic simulations of non-equilibrium idealized protein networks that we have designed to provide programmable computing. We present an example computation and describe the molecular biology, non-equilibrium statistical mechanics, and information science properties of this system.

Biomolecular systems provide models for guiding the development of molecular-based computing and self-assembly technologies. For example, chemical systems[2,3,4] and DNA-based systems for computing[5,6] have been discussed. It has been suggested that protein networks play a computational role in single cells analogous to that of neural networks in multi-cellular organisms.[7] Protein networks are of particular interest, as they carry out much of the molecular-scale directed transport, assembly, communication, and decision-making activity within and across cells. Achieving programmable computation via engineered protein networks would shed light on the extent to which biological systems perform true computing. Computational protein networks may never displace semiconductor technology for purely numerical processing. However, they suggest a novel approach to programming self-assembly processes, for both organic and inorganic structures, at the molecular scale. Consideration of engineered protein networks as platforms for molecular computing raises several issues. Can any arbitrary Turing machine be implemented? If so, what are the key properties required of the proteins, and to what extent are such properties actually exercised in living systems? How reliable would such Turing machinery be, and how would errors be corrected? To explore these issues, we present stochastic simulations of non-equilibrium idealized protein networks that we have designed to provide programmable computing. Stochastic simulations are an

* To whom correspondence should be addressed. Email: bouchar@sandia.gov

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] As an example, we present a simple simulation of computing (A*B)+(C*D)+(E*F) using an ATP-driven dynamic stochastic protein network. These information processing protein networks require simultaneous treatment of their non-equilibrium statistical mechanics, information science, and molecular biology properties to fully describe their behaviors.

We present two main conclusions. First, we find that *stochastic* versions of any deterministic Turing machine can in principle be obtained using dynamic self-assembly of proteins that exhibit commonly available properties. Second, we find that partial equilibration of the far-from-equilibrium stochastic protein networks intrinsically leads to increasing computational errors with length of computation, so that an ensemble of such computing networks diverges in configuration with time to different internal states and different computational results. This is a direct consequence of the stochastic nature of the protein networks and the second law of thermodynamics. The implication is that if natural systems do indeed perform computations with low error rates, they must employ error-correction mechanisms as part of the algorithm. Such error-correction mechanisms that might be found in nature are currently under investigation.

2 Simulation Infrastructure

We have identified a few crucial properties of proteins and their interactions that are required to enable the processes of 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" with these three properties. 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 numerical key that can either be invalid (hiding the site, preventing it from binding), or that only allows binding with a complementary site. Thus, this binding is a selective process as in biological systems (property (1)). 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). For example, when a kinase binds to a substrate, it phosphorylates the substrate and releases it (property (2)), and then hides its

own substrate-binding site until the kinase is activated again (property (3)). All of the "action" of the agent, then, is coded in the event handlers.

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, removing the site from the free-site list and 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 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 any scheduled events, those events are 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 any scheduled events, and/or corrections are 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.

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. For example, suppose each agent has a "trigger" site that activates it (its binding event handler does something interesting) and a "done" site that is exposed when its task is complete. Suppose the key of 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. 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 described above 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.

"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 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. The surface agents are analogous to transmembrane proteins in biological cells.

3 Simulations

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.[9] 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.[9] 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[9] 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.

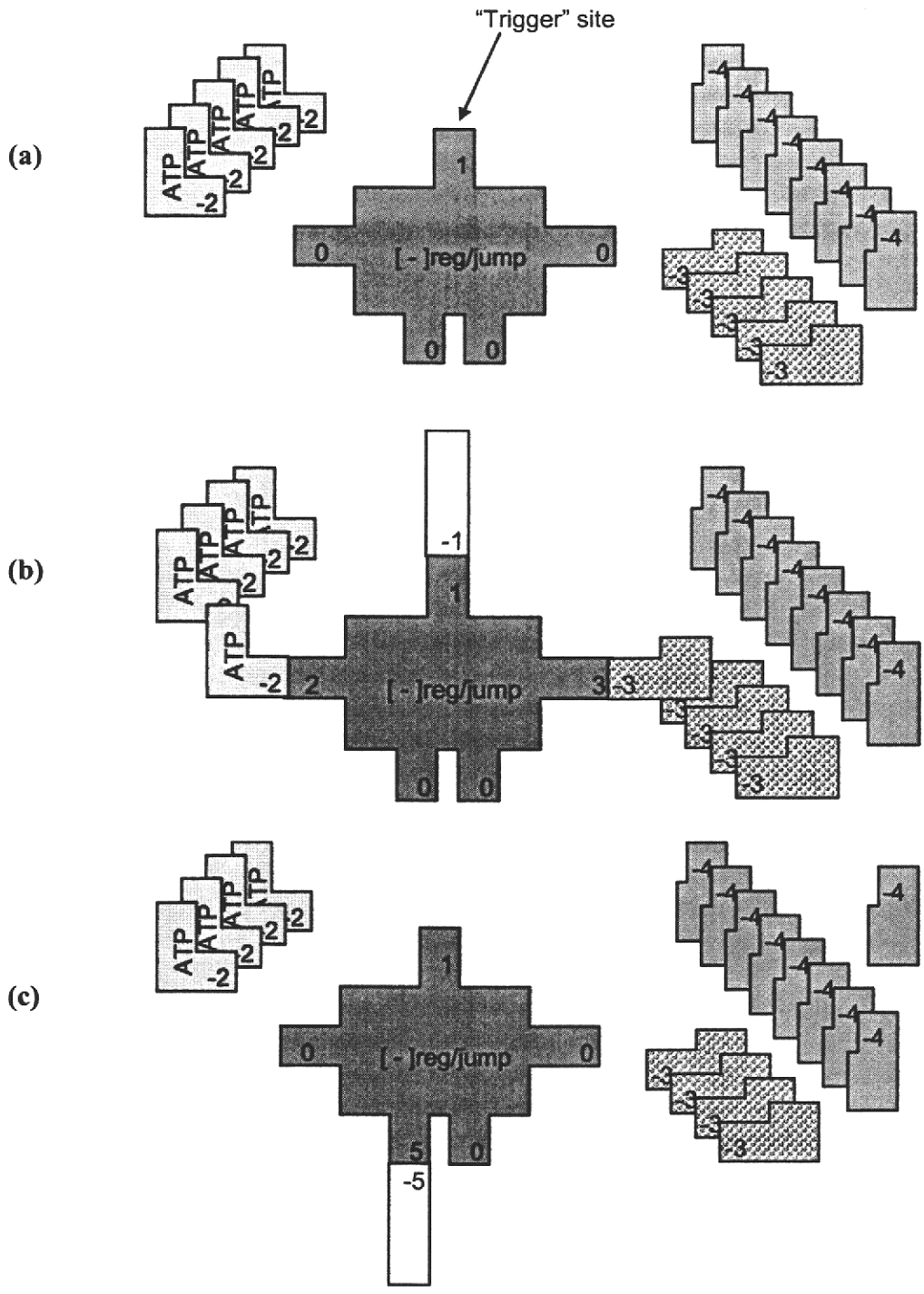

Figure 1. Illustration of the decrement operation. (a) The [-]reg/jump agent is labeled, as are the ATP agents. The two collections of agents to the right represent a single register with a value of 5 (phosphorylated proteins are textured). (b) When the [-]reg/jump agent is triggered, it binds to an 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.

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.[10] The hydrolysis of ATP drives cyclic irreversible behavior.

Figure 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 as textured. 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), releases it and the "spent" ATP, and exposes the "done" site with a key of 5. A different 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 **Figure 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 7. There is no "jump" associated with the increment operation.

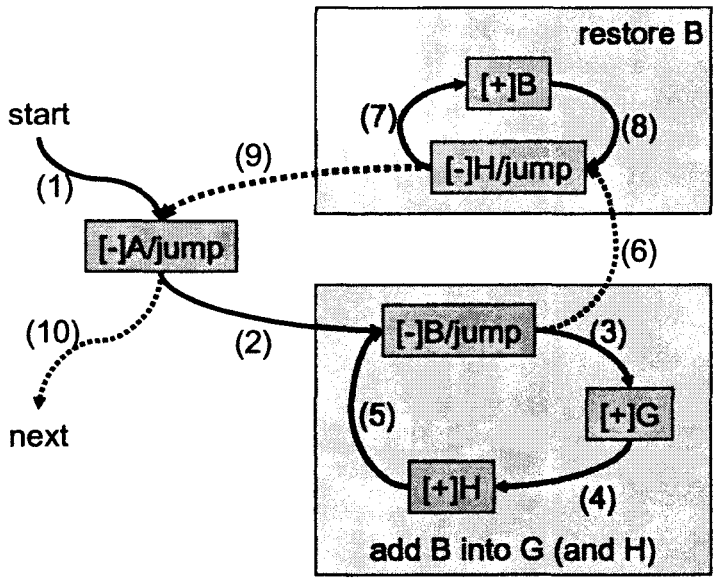

Figure 2. Schematic diagram of protein network to multiply registers A and B into register G. Each increment and decrement agent also interacts with ATP and register proteins, but these have been omitted from the figure for clearer viewing of the execution sequence. Solid arrows represent a signaling protein going from the *done* site of one agent to the trigger site of the next. Dashed arrows represent a signaling protein's pathway from the *jump* site of one agent to the trigger site of the next. The sequence begins with decrementing register A, followed by a loop in which B is added into G and H. Then B is restored from the H register, and then the entire outer loop repeats until [-]A/jump is triggered when the value of A is zero, and it jumps to the next operation.

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 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 **Figure 1**, but these are omitted from **Figure 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.

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 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. **Figure 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 dies not have sufficient energy (ATP), it cannot perform the computation correctly. **Figure 3** (right panel) shows a scatter plot of final S_{config} as a function of final yield of correct answers. These results show that ending in a more highly ordered state is clearly correlated with high yields of correct computational results, 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.

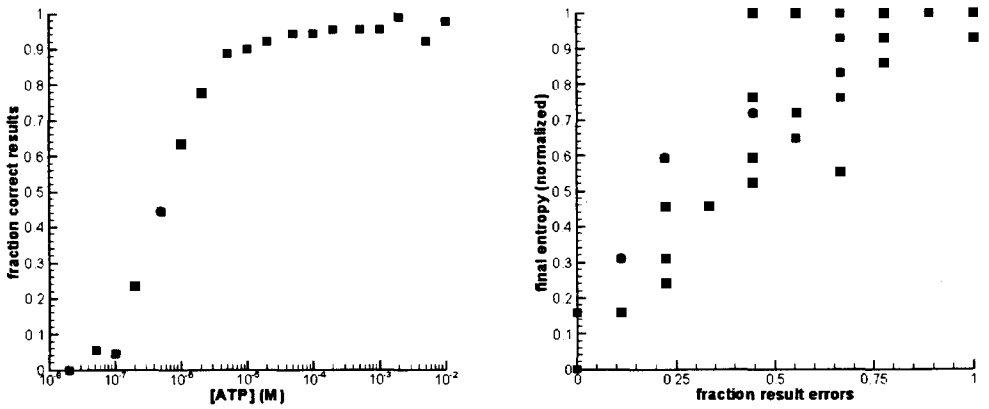

Figure 3. (left) The fraction of encapsulants with the correct final results as a function of ATP concentration. (right) Normalized final entropy vs. fraction of encapsulants with errors in their final results. See text for discussion.

The S_{config} as a function of time for a single computational run is shown in **Figure 4**. We have chosen a case where all of the encapsulants are correctly copied, and all but one of the computations 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}.

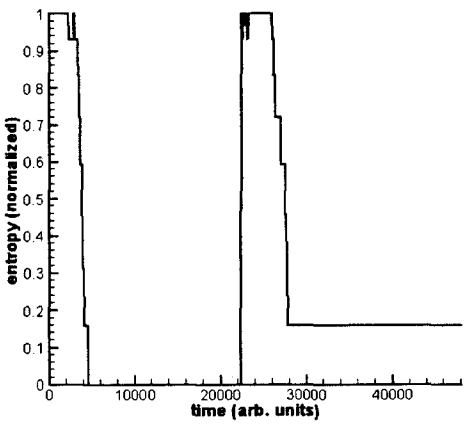

Figure 4. Normalized entropy as a function of time for a single computation where all of the encapsulants are correctly copied, and all but one of the computations achieved the correct result.

The tendency of these stochastic computational processes to increase their S_{config} 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.[11] 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 S_{config} 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 Conclusion

In this report, we have described stochastic agent-based simulations of protein-emulating agents to perform computation via dynamic self-assembly. We described the binding and actuation properties of the types of agents required to construct a RAM machine (equivalent to a Turing machine), and provided the example computation of multiplying and adding several registers. We find that partial equilibration of the far-from-equilibrium stochastic protein networks intrinsically leads to increasing computational errors with length of computation, so that an ensemble of such computing networks diverges in configuration with time to different internal states and different computational results. This is a direct consequence of the stochastic nature of the protein networks and the second law of thermodynamics. The implication is that if natural systems do indeed perform computations with low error rates, they must employ error-correction mechanisms as part of the algorithm. This is the subject of further investigation.

Acknowledgements

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.

---

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

Distribution:

1	MS-0188	LDRD office, 4001
10	MS-1423	G.C. Osbourn, 01001
2	MS-0899	Technical Library, 9616
1	MS-9018	Central Technical Files, 8945-1
1	MS-0161	Patent and Licensing Office, 11500