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

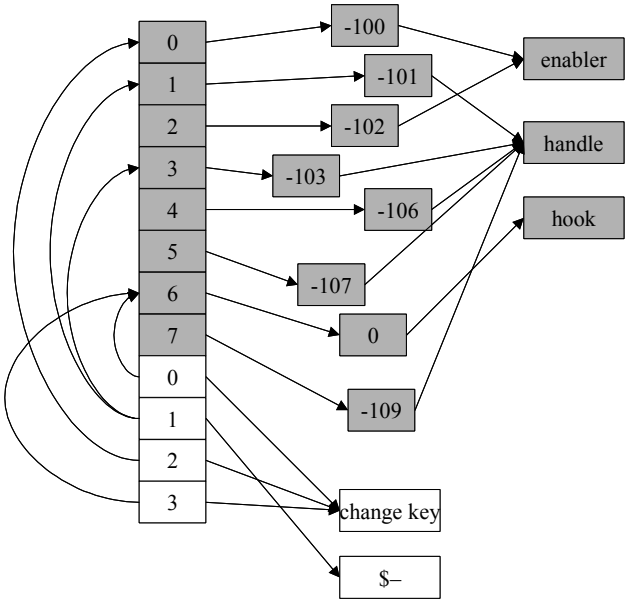

Figure 1. A schematic illustration of the data structure associating machines, controls, and actuators. Refer to the text for details.

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,

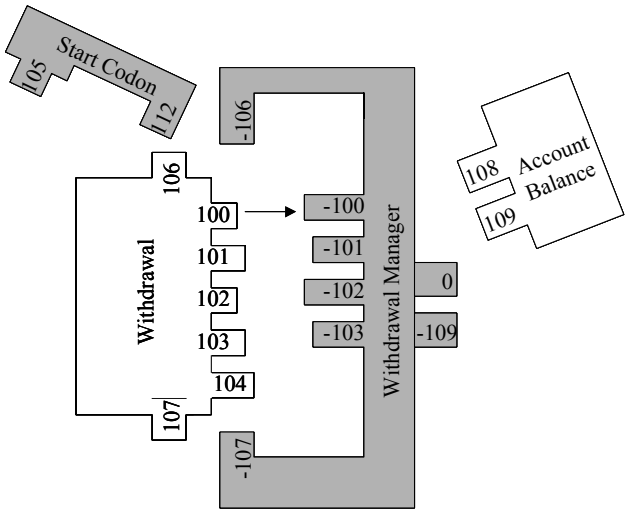

Figure 2. Initially, the Account Balance, Withdrawal Manager, and Start Codon machines are available for making bonds. A Withdrawal occurs, and many bonds with the Withdrawal Manager promptly form.

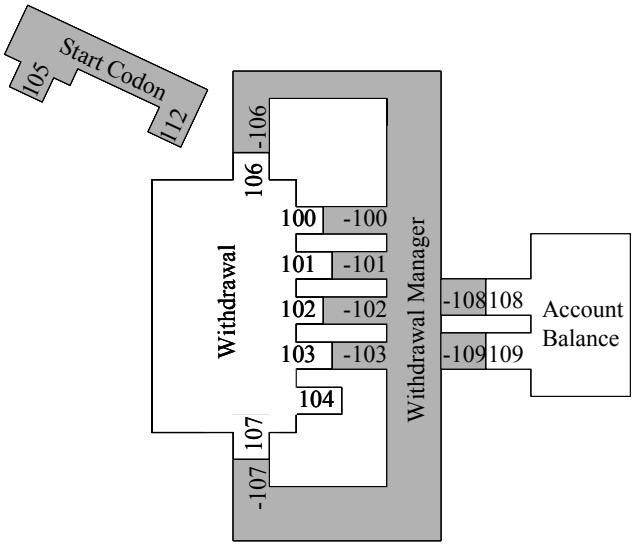

Figure 3. When the Withdrawal bonds to the Withdrawal Manager, the Withdrawal Manager changes one of its keys from 0 to –108. A bond then forms between the Account Balance and the Withdrawal Manager. The Withdrawal Manager subtracts the withdrawal amount from the balance, and updates the balance.

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

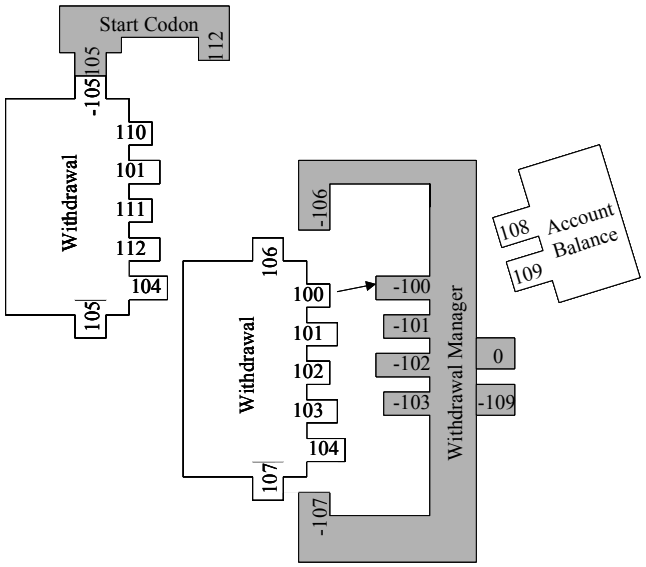

Figure 4. After completing the withdrawal transaction, the Withdrawal Manager changes the keys of the Withdrawal, so that it bonds with the Start Codon. Then the Withdrawal Manager is ready to handle a new Withdrawal.

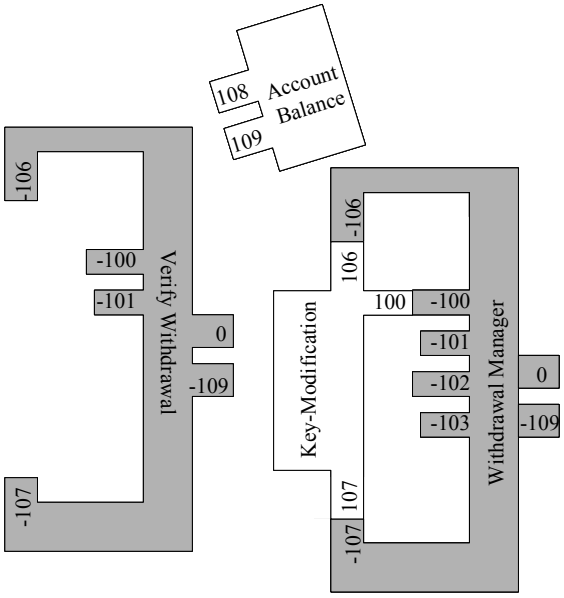

Figure 5. The Key-Modification machine bonds to the Withdrawal Manager to modify its keys. Then the Verify Withdrawal machine inserts itself.

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.

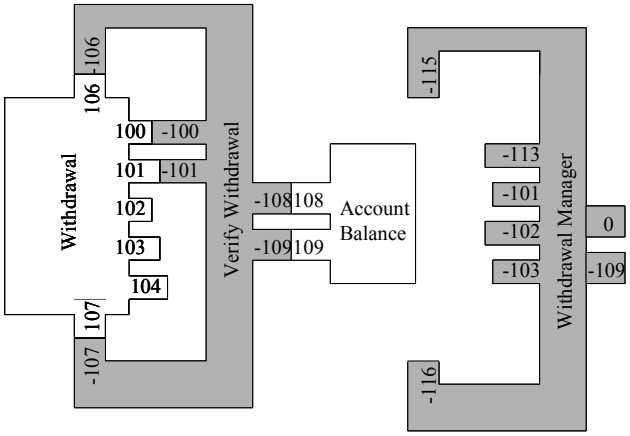

Figure 6. Now, when a Withdrawal occurs, the Verify Withdrawal machine bonds to it first.

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.

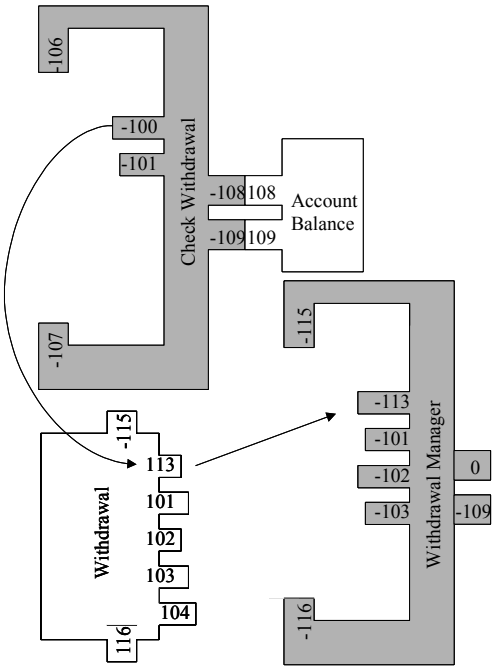

Figure 7. If there are sufficient funds, the Verify Withdrawal machine changes the keys of the Withdrawal so that it bonds with the Withdrawal Manager, and the transaction proceeds as before. If there are insufficient funds, the Verify Withdrawal machine changes the keys of the Withdrawal so that it 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

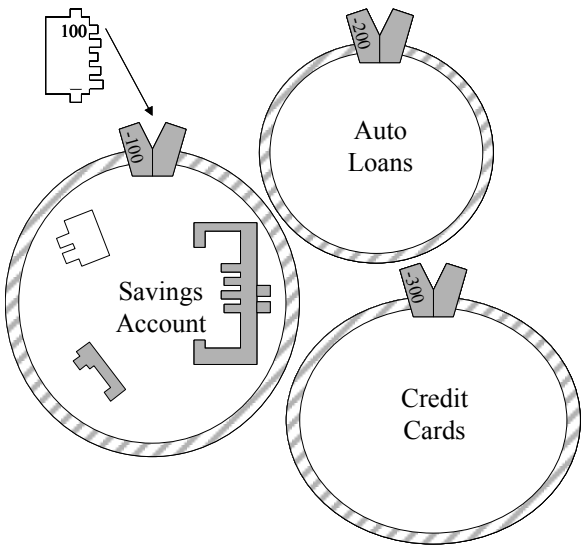

Figure 8. A schematic illustration of different encapsulants that might be in an overall banking application. Each encapsulant has a Gate machine on its surface with a specific key. The Withdrawal (upper left) will only enter the Savings Withdrawals module, because they have matching keys. Similarly, 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.