An Implementation of the Copycat Architecture Using the Starcat Adaptive Computational Framework

Joseph Lewis **Laura Demange**

Department of Computer Science
San Diego State University
5500 Campanile Dr. MC7720
San Diego, CA 92182
(v): 619-594-2014 (f): 619-594-6746
{ lewis@cs.sdsu.edu , ferrisandxena@gmail.com }

Keywords: Copycat, Starcat, adaptation, representation, concepts

Abstract—We describe the Copycat program of Mitchell and Hofstadter, an architecture demonstrating emergent representation and fluid concepts in the domain of letter-string analogies. We present the Starcat adaptive computational framework whose core design is inspired by this same program. To demonstrate that this general framework, different in many fundamental respects, is capable of the same behaviors, we describe an instantiation of Starcat that realizes some of the functionality of the original Copycat architecture. We show results of experiments indicating that Starcat supports, in this instantiation of Copycat, similar behaviors. Certain questions are raised about the impacts of design choices in Starcat on the behavior of its Copycat instantiation.

1.0 Introduction

A fundamental question in cognitive science concerns the nature of concepts. How are they represented? How do they interact? Can we build computational systems that exhibit the same fluidity as the concepts used in even run-of-the-mill human interactions? If so, how can such a system be used to solve real problems?

Artificial intelligence (AI) researchers have traditionally missed the chance to address these questions fully. This is due in part to the role representation is perceived to have in producing solutions to problems. AI is, after all, usually closer to an engineering endeavor, focused on finding solutions to problems, than it is to the more purely scientific issue of understanding how real cognitive systems found in the world engage in open-ended production of effective behavior. The disposition of AI work has mostly been that representation is something that is established a priori for a certain problem domain and then handed to a formal system for manipulation, ultimately leading to a "solution state". There are certainly exceptions, but this has been the general trend. Brooks [1] has been especially vocal in noting the limitations of this approach.

While we agree with the essential contention Brooks maintains—that representation in its usual sense is the wrong unit of abstraction, we disagree with the prescription to eschew representation altogether. We believe that the real issue lies with finding ways to imbue computational systems with the ability to generate their own representations, and in so doing to empower them to use those representations in a fluid and dynamic manner. It is this capacity that we believe underlies the profound flexibility of natural cognitive systems to apply their knowledge in unfamiliar contexts. One computational architecture stands out as having the potential to demonstrate such flexibility: the Copycat program of Mitchell and Hofstadter [2,3]. There are a few other architectures exhibiting similar functionality, Holland's classifier system [4] and Kokinov's DUAL [5] among them, but we choose here to focus on Copycat. We have discussed our position on these issues, for example in response to Brooks' criticisms, in [6,7,8,9].

In the past few years, we have been engaged in the development of Starcat, a generic framework for creating adaptive computational systems, which inherits its essential design from Copycat. One core endeavor in that project has been to instantiate a new version of Copycat, using the Starcat framework. Doing so, we hope to demonstrate that, in the generalization, nothing has been lost and to learn what, if anything, has been gained. The remainder of this paper presents the essential elements of the original Copycat program, the relevant differences between Copycat and the Starcat framework as instantiated in the Copycat problem domain, and experimental results from that new version of Copycat. We refer to the Starcat-instantiated version of the Copycat program as StarCopy. The Starcat architecture has been described in detail previously in [10,11] and its significance in the context of representation in cognitive systems has been discussed in [6,9].

The problem domain of Copycat (and thereby of StarCopy) is the completion of analogies between letter-strings. Consider the analogy "if 'abc' goes to 'abd', then what does 'ijk' go to?" We abbreviate this as "abc→abd, ijk→?" In [2,3] and elsewhere both Mitchell and Hofstadter have discussed the utility of this "microdomain". Far from being like the toy problems that misled many AI projects of the past, it is a carefully designed and deliberately simplified domain, which brings into sharp relief precisely those cognitive tensions that call into action the fluid quality of human concepts and concept usage. The point of the program is less to find a "correct" completion of the presented analogy and more to exhibit the fluid engagement of concepts in its emergent representational process. There are five target problems with which the Copycat architecture was originally tested. These create various conceptual "pressures" that explore the program's functioning. These problems will be discussed in conjunction with the discussion of our new experiments with StarCopy.

1.1 The Copycat Architecture

The Copycat architecture includes three components: the coderack, the workspace, and the slipnet. Messages from the workspace to the slipnet are handled ad hoc by tight coupling between the two components. Small pieces of executable code, known as codelets, are exchanged between the slipnet and the coderack, and also between the coderack and the workspace. Codelets are emitted by the slipnet to be enqueued in the coderack. The coderack is essentially a priority queue whose dequeue behavior is probabilistic. That is, the coderack is most likely to dequeue its most urgent codelet, but it has reasonable probability of dequeueing a somewhat lower urgency codelet, and even some non-zero probability of dequeueing its very lowest urgency codelet. Once a codelet is selected by the coderack it is sent to the workspace where it is executed. The execution of a codelet in the workspace engages in the building of a small perceptual structure. If the building of such a structure is successful, then activation is added to part of the slipnet. The probabilistic nature of the coderack ensures that many different kinds of codelet activity can be explored, gradually zeroing in on those that help produce an answer to the analogy problem.

The slipnet is essentially a deformable semantic network. There are nodes and links among the nodes that correspond, roughly, to concepts and their interrelationships pertinent to the problem domain. Each node has an activation level that varies between 0 and 100. It is somewhat more apt to say that a node and its "halo" of connected nodes of relative high activation dynamically form a currently-relevant concept from the problem domain. Activation spreads between nodes that are linked together and activation in each node decays with time. It is to the nodes that activation is added upon success of a codelet when it builds a perceptual structure in the workspace (this is an example of the ad hoc tight coupling between these two components). Nodes that are fully active are likely to emit codelets designed to further the construction of perceptual structures corresponding to that node. The likelihood of a node becoming fully active is in direct proportion to its activation level.

The workspace is the arena for the construction of perceptual structures. These structures come in several types. There are descriptors, which can be attached to an existing structure and which carry some simple bits of information about that structure. For example, a letter 'a' in the target problem "abc→abd,ijk→?" might have a descriptor attached to it (by a codelet) that indicates that it is the first letter in the alphabet. There are bonds, which are built between two structures, such as a successorship bond from 'a' to 'b' in the same target problem. There are groups built around various structures, such as the successorship group that a codelet might build around the successorship bonds in the 'ijk' sequence. Correspondences are similar to bonds but they are built between elements of different strings. For example, a codelet might build a correspondence between the two instances of 'a' labeled as alphabetically first in the strings 'abc' and 'abd'. A few remaining workspace structures support the perception of the rule at play in a given problem and its transformation to the target string to produce an answer.

The workspace provides codelets with access to these structures in a biased random fashion. Each structure has measures that rank its salience to the system at any instant. These measures derive, in part, from the degree to which the structures are integrated into the emerging set of perceptual structures and, in part, from the activation level of the associated slipnet node. This salience value is what provides the bias for the random selection. The consequence of all this is that a codelet will generally tend to select a structure on which to operate that is the most salient structure in the system, but there is still some chance of any of the other structures to be selected. A composite of these measures in the workspace leads to a value called the temperature. The temperature emerges from the structure-building activity of the codelets and is used as feedback from the system's own activity to regulate the behavior of other components, including the production of more structure-building codelets. It is this fact that gives rise to the self-organizing quality of the perceptual structure-building by the program. When the temperature is high, the system tends to make decisions more randomly and tends to explore more options in parallel and in data-driven fashion. As the temperature falls the decisions tend to become more deterministic, serial, and goal-driven [2].

The program cycles through its behavior in the following way. A codelet is dequeued from the coderack and sent to the workspace for execution. The codelet will be given a structure on which to operate in a biased random manner by the workspace, as described. If the structure-building is successful then some activation will be added to the buffer of the

associated slipnet node (to be added to the node during an update). This process is repeated some fixed number of times (15 in the original Copycat) and then updates take place. The workspace update recalculates the temperature, based on the measures of all its structures. The slipnet update does the spreading and decay of activation and, for those nodes that are fully active, generates codelets that are enqueued in the coderack. Then the cycle repeats. The program's current perceptual activity drives its tendencies, via activation of nodes in the slipnet, for its continued behavior.

In developing the Starcat framework and its instantiation as StarCopy, attention has been given to preserving the details of the architecture and its functioning as described above. There are, however, a number of significant differences in the design and implementation. In the next section we will expound upon these differences. The primary motivation for this reimplementation of Copycat as StarCopy is to test this new framework and learn what has been gained and what lost in the new design.

2.0 Design of the Starcat Architecture

The Starcat framework is a generalization of the essential ideas first presented in the Copycat architecture. Some of the differences in its design pertain to the fact that Starcat is domain neutral. It is only after a developer instantiates the framework for a particular application domain that many of the details of its functioning are realized. For example, the descriptors, bonds, groups and other workspace structures become specific to a domain by their reference to slipnet nodes for certain domain features. So, while the Starcat framework provides bonds in general, it is through instantiation that, for example, a successorship-bond becomes possible in the StarCopy domain. This means that two endeavors are central to the task of instantiating the architecture for a particular application domain. First, one must specify the slipnet for the problem domain. Second, one must provide the codelets responsible for building perceptual structures associated with the nodes of the slipnet. Slipnet specification in Starcat is done through the use of an XML configuration file, which the framework processes upon startup. Codelets are specified simply by writing the classes that extend the base classes of the framework and providing their structure-building behavior in the workspace. There are also various control parameters at both the system level and the component level which can be specified in a single location; these can be modified even at runtime and Starcat will adjust its behavior accordingly. Some of these are discussed below.

Other differences in the design of Starcat pertain to its components and their behavior. Starcat is an asynchronous, event-driven architecture. Each component runs in its own thread. In some sense, each component is a data-structure that provides particular capabilities. The coderack, for instance, has already been described as a probabilistic priority queue. The workspace is basically a storage facility with special methods for accessing its contents in particular ways (e.g., using a biased distribution for random selection). The slipnet manages activation levels and relationships among nodes whose primary job is the production of a varying population of different types of codelets. Separate from these basic data structures are the behaviors that drive their use. These behaviors are called metabolisms in Starcat. Metabolisms manage the activity of the threads for the components. The developer may either create their own metabolisms, essentially drivers for the components, or they may use or extend the ones provided by the framework. These built-in metabolisms expose most of the essential parameters of their behavior to facilitate the experimentation necessary to find the combination that best serves the application. Some of these parameters are described next.

As codelets arrive at the workspace from the coderack, they are enqueued in a local buffer. When the workspace thread is next active, it must decide how many from this buffer to process. This number is called the "execute factor" and is one of the exposed parameters of the built-in metabolisms. The duration of the inactive period ("sleeptime") for a component's thread, and values that control whether and by how much to gradually adjust the execute factor are other exposed parameters. These parameters can be adjusted independently for each component. They play a similar role in the Starcat framework as the number of codelets between updates did in Copycat.

In Starcat the function of codelets has been modified to reflect more profoundly the original metaphor between Copycat and the cytoplasm of a living cell. Like RNA, a codelet's meaning (its behavior) depends on its environment (the component in which it is executed). When a codelet is emitted from the slipnet and arrives at the coderack, its execute method simply asks the coderack to enqueue it. When a codelet is dequeued from the coderack and arrives at the workspace, its execute method engages in some structure-building. When a codelet leaves the workspace and arrives at the slipnet, its execute method adds activation to some slipnet node(s). In this way, all components are simply registered as listeners for codelet events. Codelets become the sole mediators of activity between components. One intended consequence of this is that almost any component configuration can be easily accommodated, as dictated by the needs of the application. There may be a different number of components, new kinds of components, or even changed relationships among the components. All is determined by the behavior defined for a codelet in each of the system's components.

Codelets fall into one of two categories, behavior codelets or control codelets. Behavior codelets are the kind we have discussed so far; their primary functions are to build perceptual structures in the workspace and to add activation to nodes in the slipnet. Control codelets anticipate the possibility of components whose emitted codelet events operate at the meta-

level, triggering changes in the control of other components, such as initiating an update. This feature is not part of StarCopy because it is not a functionality that Copycat possesses. We do believe that it will play an important role in future application development using the Starcat framework. Almost any of the various other parameters that affect the computation engaged by this architecture are exposed in the XML configuration files or in a runtime GUI used for their modification. Some of these include conceptual depth of slipnet nodes, codelet productivity of nodes, amounts of activation deposited by codelets, thresholds for becoming fully activated, and initial activation levels of nodes.

There are some additional features of the Starcat design which are not relevant for the StarCopy application, and so they will not be discussed in detail here. One of these is the option of a continuous model for codelet production by slipnet nodes. In Copycat, a node only produces codelets when it has become fully activated, which happens probabilistically, with a likelihood in proportion to the level of activation of the node, as long as it exceeds a certain threshold. In Starcat it is possible to have nodes emit codelets, again probabilistically, in a number proportionate to their level of activation, even if that is not at full value. Another difference is that codelets may add activation to different sets of nodes in the slipnet depending on whether they succeed or fail in building their target perceptual structure. The utility of these features is determined by the particular application being instantiated and is under investigation in other Starcat-supported projects.

3.0 Experiments with StarCopy

There are five target problems on which the original Copycat program was tested. In these initial experiments with StarCopy we have chosen to focus on three of these. They are 1) "abc→abd, ijk→?", 2) "abc→abd, kji→?" and 3) "abc→abd, xyz→?". In each of these problems, the source transformation is best expressed as "replace letter category of rightmost letter by its successor". The first problem is a straightforward test of the program's ability to recognize and apply that transformation to a different successorship string. The second problem tests the program's ability to apply the source transformation to a target string whose successorship fabric is *left*going rather than *right*going. The third problem presents a set of pressures that uniquely challenges the program. The slipnets for both Copycat and StarCopy do not allow for a circular interpretation of the alphabet. This conflicts with the application of the source transformation to replace 'z' with its "successor". Under the pressure created by the inability to apply such a transformation, the program is forced to build a more refined set of perceptual structures. These highlight, among other things, two salient facts about 'a' in the source string pair and 'z' in the target string. First, 'a' is leftmost in its string and 'z' is rightmost. Second, 'a' is first in the alphabet and 'z' is last. The concept that unites both these facts is 'opposite'. The resulting activation level of the 'opposite' node in the slipnet produces codelets that look for additional instances of that concept. In this problem, that leads to the perception of the target string as going in the opposite direction from the source strings, namely as 'zyx'. The most interesting result the Copycat program produced for this problem viewed the 'x' as the letter to replace with its *predecessor*, giving 'wyz' as an answer. In a 1000-run experiment for each problem, Copycat produces a distribution of answers [2]. Here we are primarily concerned with the ability of StarCopy to produce both the high-frequency answers and the low-temperature answers. Low temperatures generally correspond to answers that have made best use of all the concepts and pressures in the problem.

The first significant discovery about the difference between Copycat and StarCopy involves the triggering of updates. As mentioned, Copycat cycles through each component in lockstep fashion and after a fixed number of such cycles it performs an update on the workspace and slipnet, in that order. In StarCopy this behavior is controlled by the duration of inactivity of the thread for component control, asynchronously for each component. In our initial experiments we used sleeptimes for the control thread that would approximate the 15-codelet update cycle of the original Copycat. That is, we allowed the coderack to wait for only 1ms between dequeue events and we allowed the workspace and slipnet to wait for 15ms between update events (updates are where the impact of accumulated perceptual activity is incorporated into the components). We found that the results of experiments using this arrangement were very different in character from those of the original Copycat experiments. For example, the ratio of "savvy" answers (like KJJ or LJI for the KJI problem) to "rote" answers (like KJD) was much lower than in Copycat.

In order to examine systematically the effects of tuning these sleeptime parameters, a series of experiments was conducted. The results of these are summarized in Table 1. Various combinations of sleeptimes for the three components are explored, first at the low end and then with much larger values. For each combination, 100 runs of the program were measured for ending temperature, number of codelets executed, and the distribution of answers. The chosen problem is "abc→abd,kji→?", because it is less simple than the "ijk" problem (requiring more structure-building) and not as complex as the "xyz" problem. A few runs never produced an answer in the allowed number of updates; those are not included in the table. Three facts are conspicuous. One is that the ending temperature never reached the expected lower values for the lower-valued sleeptimes. Recall that temperature is a measure of how well the structures fit together and correspond to good use of the concepts and pressures involved in the problem. When the sleeptimes were increased (to 50/50/1 and 100/100/1) the temperature was lower, albeit not as low as is typical of the original Copycat. Another fact is that the number of codelets executed is much higher for lower-valued sleeptimes. Finally, the distribution of answers tended to be

more even (across the four most common ones shown in the table) for the lower-valued sleeptimes; whereas for the higher-valued sleeptimes the program clearly favors certain (more subtle) answers over those more rote ones (simply substituting a 'd' rather than a 'successor'). The behavior on all three counts, when the higher-valued sleeptimes are used, is similar to that of Copycat.

Table 1: Component speeds are shown to have significant impact on output of StarCopy (using KJI problem).

Component Speed (ms between updates)			Average values (over 100 runs)		Answer frequencies (out of 100 runs) - *excluding runs producing no answer*			
Slipnet	*Workspace*	*Coderack*	*Temperature*	*# Codelets executed*	*KJJ*	*KJD*	*LJI*	*DJI*
1	1	1	71.31	4194.12	18	27	21	31
5	10	1	82.26	1310.38	15	42	12	13
10	5	1	69.68	711.89	14	39	12	30
15	15	1	70.52	505.47	32	25	26	16
50	50	1	46.93	752.46	38	11	40	11
100	100	1	50.05	745.55	66	18	10	5

These facts are fairly straightforward to explain. In the original Copycat it was seen that. when the number of codelets executed between updates (hereafter referred to as the update period) was too low, there was not enough build-up of history to influence the perceptual activity of the program. When the update period was too high the system was too sluggish, essentially building-up too much history to be able to act upon it effectively. Here we have something similar, except that the parameters controlling the phenomenon are different. Consider the 15/15/1 configuration. It is tempting to imagine that this means roughly 15 codelets are dequeued and sent to the workspace before it awakens to execute them. However, the dequeue operation and the execute operation are not atomic (unlike the counting of events in the original Copycat); that is, they consume some of the time allotted to the thread in question. The overall effect is that the ratio of sleeptimes in StarCopy between the workspace-slipnet pair and the coderack must be much larger than the update period of the original Copycat to achieve the same effect. In newer applications built using the Starcat framework this lesson has been taken into consideration. The use of the execute factor on a component controls just how many codelets execute when the thread becomes active, but thread sleeptime is still a number whose tuning requires experiment and takes on values different in nature from those of its predecessor. This was the first important lesson from the StarCopy experiments.

In Table 2 we see the results of a 100-run experiment for each of the three problems chosen for StarCopy. The table shows the most common answers for each problem and their frequency, the average of the final temperatures of each of those answers, and the average of the number of codelets executed for each of those answers. The sleeptime configuration for these runs was 50/50/1 (workspace/slipnet/coderack, in ms). The behavior of StarCopy is similar for both the 50/50/1 configuration and the 100/100/1 configuration (though with enough difference to warrant further experiments of the type in Table 2 at other speed configurations). Several trends are evident which suggest at least qualitative agreement between Copycat and StarCopy.

Table 2: Average behavior of StarCopy over 100 runs on each problem is shown for comparison to Copycat (runs producing no answer have been excluded).

Answer and Frequency out of 100	*Average Final Temperature*	*# Codelets Executed*
ABC → ABD, IJK → ? Problem		
IJD (17)	51.49	1064.35
IJL (34)	46.63	733.41
DJK (12)	47.72	1852.17
JJK (37)	42.91	843.11
ABC → ABD, KJI → ? Problem		
KJD (11)	50.93	938.32
KJJ (38)	48.85	716.24
LJI (40)	44.33	749.25
DJI (11)	45.76	702.91
ABC → ABD, XYZ → ? Problem		
XYD (42)	50.97	1679.48
YYZ (48)	49.28	1229.42
DYZ (8)	43.67	2003.50

Consider first the "ijk" problem shown for StarCopy in Table 2. The temperature distribution across answers as well as answer frequency is similar to that of Copycat. The two more subtle answers, "ijl" and "jjk", which require perception of successorship and predecessorship relations through the strings, occur twice as often as those involving just the substitution of a 'd'. Commensurately, the temperatures of those answers are somewhat lower. Consider the "kji" problem. Again, the two more subtle answers, which require perceptual structures corresponding to the succesorship and predecessorship fabric of the strings, occur more than three times as often as those involving just the substitution of a 'd'. And again, the temperatures are somewhat lower for those answers. Note, however, that the deepest answer produced by Copycat for the "kji" problem, namely "kjh", was not obtained by StarCopy. The reason for this preference for successorship relations is not clear from these experiments. Finally, consider the "xyz" problem. Here again, the deepest answer from Copycat, "wyz" is not obtained with StarCopy. The data in the table suggests that this may also be related to a slight preference for successorship relations over predecessorship relations (though again the experiments do not suggest why this is the case). However, the only subtle answer that does involve some awareness of predecessorship, namely "yyz" is StarCopy's most frequent answer. Overall these behavioral tendencies support the claim that StarCopy's behavior is at least qualitatively similar to Copycat. Further experimentation is required to determine whether the source of differences pertains more to the details of instantiating Starcat in this domain or to the differences in the design of Starcat itself. Such questions are under exploration in the context of other applications.

One comparison between Copycat and StarCopy is clear from Table 2 (using [2], for example, as a source of data on Copycat). In Copycat, the deeper answers generally took a larger number of codelet executions to produce an answer. We see some of that trend here, in that the most difficult problem ("xyz") clearly takes more codelet executions than either of the other two, which are simpler by comparison. However, looking at the differences between the rote answers (involving a simple substitution of 'd' rather than the transplanted notion of 'successor') and the subtle ones, we see that the rote ones almost always took StarCopy a significantly larger number of executed codelets. Results of individual runs suggest that when the program does not quickly develop structures that support a subtle answer it tends to spend a lot of time investigating the same possibilities repeatedly. Since stopping and producing an answer is something the program decides to do probabilistically, it stands to reason that if it cannot find good low-temperature structures (which increase the chance of stopping) it will just keep trying until it luck would have it stop with the rote answer left as its only option. While the reason remains unclear for the reversal in the number-of-codelets-executed measure between StarCopy and Copycat, it is at least encouraging to consider that the mechanism by which this difference comes about seems reminiscent of the functioning of Copycat.

Finally, Table 3 shows the results of a 1000-run experiment for both the "ijk" and "kji" problems. Here, more details are shown regarding particular structures built (again see [2] for a detailed description of these). Most of the same qualitative comparisons between this data and the same for Copycat can be made. For example, the two more subtle answers to the "ijk" problem occur with notably more frequency than the rote answers, although as with the previous data the exact values of the distribution are somewhat different.

Table 3: For two target problems the behavior of StarCopy over 1000 runs is shown for comparison to Copycat (runs producing no answer have been excluded).

Answer	*Temp.*	*# Updates*	*Groups*	*Bonds*	*Corresp.*	*Descriptions*	*Successor Bonds*	*Predecessor Bonds*	*# Codelets*
ABC → ABD, IJK → ? Problem									
IJD (168)	49.39	45.92	1.26	4.87	2.52	35.09	2.65	2.21	3459.65
IJL (356)	47.15	22.32	1.06	4.83	2.56	33.80	2.61	2.22	773.90
JJK (332)	42.20	27.66	1.11	4.80	2.60	34.15	2.63	2.17	1408.80
DJK (141)	45.80	44.97	1.20	4.79	2.55	34.70	2.63	2.16	3688.25
ABC → ABD, KJI → ? Problem									
KJD (159)	49.44	33.45	1.28	4.89	2.51	35.03	2.43	2.46	1917.40
KJJ (397)	45.90	20.96	1.04	4.85	2.56	33.68	2.62	2.24	864.23
LJI (315)	42.47	21.43	1.10	4.90	2.61	33.93	2.59	2.29	859.70
DJI (123)	45.13	34.37	1.21	4.91	2.64	34.89	2.70	2.20	2652.42

To make this statement more precise, consider that in Copycat the frequency of the "ijl" answer is 969 out of 1000 [2]. Other differences include the aforementioned tendency to execute *fewer* codelets for the deeper answers and more for the rote ones. In some sense this is a good behavior—we would like the program to keep trying if it has not yet settled into well-integrated perceptual structures. However, discovering the precise mechanism behind this difference will require further experimentation. It is worth noting that not all applications require extended deliberation before settling on solid perceptual structures; some to which the Starcat framework is anticipated to be applied actually benefit from making quick

decisions and moving on. This may therefore become another one of the tunable features of the system as its development continues.

4.0 Conclusion

The first substantial result of the experiments with StarCopy suggest that tuning the performance of the architecture will involve markedly different parameters than Copycat, but with many of the same behavioral impacts as those in the original program. In particular, the duration of the inactive period for the threads for the components matters. The frequency of answers and ending temperature values for StarCopy are qualitatively similar to Copycat, but the number of codelets executed seems almost opposite. Further experimentation will reveal whether this derives from the differences in the underlying architectures or from the instantiation process by which StarCopy is made from Starcat. Ongoing and future work with Starcat includes an ALife application using a novel component configuration, a simplified letter-substitution application designed just for experiments on parameter tuning, a genetic algorithm for discovering appropriate parameter values, the automatic completion of musical phrases, spatial mapping and navigation in a mobile robot and distributed perception in a self-organizing network of sensors.

5.0 References

[1] Brooks, R.. "Intelligence Without Representation", A*rtificial Intelligence.* Vol. 47, pp. 139-159, 1991.

[2] Mitchell, M. *Analogy-Making as Perception*. MIT Press, 1993.

[3] Hofstadter, D., et. al. *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*. Basic Books, 1995.

[4] Holland, J. "Escaping brittleness: The possibilities of general purpose learning algorithms applied in parallel rule-based systems", *Machine Learning II* (Michaiski, R, et. al., eds.). Morgan Kauffman, Los Altos, CA, pp. 593-623, 1986.

[5] Kokinov, B. "The Context-Sensitive Cognitive Architecture DUAL", *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*. Atlanta, GA, 1994.

[6] Lawson, J. & Lewis, J. "Representation Emerges from Coupled Behavior", *Workshop Proceedings of the Genetic and Evolutionary Computation Conference 2004*. Seattle, WA, 2004.

[7] Lewis, J. *Adaptive Representation in a Behavior-Based Robot: An Extension of the Copycat Architecture*. Ph.D. Dissertation, University of New Mexico, Albuquerque, NM, 2001.

[8] Lewis, J. & Luger, G. "A constructivist model of robot perception and performance", *Proceedings of the 22nd Annual Conference of the Cognitive Science Society.* Philadelphia, PA, pp. 788-793, 2000.

[9] Luger, G., Lewis, J., and Stern, C. "Problem Solving as Model Refinement: Toward a Constructivist Epistemology", *Brain, Behavior and Evolution.* Vol. 59, No. 1-2, Karger Publishers, Basel, Switzerland, pp. 87-100, 2002.

[10] Lewis, J. & Lawson, J. "Computational Adaptive Autonomy: A Generalization of the Copycat Architecture", *Proceedings of the 2004 Las Vegas International Conference on Computer Science.* CSREA Press, Las Vegas, NV, pp. 657-663, 2004.

[11] Lewis, J. & Lawson, J. "Starcat: An Architecture for Autonomous Adaptive Behavior", *Proceedings of the First Annual Hawaii International Conference on Computer Sciences*. Honolulu, HI, pp. 537-541, 2004.