To the Graduate Council:

I am submitting herewith a dissertation written by Myles Brandon Bogner entitled "Realizing 'Consciousness' In Software Agents." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Mathematical Sciences.

_____
Stan Franklin, Ph.D., Major Professor

We have read this dissertation
and recommend its acceptance:

_____
Dinpankar Dasgupta, Ph.D.

_____
Art Graesser, Ph.D.

_____
Jonathan Maletic, Ph.D.

_____
Edward Ordman, Ph.D.

Accepted for the Council:

_____
Linda L. Brinkley, Ph.D.
Vice Provost for Research
& Dean of the Graduate School

Realizing “Consciousness” In Software Agents

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Memphis

Myles Brandon Bogner

December, 1999

DEDICATION

To Rose Greenbaum

For all of her struggles

ACKNOWLEDGMENTS

I would like to express thanks to my advisor, Dr. Stan Franklin, for his willingness to share his ideas and allow me to explore my own. Dr. Franklin has met me many times on late Friday afternoons when all others have deserted the university. Under his direction I have thoroughly enjoyed pursuing my graduate degrees. I would like to thank Dr. Jonathan Maletic, a committee member and friend, for his encouragement. Dr. Maletic's guidance significantly enhanced my knowledge of software reuse. I would like to acknowledge Dr. Chip Ordman for his teachings. From Dr. Ordman I learned many of the distributed computing techniques used in my research. He and I have traced through numerous paths on his office chalk board. I would like to thank Dr. Art Graesser for sharing with me the ideas of cognitive modeling. Dr. Graesser made sure I stayed the course when testing my implementation. I would also like to acknowledge Dr. Dipankar Dasgupta, for his assistance and comments. I have often paused in front of Dr. Dasgupta's office door to read his postings.

I would like to thank the "Conscious" Software Research Group for our inspiring weekly meetings. I would like to acknowledge the United States Navy's Office of Naval Research and Personnel Research and Development Center for funding this research under grant N00014-98-1-0332.

I would like to thank Michaela, for her courage and support. I would like to thank my sister, Alexis, for helping to hold up the St. Louis fort while I have been away. Finally, I would like to thank my parents, who willingly continue to be wonderful guides.

ABSTRACT

Realizing "Consciousness" In Software Agents describes the first design and implementation of Bernard Baars' global workspace theory. Global workspace theory is a leading psychological model of human consciousness. The "Conscious" Software Research Group at the University of Memphis has labeled agents which implement this theory as "conscious" software agents. As background material for the reader, this work also discusses agents, other existing cognitive architectures, and current software reuse methodology.

This dissertation describes in depth the "Conscious" Agent framework (ConAg), developed by this author. ConAg is a reusable software framework that carefully follows software reuse methodology. ConAg provides a solid foundation for building "conscious" software agents, and in particular, "consciousness" within these agents. A description of two agents built with ConAg are described, as well as the framework's structure. It is beyond this work's scope to address whether or not agents built with ConAg are sentient.

There are several motivators for this research. First, it is hypothesized that a global workspace gives a multi-agent system several advantages. For example, it offers individual agents in the system a means of recruiting the system's other agents to help solve novel and ambiguous problems. Also, it gives a method for attentional focus for the overall system. This provides a means for associative learning and metacognitive functions to take place. This dissertation gives an in depth discussion of the specific

functions of the global workspace in ConAg. It is hoped that the software implementation advantages gained when using a global workspace are evident.

As a second motivation, this research hopes to provide new hypotheses about human consciousness for cognitive scientists and neuroscientists. As a cognitive science theory, Baars' theory does not contain many of the low-level design specifications necessary for a computer scientist's implementation of the theory. To implement the theory, these design decisions had to be made. Many of these decisions can be considered hypotheses on human consciousness. It is hoped that the members of the above disciplines will view these implementation decisions as springboards for the further study of consciousness.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
An Introduction To A Software Implementation of "Consciousness"

Motivations

This work describes a software design and implementation of global workspace theory, a cognitive science theory of consciousness (Baars, 1997). This implementation focuses on the theory's consciousness portion. The theory extends to cover other cognitive mechanisms such as learning and metacognition. Global workspace theory postulates that in humans, consciousness provides for numerous functionalities. These include adaptation, learning, and prioritization. This research is motivated, in part, to show that these cognitive mechanisms, many of which have partly been implemented on machines previously, can be enhanced through "consciousness." Hopefully, this will lead to "smarter" software.

Second, it is important to stress that global workspace theory is a cognitive science hypothesis. As such, it is at a much higher conceptual level than necessary for a software implementation. Therefore, this writer, in consultation with the "conscious" software research group, including both computer and cognitive scientists, made lower-level design decisions for the theory's implementation. These design decisions can be considered hypotheses on how humans minds work. Hopefully, they will be

analyzed by cognitive scientists, neuroscientists and philosophers and lead to a further understanding of mind.

The “conscious” software research group labels software agents (see chapter 2) which implement global workspace theory as “conscious” software agents. As seen throughout this work’s later chapters, these agents are quite complex and very time-intensive to build. They are also closely coupled to their domain. Therefore, a goal for this author’s implementation has been to incorporate current software reuse methodologies to create a reusable framework. New “conscious” software agents, when incorporating this framework, do not have to be implemented from scratch. Instead, large portions of each new agent need no code modification, while other portions need not be reimplemented from scratch. As discussed later, reuse has shown to significantly increase productivity while decreasing defect density, rework, and development costs (Basili, Briand, & Melo, 1996).

This framework, named ConAg: The “Conscious” Agent Framework, is implemented in Java beans (Eckel, 1998). The use of Java beans increases ConAg’s chances of truly being a reusable framework. As of July 1, 1999, ConAg had been developed solely by this writer, and consisted of approximately 60,000 lines of code and over 280 Java beans. While other agent frameworks do exist (Reticular Systems, 1999), this is the first one designed for building “conscious” software agents.

Contents and Author's Contributions

This work contains seven chapters. Chapter 2 gives a literature search overview. It discusses agents, focusing on software agents (Franklin & Graesser, 1997) and their relation to "conscious" software agents. The chapter then describes computational models, comparing "conscious" software agents to established ones. The chapter ends with a discussion of software reuse methodology, focusing on its importance and components. ConAg is designed under software reuse methodology, and ConAg's heavy reliance on this methodology will be evident throughout the later chapters.

Chapter 3 sets the stage by describing global workspace theory and Virtual Mattie (VMattie), the "conscious" software research group's first project (Song, 1998). VMattie is not a "conscious" agent, but contains many of the building blocks for agents which do implement global workspace theory (Zhang, Franklin, Olde, Wan, & Graesser, 1998). The author was actively involved in VMattie's design.

Chapter 4 discusses "conscious" software agents in depth, focusing on "Conscious" Mattie (CMattie) and the Intelligent Distribution Agent (IDA) (Bogner, Ramamurthy, & Franklin, in press 1999). Chapter 4 describes the new architectural style developed by the author for high level "conscious" software agent description (Bogner, Maletic, & Franklin, in press 1999). This author has been actively involved in all stages of the design and implementation of CMattie, and as seen in the reference section, has publications based on work relating to the agent. CMattie is the first "conscious" software agent. IDA is the "conscious" software research group's proof of concept

project. IDA is funded by the Office of Naval Research and will, hopefully, turn into a useful software package. The present writer has been actively involved in all stages of the design and implementation of IDA, including meetings with naval personnel.

For both CMattie and IDA, this author's major contributions have been:

1. The creation of a new architectural style and general architecture to succinctly describe "conscious" software agents.
2. The design from scratch and implementation of these agents' "consciousness" mechanisms. Since "consciousness" provides the backbone in these agents' structures, many of these agents' core components are within this "consciousness" mechanism.
3. The successful implementation and extension of John Jackson's pandemonium theory (Jackson, 1987). Extending pandemonium theory's ideas helped provide a means for implementing global workspace theory's consciousness. This is the first known implementation of pandemonium theory.
4. The design from scratch and implementation of these agents' base package of classes, known as the base codelet package. All codelets in the system utilize this package.
5. The creation of a reusable software framework for utilization in both these agents and in future "conscious" software agents. This framework follows software reuse methodology to create a truly "reusable" framework. It includes a graphical user interface for a view of what is internally occurring

within these agents; in particular, what is occurring within these agents' "consciousness" mechanism.

6. The successful provision for each of Baars' nine major functions of consciousness. These are described in chapter 3, and chapter 9 includes discussion on how the framework successfully provides for each of these functions.

In addition, the present writer has worked on a project to develop an intelligent tutoring system. During that time, he was a primary developer of natural language curriculum scripts which are spoken by the tutoring system. Many of these script innovations are planned for use in IDA's natural language generation portion. Chapter 4 gives a brief tour of these scripts (Graesser, Franklin, & Wiemer-Hastings, 1998).

Chapter 5 describes the design of CMattie and IDA's "consciousness" mechanism. This includes how information is brought to "consciousness," how the "consciousness" apparatus functions, how the "conscious" information is disseminated, and what "consciousness" provides to these agents. Chapter 6 delves into depth on the structure of ConAg, illustrating how it implements Chapter 5's design.

To be a successful software framework, it has to be straight forward in its use. Chapter 6 also describes how different modules, developed by other "conscious" software research group members, integrate with ConAg and utilize its features. In addition, it gives testing results. Finally, chapter 7 specifically answers the question of how ConAg provides for Baars' nine functions of consciousness. In addition, it draws conclusions, hypotheses, and future speculations.

Chapter 2
Agents, Models, & Reuse

Introduction

This chapter gives overviews of research relevant to this dissertation's focus. It begins with a discussion of agents, focusing on software agents. "Conscious" software agents are cognitive software agents. It then gives an overview of three computational models. Described in this chapter and further beginning in chapter 4, "conscious" software agents are different than these models as they follow the action selection paradigm of mind (Franklin, 1995) and focus on consciousness directly in their implementation global workspace theory (Baars, 1997). Finally, this chapter describes current software reuse methodology. The "Conscious" Agent Framework (ConAg) draws heavily on this practice to create a reusable framework.

Agents

On the software agents mailing list (*http://www.csee.umbc.edu/agentslist/)*, frequently asked questions include: what is an agent; why are agents needed; can I call this an agent; is there a dummy agent available on the web; what makes an agent distinct from other software; what are the differences between multi-agent and single-agent systems; what are the differences between agent architectures, cognitive architectures, and software architectures; can agents be used in the medical area; and many more along

the same lines. A main reason for these questions is the increased interest in agents as they continue to rapidly grow in popularity. Also, there are many varied definitions of what an agent even is. In addition, agents are now classified: artificial life agents, autonomous agents, cognitive agents, goal-based agents, irrational agents, mobile agents, multi-agents, rational agents, reflex agents, robotic agents, utility-based agents, and viruses are all examples.

Russell and Norvig have a highly regarded definition of an agent: it is anything that can be viewed as having sensors through which it perceives its environment and having effectors with which it acts upon this environment (1995). Under this definition, humans are agents with sensors such as eyes and ears and effectors including hands and feet. Described in later chapters, "conscious" software agents sense through their perception modules and act upon their environment in numerous ways such as sending out email messages through mail output modules.

However, by this definition, a refrigerator's thermostat is an agent. It senses its environment, the air temperature, and it acts upon its environment by causing the cooling of the refrigerator. Obviously, more is needed to identify agents with "intelligence." One way to do this is to classify agents as irrational or rational. Rational agents are said to do the right thing, meaning that they will take actions which allow them to achieve the greatest success. CMattie is a rational agent, and, presumably, this is the goal of rational humans as well. Rational agents can be classified as ideal rational agents. These perform actions which are expected to maximize the agents' success. These actions are taken

based on the agents' built-in knowledge and what they have perceived up to this point. Notice, thermostats are ideal rational agents.

Numerous researchers have defined autonomous agents (Franklin & Graesser, 1997). Franklin and Graesser provide a now prominent definition, stating that an autonomous agent is a system situated in, and part of, an environment, which senses that environment, and acts on it, over time, in pursuit of its own agenda. It acts in such a way as to possibly influence what it senses at a later time. In other words, it is structurally coupled to its environment (Maturana 1975; Maturana and Varela 1980). Biological examples of autonomous agents include humans and most animals. Non-biological examples include some mobile robots, and various computational agents, including artificial life agents, software agents and computer viruses. "Conscious" software agents are designed to be autonomous agents under this definition. It so happens that the thermostat is one also.

The thermostat is lost, however, when classifying agents as cognitive agents (Franklin, 1997). Both humans and "conscious" software agents can also be considered cognitive agents. These are autonomous software agents that are equipped with cognitive (interpreted broadly) features chosen from among attention, concept formation, decision making, emotions, learning, long and short-term memory, multiple senses, perception, problem-solving, etc. While this definition is not crisp, cognitive agents can play a synergistic role in the study of human cognition, including consciousness (Bogner, Ramamurthy, & Franklin, in press 1999). This dissertation uses cognitive features such as attention both in the folk-psychological and technical senses.

Due to their cognitive nature, “conscious” software agents are currently relatively unique; this can be seen through CMattie. CMattie is designed to fully function as a “human” seminar announcer. She communicates entirely via the natural language found in email messages. Her architecture combines numerous artificial intelligence techniques to model the human mind. While CMattie’s role will not fully be discussed until chapter 4, it is beneficial to point out that there are in fact other email and scheduling agents. For example, the Calendar Agent automates a user’s scheduling process by observing the person’s actions and receiving direct feedback (Kozierok, 1993). The Maxims system is an email filtering agent which learns to process a user’s incoming mail messages (Lashkari, 1994). These two systems employ other agents that collaborate to overcome the problem of learning from scratch. Re:Agent is an email management system (Boone, 1998). This agent routes email to handlers that delete, download, sort, and store these messages on palmtop computers and pagers. Re:Agent learns the emails’ features in order to learn how to appropriately classify the messages. The Visitor-Hoster system is aimed at helping a human secretary organize a visit to an academic department (Sycara, 1994). The secretary is presented with a user interface where she inputs relevant information to the agent about the incoming visitor. The agent then plans the visit, and returns to the secretary for confirmation. In addition to differences in tasks, CMattie’s architecture, method of communication, degree of autonomy, and emphasis on “consciousness” make her relatively unique among these types of agents. IDA continues the trend.

Probably the types of agents gaining the most prominence are internet agents, often known as mobile agents. These agents are beginning to be commonly found on the internet, performing tasks such as information retrieval (Menczer, Belew, & Willuhn, 1995), network routing (Bonabeau, Henaux, Guérin, Snyers, Kuntz, & Theraulaz, 1998), and security (Crosbie & Spafford, 1995). At times these agents may need to communicate (*http://www.fipa.org*) and even barter with one another (Wurman, Wellman, & Walsh, 1998).

Computational Models

Several researchers have created computational models which attempt to model human intelligence. Often these models try to depict a specific aspect of cognition, such as the learning performed by connectionist systems (Haykin, 1994). Some attempt to approach complete cognitive architectures. This section touches on three that begin to: Soar (Laird & Rosenbloom, 1996), CAPS (Just, Carpenter, & Hemphill, 1996), and Pacrat (Johnson & Scanlon, 1987). There are others, including the well known ACT* (Anderson, 1991), OSCAR (Pollock, 1995), and "conscious" software agents. This section also touches on why "conscious" software agents do not utilize Soar and CAPS, the two most prominent cognitive architectures.

Soar

Soar is a cognitive architecture designed for general intelligence (Laird & Rosenbloom, 1996). It provides the means to study general properties of intelligence. In this way, Soar goes beyond many systems which solely deliver the ability to analyze specific algorithms

modeling specific problems. This design is motivated by cognitive issues such as humans having a variety of behaviors. Soar's underlying structure is built so that a full range of cognitive tasks are available (Franklin, 1995). Soar's creators declared specific measures in order to monitor the progress of how close Soar is to general intelligence (Rosenbloom, Newell, & Laird, 1991). These include the ability to work on a full range of tasks, to be able to use the full range of problem-solving methods and varieties of knowledge, to be able to interact with the outside world in real time, and to learn about the world and the system's own performance. Soar has achieved significant progress in numerous areas such as learning, outside interaction, problem solving, and range of tasks.

Soar's hypothesis is that general intelligence systems must be realized by symbolic systems, which may be implemented on a lower-level architecture. Therefore, Soar relies on production systems (Russell & Norvig, 1995) in its quest to perform a full range of tasks from routine to extremely difficult. Soar has very little built in default knowledge; most of the knowledge is domain-specific knowledge provided by the user.

Soar is often analyzed along four levels. The knowledge level is the most abstract level and is used to characterize the system's behavior. When Soar acquires knowledge, it is available for all future goals. There is no capacity limitations on the amount of knowledge that can be available, or on Soar's ability to utilize it in the selection of actions that achieve its goals. The essential feature of the knowledge level is that Soar's behavior is determined by the content of its knowledge, not by aspects of its internal structure. Soar's intelligence can be measured by how well the system applies its knowledge to its tasks. The problem space level is concerned with the characterization of

problem spaces, operators, relationships between goals and subgoals, and states. The problem space determines the set of operators and states that can be used during the processing for goal attainment. One of Soar's unique properties is that it is quite entrenched in the problem space level as it determines how tasks are formulated. The symbolic level contains the details of control flow and memories. More specifically, it provides the basic control structure, memory organization, and processing structure for supporting the knowledge level. The implementation level is the underlying technology on which the symbolic architecture is built. The implementation details are largely irrelevant to the Soar architecture. However, this level does show boundedness, correctness for the execution of the symbolic level's processes, and efficiency. In addition to these knowledge levels, Soar contains declarative, episodic, and procedural knowledge, and its primary task is to perform procedural knowledge.

One of Soar's most notable features is its ability to subgoal. When goals cannot be reached, due to a lack of knowledge or a tie in the decision procedure, it is known as an impasse and subgoals are created. Soar's subgoaling is known as automatic subgoaling as the architecture automatically generates subgoals based on an inability to make progress on the task. Soar also does universal subgoaling as it creates any and all types of goals including metagoals (goals for deciding what operator to select), goals for achieving operator preconditions, and goals for performing certain operators. Soar can create subgoals of subgoals, creating a hierarchy of goals, in order to help achieve the overall goal. To get through the created subgoals, Soar uses weak methods. These types of methods are heuristic searches (Russell & Norvig, 1995) which control the search through

the problem spaces. Weak methods do not require significant built-in knowledge and include common artificial intelligence search techniques such as generate and test, hill-climbing, and means-ends analysis. The Soar architecture automatically terminates the subgoals when the impasse is resolved.

When a subgoal is realized, Soars collapses the subgoal into a chunk. This is a method of one shot learning known as chunking. Chunking creates a production that contains both the condition, describing the situation leading up to the impasse, and the weak methods, utilized to resolve the impasse. In other words, chunking summarizes the problem solving of the subgoals, so that in the future, chunks fire in situations which would have previously led to subgoals. Chunking is a background process which is invoked automatically whenever a subgoal result is produced. Learned chunks are usable throughout the entire system. Described in later chapters, "conscious" software agents also utilize a chunking technique, but its form is different largely due to architectural differences.

Out of Soar's goals and subgoals come two hypotheses about the relationships among goals in intelligent systems. First, subgoals are created to obtain knowledge so that the pursuing of a goal can continue. Second, the functions for creating and selecting goals are embedded into these systems' architectures. This impasse-driven mechanism has the unique property of eliminating the need for deliberate goals. "Conscious" software agents' goals are built in or learned.

Soar has been used in numerous successful applications including expert systems, natural language parsing, resolution theorem proving, and robotic arms. In addition, Soar

has used in applications such as algorithm design, real-time control of simulated aircraft (Tambe, Johnson, Jones, Koss, Laird, Rosenbloom, & Schwamb, 1995), medical diagnosis, blood analysis, production-line scheduling, chemical process modeling, and intelligent tutoring. Soar has been used to model humans including concept acquisition, immediate reasoning tasks, instruction taking, natural language understanding, number conservation, problem solving, syllogisms, verbal reasoning, and visual attention. However, Soar is still incomplete in respect to being a full unified theory of cognition in some ways, such as having unbounded working memory.

CAPS

CAPS has been used to model problem solving, spatial reasoning, and text comprehension. Like Soar, CAPS is a production system. CAPS, however, differs from classical production systems in several ways. CAPS is actually a hybrid as it combines a production system with an activation-based connectionist (Haykin, 1994) system. Unlike traditional production systems, each element in CAPS has an associated activation level. This allows elements, which can represent grammatical structures, thematic structures, and words, to have varying degrees of activation. The production's condition portion specifies not only the presence of an element but also the minimum activation level (a threshold) at which the element satisfies the condition. In addition, the precondition elements are weighted, so that all preconditions do not need to be met for a production to fire. If an element's activation level is above threshold, it is considered to be in working

memory and available to initiate other computational processes. These can either be actions or influence mental parameters (the contents of working memory).

Productions in CAPS change an element's activation level by passing the source element's activation level, with some degradation, to this output element. In addition, a reiterative action allows for symbolic manipulation. More specifically, productions fire reiteratively over successive cycles, allowing for the output elements' activation levels to be gradually incremented until a threshold is reached. CAPS also allows multiple productions with fulfilled preconditions to fire in parallel on a given cycle.

CAPS' learning adjusts both productions' condition weights and their firing thresholds. This method of learning captures adaptation, as over time, weights and thresholds can be lowered allowing frequent actions to occur more readily. Unlike Soar, new productions are not learned via chunking.

Why "Conscious" Software Agents Do Not Use Soar or CAPS

Soar and CAPS have relatively long traditions of being comprehensive cognitive models. Even so, in the creation of the "conscious" software agent architecture, the "Conscious" Software Research Group is developing a quite detailed computational model of cognition. This model has several distinctions when compared to Soar and CAPS. As previously described, both Soar and CAPS are heavily engrained in action selection. Each of these models most likely could have been the action selection mechanism for "conscious" software instead of behavior networks (Maes, 1989). Certainly, there would have been advantages and disadvantages to their use, different integration problems

would have arose, and different extensions would be necessary. Behavior networks simply work well in this context as they provide an appropriate level of abstraction and relatively easy tuning once the behavior streams are in place. For the "conscious" software architecture, the most important feature of the behavior network is its ability to be extended to provide for global workspace theory's goal contexts (see chapter 4). As comprehensive cognitive models, Soar and CAPS do not provide the focus on consciousness for which the "conscious" software agent architecture strives. The "conscious" software agent architecture has been developed under the premise that cognition is not unified. Instead, the "conscious" software agent model assumes that humans are an evolutionary kludge. The "Conscious" Software Research Group does not believe that a single mechanism can cover all of cognition.

Pacrat

Pacrat's designers hoped to duplicate the functions of the "mammalian brain" by designing and building "feeling-thinking" machines. These designers hoped to produce the functions of the brain in electronic circuitry. In addition, they hoped to provide insight into how the brain potentially works. In fact, the authors explicitly state that a distant goal is to create a machine that acts and thinks like a person.

Pacrat is an artificial life agent created to be a feeling-thinking machine. Pacrat feels, learns, and thinks about what it has learned. Pacrat's actions are driven by eight brain centers: amygdala, cingulate gyrus, hippocampus, hypothalamus, isocortex, medial forebrain bundle, reticular ascending substance, and the thalamus. These brain centers

model the functional relationships between mammalian centers, but not the specific electrical activity. The brain centers' interaction gives rise to a sophisticated structure. In Pacrat, individual neurons are not simulated, but codons, the activities of assemblages of neurons, are.

Pacrat's universe can be represented by a grid. Pacrat has the ability to move around this universe through four motor neurons driven by the isocortex's motor area. Pacrat's moves are based on the sensory input and the prevailing emotion. In addition, Pacrat feels hunger. The activity of the hunger center is tied to the contractions and expansions of the stomach. As the stomach empties, the hypothalamus' hunger center becomes more active. Pacrat also feels anger and frustration, which are inhibited by eating.

Pacrat experiences agoraphobia, the fear of open places, when his back is uncovered. This fear is determined by the level of activity in the cingulate gyrus. Pacrat has curiosity, which arises when codons in the isocortex that have not previously been excited become activated. Pacrat has habituation, as curiosity fades due to continuous excitement of codons in the isocortex. Pacrat contains location sense as each location has a different sensory neuron which becomes active in that location. When Pacrat is first started, he does not know where one location is in relation to another. He does know where he is, and he has an aversion to returning to where he's already been.

Pacrat can move all four cardinal directions within the boundaries of his universe. Pacrat is motivated by fear and hunger. Food is placed randomly in Pacrat's universe in three locations. Pacrat gets angry when food is not where it is expected. Pacrat also

sleeps, due to his reticular ascending substance becoming less active, and awakens when hungry enough. Pacrat also contains a reward-punishment mechanism. When food is found, this reward mechanism drives the learning of preferred direction of movement when similar codons are active in the future. Pacrat's fear is masked by hunger, and, therefore, drives Pacrat when hunger is satisfied. Fear drives Pacrat back to his burrow. When Pacrat's burrow is reached, his back is covered, and once again the reward mechanism is activated to drive learning. Pacrat's learning is for survival.

Randomness forces Pacrat out of obsessive behavior patterns. His amygdala mediates anger. Pacrat performs three forms of thinking. First, he evaluates the consequences of his last moves. Second, Pacrat performs recognition when he moves to an area of interest. Finally, Pacrat has insight, "the basic mechanism of rational thought," (Johnson & Scanlon, 1987) when he finds more efficient paths to food.

Software Reuse

What Is Software Reuse?

Software reuse is the use of existing software to create new software instead of building the new system from scratch (Krueger, 1992). Reuse involves using both previously defined higher-level concepts such as ideas and knowledge and lower-level specific components such as objects in new situations. The software reuse process is commonly thought to involve three steps (Prieto-Dias & Freeman, 1987):

1. Accessing and choosing a reusable artifact.
2. Understanding and adapting the artifact to the application's purpose.
3. Integrating the artifact into the product currently being developed.

Portions which can be reused include design structures, documentation such as manuals and specifications, and source code. Reuse involves both black-box techniques, utilizing a component as is, and white box techniques or code scavenging. White box reuse occurs when existing components are modified to fit the needs of a new system. Adaptation is much more common than straight reuse as available components usually do not match the desired functionality.

Reuse Is Often Not Standard Practice

Software reuse is often not standard practice in software development organizations. Reuse is difficult as abstractions for large and complex systems are typically complicated. It is often difficult for developers to learn these abstractions. Research has shown that a reuser's skill is important in determining their levels of reuse. Many developers are not trained in reuse, and those with training are often not pressed to practice it. This is often the case in the profit-driven corporate setting, where implementing an effective reuse mechanism has a high initial investment (Kaspersen, 1994). This investment takes several forms including new training for existing personnel, the establishment of reuse repositories and hiring of maintainers for them, and an incentive program for establishing reuse. In addition, an organizational-wide classification scheme is often needed. For reuse to be attractive, the effort to use existing code must be less than the effort needed to

write new code, which is often not the case. For example, without an adequate classification scheme, reuse becomes less attractive as it is difficult to distinguish between similar items.

The Motivations for Software Reuse

Reuse techniques have been gaining momentum as they have potential to significantly reduce development costs, maintenance costs, and unrealizable schedules. Software reuse has repeatedly been suggested as a means for successfully combating the software crisis: the problem of building reasonably costing but large and reliable software systems (Mellor & Johnson, 1997). Reusability is widely believed to be a key to improving software development quality (Biggerstaff & Richter, 1987). Reuse results in a completed systems' containing fewer total symbols with less time having been spent on the symbols' organization. Therefore, in a sense, reuse enhances software developers' capabilities, and most developers prefer to reuse than write code from scratch when given the option (Frakes & Fox, 1995).

According to Biggerstaff and Richter, a good reuse system addresses four problems:

1. The ability for developers to be able to find necessary components, both exact and similar matching ones.
2. A means for easily understanding the components. This is particularly key when components need modification.

3. A method by which components can be modified in order to apply them to new domains.
4. A way to appropriately document newly composed components. This representation should illustrate the composed components both as independent entities as well as showing how they can be modified to fit new domains.

Using common object-oriented development with standard tools, reuse has been found to significantly reduce both defect density and rework while significantly increasing productivity (Basili, Briand, & Melo, 1996). This illustrates the potential to decrease software development costs and cycle time as human time and effort are reduced in software construction. For black-box modifications, there appears to be no observable difference between verbatim used and slightly-modified code. For white-box reuse techniques, reuse has been shown to decrease rework, especially for experienced developers, even when extensive code modification is required.

How Does Reuse Work?

Software reuse involves four dimensions:

1. Abstraction is the central feature of software reuse. Abstraction allows for a succinct description of an item, highlighting the important information while leaving out what is unimportant. A common example of an abstraction technique are object-oriented languages' class definitions. These languages' provision for inheritance also allow for a reuse class hierarchy. These subtype hierarchies are helpful in finding reusable items.

2. Selection provides classification schemes for organization and the finding of reusable artifacts. Selection works well when the representation is clear on what the artifact does. To be effective, the classification schemes must allow developers to find components faster than write them.
3. Specialization allows developers to modify general components to fit their specific needs.
4. Integration allows developers to combine their specialized components into a new software system.

There are many techniques which when utilized help foster software reuse. Some of the main techniques are the use of architectural styles, design patterns, and objects.

What are Architectural Styles?

Research has illustrated that design reuse does have several advantages over simple code reuse (Johnson, 1997). Design reuse is common as it can be applied to many contexts. In addition, as has been the case in the development of "conscious" software agents, the design process can be applied earlier in the development process, and, thereby having a larger impact on the project. Also, true to form with "conscious" software agent development, most design reuse is informal and happens with experienced developers. Design reuse allows for open systems, and it allows the "Conscious" Software Research Group's developers to share a common vision.

Architectural styles are a form of design reuse. Architectural styles provide a collection of constraints, building-block design elements, and rules for composing a

system (Monroe, Kompanek, Melton, & Garlan, 1997). There are several benefits to architectural style usage. For example, routine solutions with well-understood properties can be reapplied to new problems with confidence, potentially leading to significant code reuse. In addition, architectural styles can be applied to a broad range of problems, such as the different domains for "conscious" software agents.

Each architectural style has its own notation, or specialized design language, describing:

- The structural and semantic properties of systems falling within the style.
- A common vocabulary such as "blackboard system," "client-server system," and "database." A semantic interpretation is also provided so that the composition design elements have well defined meanings.
- The patterns of interaction of systems built within the style. These design rules (constraints) determine which design element compositions are permitted. For example, all "conscious" software agents' processes have access to a single blackboard.
- Analyses that can be performed on systems built in the style.

What Are Frameworks?

Frameworks are often not well understood and misused outside the object-oriented community. Frameworks are reusable designs of all or part of systems. They are commonly represented by a set of abstract classes and the way these classes' instances interact. A framework's purpose is to provide an application skeleton that can be

customized by developers. Framework's are powerful as they can significantly reduce the amount of effort necessary to develop customized applications, thereby saving organizations time and money.

Frameworks are a form of design reuse as they express reusable designs. They are at a lower level than architectural styles as they are more concrete. In fact, frameworks are actual programs, and, therefore, users of frameworks are often tied to a programming language. Because of this, frameworks are more closely tied to their domain than architectural styles. Therefore, successful frameworks must be consistent throughout more so than architectural styles. Since frameworks are programs, they are often easier for programmers to learn and apply than architectural styles. This occurs partially because only a compiler is needed, not special design notation or software tools often utilized when creating architectural styles.

When using frameworks, developers often think they are just using an object-oriented language's class library. However, frameworks are different than class libraries as frameworks reuse high-level design. With frameworks, there is more to learn before classes can be reused. For example, a set of classes must typically be learned at once, and classes are not reused in isolation. A framework can usually be distinguished from a class library if there are dependencies among components and developers learning the library comment on its complexity. Because of this complexity, frameworks require quality documentation. Even with the difficulty which comes in learning a framework, expert developers normally prefer frameworks over special-purpose languages as they are easier to extend.

What Are Components?

Components are actual working code portions and are designed for reuse. Ideally, components should be easy to learn. Often, with black-box reuse, developers do not need to learn how components are implemented. Components are simply connected to create a new system. By using existing components, more reliable systems are usually created and are, therefore, easier to maintain. As components increase in generality, the payoff for use in narrow focused domains diminishes. On the flipside, with component growth, the payoff when reusing the component increases more than linearly due to the complexity costs. Larger components, however, often become more specific which increases the costs when modification is required.

Frameworks and Components

Frameworks are intertwined with components, and they are cooperating technologies. Frameworks make it easier to develop new components. For example, frameworks provide a standard way for components to do data exchange, error handling, and invoke operations on other components. In other words, frameworks allow components to make assumptions about their environment, making component integration easier. Frameworks provide specifications and templates for new components and allow new components to be built out of smaller components. Frameworks can actually be viewed as components in the sense that applications might use several components, and vendors sell them as products. As a whole, frameworks are more customizable than components and have more complex interfaces, again highlighting the difficulty of learning a framework.

What Are Design Patterns?

Frameworks are composed of micro-architectural elements called design patterns. Design patterns describe solutions to recurring problems and a context for which the solution works. They include the solution's costs and benefits. Design patterns represent the common idioms found repeatedly in software designs and makes them codified, explicit, and applicable to similar problems. Patterns emphasis is placed on documentation and literary style rather than code generation or tools. Design patterns are useful as a documentation tool for classification of design fragments, making it easier for a development team to add new members (Cline, 1996). Design patterns provide a standard vocabulary for developers. They communicate information between designers, programmers, and maintenance programmers at a significantly higher level than individual classes. They provide a list of items to look for in a design review. Maintenance programmers are less likely to break existing code when they understand and work to preserve the integrity of design patterns during maintenance changes. Patterns are particularly useful for building robust designs in situations where the trade-offs are well understood. When specifying and reusing design patterns, there are three fundamental requirements to be followed. First, the design domain must be well understood. Second, the patterns must support the encapsulation of design elements. Finally, the design patterns must be responsible for a collection of well-known and proven design idioms.

Conclusions

This chapter described agents, computational models, and software reuse. Software reuse is relied upon heavily to create ConAg as a reusable framework based on an architectural style for “conscious” software agents. “Conscious” software agents are cognitive agents that implement Bernard Baars’ global workspace theory. This theory’s description is a major portion of chapter 3.

Chapter 3
Setting The Stage

Introduction

This chapter discusses Virtual Mattie (VMattie) (Song, 1998; Zhang, Franklin, Olde, Wan, & Graesser, 1998) and global workspace theory (Baars, 1997). VMattie is a software agent containing many of the building blocks necessary for agents implementing global workspace theory. While VMattie is not a "conscious" agent, she directly preceded CMattie, the first "conscious" software agent. "Conscious" software agents significantly extend VMattie's design and, they implement global workspace theory. This chapter, therefore, sets the stage for a description of "conscious" software agents.

VMattie

Introduction

VMattie is an intelligent autonomous agent. She functions in a clerical role. Specifically, she coordinates departmental seminar information, carrying out a role originally performed by the department's former secretary, Mattie. The agent was developed by Stan Franklin and the "conscious" software research group and programmed by Honjung Song, Zhaohua Zhang, and Aregahegn Negatu. She is implemented in Java.

This section first discusses VMattie's job description. Next, the reasons why VMattie can be considered an autonomous agent are addressed. The agent's architecture is then presented. Finally, testing results are given, and conclusions are drawn with mention of how this agent can be improved.

VMattie's Job Description

VMattie is an unsupervised agent. She functions to announce the University of Memphis' Department of Mathematical Sciences' weekly seminars. VMattie communicates completely via email. Below is a discussion of the different tasks performed by the agent.

VMattie gathers seminar information from seminar organizers. She accepts email from organizers about their upcoming seminars. Since there is no predetermined format which the organizer's email messages must take, VMattie has natural language processing ability (Zhang, Franklin, Olde, Wan, & Graesser, 1998). The agent generates and sends acknowledgments to emailers for every incoming message.

VMattie composes the upcoming week's seminar announcement. She composes messages stating she has incomplete information. She also writes messages saying a received message was not understood.

VMattie emails the composed seminar announcements to a recipient's list at a specified time. To do this, the agent maintains a list of people who receive the weekly seminar announcements. Therefore, VMattie accepts incoming email for requests to join and leave the seminar list.

VMattie As An Autonomous Agent

Based on Franklin and Graesser's definition of an autonomous agent (1997), described in chapter 2, VMattie has many properties which enable her to be an autonomous agent.

- Her environment is the UNIX operating system.
- The agent's niche is the maintenance of seminar announcements.
- VMattie can sense incoming email. The degree to which she actively understands these messages corresponds to different perceptual levels. She also is aware of dates.
- The agent's multiple drives are diverse and explicitly represented.
- VMattie has a distinctive action selection mechanism, known as the behavior network, which is not controlled by a central executive.

VMattie's Architecture

Figure 3.1 illustrates VMattie's architectural components. VMattie's architecture is an original high-level agent architecture. The architecture is largely based on the behavior networks developed by Pattie Maes (Maes, 1990) and the model of perception found in Hofstadter and Mitchell's Copycat project (Hofstadter & Mitchell, 1994). Both architectures have been modified and significantly extended for VMattie.

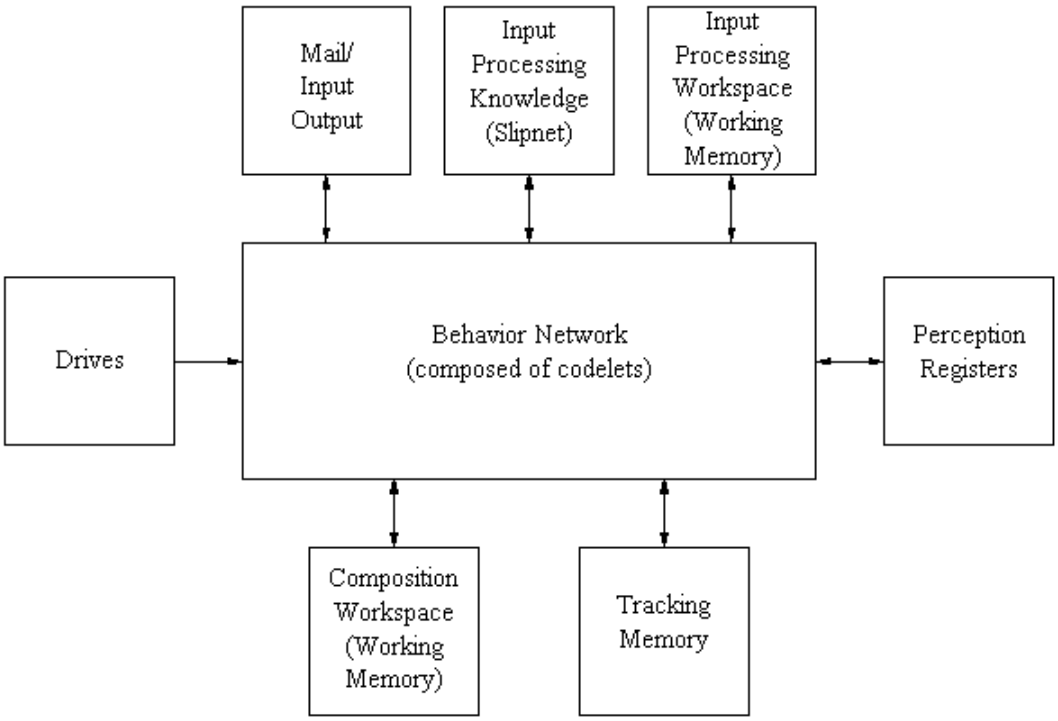

Figure 3.1: VMattie's Architecture
(Bogner, 1998)

Drives

The drives portion of the architecture is based on Maes' goals. The agent's drives correspond to her tasks found in the above Job Description section. All of VMattie's drives are built into the agent. These drives can operate in parallel. Some drives vary in urgency. For example, the urgency level for sending out a seminar announcement may be higher as it gets closer to the time to send the announcement. This varying in the level of urgency is an addition to Maes' original work. Each drive activates behaviors which work to fulfill the drive.

Behavior Network

The behavior network is composed of behaviors whose role is to fulfill the drives. Behaviors in VMattie's architecture correspond to Maes' competencies. Behaviors have an activation level. In general, this activation level is affected by the drives, the agent's internal conditions, and the perception registers. The perception registers serve as the behavior network's environmental inputs. Behaviors have preconditions that must be met. For example, a behavior's preconditions might be fulfilled if an organizer's email message contains specific items of information. A behavior's activation level increases as more of its preconditions are met.

A behavior's activation level is spread to other behaviors. Broadly speaking, its activation spreads to three locations. A behavior's activation is spread to those behaviors which can fulfill this behavior's unmet preconditions. Also, the behavior's activation is spread to the behaviors whose preconditions can be filled by this behavior. Third, the specific behavior sends inhibition, causing a reduction in activation level, to all behaviors which can remove one of its currently met preconditions. Due to the way behaviors spread forward and backward activation, each behavior can be thought to be a part of a behavior stream. If a behavior has a high enough activation level and all of its preconditions are met, it has the potential to become active. Only one behavior in a behavior stream can be active at a time. The active behavior is determined by choosing the executable behavior in the behavior stream with the highest activation level above a

threshold level. Each behavior stream can also be thought of as a plan, created and executed without the building of a search tree.

As a behavior's activation spreads, it diminishes in strength. Also, activation level continually decays at a slow rate. Once a behavior performs its function, its activation level returns to zero. The behavior network is tunable through global parameters.

Perception

VMattie's sensory data are the incoming email messages she receives. Perception for the agent occurs by her comprehending these email messages. VMattie contains three perceptual components: the input processing knowledge, the input processing workspace, and the perception registers.

Input Processing Knowledge (Slipnet)

VMattie's input processing knowledge, also known as the slipnet as it is based on Copycat's slipnet, contains the knowledge needed to understand incoming email messages. Two years worth of email messages to the department secretary were studied in order to generate the knowledge utilized by VMattie's slipnet. Items in the input processing knowledge include the message type, ways to identify the different portions of email messages such as the name of the seminar and speaker, and the abbreviation of words. A common case of abbreviation found in departmental email is seen in the writing of building names, such as Dunn Hall being abbreviated DH, D.H., or D. Hall.

Another common example is in the days of the week, where Thursday can be abbreviated Thu, Thurs, Th, etc.

VMattie's input processing knowledge contains knowledge of nine message types. Examples of message types include messages declaring the establishment of a new seminar and messages stating the upcoming speaker and topic for a seminar. VMattie uses surface level natural language processing in conjunction with a feed-forward neural network to determine an email's message type. The highest output value from the neural network is taken to be the candidate message type. If the output of the neural network is inconclusive, VMattie sends back an acknowledgment saying the message was not understood.

Input Processing Workspace (A Working Memory)

Sometime after a candidate message type is determined, a message template of this type is placed in the input processing workspace. Codelets, described below, work to fill the template's fields. Similar to Copycat, as mandatory fields in the template are filled, the temperature, representing the proximity to completeness, falls. If the temperature falls low enough a message is considered understood. However, if a certain number of mandatory fields remain empty after the codelets have completed their tasks, the next highest output of the neural network is tried as the appropriate message type. For safety in the event a message is classified incorrectly, VMattie acknowledges every message. If the acknowledgment conveys an incorrect understanding of the message, the seminar organizer can send a reworded message.

Perception Registers

Once a message template has been filled, its contents are moved to the perception registers, and the perception module can begin working to understand another message. The perception registers are similar to a blackboard. Information which is placed in the perception registers is available for utilization by VMattie's other modules such as the message composition component described below.

Codelets

Each codelet can be thought of as a small distinct agent designed to perform one task. The term codelet originated with Copycat. VMattie's codelets correspond to global workspace theory's processes, described later in this chapter. VMattie's behavior network and perceptual module, both described above, are largely implemented via codelets. For example, one codelet's task in the slipnet is to fill a message template's speaker name field. Codelets perform the vast majority of VMattie's actions.

Most codelets serve to implement a behavior or a portion of the Slipnet. However, primitive codelets also exist. A primitive codelet does not serve the behavior network or slipnet. Instead, it functions independently to perform housekeeping functions. For example, a primitive codelet might poll for an incoming email message addressed to VMattie.

Tracking Memory

Tracking memory contains the information utilized in composing outgoing email messages. Tracking memory contains the default information on seminars, such as the day of the week each one occurs. It saves the current seminar announcement mailing list. Both the seminar and mailing list information are updated via codelets attached to behaviors. Tracking memory also stores the templates for different types of outgoing messages. The corpus of email messages collected for two years contributed to the building of tracking memory.

Composition Workspace (A Working Memory)

All outgoing messages are composed in the composition workspace. Message composition consists of filling the fields of an outgoing message template. The information used to fill the template fields comes from the tracking memory and the perception registers. There is always a copy of the current seminar announcement being generated in the composition workspace. As new information arrives in the perception registers and tracking memory, the template fields are filled. When a seminar is announcement is mailed, a new default announcement template is placed in the composition workspace.

Mail Input and Output

VMattie's mail input and output portion deals with the actual receipt of incoming email messages and the sending of outgoing ones. Incoming email messages are first received

by the mail input portion. Messages are moved from here to the perception module. Once an outgoing message is fully composed, it can be moved to the mail output portion. Mail output hands off the outgoing message to the operating system.

Results and Limitations

VMattie is able to accurately perform her duties. She was tested over a period of four weeks, with tests designed to simulate real world settings. During testing, she received 55 messages comprised of 10 message types. The majority of messages received fell into the categories of add to mailing list, seminar conclusion, seminar initiation, and a speaker's presentation of a topic (speaker-topic). She received 5 messages which were irrelevant to her domain.

VMattie was able to correctly fill all of the perception registers for 96.4% of the messages she received. She chose the date of seminar and title of talk incorrectly for only two speaker-topic messages as two words were collapsed together without a space in the incoming messages. Even with this misperception, she correctly composed acknowledgment messages and sent them to the senders of each received message.

The behavior network used this perceived information to generate seminar announcements. VMattie was 100% accurate in generating and sending out the seminar announcements. This included correctly recovering missing information from her tracking memory for default values with full accuracy. VMattie was able to correctly change her mailing list upon receipt of add to mailing list and remove from mailing list messages.

VMattie sent 7 reminder messages to seminar organizers on time during this testing. She received 5 replies to her reminders before the seminar announcement distribution date. She correctly inserted "TBA" for the remaining 2 instances in the seminar announcements.

To effectively coordinate departmental seminars, VMattie's job description needs expansion. The agent cannot accurately handle an incoming message containing two message types. She cannot deal with one time events such as a colloquium. VMattie does not perform any learning. Learning is very useful in several areas, such as learning new message types and new behaviors.

Global Workspace Theory

Introduction

This section describes Bernard Baars' global workspace theory. Global workspace theory is a cognitive science model of human consciousness. It also discusses other cognitive processes such as action selection and learning. This chapter focuses on consciousness. It first gives the operational definition of consciousness and describes its functions. The components of global workspace theory are discussed, followed by a presentation on how the theory works. Finally, a detailed example is used to trace through the theory.

Global workspace theory is a significant step towards a concrete description of human consciousness. Its computer science implications will be examined throughout the remainder of this work.

Consciousness

Baars states that throughout human history, consciousness has been extremely difficult to define. He states, "Even today, more and more nonsense is spoken of consciousness, probably, than of any other aspect of human functioning" (Baars, 1988, p. 4). According to Baars, consciousness, while it can be inferred from reliable evidence, is a theoretical construct. Consciousness, therefore, is defined in terms of what makes up the human conscious experience. A two-part definition is necessary to define what it means to be conscious of an event. The first part states that a person is defined to be conscious of an event if the person states that they were conscious of it immediately after the event occurs. Events may be conscious for only hundreds of milliseconds. The second portion says that the experiencer's report must be able to be independently verified.

The reader may now be wondering how this psychological definition ties into a software agent implementation. As seen in later chapters, it contains two main implications. First, "consciousness" will contain elements which relate to events. Second, elements may be "conscious" in the agent for an extremely short period of time.

Functions of Consciousness

Baars states that consciousness has nine major functions. The reader will encounter these functions again in chapter 7 when it is shown how ConAg's implementation of "consciousness" fulfills these functions. The first function is Definition and Context-setting. An example of this occurs when one focuses on a distant tree in a forest. While multiple visual stimuli are present, a coherent image is able to be retrieved. The

second function is Adaptation and Learning. For example, extremely difficult material is often pondered for a great deal of time when attempting to learn it. The third function is Editing, Flagging, and Debugging. This is evident in biofeedback training, where persons use flagging in order to gain conscious voluntary control over usually unconscious systems. Fourth is the Recruiting and Control function. An example of this function's use occurs when attempting to answer a question. While one is conscious of a question, the candidate answers to that question are recruited unconsciously and brought to consciousness. The fifth function is Prioritizing and Access-control. This occurs when learning a foreign language. One may wish to prioritize words which are difficult to pronounce. The Decision-making or Executive function is useful in controlling thought and action. A decision-making question is "Should I go to the mall or to the park?" The Analogy-forming function occurs when people make analogies to compare a novel experience to known ones. An example of this is "Hate is the wrong road to travel." The eighth function is the Metacognitive or Self-monitoring function. One example of this is humans' ability to pinpoint and express their current feelings. The final function is Autoprogramming and Self-maintenance. This can be seen in the desire to exercise and eat properly in order to keep the body healthy.

The Components of Global Workspace Theory

Global workspace theory is an attempt to integrate the large amount of information about consciousness into one model. To comprehend the theory, its underlying components must first be understood.

Processes and Coalitions

Global workspace theory states as a premise that the nervous system is composed of processes. Each process is autonomous and has a narrow focus. It is very efficient, works at high speeds, and makes very few errors. Each process can act in parallel to others. This allows for the creation of a high capacity system such as the central nervous system. In "conscious" software agents, as in VMattie, processes are called codelets, taken from the Copycat architecture (Hofstadter & Mitchell, 1994) which inspires much of these agents' perception apparatuses (Bogner, Ramamurthy, & Franklin, in press 1999).

A coalition is a set of processes which work together to perform a specific task. For example, it takes numerous processes for breathing to take place. Coalitions are recursive in nature. For example, a coalition may be composed of several coalitions. A process may be a member of more than one coalition. Coalitions normally perform routine actions. However, coalitions may also perform duties relating to ambiguous, conflicting, or novel events for the system. These coalitions have the potential for entering consciousness. Baars states that coalitions may have an activation level. When performing routine actions, a coalition's activation level is low. Coalitions performing more uncommon tasks have a higher activation level. A higher activation level gives a coalition a greater chance to enter consciousness. Baars is careful to point out that a high activation level may be a necessary but possibly not a sufficient condition for a coalition to enter consciousness. As seen later in ConAg, this holds true.

Contexts

Contexts are relatively stable coalitions of processes which affect consciousness. Contexts are normally unconscious. Therefore, contexts are not usually experienced directly. Contexts interact rapidly with what is occurring in consciousness. One example of a context is found in a collegiate classroom. Both students and faculty know that a certain classroom behavior is expected in an engineering class, while not always being conscious of it. This example illustrates cultural context, a main context according to Baars. Perceptual, conceptual, and goal contexts are the three other main contexts. In "conscious" software agents, the current goal context corresponds to the agents' current behavior. Contexts provide an underlying level of stability. In the above classroom example, if the professor began singing during class, the students would quickly become conscious of the fact that the professor was exhibiting abnormal behavior. The professor's behavior went against the prevailing cultural context. In this sense, contexts allow for novel events to become conscious. New contexts can be learned, allowing for reality to be perceived in a further enhanced way.

The Global Workspace

The crux of the theory's architecture is the global workspace. The global workspace is intended to implement consciousness. The global workspace is a working memory which gives a central location for one coalition to interact with the system's other processes. Therefore, the global workspace can be considered analogous to a classroom blackboard. The next section discuses how this information exchange works and what information

this exchange provides. As seen later, ConAg's attention package implements the global workspace.

How Global Workspace Theory Works

Global workspace theory can be compared to a theater's stage. In a theater, a spotlight is often used to focus the audience's attention. The spotlight roams around the stage, and there is normally only one person in the spotlight at a time. A person is usually in the spotlight when they are performing new actions. More is usually occurring in the production than what is currently in the spotlight. This global workspace theory theater is an interactive one. When members of the audience see something in the spotlight they can relate to, they begin acting. These members may join the actors in the spotlight, or they may begin acting outside the audience's main focus.

In global workspace theory, consciousness is the spotlight which roams over the active unconscious processes. This spotlight shines on coalitions attempting to solve difficult problems. It shines on coalitions performing tasks relating to ambiguous and conflicting items. It also focuses on coalitions dealing with novel situations. Many unconscious coalitions and processes are working even while a particular coalition is in consciousness. In ConAg, the spotlight controller serves as this attentional focus mechanism.

The audience is the unconscious processes not on the stage. When coalitions enter consciousness, they broadcast information to all processes. Some audience members which understand this information become active and perform their specific

functions. These processes, therefore, potentially contribute to the work of the conscious coalition.

The spotlight can shine on only one coalition at a time. Baars states that humans must think of two alternatives one after the other; they cannot be addressed at the same time. Due to its serial nature, consciousness is a much smaller capacity system compared to the large capacity system created by the numerous unconscious processes acting in parallel.

Overall Steps In Global Workspace Theory

Figure 3.2 shows an illustration of global workspace theory (Baars, 1988). It is important for the computer scientist to remember that global workspace theory is a high level model. To understand the theory, it is helpful to think of processes going through five stages. These are:

1. Unconscious processes, each working towards achieving a portion of an overall goal, form a coalition. Unconscious processes working on ambiguous, conflicting, or novel information have a greater chance of entering consciousness.
2. Coalitions compete for access into the global workspace.
3. The coalition which enters consciousness broadcasts information to all unconscious processes. This broadcast allows the conscious coalition to recruit other processes which can contribute to the conscious coalition's tasks.

4. All unconscious processes will receive the broadcast message. However, only certain ones will be able to understand its contents.
5. The processes which understand the message and which need to take action do so.

This five step process is implemented in ConAg.

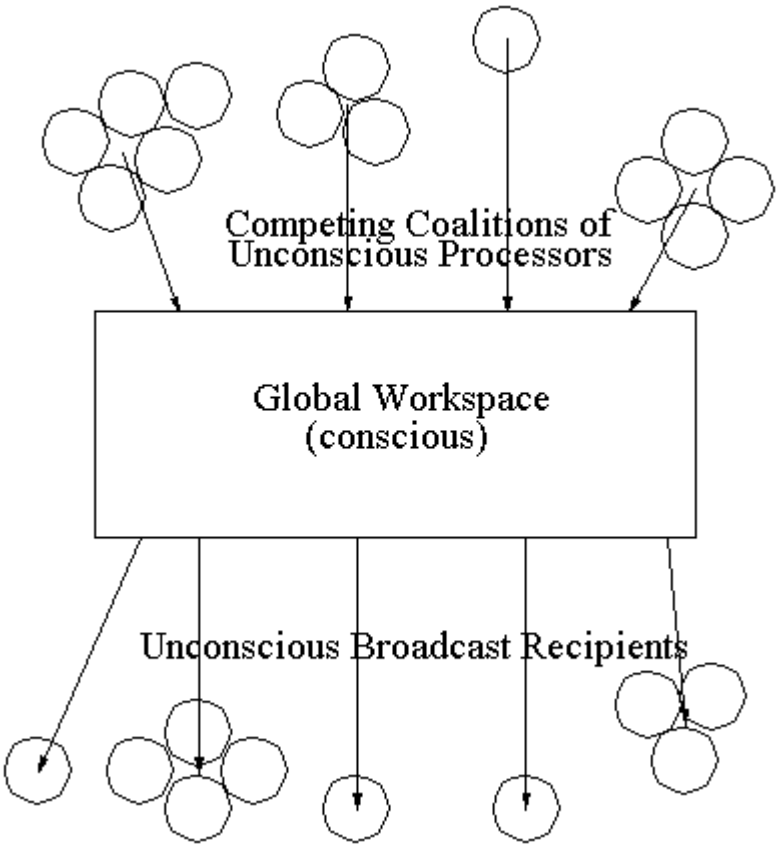

Figure 3.2: Global Workspace Theory
(Bogner, 1998)

An Example To Illustrate Global Workspace Theory

Tracing through an example helps in understanding global workspace theory. Patricia is a fourth year piano major who began playing the instrument her first collegiate year. At this point Patricia is well accomplished, but does not yet have a maestro's skill. She is currently performing the fifth piece of her memorized ten piece recital, and has so far faced no problems in her performance.

By this point in the recital, Patricia is relatively relaxed. She is playing habitually. While she is conscious of what she is hearing, she basically is letting her hands move themselves through this piece's long runs. Suddenly, however, in the quiet auditorium, Patricia hears the loud, sharp bark of a dog. This causes her to slightly lose concentration and hit two wrong notes. Patricia, with no recollection of where her fingers are to go next, jumps to a later portion of the piece where she continues to play. Only the keenest audience members know there was something in her playing out of place.

This example illustrates numerous aspects of global workspace theory:

- With her long hours of practicing, Patricia's ability to play her current piece has become habitual. Unconscious processes are controlling her finger and pedal movement. Patricia is mainly focusing on each note she hears. In this case, the components involved in listening to the piano notes form one or more conscious coalitions.
- The dog's loud bark introduces a new, unexpected, and potentially dangerous event. A coalition of processes involved in determining the presence of

danger forms. This coalition may have a process to determine a sound's volume, another to determine a sound's pitch, another to determine the direction a sound is coming from, another which detects novel sounds, etc. This coalition, having a very high activation level relative to the piano notes, gains access to consciousness. It then broadcasts information. This information is received by the unconscious processes. Visual and auditory processes which can help determine if there is any immediate danger respond.

- Patricia, however, rapidly realizes that there is no immediate risk and she is in front of an audience. In this case, a new coalition arises into consciousness. This one's goal is to get her back on track in her piece. To do this, this coalition broadcasts asking for help. Processes in auditory, memory, and visual retrieval respond. Those processes containing the solution gain a high enough activation to reach consciousness.
- While in the short term thoughts of the dog and the crowd's response come in and out of consciousness due to their remaining high activation level, as their activation falls Patricia relaxes into her practiced habitual playing mode.

Conclusions

Humans utilize consciousness extensively. Global workspace theory provides a high-level model describing human consciousness. It provides a means of cooperation for coalitions. This fosters conflict resolution, learning, and perceptual clarification.

VMattie's modules provide an agent implemented by combining and extending several recent artificial intelligent mechanisms. The agent implements several portions of Baars' global workspace theory. This correspondence will be discussed in chapter 7. The next chapter includes a presentation of CMattie, the successor to VMattie which is designed under the framework of global workspace theory.

Chapter 4
"Conscious" Software Agents

Introduction

For the past several years, the "conscious" software research group has been developing "conscious" software agents. "Conscious" software agents are cognitive agents (see chapter 2) that integrate numerous artificial intelligence mechanisms to implement global workspace theory (see chapter 3). "Conscious" software agents are designed to be "smarter" software. These agents can range in functionality, from academic seminar organizers (Bogner, Ramamurthy, and Franklin, in press 1999), to navy detailers responsible for naval personnel placement (Franklin, Kelemen, & McCauley, 1998), to personal travel agents. From the onset, and continually more so as development progresses, it is clear that these agents are extremely complex and time-consuming to develop and implement. This chapter first describes "conscious" software agents' architectural style and a general architecture for them (Bogner, Maletic, & Franklin, in press 1999). "Conscious" Mattie (CMattie), the first "conscious" software agent, is described in depth (Franklin, submitted). IDA, the "conscious" software research group's proof of concept project, is then presented (Franklin, Kelemen, & McCauley, 1998). Left for the next chapter is a description of these agents' "consciousness" mechanism.

"Conscious" Software Agent's Architectural Style

Action Selection Paradigm of Mind

As described in chapter 2, research has shown that design reuse has many advantages (Johnson, 1997). "Conscious" software agents are designed following the action selection paradigm of mind, a design philosophy providing principles for cognitive agent architectures (Franklin, 1995; Franklin, 1997). The action selection paradigm states that minds are autonomous agents' control structures. Minds' task is to produce the next action. Minds should be viewed as continuous instead of boolean. Sensations, such as perception, are operated on by minds to create information for their own use. A multitude of disparate mechanisms enable minds, and there is little communication between them. Minds and action selection are limited to autonomous agents. Agents are situated in environments, and agents' actions are selected in the service of drives. Prior information (memories) are re-created to help produce actions. Cognitive functions such as categorizing, inferencing, planning, recalling, recognizing, and sensing all serve to help determine what to do next.

Global Workspace Theory Recap

"Conscious" software agents also fall under Baars' global workspace theory (see chapter 3). Particularly important from the theory is that the system is comprised of numerous small processes, known as codelets (Hofstadter & Mitchell, 1994) in "conscious" software agents. Some of these codelets form coalitions and compete for consciousness.

When a coalition reaches consciousness, its information is broadcast to the entire system. Becoming conscious is sufficient for learning. Processes act under the auspices of contexts: conceptual contexts, cultural contexts, goal contexts, and perceptual contexts. Each context is a coalition of processes.

Pandemonium Theory

Also key to these agents' design is Jackson's (1987) pandemonium theory, which extends Selfridge's (1959) original work. Pandemonium theory's components interact like people in a sports arena. Both the fans and players are known as demons. Demons can cause external actions, they can act on other internal demons, and they are involved in perception. The vast majority of demons are the audience in the stands. There are a small number of demons on the playing field. These demons are attempting to excite the fans. Audience members respond in varying degrees to these attempts to excite them, with the more excited fans yelling louder. The loudest fan goes down on the playing field and joins the players, perhaps causing one of the players to return to the stands. The louder fans are those who are most closely linked to the players. There are initial links in the system. Links are created and strengthened by the amount of time demons spend together on the playing field and by the system's overall motivational level at the time.

The Architectural Style

As described in chapter 2, architectural styles provide a collection of building-block design elements that can be applied over a broad range of problems. A common example of an architectural style is a blackboard system. Figure 4.1 illustrates "conscious"

software agents' high level architectural style, comprised of many cognitive features from the action selection paradigm.

Beginning at the bottom, figure 4.1 depicts "conscious" software agents' numerous cognitive mechanisms, such as behaviors and perception. These mechanisms are in reality driven by the small single-task codelets corresponding to global workspace theory's processes and pandemonium theory's demons. The attention manager gathers the necessary information from the codelets and chooses the appropriate ones for "consciousness." It updates the short-term memory blackboard with the "conscious" codelets' information and sends these codelets' information to all of the cognitive modules.

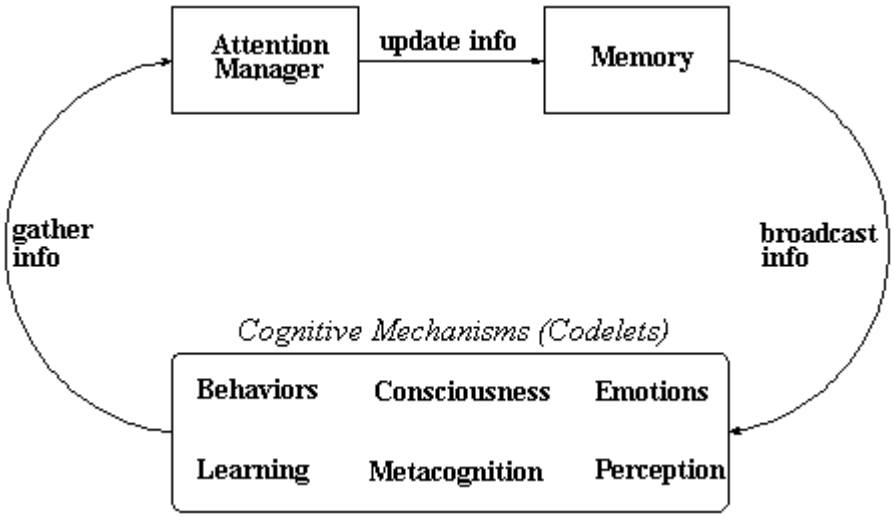

Figure 4.1: Architectural Style For "Conscious" Software Agents
(Bogner, Maletic, & Franklin, In Press 1999)

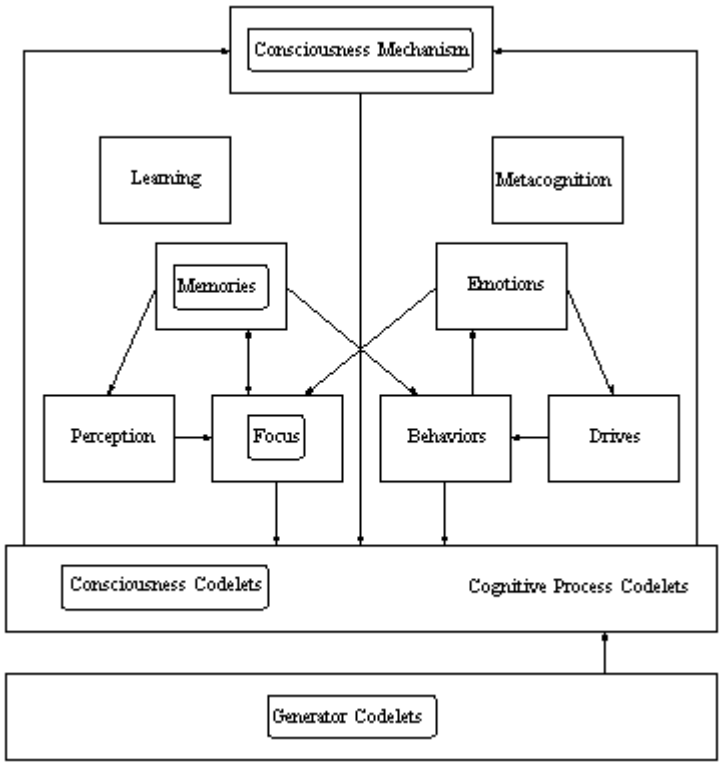

Figure 4.2: An Architecture For "Conscious" Software Agents
(Bogner, Maletic, & Franklin, In Press 1999)

The General Architecture

High Level Cognitive Modules

By extrapolating this architectural style at a lower level, figure 4.2 illustrates the particular architecture that is used in "conscious" software agents. Described in chapter 6, this architecture forms the basis for the ConAg framework. Items specifically dealing with "conscious" software agents' "consciousness" are circled.

In this architecture, codelets comprise emotions (McCauley & Franklin, 1998), behaviors (Maes, 1990; Song, 1998), metacognition (Zhang & Franklin, 1998),

perception (Ramamurthy, Bogner, & Franklin, 1998; Zhang, Franklin, Olde, Wan, & Graesser, 1998), and portions of "consciousness" (Bogner, Ramamurthy, & Franklin, 1999). Emotion codelets are dispersed throughout "conscious" software agents, looking for situations which will influence the systems' overall emotional state. Systems' emotional states are a composite of several emotions, such as happiness, sadness, anger, and fear. Behaviors serve to perform the systems' major actions. For example, for agents which communicate via email, a behavior might be to compose a reply to an email. Drives are built into "conscious" software agents, and they operate in parallel. Drives activate behaviors, and behaviors work to fulfill them. Perception varies depending on the domain; it can range from receiving voice in tutoring systems to natural language email messages in department seminar organizers.

The focus is the location where perceptual information is created for the agents' own use. Here this perceptual information is associated with emotions and memories. "Conscious" software agents contain numerous memories, including associative (Kanerva, 1988), episodic (Kolodner, 1993), short-term memory associated with what has become "conscious," and numerous working memories. Metacognition keeps track of agents' internal conditions. If necessary, it can influence the behaviors, perception, learning, and where the spotlight of "consciousness" shines. For example, metacognition can make the agent more goal-oriented or opportunistic, and cause voluntary attention by influencing the chances that a coalition of codelets will make it to "consciousness." Learning takes many forms in these agents such as the ability to learn new behaviors. The primary responsibility of "consciousness" codelets are to bring novel or conflicting

information to “consciousness” (Bogner, Ramamurthy, & Franklin, 1999). This includes new perceptual information. It also includes conflicts between what is perceived and what is remembered, and conflicts in the potential communication output of the agents.

Low Level Codelets

All codelets have activation levels corresponding to how important they perceive their action to currently be. When appropriate, these activation levels are also directly associated with the higher level concept the codelet serves, such as a behavior currently being executed. Codelets also contain associations with other codelets, corresponding to the links of pandemonium theory’s demons. They also carry information such that, if the codelet were to become “conscious,” this information would be broadcast to the entire system.

In some cases, there must be multiple, concurrent instances of the same kind of codelet based on what is “conscious.” Generator codelets, each corresponding to a specific kind of codelet, are used in these situations. Generator codelets receive the “conscious” broadcast and instantiate copies of themselves with the correct information. Chapter 5 describes codelets in much more detail.

A Brief Overview Of Codelets Reaching “Consciousness”

All codelets which are actively performing their tasks join the playing field, also inspired from pandemonium theory. The playing field is a portion of the “consciousness” mechanism. This mechanism also contains a way to form coalitions of codelets. Specifically, a coalition manager works to group codelets into coalitions based on their

associations to other codelets. A coalition must be selected for "consciousness" from among the formed coalitions. The "consciousness" mechanism also contains a spotlight controller that chooses the next coalition for the spotlight of "consciousness" based on coalitions' activation levels. Once the "conscious" coalition has been selected, this mechanism's broadcast manger sends out the coalition's information. This information is also placed in the module's short-term memory as it is known that approximately seven recently "conscious" items remain in short-term memory. It is also passed to the "consciousness" module's chunking manager. The chunking manager forms chunks out of the different "conscious" coalitions. The chunks are later broadcast as potential items to be learned. All codelets in the system are able to receive all of the "conscious" broadcast.

"Conscious" software agents are extremely domain-specific entities. Following the action selection paradigm, what an agents perceives, its drives and corresponding behaviors, etc. are coupled to its environment (Maturana, 1975; Maturana & Varela, 1980; Varela, Thompson, & Rosch, 1991). One of the few relatively domain-independent portions is these agents' "consciousness." This is ConAg's main focus, and is described throughout the later chapters.

"Conscious" Mattie

"Conscious" software agents' general architecture is more readily understandable through concrete examples. CMattie is the first software agent intended as an implementation of global workspace theory. As such, she is "conscious," and socially situated (Bogner,

Ramamurthy, & Franklin, in press 1999). CMattie is able to interact, learn, and adapt in a social environment comprised of human agents. CMattie "lives" in a real world computing system, a Unix-based system. No claims are made that CMattie is "conscious" in the sense of being sentient. As described in chapter 1, this author has contributed to several portions of CMattie, with the focus being the agent's "consciousness." The other contributors are Stan Franklin as project leader, Art Graesser as testing leader, Lee McCauley for emotion, Aregahegn Negatu for action selection, Uma Ramamurthy for perception, and Zhaohua Zhang for metacognition.

CMattie is designed for a specific, narrow domain. She functions in an academic setting, gathering information from humans regarding seminars and seminar-like events such as colloquia, defenses of theses, etc. Using this information, she composes an announcement of the next week's seminars, and mails this announcement weekly to members of a mailing list that she maintains, again by email interactions with humans.

CMattie's implementation follows "conscious" software agents' general architecture. Her modular architecture, as illustrated in Figure 4.3, carries over and significantly extends several portions of VMattie (see chapter 3). These include behavior networks (Maes, 1990) for action selection, the Copycat architecture (Hofstadter & Mitchell, 1994; Mitchell, 1993) and natural language understanding (Allen, 1995) for email comprehension, and tracking memory. In addition, CMattie contains a sparse distributed memory (Kanerva, 1988) for long-term, associative memory, pandemonium theory (Jackson, 1987) for agent grouping, and case-based memory (Kolodner, 1993) for

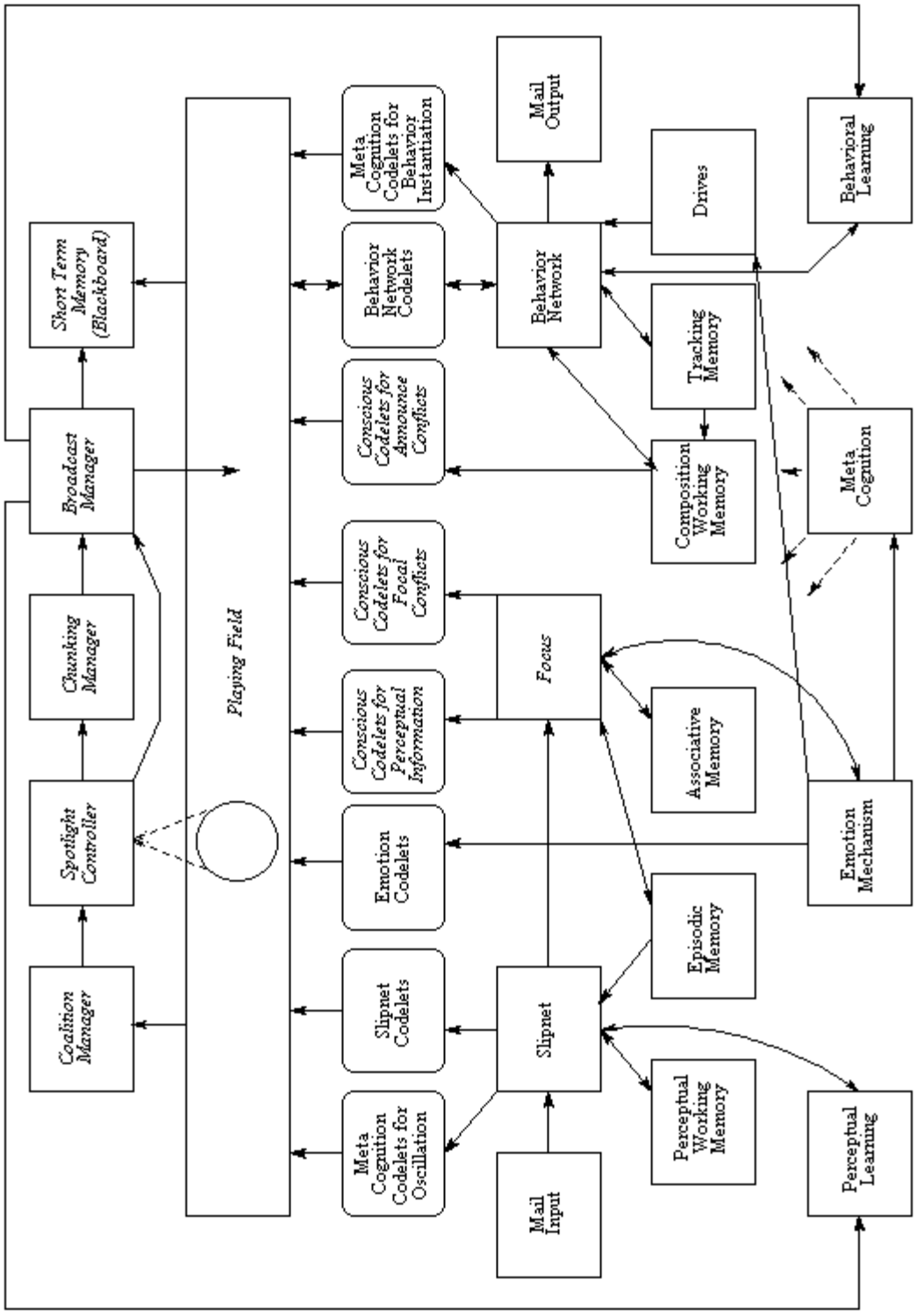

Figure 4.3: CMattie's Architecture

intermediate term, episodic memory. Each of these mechanisms has been significantly extended in order to merge with the others, and to meet the needs of this domain.

As specified in the architectural style, the real work of almost all of CMattie's modules is performed by codelets. Codelets lie underneath CMattie's modules including her behavior network, emotion, metacognition, perception, and portions of "consciousness." CMattie's codelets coalesce into coalitions, become "conscious," broadcast their information to all other codelets in the system, and receive the "conscious" broadcast. CMattie, follows a tenet of the action selection paradigm by being a multi-agent system.

Roles of CMattie's Modules

Mail Input and Output

CMattie's sensory data are, for the most part, the incoming email messages she receives. Mail input and output provides CMattie's interface to her domain. Using this unit, she receives and sends out email messages related to seminars, seminar-like events such as colloquia, and maintenance of the recipient mailing list. Mail input and output can process more than one email message at a time, enabling the perception module to perceive and understand emergency events in CMattie's world. This aids in maintaining her sense of self-preservation as she proactively reacts to her changing resource needs. She immediately reacts to the status of the Unix-host system wherein she "lives."

Perception

Perception for CMattie occurs when she "comprehends" an email message. As in VMattie, incoming email messages, received by the mail input portion, are moved to the perception module. The perception module was inspired by the Copycat architecture (Hofstadter & Mitchell, 1994), and CMattie's perception follows Copycat more closely than VMattie's. (Ramamurthy, Bogner, & Franklin, 1998). When an incoming message is understood, every significant word or phrase has been classified, and the email message has a whole has been categorized into a "message type," such as "add me to your seminar announcement mailing list" and "I'm initiating a new seminar."

Focus

The focus is a portion of the "consciousness" mechanism and is described in great detail in the next chapter. It serves as an interaction point for several of CMattie's modules, specifically associative memory (sparse distributed memory), "consciousness," episodic memory (case-based memory), emotions, and perception. The focus includes four vectors: the perception registers, the output of associative memory, the output of episodic memory, and the input to both these memories. First, the perception module places the components of the understood email message into the perception registers. Next, associative memory is read with the current percept as the address. Also, episodic memory is read with the same address. These reads are designed to gather the information most relevant to what was just perceived. At this point, the contents of the Focus constitute the current percept. After the current percept has become "conscious,"

the behavior network and emotions potentially choose new states based on their receipt of this “conscious” broadcast. These modules then write their current states to the focus. This along with the current percept, is written to both memories.

Associative and Episodic Memories

Sparse distributed memory is a content addressable memory that serves as long-term, associative memory for CMattie (Anwar & Franklin, forthcoming). This memory stores the contents of the perception registers as well as her actions and emotions. Default information, such as room and time can often be recovered, contributing to the understanding of incoming messages. Recovering remembered actions and emotions helps with action selection in the new situation.

Case-based memory is used as CMattie’s episodic memory. In it she stores the sequences of email messages that form episodes. This memory acts as an intermediate term memory, and the information stored there is used to learn domain knowledge. This allows her to relate new events to similar past events. She understands these past events using her built-in domain knowledge. Case-based memory aids her in learning new perceptual concepts through case-based reasoning. Also, it aids in learning the new action sequences necessary when new perceptual concepts learned by CMattie.

“Consciousness” Codelets

A “consciousness” codelet is one whose function is to bring specific information to “consciousness” (Bogner, 1998). They are described in greater depth in subsequent chapters. After the memory reads, perceptual “consciousness” codelets bring information

from the focus to “consciousness.” One such “consciousness” codelet is associated with each of the perception registers and carries the specific piece of perceived information from that register. For example, one codelet carries the speaker’s name, and another carries the seminar’s time.

Specific “consciousness” codelets spring into action when the information in the perception registers is relevant to them. For example, if what is perceived is a request to be removed from the seminar announcement mailing list, the “consciousness” codelet which carries a person’s email address becomes active. It then joins the playing field on its quest for “consciousness.”

In addition, some “consciousness” codelets check for conflicts amongst the relevant items returned from the percept and the memory reads. For example, a conflict occurs if the perceived date, room, and time for the Cognitive Science Seminar are the same as case-based memory’s output of these same features for the Graph Theory Seminar. The “consciousness” codelet recognizing the conflict joins the playing field and raises its activation level. Since it is associated with the other perceptual “consciousness” codelets, the “consciousness” mechanism groups them together to form a coalition. “Consciousness” codelets also check for conflicts in the seminar announcement template being generated by the behavior network.

“Consciousness”

CMattie contains a global workspace based on Baars’ theory of consciousness. Chapter 5 describes this “consciousness” mechanism in depth. The agent’s “consciousness”

mechanism serves to gather the active codelets into coalitions, choose the "conscious" coalition, and broadcast this coalition's information to all codelet's in the system. CMattie's global workspace gives the agent several important performance features. It allows for coalitions of codelets to gain attention and have their information broadcast to all of the agent's other codelets. Recipients of this broadcast become active themselves if enough of the information is understood, and if it is applicable. In this way, the broadcast recipients have the potential to contribute towards solving the problem raised by the "conscious" coalition. This broadcast also allows metacognition a view of the events taking place in the system. Learning also uses the information in "consciousness" to learn to associate codelets as a coalition.

Behavior Network

Like VMattie, CMattie has several drives, some corresponding to her tasks (sending seminar announcements, reminding organizers to send information, and acknowledging messages). As specified in the "conscious" software agent general architecture, these drives activate behaviors that work to fulfill them, are explicitly built into the agent, and operate in parallel.

Behaviors in CMattie correspond to global workspace theory's goal contexts. Each behavior has an activation level affected by drives, other behaviors adjacent to it in the behavior net, internal conditions, and environmental inputs (incoming email messages). Only one behavior can be active at a time. A behavior's activation is spread to those behaviors that can fulfill its unmet preconditions and to behaviors whose

preconditions can be satisfied by this behavior. Thus, each behavior can thus be considered part of a behavior stream. For example, there's a behavior stream that composes the seminar announcement. One behavior in that stream might fill the Cognitive Science Seminar's portion of the seminar announcement.

The behavior network uses tracking memory to store templates used in composing outgoing email messages of different types. It also keeps track of the current seminar announcement mailing list. Tracking memory is external to CMattie, acting as a cognitive prosthesis for the agent. As of now, this memory also stores default information on seminars, such as the day of the week each one occurs. This function may be subsumed by associative memory.

All outgoing messages are composed in the composition workspace. Message composition consists of filling the fields of an outgoing message template. The information used to fill these fields comes from the perception registers and any of associative, case-based, or tracking memories. A current seminar announcement template is always being generated in the composition workspace. As the behavior network receives new perceptual information from "consciousness," the announcement template fields are filled. When a seminar announcement is moved to mail output and mailed, a new announcement template is placed in the composition workspace.

Emotions

CMattie's emotions play two roles (McCauley & Franklin, 1998). First, emotions indirectly affect a behavior stream's activation level by affecting the strength of drives.

Emotions allow CMattie to be pleased about sending out a seminar announcement on time and to be anxious about an impending system shutdown. In these cases, emotion might increase a behavior stream's activation level since it is pleasing for CMattie to complete these streams promptly. Second, metacognition uses emotions to help determine its course of action. For example, if CMattie is "happy," metacognition makes her more reactive. If CMattie is "sad," metacognition makes her more thoughtful.

Metacognition

By monitoring what is in "consciousness," the activation of drives, emotional states, parameters in the behavior network, and the perception module, metacognition keeps track of CMattie's internal conditions (Zhang, Franklin, & Dasgupta, 1998). Using a classifier system (Holland, 1986), metacognition makes inferences about CMattie's state. If necessary, it can influence the behavior network, "consciousness," learning, and perception. For example, metacognition can change the behavior network's activation level threshold to make the agent more goal-oriented or more opportunistic. It can cause voluntary attention by influencing the activation levels of certain coalitions of processes. It keeps the perception module from oscillating indefinitely when deciding on a message type. Metacognition plays the role of an overseer, trying to keep CMattie's action selections on a productive track.

Learning

Learning via several types of mechanisms allows CMattie to become more closely coupled to her environment. She can learn new behaviors, for example, a new step in

preparing for a system shutdown. She might also learn a new strategy for sending out reminders to seminar organizers. Much of her learning uses case-based reasoning. She learns new concepts in her slipnet allowing her to better understand incoming messages. Described in the final chapter, "consciousness" is essential for this form of learning. CMattie creates (learns) new codelets by modifying existing codelets enabling her to perform newly learned behaviors and perceptual techniques. As described in the next chapter, coalitions of codelets are learned via association a la pandemonium theory. This allows the agent's codelets greater ease in communicating and recruiting other codelets to help in performing tasks. Associative learning also occurs in sparse distributed memory as actions, emotions, and events are associated with one another when placed in this memory.

CMattie's Performance

Design and development of CMattie has been ongoing for several years. As of this writing, the framework used for building CMattie's "consciousness" mechanism is "complete." Chapter 6 describes this framework's structure and gives testing results. In addition, all of the different modules' communication protocols have been agreed upon and have been integrated. However, several of these modules are still actively being implemented. Therefore, overall test results for CMattie are not expected until Spring, 2000. Rigorous testing of CMattie is planned.

If VMattie is a guide, CMattie's performance should be very satisfactory. CMattie's implementation of global workspace theory makes her an agent significantly

more complex than VMattie. At the moment, even without the test results, CMattie's role as an implementation of global workspace theory makes her valuable as a conceptual model of mind.

The Intelligent Distribution Agent (IDA)

IDA is an intelligent distribution agent being designed and prototyped for the United States' navy (Franklin, Kelemen, & McCauley, 1998). IDA is designed to perform as one navy detailer. At the end of sailors' tour of duties (approximately 3-6 years), sailors are assigned to new billets (job positions). This job assignment process is known as distribution. These new assignments are made by approximately 200 full-time navy personnel, known as detailers. Currently, employing these detailers costs approximately $20,000,000 annually.

IDA is the "conscious" software research group's proof of concept project, and much of her is still in the design phase. IDA utilizes CMattie's modules and significantly extends them. Like CMattie, IDA must communicate, this time to sailors, in natural language. In addition, she must access and understand the content of several naval databases. In addition, IDA has constraint satisfaction issues in satisfying the Navy's needs. For example, she must make sure that a destroyer has the required number of sonar technicians and all have the appropriate training. She must keep down the costs associated with moving sailors. Also, she must cater to the desires and needs of sailors as much as possible.

IDA's Architecture As An Extension Of CMattie's

Like CMattie's is to VMattie, IDA's architecture is a significant extension of CMattie's. Specifically, enhancements include:

- IDA's codelets in her action selection mechanism access and perceive external, naval databases.
- IDA uses a linear functional as the first line of assessment when assigning a sailor to a billet. This linear functional encodes the common issues a detailer considers. It includes concerns such as: are there women's quarters on the ship, does the sailor need training for the new position, does the sailor need to change coasts for the new position, is the appropriate health care available for the sailor's family, is the cost of moving within the budget, the sailor's home port preference, etc.
- There are cases where the linear functional will not be able to encode all the potential situations a sailor might face when switching positions. For example, a sailor may desire to move his base location from the United States' east coast to the west coast due to his marrying a Californian. In such cases, IDA will deliberate on the different scenarios possible in the billet assignment.
- IDA utilizes a naval order generation program in her creation of assignments.
- IDA's email communication with sailors is over a much wider range of topics than CMattie's. Therefore, IDA uses scripts to help her generate natural language.

IDA's Natural Language Generation Scripts

IDA's scripts are in the early design stage. However, they are largely inspired by AutoTutor's (Graesser, Franklin, & Wiemer-Hastings, 1998) curriculum scripts (Hacker, Bogner, Yetman, & Klettke, 1998). AutoTutor is an intelligent tutoring system that contains a talking head and currently tutors students in computer literacy. Curriculum scripts are used to present the students with questions and provide them hints and prompts in helping guide them towards the appropriate answers. While IDA does not ask students about computer literacy, she does interact with sailors in a similar manner about billet assignment. The remaining portion of this subsection describes AutoTutor's scripts.

According to Putnam, a curriculum script is "a loosely ordered but well-defined set of skills and concepts students are expected to learn, along with the activities and strategies for teaching this material" (1987, p. 17). AutoTutor's curriculum scripts are based on research indicating that tutors appear to follow a predetermined script, with greater attention given only to those elements of the script the student has missed (Graesser & Person, 1994; Graesser, Person, & Magliano, 1995). In fact, this research indicates that adherence to a tutor-driven script may be nearly absolute. Contained within the curriculum script are predetermined sequences of examples, lessons, questions, problem types, and subtopics that are used to instruct a discrete topic area. Within the curriculum script macrostructure is the agenda, or microstructure, that is to be followed during a tutoring session. The microstructure is the set of desired goals for a particular

lesson. In addition, the microstructure contains policies and microplans that may be used when difficulties or misconceptions arise (McArthur, Stasz, & Zmuidzinas, 1990). These policies and microplans can serve as procedures that determine when a specific line of questioning is to be terminated, what new subtopic is to be presented, and the number of examples to be presented.

AutoTutor's curriculum script's are arranged hierarchically. At the broadest level is the knowledge domain, which is a global body of knowledge that the tutoring addresses. AutoTutor's knowledge domain is currently computer literacy. Within the knowledge domain are topics, natural chunks of knowledge characterized by common themes. At the next level are subtopics, which are subchunks of knowledge characterized by more discrete themes within each topic. Each subtopic is a structured database that is further divided into smaller levels, or fields, each focused on a specific component of the tutor-student dialogue.

Each subtopic field contains a list of one or more English words, sentences, or paragraphs, most in conversational form. The fields include the focal question, which is the main question being asked in the subtopic. The ideal answer is the desired response to the posed focal question. The subtopic also includes lists of good answers containing relevant information, lists of different bad answers and misconceptions, lists of hints to help the student, lists of prompts to try and get the student to divulge more information, a succinct summary of ideal answer, a list of anticipated student questions and answers to these questions, and lists of good and bad keywords which help in the assessment of how the student is doing.

Conclusions

"Conscious" software agents are unique largely because they implement a cognitive theory of consciousness. In addition, they integrate and extend numerous mechanisms from the "new ai." This chapter described "conscious" software agents' architectural style, general architecture, and two example agents. Not discussed, however, is how these agents' "consciousness" actually works. This is chapter 5's topic.

Chapter 5
Realizing "Consciousness"

Introduction

"Conscious" software agents are unique in part because they implement global workspace theory. This chapter discusses these agents' "consciousness" at the procedural level. A description of the "consciousness" mechanisms' locations and use in ConAg is left for chapter 6. "Consciousness" in these agents includes the base codelet, broadcast manager, chunking manager, coalition manager, "consciousness" codelets for conflict detection, "consciousness" codelets for perceptual information, focus, playing field, short-term memory, and spotlight controller. Each of these components has been originally designed and implemented by this author in consultation with the "conscious" software research group. Throughout this chapter, it is helpful to revisit figure 4.3.

Base Codelet

All codelets in "conscious" software agents utilize the base codelet through inheritance (Eckel, 1998). The base codelet is a class (Eckel, 1998) containing the variables and methods common to all codelets in "conscious" software agents. Action selection codelets, "consciousness" codelets, emotion codelets, metacognition codelets, and perceptual codelets all extend the base codelet for their needs. This structure contains all of the information necessary for the "consciousness" mechanism to access and

manipulate the codelets with which it is working. All codelets extend the base codelet as "consciousness" provides the backbone for communication amongst these codelets. Below is a description of the base codelet's properties, and, therefore, the properties common to all codelets in "conscious" software agents.

Thread. All codelets are threads (Eckel, 1998). This means that each codelet runs in at least simulated parallelism to one another on a single processor machine.

Broadcast Listener. All codelets, when they become instantiated, are broadcast listeners. This means that all "alive" codelets receive the broadcast from "consciousness." This follows global workspace theory's premise that all processes receive the broadcast. Just as in Baars' theory, while codelets receive the broadcast, they do not necessarily act upon it. They do so only if they understand it and it is applicable.

Serializable. All codelets are serializable (Eckel, 1998). Serialization is a Java construct. It provides for objects to be turned into a sequence of bytes. Later, these bytes can be restored to the original object. Serialization is commonly used in sending objects over a network. "Conscious" software agents use serialization for self-preservation. Specifically, codelets' states are saved in the event of a system shutdown. Upon system startup, they can be returned to their last running state.

Name. Each codelet has a name providing for description throughout the system.

Unique ID. For help in keeping track of the codelets in the system, they each have a unique id. By default, this id is a randomly selected number. This unique id is quite important for generator codelets. Generator codelets are like all codelets in that they listen for the "conscious" broadcast. However, generator codelets do not directly

spring into action upon receiving a relevant broadcast. Instead, they instantiate copies of themselves with the appropriate information. This unique id helps in identifying these instantiated codelets.

Activation Level. Each codelet has an activation level. The activation level is a fuzzy variable; codelets can set it to none, low, medium low, medium, medium high, high, and max. They also can call methods to increase or decrease their activation level to the next closest value. All activation values fall between zero and one. A codelet's activation level is tied to how important it perceives its current task to be.

Associations. Codelets have associations to one another, corresponding to pandemonium theory's links (see chapter 4). It is these associations which determine if codelets are placed together in coalitions. Each association consists of a handle (Eckel, 1998), or pointer in C/Pascal terminology (Kernighan & Ritchie, 1988), to the associated codelet along with an association strength, kept between zero and one. Physical memory is the only limitation on the number of associations each codelet may have.

Codelets also have an association decay rate, and all contain a standard method for decaying associations. Codelets independently choose how frequently to decay their associations. Both the coalition manager and spotlight controller form new associations and strengthen existing ones.

Consciousness Indicator. The spotlight controller sets this flag to signify that a codelet has become "conscious." Currently, the flag's two options are that the codelet has not yet become "conscious" and that the codelet can leave the playing field. This flag is used by the "consciousness" codelets to know when their job is complete (they have made

it to "consciousness"). In addition, it is useful in tracking codelets' states. In the future, for mechanisms such as deliberation, a third option, that the codelet must stay on the playing field, may be used. This would be set by the spotlight controller to force a codelet to remain on the playing field, even if it perceived its task to be complete.

Broadcast Information. The broadcast from "consciousness" takes the form of a hashtable (Aho & Ullman, 1992). Codelets receiving the broadcast search for relevant keys. If one is found, they view the corresponding message. With this system, all codelets carry broadcast information in the event they reach "consciousness." This information includes their key and message. Individual codelets determine their own key and message. An interesting future study might be an analysis of the "language" created by "conscious" software agents' keys and messages.

Access To The Playing Field. Codelets contain the means to join and leave the playing field. The playing field gives codelets potential access to "consciousness." Codelets join the playing field just before they perform their actions, and they leave once their actions are complete.

The Focus

As mentioned in chapter 4, the focus is a main interaction point for perception, emotions, "consciousness," associative memory, and episodic memory. The specific properties of the focus are now described.

Perception Registers. The perception registers are set by the perception module. These registers are a fixed size array of strings. Fixing the size of these arrays is

necessary largely due to the interactions with the associative and episodic memories. IDA, while they are not yet fully determined, has significantly more perception registers than CMattie. CMattie's perception registers, listed below, are the ones currently contained in ConAg.

1. Seminar Name
2. Seminar Organizer
3. Speaker's Name
4. Speaker's Affiliation
5. Talk Title
6. Day
7. Date
8. Time
9. Location
10. Message Type
11. Email address
12. Emergency Indication
13. Unrecognized word one
14. Unrecognized word two
15. Unrecognized word three
16. Unrecognized word four
17. Unrecognized word five

"Conscious" software agent's perception is largely driven by perceived events. In IDA, these events include leaving a current billet, training, arriving at a new billet, etc. For CMattie, the events are the different message types. These are:

1. Add person to seminar announcement mailing list
2. Answer to CMattie
3. Change of seminar's location
4. Change of seminar's time
5. Change of seminar's topic

6. CMattie copy message to herself
7. Remove person from seminar announcement mailing list
8. Incoming message contains multiple message types
9. Negative message in response to CMattie's action
10. Initiate a new seminar
11. No seminar at the specific time
12. CMattie sees message as nonsense
13. Question message for CMattie
14. Seminar has a speaker discussing a certain topic
15. Seminar is permanently ending
16. Message from the system administrator

Associative Memory Output Registers. After the perception registers are set, a read from associative memory occurs. The result of this read is placed in the associative memory output registers. These registers are the same set as the perception registers. In addition, they contain the remembered emotion and behavior associated with the remembered perception. In CMattie, these registers, not including the perception registers, are:

17. Happy emotion
18. Sad emotion
19. Anger emotion
20. Fear emotion
21. Behavior's name
22. Behavior's activation

Episodic Memory Output Registers. In addition to a read from associative memory, a read from episodic memory occurs. The episodic memory output registers are in the same format as associative memory's.

Memories' Input Registers. Once the perceptual information has reached "consciousness" and has been broadcast, a new emotional state and behavior are potentially chosen. After this assessment, the current emotional state and behavior are written to the focus into the memories' input registers. The memories' input registers have the same format as the associative and episodic output registers. Once both the emotion module and behavior network have written to the focus, the focus writes the memories' input registers to both the associative and episodic memories.

Cannot_Fill_Registers_Counter and Registers_Can_Be_Filled_Indicator. A significant design decision made for CMattie and IDA is that all of the perception registers must become "conscious" before the perception registers can be filled again. The cannot_fill_registers_counter is initially set to two. When the perception registers are newly filled, "consciousness" codelets check to see if there is information for them to carry to "consciousness." If so, each codelet increments the cannot_fill registers_counter and picks up this information. Once they've been "conscious," these codelets decrement this counter. In addition, the counter is decremented twice more: once with each memory write.

At this point, the cannot_fill_registers_counter is reset to two, and the registers can_be_filled_indicator is set to true. The perception module uses this boolean indicator to determine if the next perception can be placed into the registers. This system ensures

that all “consciousness” codelets and memory accesses occur before the next perceived email message is placed in the focus.

Perception Register, Associative Memory Output, and Episodic Memory Output Broadcaster. Certain “consciousness” codelets and emotion codelets spring into action immediately after the perception registers are filled. Upon system startup, these codelets register to receive the focus’ perception register broadcast. When the perception registers are newly filled, the focus broadcasts to all desired recipients, letting them know the perception registers have been filled. This same process is used for the “consciousness” and emotion codelets desiring to know when the associative memory read has been completed, and those wanting episodic memory read’s completion.

Associative and Episodic Convergence Indicators. At times, “consciousness” codelets and emotion codelets need to know whether or not associative and episodic memory have converged. The focus stores this information.

Playing Field

The playing field, inspired from pandemonium theory, provides codelets access to “consciousness.” Each codelet joins the playing field immediately before it begins to perform its task. Each codelet currently leaves the playing field immediately after it has completed its task. The playing field stores handles to each codelet on the playing field.

Coalition Manager

The coalition manager traverses the codelets on the playing field, forming them into coalitions based on their associations. The coalition manager is a thread. It sleeps for a short period of time as it must keep its coalition table current. Specifically, the coalition manager performs the following loop:

1. It sleeps for a short period of time
2. It creates a new, temporary, coalition table
3. It traverses the playing field's codelets. Each codelet encountered is initially placed in its own coalition. This provides for singleton coalitions.
4. It looks at each of the coalition's associations as a whole. In other words, what is being viewed are the composite associations from each of the coalition's codelets. For each of these associations, the coalition manager searches for the associated codelet on the playing field that is not already in the coalition. If the searched for codelet is found, it is added to the coalition.
5. It overwrites the coalition table with its new temporary coalition table.
6. It updates all of playing field codelets' associations to one another. Specifically, if a codelet is on the playing field and is not associated to another codelet on the playing field, an association is made. If associations exist, they are strengthened by a small amount, assuming they are not at the maximum value. Updating codelets' associations on the playing field is inspired by

pandemonium theory and, potentially, allows the system's behavior to evolve over time.

Spotlight Controller

The spotlight controller determines the contents of "consciousness." Specifically, as specified by global workspace theory, it is the spotlight of "consciousness" shining down on a coalition of codelets. The spotlight controller is its own thread, looping and performing the following steps:

1. It sleeps.
2. It calculates the average activation level of the codelets in each of the coalition manager's coalitions. Average activation is used as opposed to total activation to ensure that larger coalitions do not have an advantage for "consciousness."
3. For the "coalition" with the highest average activation level, it sees if this coalition is above the threshold for entering "consciousness."
    1. If so, it selects this coalition as the "conscious" coalition.
    2. If not, it drops the threshold for "consciousness" by a percentage (currently ten percent), and starts again. This threshold dropping is inspired by Maes' (1990) behavior networks.
4. It updates the codelets' associations to one another as codelets are now in "consciousness" together. This association level increase is significantly greater than the association increase gained when codelets are on the playing field together. In the rare event some of "conscious" codelets have not been

yet been associated by the coalition manager (as they're also still on the playing field together), the spotlight controller sets up new associations as well.

5. It passes the "conscious" coalition to the broadcast manager.
6. It passes the "conscious" coalition to the chunking manager.
7. It sets each codelet's consciousFlag to indicate that they have previously been in "consciousness." Even with this flag set, there is no limit to the number of times a codelet can be chosen for "consciousness," as long as it is on the playing field.
8. It resets "conscious" coalition to indicate the lack of a "conscious" coalition.

Broadcast Manager

The broadcast manager disseminates the "conscious" information to all those listening. Specifically:

1. It traverses the "conscious" coalition, taking from each codelet its information to be broadcast. All codelets' information is placed in a single hashtable.
2. It inserts a time-stamp into the hashtable.
3. It sends out the broadcast to all listeners.
4. It passes the broadcast hashtable to short-term memory.

Described below, the chunking manager also sends out information via the "conscious" broadcast, and it uses the broadcast manager to do so. In these cases, step 1 is bypassed.

Chunks and the Chunking Manager

Currently, a chunk consists of three items:

Chunks' Current Strength. This value determines how close a potential chunk is to becoming a chunk.

Is Chunk Indicator. This flag is false if the chunk is only a potential chunk and true otherwise.

Chunk's Codelet Names. A chunk contains the names of all codelets in the chunk, sorted and delimited by colons.

The chunking manager receives the latest "conscious" coalition from the spotlight controller. The chunking manager's role is determine new chunks out of the potential chunks. Each "conscious" coalition is a potential chunk. Chunks are used by the behavioral and perceptual learning mechanisms to help determine new concepts to learn. They are inspired by pandemonium theory's associations which develop over time into concept demons.

The chunking manager:

1. Gathers the codelets' names from "conscious" coalition, sorts these names, delimits them, and places them into a single string.
2. Checks if this string is already in the list of chunks.
    1. If it is not, the string is added to the list of chunks as a potential chunk.

2. If it is, the potential chunk's strength is increased. If this potential chunk is now above threshold, it is considered a chunk. In this case, it is passed to the broadcast manager and sent out.
3. Decays all of the chunks and potential chunks. Chunks do not regress to potential chunks, even though their activation level might deem it one. Therefore, chunks are broadcast only once. The entire chunk list can be retrieved from the chunking manager by the learning mechanisms.

Short-Term Memory

Short-term memory serves to hold the last several coalitions broadcast from "consciousness;" CMattie's is currently a maximum of seven. It provides a means for the systems' modules, including codelets, to look at the recent contents of "consciousness." This is particularly important for codelets which sleep for a relatively long period of time. In these cases, the codelets still do receive all broadcasts as specified by Baars' theory. However, they potentially were asleep as received broadcasts overwrote each other. When the codelets awaken, they have missed some broadcasts. Codelets can use the broadcast's time-stamp to determine which ones in short-term memory are relevant. In addition, short-term memory provides an additional data-analysis source for the learning mechanisms.

"Consciousness" Codelets

"Consciousness" codelets are base codelets with additional functionality. Following Baars' theory on consciousness' role, "consciousness" codelets' primary role are to bring novel and conflicting information to "consciousness." In CMattie and so far in IDA, these codelets work in two areas: the focus and the behavior network's composition working memory. "Consciousness" codelets bring to "consciousness" both novel and conflicting information from the focus and bring conflicting information from the compositional working memory.

From the Focus

When the focus' perception registers are filled, the focus notifies the "consciousness" codelets listening for this. For each of the perception registers excluding the message type register, there is one "consciousness" codelet per register. These codelets are daemon codelets, listening for the perception registers to be filled and acting if their specific register is in fact filled. Initially, these codelets have no associations with other codelets; associations are simply allowed to develop over time.

Watching the message type register is one "consciousness" codelet for each message type. When the appropriate message type is encountered, the corresponding codelet springs into action. It picks up the message type from the perception register in order to bring it to "consciousness." Each message type codelet is initially strongly associated with the perception register's non-message type "consciousness" codelets that carry the information describing the message. In this way, when the message type codelet

is on the playing field, it will be grouped with the relevant "consciousness" codelets such as the one carrying the date, speaker's name, time, etc. assuming they are on the playing field. In CMattie, the message type codelets are initially associated with all of these non-message type "consciousness" codelets as all are potentially relevant. In IDA, this will most likely not be the case because of the greater number of perception registers. CMattie's message type "consciousness" codelets correspond to those which carry IDA's different events such as leaving the current billet.

After the associative memory read is complete, if there is convergence, "consciousness" codelets listening for the read's completion spring into action. Some of these codelets work to bring the remembered register contents to "consciousness." These go into effect if the corresponding perception register is empty. For example, for a message from a seminar organizer about an upcoming speaker and topic for a seminar but not containing the time of the seminar, the remembered time is carried to "consciousness." There are also "consciousness" codelets which listen for episodic memory convergence. Some of these codelets work to carry episodic memory information to "consciousness" if both the corresponding perception register and associative memory register are empty. Like the "consciousness" codelets which bring the remembered associative memory information to "consciousness," these only bring the appropriate remembered information such as the time of the seminar. Items such as a seminar's speaker are not brought to "consciousness" as those change weekly.

"Consciousness" codelets also check for conflicts amongst the perception registers and memory registers if either of the memories have converged. Currently in CMattie,

one conflict is searched for in the focus: an overlap in the room, time, and date of the remembered seminar and the seminar in the perception registers. Conflict "consciousness" codelets work differently than those which simply carry information to "consciousness." The codelet which is actually searching for the conflict is a daemon codelet. In the event a conflict is found, this codelet instantiates codelets associated to it that pick up the information in both the memory and perception registers. This stems from the fact that conflicts tend to be temporary in nature. The conflict codelet and these new codelets carrying the information about the conflict then join the playing field in their quest for "consciousness." After the temporary "consciousness" codelets reach consciousness, they die and are garbage collected.

From the Composition Working Memory

As the seminar announcement is being generated in the composition working memory by the behavior network, "consciousness" codelets searched for conflicts in this template. These conflict detecting codelets function in the same way as those in the focus. In CMattie, two conflicts are detected. One is the same as detected in the focus: two seminars overlapping on the same day at the same time in the same room. In addition, a conflict is detected if the same speaker is speaking in two seminars simultaneously. Unlike CMattie, IDA will most likely check for conflicts in all outgoing messages.

An Example "Consciousness" Codelet

This subsection illustrates a representative example of a "consciousness" codelet named NewSeminarMessageTypeCarrier. This codelet waits until the perception registers are

filled with the message type signifying that a new seminar is being initiated. NewSeminarMessageTypeCarrier inherits the properties of a percept register carrier codelet. These codelets have the following properties.

Codelet. All percept register carrier codelets extend the base codelet class, and, therefore, have all the properties of the base codelet. These codelets do not override any of the base codelet's defaults.

Percept Registers Listener. These codelets listen for notification from the focus that the perception registers have been filled.

Check All Other Perceptual "Consciousness" Codelets Are Alive For Association Setup. Many of the perceptual "consciousness" codelets are those which carry the different message types to "consciousness." So that these message type carrying codelets have a high likelihood of being associated with the codelets carrying information such as the time, they are initially associated with each of the other non-message type perceptual information carrying codelets. Before these associations can be assigned, however, it must be ensured that the codelets to which the associations are being made have actually been initialized.

Refresh Codelet. Upon reaching "consciousness," all of the "consciousness" codelets carrying the perception registers' information reset the their variables. Specifically, they:

1. Leave the playing field.
2. Set their activation back to low.
3. Set their "conscious" indicator to not yet "conscious."

4. Reset their broadcast information.
5. Decrement the focus' cannot fill the perception registers counter.

NewSeminarMessageTypeCarrier performs the following algorithm.

1. Once all perceptual codelet's are alive:
    1. Setup the initial codelet's information such as the codelet's name and initial activation level.
    2. Setup the initial codelet's associations.
2. Loop forever:
    1. Sleep.
    2. Decay associations.
    3. If on the playing field:
        1. If the "consciousness" indicator states the codelet can leave the playing field, refresh the codelet.
        2. Else, decay the activation level.
        3. Return to the start of loop.
    4. If the perception registers are not newly filled, return to the start of the loop.
    5. If the perception register's message type is not of type initiate a new seminar, return to the start of the loop.
    6. Increment the focus' cannot_fill_registers_counter.
    7. Place the appropriate key and message into the codelet's information to be broadcast.

8. Join the playing field.
9. Increase activation to high.

Conclusions

This chapter describes how “consciousness” is realized in “conscious” software agents. “Consciousness” codelets, extensions of the base codelet, bring novel and conflicting information from the focus and the behavior network’s composition workspace. These codelets compete for “consciousness” along with all other codelet’s in the system. The playing field, coalition manager, spotlight controller, chunking manager, and short-term memory all serve to complete the implementation of global workspace theory’s “consciousness.” Not described in this chapter, however, is how “consciousness’ ” different components are arranged in the code and how others’ modules interface with it. This is the subject of chapter 6.

Chapter 6
ConAg

The “Conscious” Agent Framework

The “Conscious” Agent Framework (ConAg) is a software framework for implementing “consciousness” in software agents. ConAg is intended to implement “consciousness” according to global workspace theory (Baars, 1997), and its algorithms are those described in chapter 5. ConAg is designed to carefully follow software reuse methodology, discussed in chapter 2. ConAg is designed under the architectural style for “conscious” software agents, detailed in chapter 4. This chapter first discusses the rational for implementing ConAg in Java. This includes giving an overview of Java beans since all of ConAg’s classes which can be beans are. ConAg’s primary goals are then detailed. The framework’s package structure is described. For unfamiliar topics in this section, please refer chapter 5. Next, the well known design patterns which ConAg incorporates are presented. The techniques that other modules use to integrate with ConAg is then described. ConAg’s current graphical user interface is presented to help ground the framework and illustrate how portions of the testing results are gathered. Finally, testing results are presented.

Why Java?

Java was introduced in late 1995 by Sun Microsystems with a large marketing blitz. Opening any Java book's introduction or visiting *http://java.sun.com* illustrates this fanfare. After careful consideration amongst the "conscious" software research group including this author, the group decided to use Java for several reasons. First, Java programs run on a virtual machine. Therefore, they run on all operating systems for which a Java virtual machine has been developed. This machine provides a layer of abstraction over the operating system so that, to Java programs, all operating systems are the same. In particular, this is helpful in the development of user interfaces and in multi-threading programming; both are often operating system dependent. At decision time, the operating systems of choice among the research group's developers were Windows 95, Windows NT for Intel, Solaris for Sparc, Linux for Intel, and Mac OS. Java has relatively strong virtual machine support for each of these platforms, making it a quite portable language.

Java has solid multi-threading support. Each thread works in parallel, or in a single-process environment, in simulated parallelism to all other threads. In Java, threads are easy to create. This helps in modeling both global workspace theory and pandemonium theory, as both describe numerous processes being in action simultaneously. Java is also highly object-oriented. Objects help provide for encapsulation, allowing for both a structure's data and methods to exist within the same class. In addition, it provides for inheritance. Inheritance helps facilitate the sharing of

properties among a group classes. Object-oriented programming helps foster the independent development of different modules by different research group members as there is minimal interaction needed among developers. Simultaneously, inheritance helps in providing a common interface across modules for all of the agents' codelets.

Java Beans

ConAg is implemented following the Java bean convention. Beans are simply classes, with specific naming conventions for events, methods, and variables. By making each class a bean, it can be opened and modified by any Java development environment supporting beans such as Sun's Java Workshop. This is possible as each bean's properties and events follow the same convention and, therefore, they can be easily exposed. A simple example of this convention is the naming of variables. Variable names begin with a lower case letter, such as activationLevel. Two methods, a get and set method, then follow a standard naming convention to provide access to this variable. In this case, the get method is named getActivationLevel and the set method is named setActivationLevel.

A jar file is created when the compiled class files are placed into a single compressed file. After ConAg's source files are compiled, they are placed into jar files. Jar files can be read by any bean-development tool, such as Sun's free Bean Box. The Bean Box is a simple tool intended to facilitate a shared vision for bean developers. ConAg's jar files can be loaded into the Bean Box, and its events and variable names are

visible. With this, other module developers utilizing ConAg can see the framework's properties without having to dive into the code.

The Framework's Primary Goals

As a framework, ConAg serves four primary goals:

1. To fit within the boundaries of the architectural style for "conscious" software agents (described in chapter 4).
2. To provide a drop-in implementation for the domain-independent portions of these agents' "consciousness" mechanism.
3. To provide working, easily customizable, and properly documented domain-specific portions of the "consciousness" mechanism, such as "consciousness" codelets which look for a specific conflict.
4. To provide quality, working stubs for the cognitive mechanisms such as behaviors and emotions found in "conscious" software agents. These illustrate how these modules should work with the "consciousness" module and provide a starting point for the mechanisms' development.

ConAg's structure

Explicit care has been taken to ensure ConAg follows good coding practice. ConAg's source code is 100% pure Java, and it utilizes Java's beans framework as well as the AWT and Swing frameworks (Eckel, 1998) for its graphical user interface. All possible classes are Java beans, helping contribute to both black-box reuse, and when necessary,

easy modification for white box reuse (see chapter 2). Each source file has detailed header comments, and almost every line of code is commented. All comments are catered for Javadoc use. Javadoc comes with Sun's standard Java distribution, and it allows for comments to contain html code and be turned into readable html files.

ConAg has a detailed package structure allowing for its components to be easily found and identified. The next several figures and subsections describe these packages at a high level. At the root is the "conscious" software agent directory, shown in figure 6.1. This directory is not actually a package within ConAg. It is simply the location where each of the different modules' jar files are placed. For example, besides ConAg, this directory contains emotion and perception, developed independently. In reality, these jar files could be placed anywhere on the disk, as long as the Java virtual machine is told where to find them. ConAg's package branches off into two main groups, the base codelet and "consciousness" packages. Each are described below.

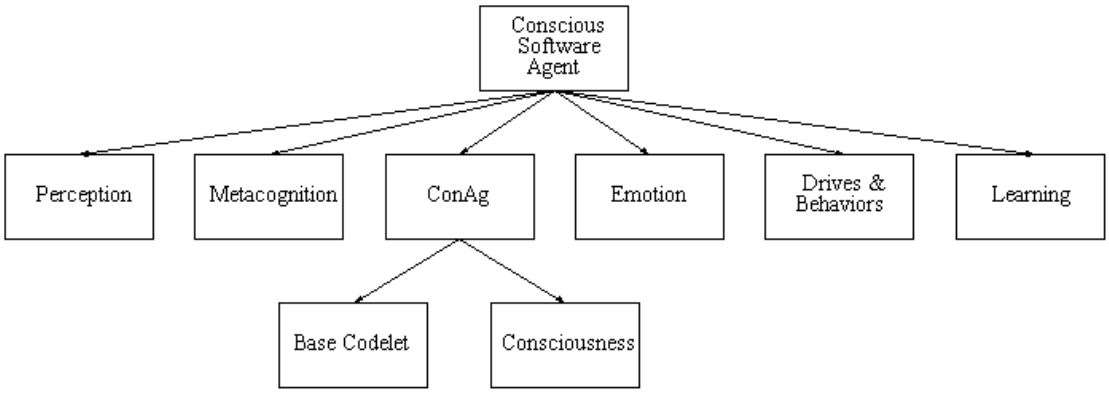

Figure 6.1: "Conscious" Software Agent Directory
(Bogner, Maletic, & Franklin, In Press 1999)

ConAg's Domain Independent Portions

ConAg can be viewed partially as a "generic" framework similar to frameworks for building graphical user interfaces. It implements portions of the "consciousness" mechanism intended to work across domains. In other words, many portions of the framework can be dropped in an agent being developed such as a distribution agent or travel agent.

Codelet Definitions

Conscious software agents' base codelet resides in ConAg's base codelet package. Similar to all of ConAg's packages, the base codelet package contains several classes. Since the base codelet package is crucial to "conscious" software agents, it is worthwhile to mention the classes within this package. The codelet class inherits or instantiates all of the other classes in the package.

ActivationLevels.class: This bean contains all the information needed for each codelet to set its activation level.

AssociationElement.class: This bean stores all information about each codelet's associations.

BroadcastElement.class: This bean contains the key and message the codelet carries in the event it becomes "conscious."

BroadcastEvent.class: This is the object that is actually "thrown" by the Broadcast Manager when it sends out a broadcast. It contains the hashtable that codelets

query as they look for relevant messages. All codelets know how to listen for and work with this object.

BroadcastListener.class: This is an interface class, meaning that it contains empty methods that must be declared by all classes that extend it. This class contains the method necessary to receive the broadcast from the broadcast manager.

BroadcastListener.class is extended by Codelet.class, which implements the appropriate method for broadcast receipt. In this way, all codelets in the system receive the "conscious" broadcast.

Codelet.class: Described in the previous chapter, this bean contains the base information common to all codelets in "conscious" software agents, including "consciousness" and generator codelets.

As seen in figure 6.2, ConAg also includes a package containing classes common to "consciousness" codelets. This package highlights that "consciousness" codelets carry information, whether it be a conflict or novel information. Further specificity for "consciousness" codelets is included in ConAg, and this is discussed in the Domain Dependent Portions section below.

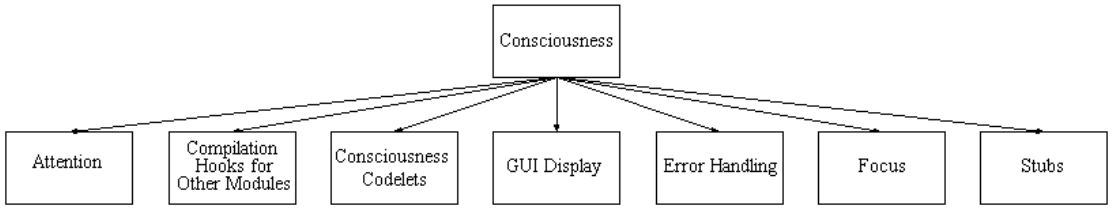

Figure 6.2: "Consciousness" Package
(Bogner, Maletic, & Franklin, In Press 1999)

Attention

Figure 6.2 also shows where attention is located. The attention package includes the broadcast manager, chunking manager, coalition manager, playing field, short-term memory, and spotlight controller. It also contains the definition of a chunk. These are domain independent, and each of these mechanisms can be modified to the developer's satisfaction. For example, a new algorithm for forming coalitions can be created based on the one included in the framework.

Compilation Hooks

Figure 6.3 shows the packages in the compilation hooks package. ConAg's compilation hooks allow the framework to be compiled without reliance on the other module's jar files. They are needed as ConAg currently has eleven locations where method calls are made to cognitive modules built by other "conscious" software research group developers. These are:

- The read and write calls to associative memory by the focus.
- The read and write calls to episodic memory by the focus.

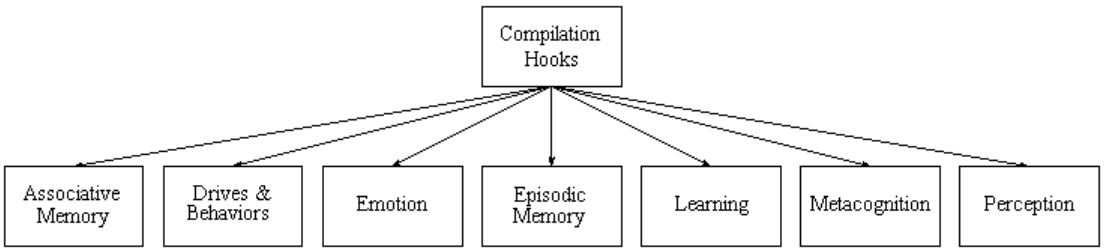

Figure 6.3: Compilation Hooks Package

- A read of the behavior network's composition working memory by "consciousness" codelets working to detect conflicts in the outgoing announcement.
- The ability to query, from ConAg's graphical user interface, methods from other cognitive modules stating the last items they passed to the focus. Specifically, the modules are the associative memory, behavior network, emotion, episodic memory, and perception. This system is used to help ensure that the focus is correctly integrated with the corresponding modules.
- "Conscious" software agents' main method, starting up these agents, is found in the ConAg package, illustrated in figure 6.1. Different classes within this package and instantiated by the main method allow for different startup scenarios. These classes make calls to the other cognitive modules to start them up.

The compilation hooks package contains its own packages, one for each of the modules to which ConAg makes method calls. For each cognitive module, only the classes that ConAg imports are included. For each included class, only the methods ConAg accesses are included, and these methods are normally empty or contain only bare functionality. These compilation hooks are intended to be written and maintained by the developer(s) of ConAg, not of the other cognitive modules. They can only be written, however, after agreement is reached with ConAg's developer(s) and the other cognitive module developers on how the integration takes place.

Graphical User Interface

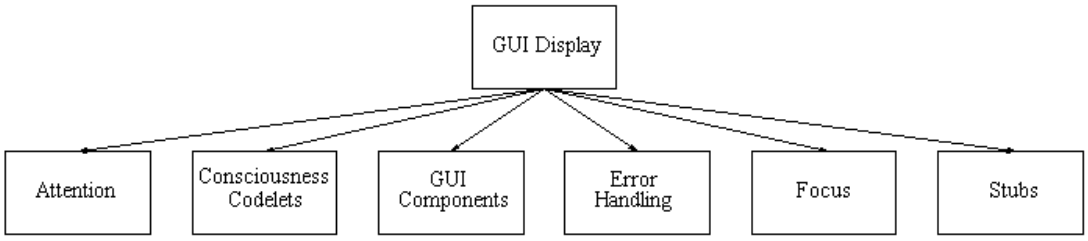

Figure 6.4: Display Package
(Bogner, Maletic, & Franklin, In Press 1999)

ConAg's graphical user interface, written using Java's AWT and Swing frameworks, serves two roles. First, if desired, it allows different ways for starting up ConAg. For example, ConAg can be started using all of CMattie's other cognitive modules or just its own stubs (described below). Of note, it also allows the "consciousness" module to be started independently of the other modules. While yet unproven, this may provide a testing ground to compare how these agents run "consciously" versus entirely "unconsciously." Second, the gui provides a window into the inner workings of the system. Screenshots of portions of this are seen later in the chapter, as illustrations on how testing information is gathered. Currently, the gui is catered to developers' use as what is displayed is fine-grained. Figure 6.4 illustrates the gui package structure. Notice that it is a close mirror to the "consciousness" package shown in figure 6.2. This allows developers to easily find its components. The differences to the "consciousness" package include a components package, where beans used throughout the display module reside. All of the gui is domain-independent, except in the case of "consciousness" codelets as

these are domain dependent. In cases where “consciousness” codelets are added (or removed) for a new domain, the gui can be easily modified to view these new codelets. More importantly, ConAg does not depend on a gui to run; the package could be completely removed. This provides a means for a user-interface based on a different toolkit to be provided.

Error Handling

As seen in figure 6.2, ConAg provides a common mechanism for handling errors throughout the system. To help foster ConAg’s independence from the need for the user-interface, the provided GUI contains its own error handling mechanism. It currently uses identical techniques as the framework’s main one. This technique allows the developer a single point in which to code for handling additional errors while also providing consistent debugging methods throughout the framework.

Other Cognitive Module Stubs

ConAg provides its own stubs for the other cognitive modules such as the behavior network. These stubs simulate the basic functionality of the actual modules. For example, the behavior network stub listens for an appropriate broadcast from “consciousness” and sets the focus with a behavior. On startup, all or some of these stubs can be run instead of the other cognitive modules. These stubs get their data from text files, making them ideal for testing.

It is important to realize the rational for separating the compilation hooks from the stubs. The compilation hooks are created only after discussion with other developers, and

they are subject to the changes the other developers make in terms of class names and method calls. On the flipside, ConAg(s) developers control the course of the stubs, and they provide functionality even if the other cognitive modules' are not ready for integration. These stubs often do not share method names with the real cognitive modules. In addition, many of "conscious" software agents' modules are domain specific, such as perception. These stubs, however, are domain independent. Within ConAg's modules such as the focus, checks are done to see if the actual cognitive module or the corresponding ConAg stub has been started. The appropriate method calls are then made.

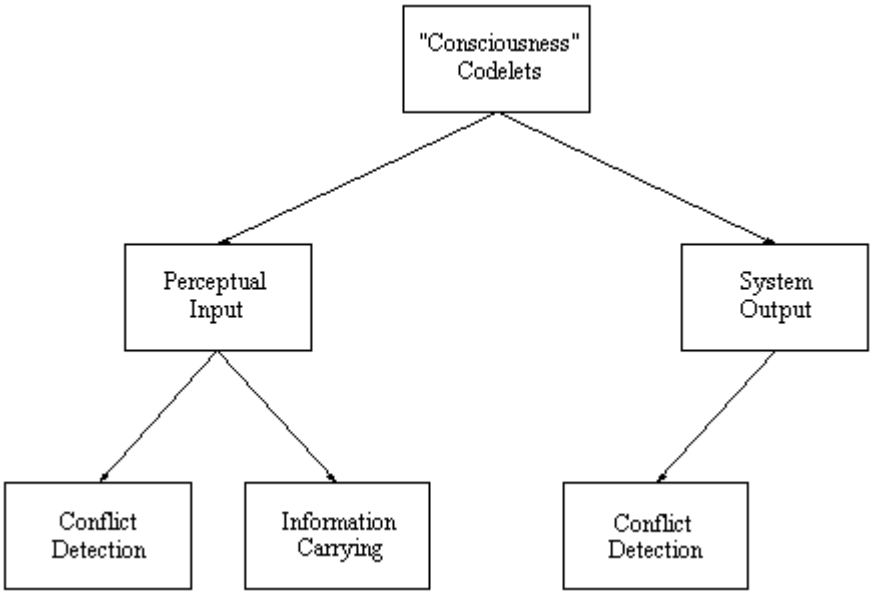

Figure 6.5: "Consciousness" Codelet Package
(Bogner, Maletic, & Franklin, In Press 1999)

Domain Dependant Portions

ConAg's packages for "consciousness" codelets and the focus are domain-dependent, with the exception being the "consciousness" codelet components common to all of these codelets. As seen in figure 6.5, ConAg provides "consciousness" codelets to detect conflicting and novel information for perceptual input and conflict detection for the systems' output. "Consciousness" codelets are domain specific. Currently, ConAg's "consciousness" codelets are tailored for use in CMattie. These provide a basis for white-box reuse in order to apply these components to new domains. As "consciousness" codelets are Java beans, often times code changes to the graphical interface are not needed when "consciousness" codelets are applied to new domains.

As previously described, the focus is the location where perceptual information is created for the agents' own use. This perceptual information is associated with the agents' memories, and "consciousness" codelets bring this new and potentially conflicting information to "consciousness." Perceptual information is domain specific as are systems' memories about their taken actions in relation to what has been perceived. Therefore, the focus, while an integral part of "conscious" software agents, is domain-dependent. Even so, ConAg provides common methods for a focus' use across domains.

ConAg's Design Patterns

Design patterns, an important aspect of software reuse methodology and described in chapter 2, are heavily utilized throughout ConAg. Illustrated here is ConAg's use of

several patterns, all described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1995). The abstract factory pattern provides an interface for creating families of related or dependent objects without their concrete classes needing specification. Abstract factory patterns are used throughout ConAg. Examples are seen in the base codelet and base perceptual "consciousness" codelet definitions.

The singleton pattern ensures that there is only one instance of a class and that it is accessible globally. ConAg relies on this pattern for each of the components that start up its different modules. For example, the attention startup bean provides single access to the attention components for the other cognitive modules, namely those that access the playing field and the broadcast. "Consciousness" codelet startup provides access to all of the "consciousness" codelets. Focus startup provides a single point of access to the perceived information and the memories associated with it.

Used throughout ConAg is the façade pattern, which defines a higher-level unified interface to a subsystem, making these subsystems easier to use. In ConAg, active codelets join the playing field. The playing field's structure is hidden from them; there is simply a common way to exit and join the field. Completed perceptual information is set in the focus for use by the entire system; the actual process of perceiving is hidden. Codelets receive the broadcast information; hidden from them is how this information is collected and arranged for broadcast.

The strategy pattern defines a family of algorithms, encapsulating each one and making them interchangeable. This allows algorithms to vary without directly affecting those which utilize it. Throughout ConAg's attention package, great care has been taken

to follow this pattern. For example, a different algorithm for forming coalitions can be used in the coalition manager; the same holds true for the spotlight controller's choosing a "conscious" coalition. The methods for gathering the information to be broadcast and the actual manner with which it is broadcast is also interchangeable. The same holds true for the representation of short-term memory.

The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependencies are notified and automatically updated. A prime example of this in ConAg occurs with the "conscious" broadcast, where one broadcast is received by all codelets in the system. In addition, when the focus receives a new percept, one announcement of this fact is sent out to all of the system's perceptual "consciousness" codelets and certain emotion codelets.

The memento pattern provides a way to capture and externalize an object's internal state, without violating encapsulation, so that the object can be restored to this same state later. "Conscious" software agents often have a self-preservation mechanism. For agents with this mechanism and written in Java, the base codelet class has the option to utilize Java's serialization techniques. Serialization achieves the memento pattern, and since all codelets inherit the base codelet component, all codelets' states can be captured and restored. In this future, this may apply to the system's short-term memory as well.

How Other Cognitive Modules Integrate With ConAg

Currently, seven cognitive modules developed by other conscious software research group members integrate with ConAg. Five of these modules, the behavior network,

emotion, learning, metacognition, and perception, extend the base codelet class. They do this by importing ConAg.BaseCodelet.Codelet. Most developers choose for their codelets to access short-term memory as well. In this case they import ConAg.Consciousness.Attention.AttentionStartup and access short-term memory, for example, via AttentionStartup.shortTermMemory.getMemory().

The behavior network, emotion, and perception modules all integrate with ConAg to access the focus. To do this, they import ConAg.Consciousness.FocusPackage. FocusStartup. They can then access the focus, for example, via FocusStartup.focus. setBehaviorRegisters (String[] behavior), FocusStartup.focus.setEmotionRegisters (String[] emotions), FocusStartup.focus.getRegistersCanBeFilled(), etc.

Associative and episodic memories integrate with ConAg. Specifically, they import the MemoryReadReturn class in ConAg.Consciousness.FocusPackage. This class contains a boolean value specifying whether or not associative memory converges. It also contains the returned memory registers which are set if there is convergence. Both associative and episodic memory return a MemoryReadReturn class when a read is made from the focus to either memory.

ConAg's Graphical User Interface Revisited

Currently, two methods are available to get information on what is occurring inside the system: a log file and ConAg's gui display. Below is a small excerpt from the log file when ConAg is running with its stubs. When ConAg is running with the real cognitive modules, the log file is more difficult to follow as each module independently writes to

the file. The illustration below takes up after “consciousness” codelets have picked up their respective information from the focus; also, the reads from associative and episodic memories have already occurred.

- The spotlight controller has determined that no coalition has a high enough average activation for “consciousness.”

```
A NEW COALITION WAS NOT CHOSEN FOR CONSCIOUSNESS.  THRESHOLD FOR
CONSCIOUSNESS: 0.47829682
```

- The coalition manager has rapidly traversed the playing field, forming coalitions (twice)

```
Coalition Manager states playing field's size is: 10
Coalition Manager states playing field's size is: 10
```

- The spotlight controller once again traverses the coalition manager’s coalitions.

```
The Spotlight Controller's coalitions:
  Coalition:0
    Name:    SpeakerTopicMessageTypeCarrier
    Id:      0.9185863266452637
    Act Lvl: 0.7

    Name:    DateCarrier
    Id:      0.3318647383472735
    Act Lvl: 0.45

    Name:    DayCarrier
    Id:      0.6290151970713053
    Act Lvl: 0.45

    Name:    EmailAddressCarrier
    Id:      0.9806771259346015
    Act Lvl: 0.45

    Name:    PlaceCarrier
    Id:      0.4155485895454599
    Act Lvl: 0.45

    Name:    SeminarNameCarrier
    Id:      0.3213166329772882
    Act Lvl: 0.45
```

```
    Name:    SeminarOrganizerCarrier
    Id:      0.2814919677465175
    Act Lvl: 0.45

    Name:    SpeakerNameCarrier
    Id:      0.044040281456874886
    Act Lvl: 0.45

    Name:    TalkTitleCarrier
    Id:      0.5934670356478824
    Act Lvl: 0.45

    Name:    TimeCarrier
    Id:      0.4145744269360071
    Act Lvl: 0.45

  Coalition:1
    Name:    TalkTitleCarrier
    Id:      0.5934670356478824
    Act Lvl: 0.45

  Coalition:2
    Name:    EmailAddressCarrier
    Id:      0.9806771259346015
    Act Lvl: 0.45
```

- In this case, while not listed, there are a total of ten coalitions.

- The spotlight controller now computes each coalition's average activation level. The first three log entries for this are shown.

```
Coalition 0 average activation level: 0.475
Threshold for consciousness: 0.43046713
Coalition 1 average activation level: 0.45
Threshold for consciousness: 0.43046713
Coalition 2 average activation level: 0.45
Threshold for consciousness: 0.43046713
```

- The spotlight controller selects a "conscious" coalition.

```
The conscious coalition's average activation level: 0.475

THE CONSCIOUS COALITION:
    SpeakerTopicMessageTypeCarrier 0.9185863266452637
    DateCarrier 0.3318647383472735
    DayCarrier 0.6290151970713053
    EmailAddressCarrier 0.9806771259346015
    PlaceCarrier 0.4155485895454599
    SeminarNameCarrier 0.3213166329772882
    SeminarOrganizerCarrier 0.2814919677465175
    SpeakerNameCarrier 0.044040281456874886
    TalkTitleCarrier 0.5934670356478824
    TimeCarrier 0.4145744269360071
```

- The broadcast manager receives the “conscious” coalition, and selects each codelet’s key and message.

```
In Broadcast Manager,  information about to be broadcast:
  Codelet: SpeakerTopicMessageTypeCarrier 0.9185863266452637
    Key: speakerTopicMessageType
    Message: SpeakerTopicMessageTypeCarrier
  Codelet: DateCarrier 0.3318647383472735
    Key: prDate
    Message: DateCarrier:8th January '99
  Codelet: DayCarrier 0.6290151970713053
    Key: prDay
    Message: DayCarrier:Friday
  Codelet: EmailAddressCarrier 0.9806771259346015
    Key: prEmailAddress
    Message: EmailAddressCarrier:linki@msci.memphis.edu
  Codelet: PlaceCarrier 0.4155485895454599
    Key: prPlace
    Message: PlaceCarrier:Dunn Hall 351
  Codelet: SeminarNameCarrier 0.3213166329772882
    Key: prSeminarName
    Message: SeminarNameCarrier:Computer Science seminar
  Codelet: SeminarOrganizerCarrier 0.2814919677465175
    Key: prSeminarOrganizer
    Message: SeminarOrganizerCarrier:David Lin
  Codelet: SpeakerNameCarrier 0.044040281456874886
    Key: prSpeakerName
    Message: SpeakerNameCarrier:Sudipkumar P. Karnavat
  Codelet: TalkTitleCarrier 0.5934670356478824
    Key: prTalkTitle
    Message: TalkTitleCarrier:Knowledge Discovery for Time Series
(Master Thesis defense)
  Codelet: TimeCarrier 0.4145744269360071
    Key: prTime
    Message: TimeCarrier:2:00 pm
```

- The broadcast manager sends the “conscious” information to all codelets.

```
Broadcast Manager sending broadcast at: 5/2/99 4:44 PM
```

- ConAg’s behavior network and emotion stubs determine a message type was broadcast. This signifies a new perception has been received. The current behavior and emotion are then written to the focus.

```
ConAg stub detected a message type was just broadcast:
speakerTopicMessageType
ConAg's emotion stub detects a broadcast message type, setting the
Focus.
Focus reports that the current emotions have been set.
```

```
ConAg stub detected a message type was just broadcast:
speakerTopicMessageType
ConAg's Behavior Network stub detects a broadcast message type, setting
the Focus.
```

- The focus determines that both the behavior network and emotion modules have written to it. The focus now writes to associative memory.

```
Focus reports the current behavior has been set.
Focus reports it has just written to associative memory.
Focus reports it has just written to case based memory.
```

- Even while ConAg's stubs had received the broadcast and performed their actions, others still had not. Here, the broadcast manager has completed sending the "conscious" information to all listeners. Left out is the printout of the hashtable actually broadcast.

```
Broadcast Manager has completed sending broadcast to all listeners.
```

- Short-term memory receives the "conscious" contents. Here the hashtable received is printed.

```
Short term memory received from broadcastManager:
{prPlace=PlaceCarrier:Dunn Hall 351, prDay=DayCarrier:Friday,
prDate=DateCarrier:8th January '99,
prSeminarName=SeminarNameCarrier:Computer Science seminar,
prSpeakerName=SpeakerNameCarrier:Sudipkumar P. Karnavat,
prTalkTitle=TalkTitleCarrier:Knowledge Discovery for Time Series (Master
Thesis defense), broadcastTime=5/2/99 4:44 PM, prTime=TimeCarrier:2:00
pm, prEmailAddress=EmailAddressCarrier:linki@msci.memphis.edu,
speakerTopicMessageType=SpeakerTopicMessageTypeCarrier,
prSeminarOrganizer=SeminarOrganizerCarrier:David Lin}
```

- The chunking manager receives the codelets, sorts them, and determines if a new potential chunk is necessary.

```
Chunking Manager received these codelets:
DateCarrier:DayCarrier:EmailAddressCarrier:PlaceCarrier:SeminarNameCarri
er:SeminarOrganizerCarrier:SpeakerNameCarrier:SpeakerTopicMessageTypeCar
rier:TalkTitleCarrier:TimeCarrier
Chunking Manager reports conscious coalition has not been in
consciousness before, creating new chunk.
```

- The spotlight controller notifies the “conscious” codelets that they have been “conscious.” For “consciousness” codelets, which are all of the codelets currently in “consciousness,” this signifies they can leave the playing field.

```
Spotlight Controller has set the conscious codelets'
canLeavePlayingField flag to true.
DateCarrier just left the playing field.
DayCarrier just left the playing field.
Coalition Manager states playing field's size is: 8
EmailAddressCarrier just left the playing field.
PlaceCarrier just left the playing field.
SeminarNameCarrier just left the playing field.
SeminarOrganizerCarrier just left the playing field.
SpeakerNameCarrier just left the playing field.
SpeakerTopicMessageTypeCarrier just left the playing field.
TalkTitleCarrier just left the playing field.
TimeCarrier just left the playing field.
```

- All of the perceptual information has made it to “consciousness,” and the focus now awaits for a new percept.

```
The Focus is now ready to receive a new percept.
```

ConAg’s graphical user interface provides a view into the internal workings of the system. Figure 6.6 illustrates the current startup screen with its menu-bar resembling today’s common applications. Figure 6.7 shows ConAg’s File menu. If ConAg’s display is started as part of a running “consciousness” module, the Start Consciousness Module menu item’s choices are disabled. However, if only ConAg’s display is started, it can be used to start “conscious” software agents in four ways:

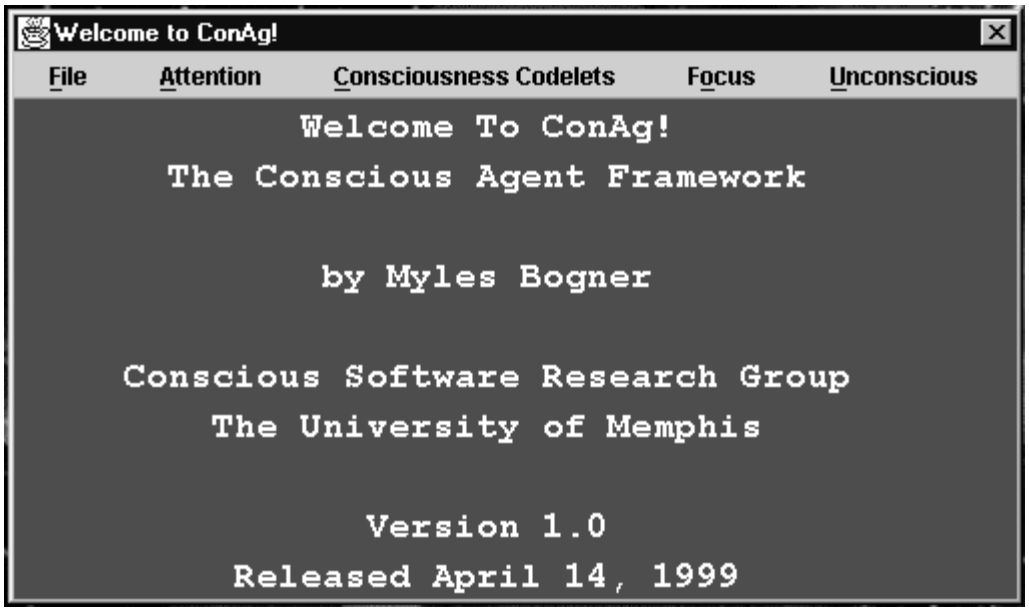

Figure 6.6: ConAg's Startup Screen

1. "Consciousness" as well as all the other cognitive modules and their user interfaces.
2. "Consciousness" as well as all the other cognitive modules without their user interfaces.
3. "Consciousness" with ConAg's stubs serving as the other cognitive modules.
4. Without starting ConAg or its stubs. This can be used to start "consciousness" after all other cognitive modules are running. This can potentially be used to test the agent running with or without "consciousness."

Different startup options can easily be integrated into the framework based on the provided ones.

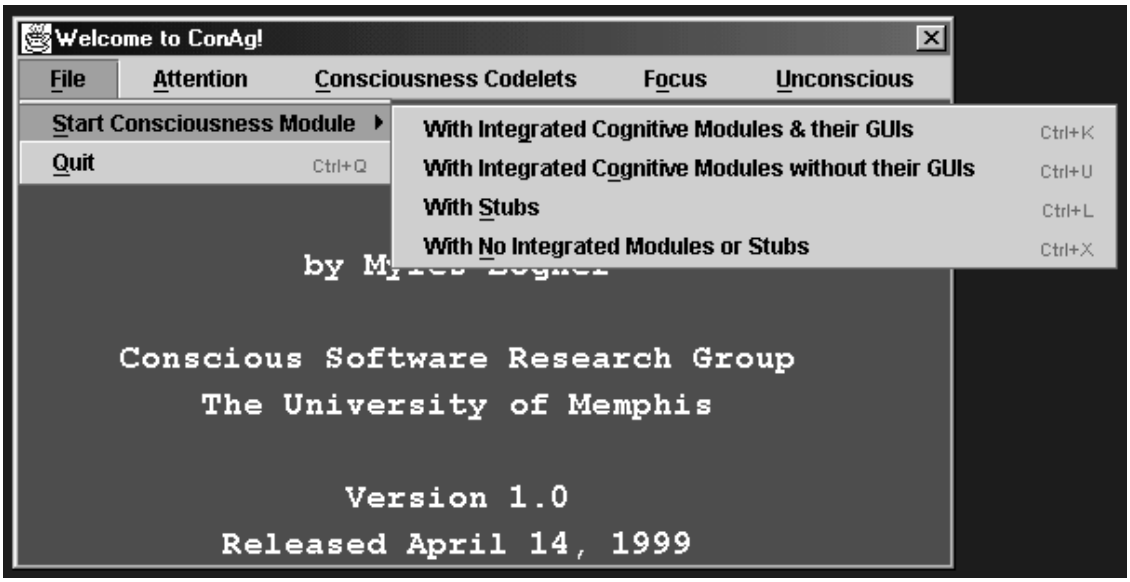

Figure 6.7: ConAg's File Menu

Figure 6.8 shows ConAg's Attention menu. This menu allows for viewing each of the portions of the Attention package. Currently, throughout ConAg's menus, selecting a menu item, brings the screen shown in figure 6.9. The top portion of the screen gives instructions. Pressing the button on the left updates the information immediately. Entering a number in seconds at the bottom and pressing return displays the desired information at the selected interval. Pressing the Reset button stops this interval display. While displaying the information at intervals, the Update Now button can be pressed at any time. Figure 6.9 shows the codelet's listening for the "conscious" broadcast. Listed are the codelets' names and unique ids. Figure 6.10 illustrates the framework's "consciousness" codelets menu. Submenus and menu items can be easily

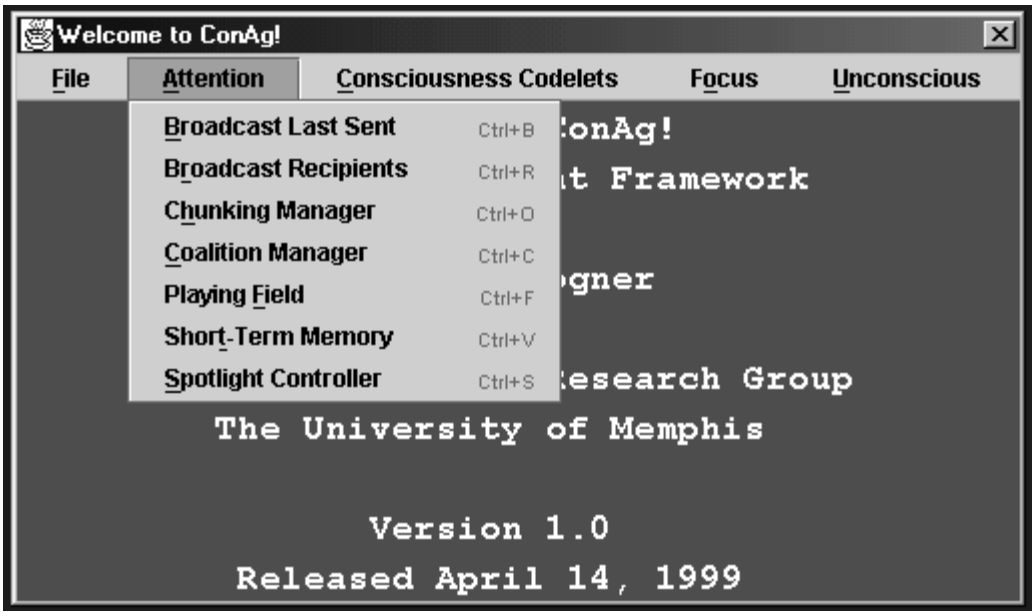

Figure 6.8: ConAg's Attention Menu

added for the addition of new "consciousness" codelets. Figure 6.11 shows a portion of the information seen when looking at a "consciousness" codelet.

Figure 6.12 shows the Focus menu. This menu allows the viewing of the focus' memory and perception registers. Figure 6.13 illustrates a perception register output. Figure 6.14 shows the "Unconscious" menu. This menu allows the viewing of the "unconscious" modules developed by the other "conscious" software research group members. By viewing this information, the last information associative memory, behavior network, emotion module, etc. intended to set to the focus is viewable. This information can be compared to that actual focal contents to determine if these modules are communicating appropriately.

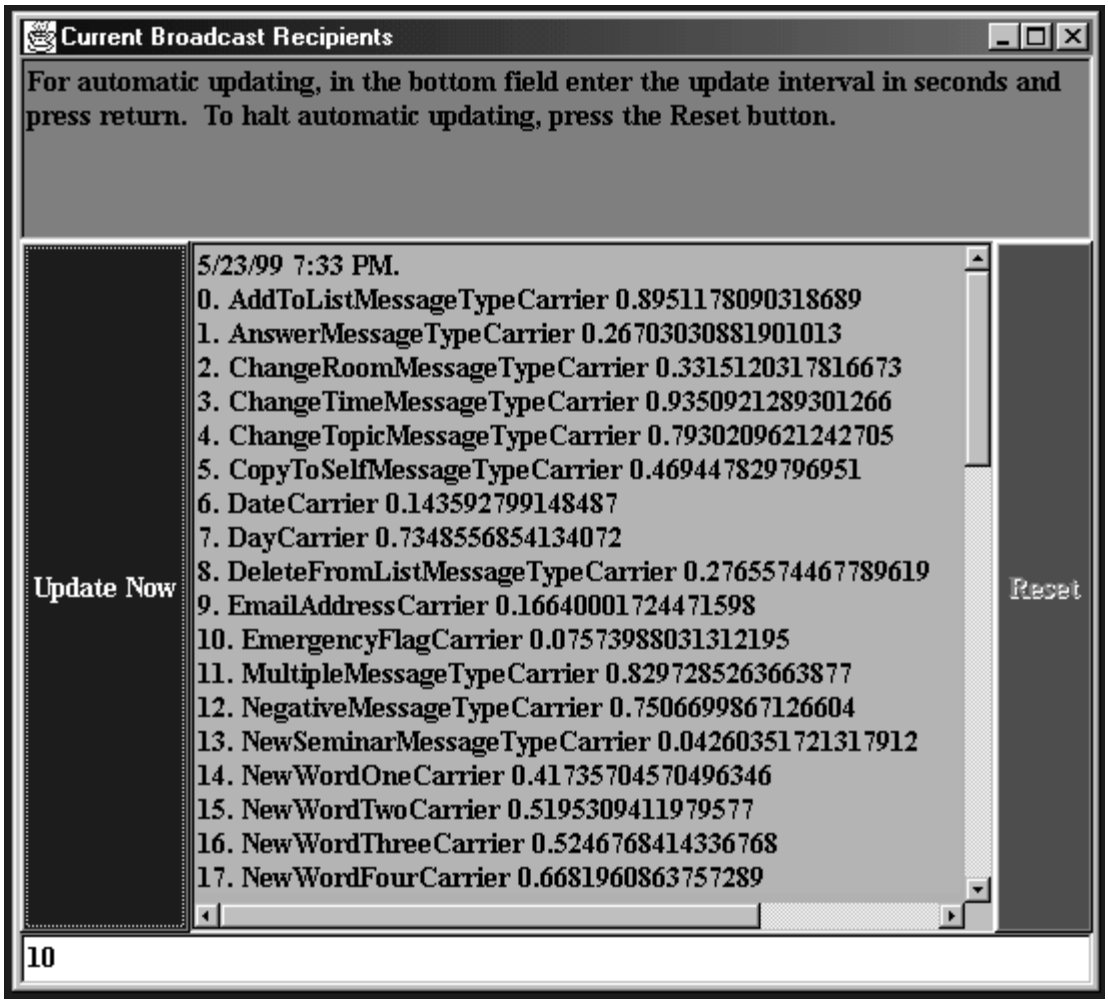

Figure 6.9: ConAg's Broadcast Recipients View

Testing Results

In parallel to its continual development, ConAg is being thoroughly tested. As stated previously, at this time a fully realized "conscious" software agent is not available. Therefore, the testing on ConAg has been fine-grained in nature, making sure the framework does what it is supposed to do. As an example of the tests performed, this

section presents eight tests performed. All tests had very solid results. Each test performed was for ConAg's use in CMattie.

Test One

This test asked three questions. First, do the focus' perception registers receive the data correctly from the perception module? If so, do all the appropriate "consciousness" codelets pick up the information? If this occurs, does all the information get chosen for "consciousness?"

To perform this test, ten different message types were chosen from email messages previously sent to the departmental secretaries. As currently there is not a completed perception module, these messages were placed in ConAg's perception module stub file. They were placed in the stub file in such as way as to mimic how CMattie's perception module should ideally perform. ConAg was run with these ten messages as input and then stopped. Upon completion, the log file was analyzed to ensure all of the perception registers were filled correctly for each message. Also, it was checked to make sure all of the "consciousness" codelets which were supposed to pick up the information did in fact spring into action. Finally, it was check that the "conscious" broadcast contained the same information as the initial stub file.

Testing revealed that the perception registers were able to be set correctly 100% of the time. The "consciousness" codelets picked up their information 100% of the time. The data broadcast from "consciousness" matched the initially perceived information 100% of the time.

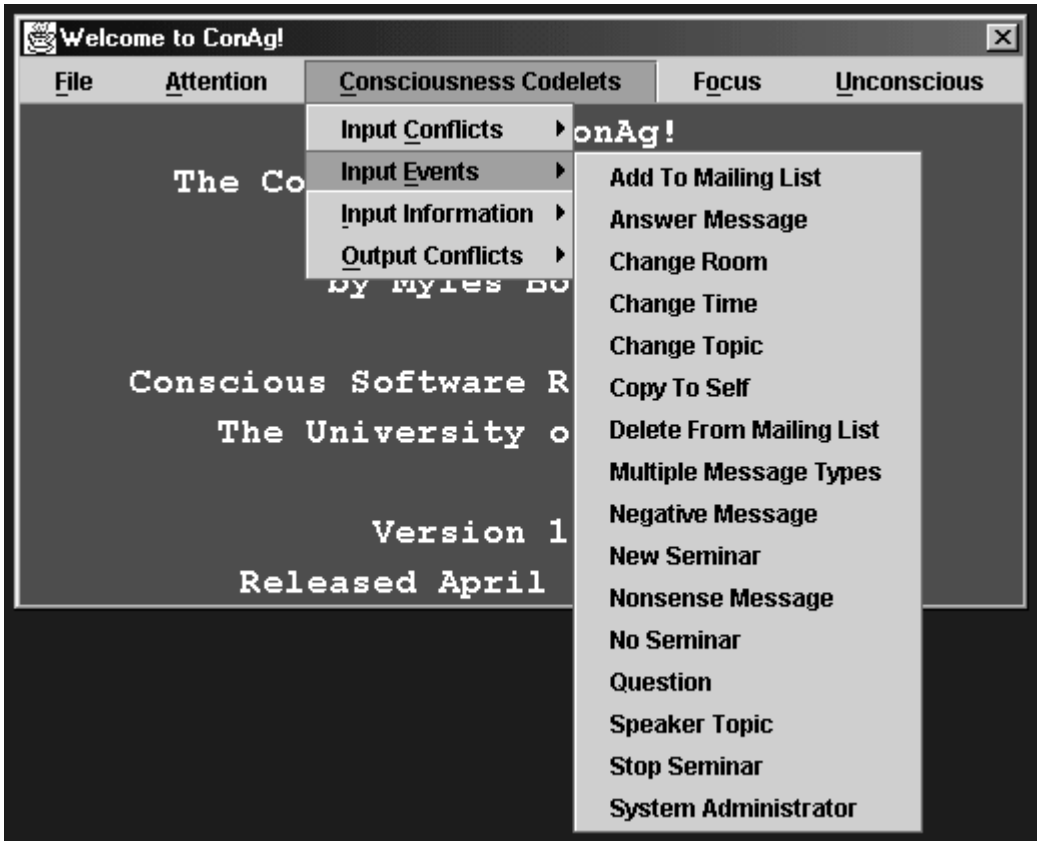

Figure 6.10: ConAg's "Consciousness" Codelets Menu

Test Two

This test made sure that in all cases after receiving the perception registers, the focus performed a read from associative memory. This test also made sure that in each of these cases, the focus performed a read from episodic memory.

To perform these tests, test one's sample of different ten message types was used, as well as ConAg's associative and episodic memory stubs. For all ten messages, after the perception registers received a new perception, 100% of the time a read from associative memory and episodic memory occurred.

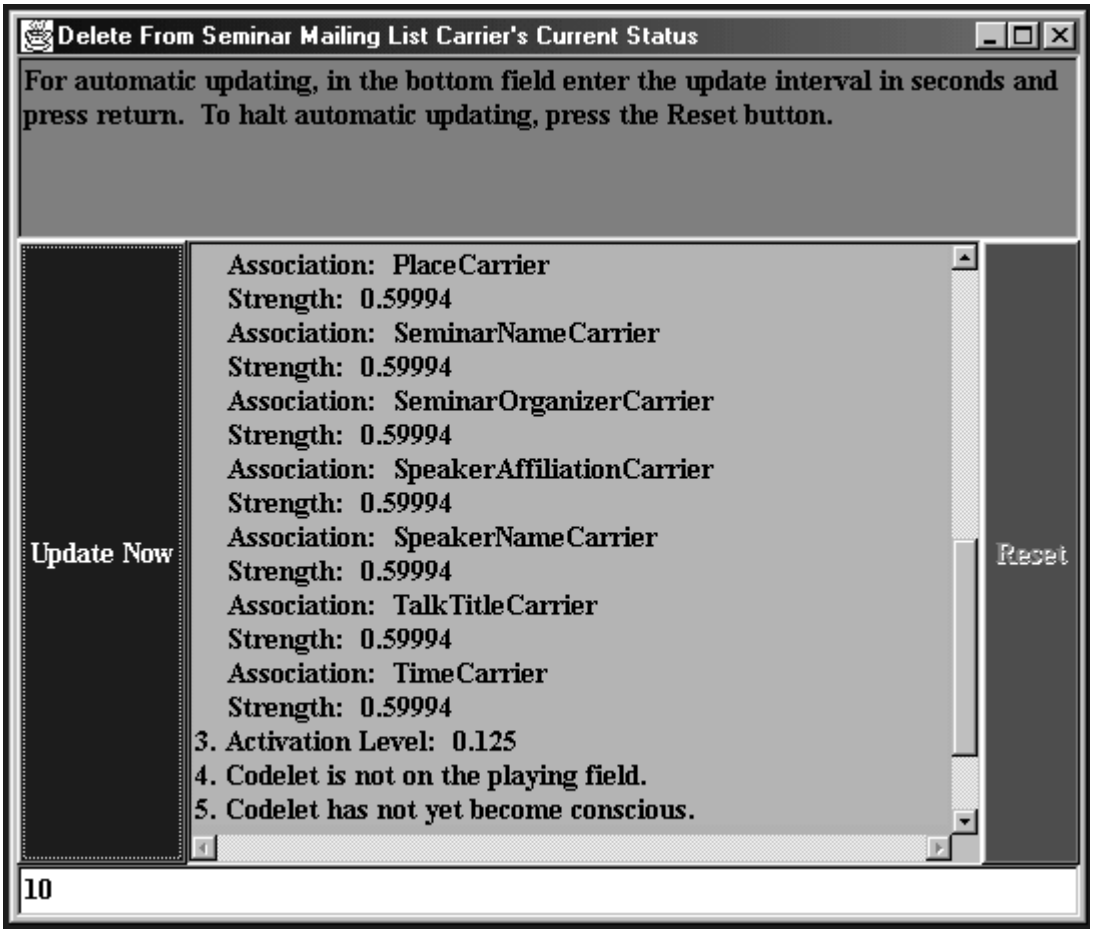

Figure 6.11: "Consciousness" Codelet View

Test Three

This test asked if all codelets in the system are in fact registered to listen for the broadcast. To perform this test, test one's input was used. At three, five minutes intervals, it was checked to see if all codelets running in the system were broadcast listeners. This test was then repeated in its entirety. At each of the six check points, 100% of the codelets alive in the system were listening for the broadcast.

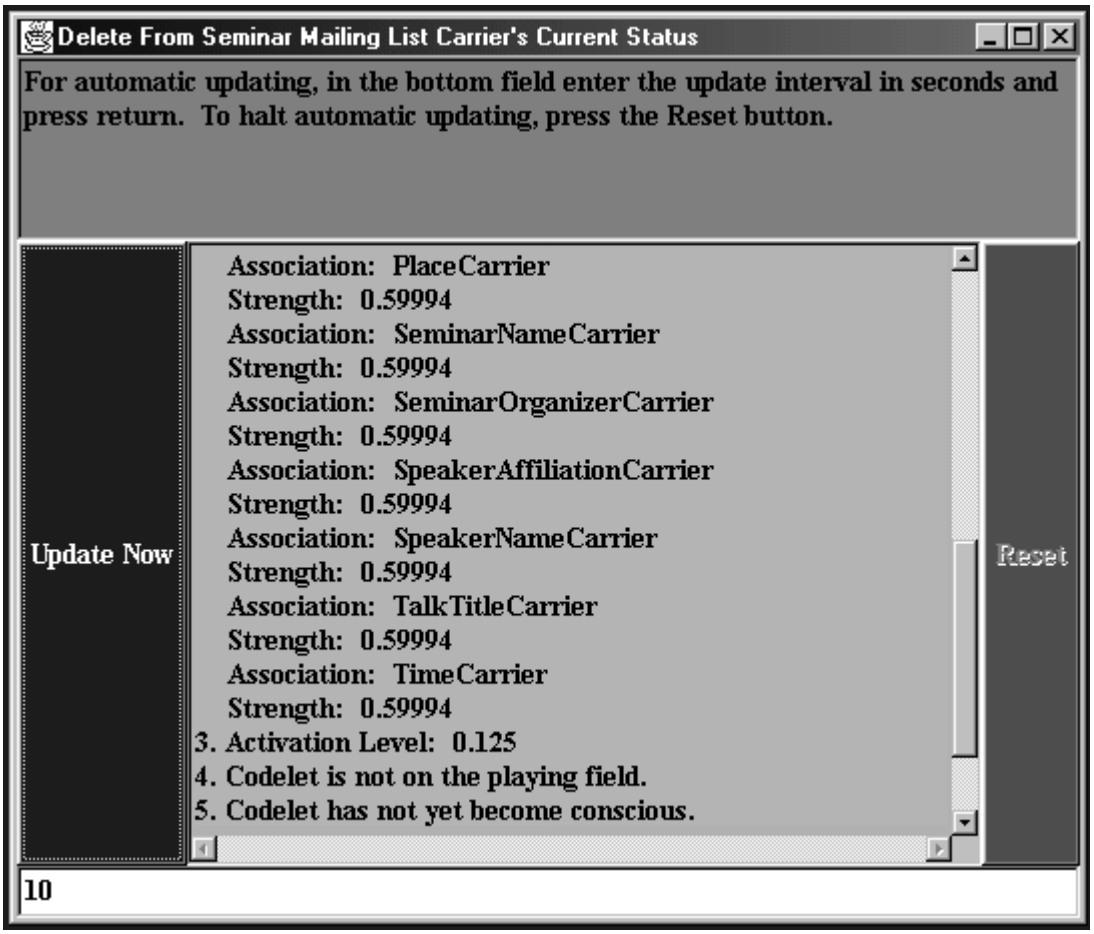

Figure 6.12: ConAg's Focus Menu

Test Four

This test asked if all codelets, when on the playing field, were grouped into coalitions. Test one's input data was used. For each input message, the system's subsequent playing field codelets were logged. These were compared to the codelets in the coalition manager's coalitions. For each of the input messages, 100% of the time the codelets on the playing field were in fact in the coalition manager's coalitions. Do note that in

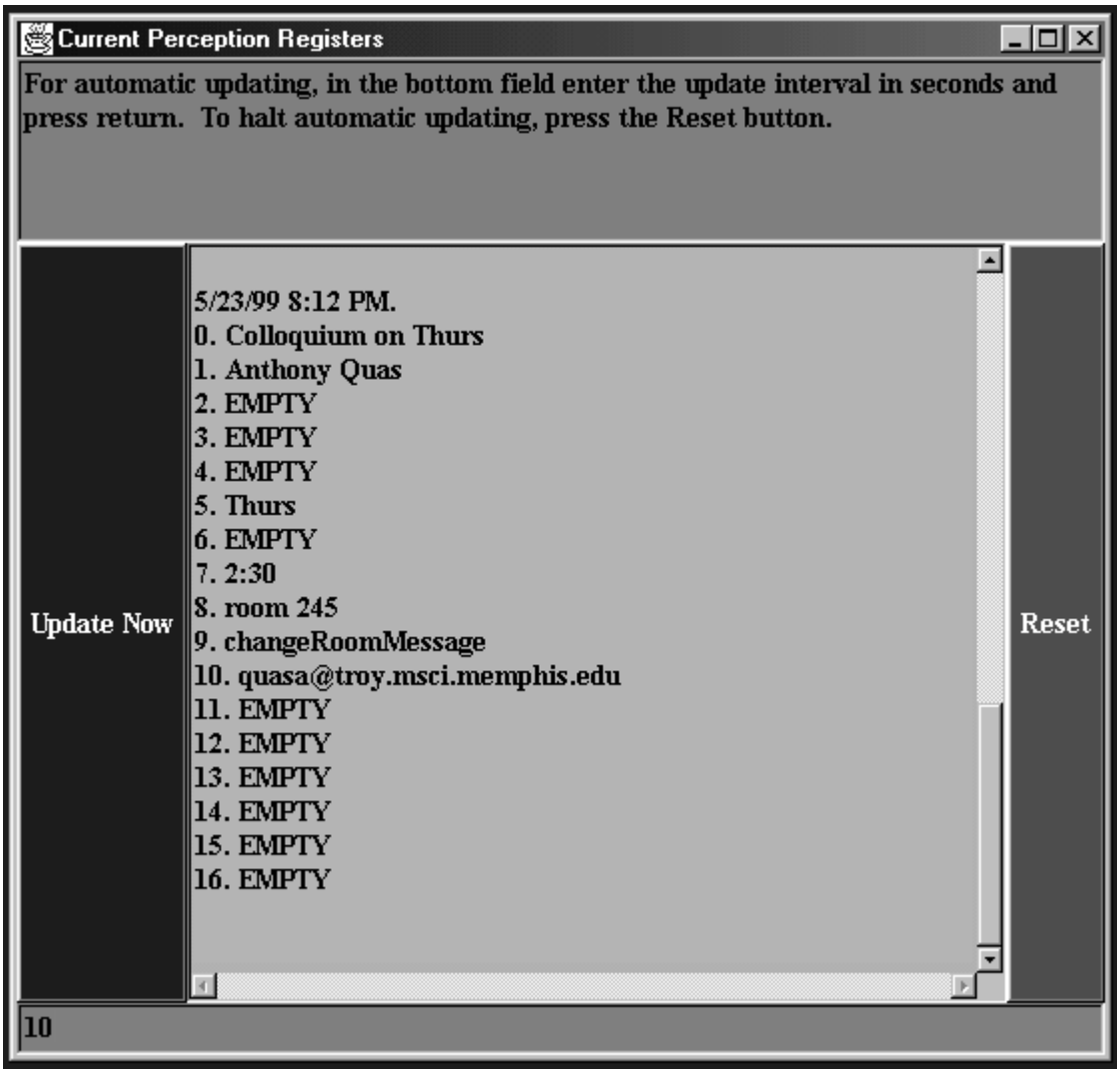

Figure 6.13: Current Perception Registers Snapshot

general, this is not always the expected case, especially if a working codelet joins and leaves the playing field while the coalition manager's thread is asleep.

Test Five

Test five made sure that the coalition with the highest average activation level is always the one chosen for "consciousness." This test used test one's input data. The list of

coalitions were viewed. For the first ten "conscious" coalitions, this data was analyzed to check if the highest average activation coalition became the "conscious" one. This occurred 100% of the time.

Test Six

This test checked to make sure that the broadcast is always prepared correctly. Question one's input data was used. By analyzing the output log, the "conscious" coalition was compared to the broadcast. Checked was whether or not all of the "conscious" codelets' information was picked up by the broadcast manager. For each codelet's information, it was made sure that this data was formatted correctly by the broadcast manager.

The results found that 100% of the time, the "conscious" codelets were the codelets broadcast. 100% of the time this broadcast information was formatted correctly by the broadcast manager.

Test Seven

Question seven asked if short-term memory received the items from "consciousness" correctly. If so, it then asked if short-term memory contained a maximum of the last seven items in "consciousness." This test used question one's input. For each of the input messages, the data was analyzed to ensure the short-term memory items were correct. 100% of the time short-term memory correctly received the "conscious" coalition's information. 100% of the time short-term memory had a maximum of seven items.

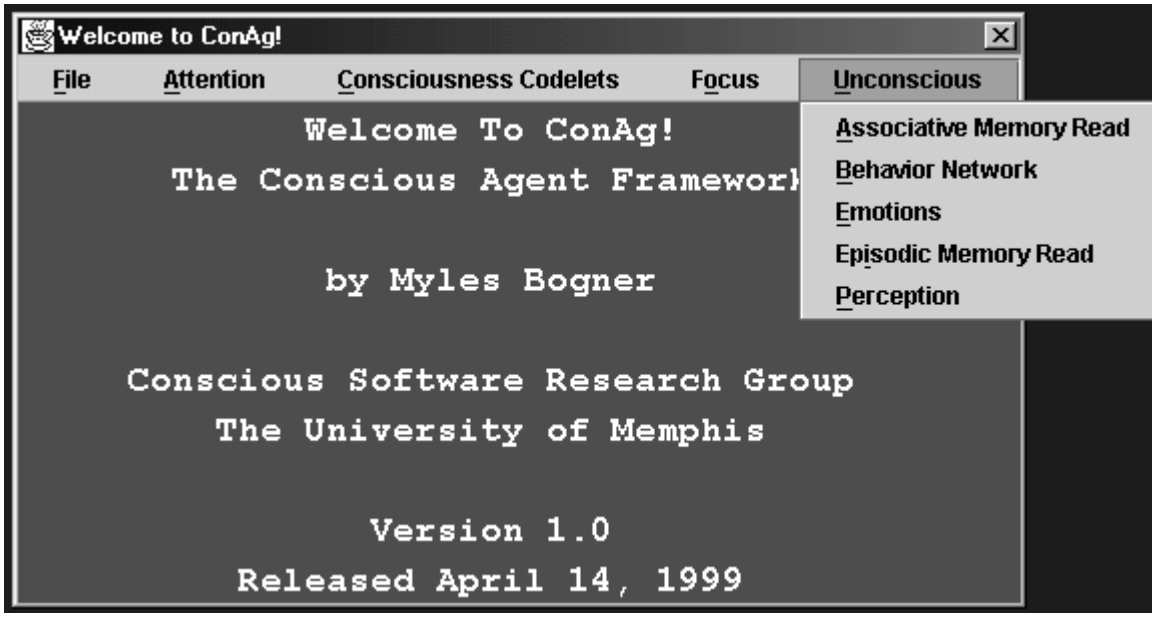

Figure 6.14: ConAg's "Unconscious" Menu

Test Eight

This test analyzed whether or not the chunking manager received the broadcast correctly and appropriately prepared the (potential) chunk. The test data was each of test one's input messages. The chunking manager performed appropriately 100% of the time.

Conclusions

This chapter describes the framework's structure. This framework is designed to be easily extended for the many "conscious" software agent environments. Examples of the fine-grained testing results, hopefully, illustrate to the reader that the framework as a program does appear to function correctly. The real underlying question, however, is

does ConAg implement Baars' global workspace theory's consciousness. This is a subject of the final chapter.

Chapter 7
Computational "Consciousness"

Conclusions

This work gives overviews of software agents, cognitive models, and software reuse. Global workspace theory is detailed. VMattie, a predecessor to "conscious" software agents, is described. "Conscious" software agents' architectural styles and general architecture are then discussed. This includes an illustration of CMattie, the first "conscious" software agent, and a description of IDA, the "conscious" software research group's proof of concept project. "Conscious" software agents' "consciousness" mechanism is then detailed. ConAg, the "Conscious" Agent Framework, a software framework for implementing these agents' "consciousness," is described.

ConAg serves as the backbone for "conscious" software agents, providing for these agents' base codelet class, "consciousness" codelets, a focus where incoming perception is associated with remembered information, an attention mechanism which includes short-term memory and chunking for learning, and conflict resolution. It is the first framework to implement pandemonium theory in its realization of global workspace theory's "consciousness."

The research leading to this dissertation concentrates on how to implement global workspace theory, with a focus on the theory's consciousness. This work is necessary as

global workspace theory describes when and why consciousness occurs. It leaves out how the consciousness mechanism actually works, such as a coalition manager grouping processes for consciousness. This research hopes to shed light on these mechanisms and, therefore, to further extrapolate global workspace theory. Therefore, ConAg, as a realization of global workspace theory's consciousness, not only extends global workspace theory conceptually, but also provides a grounds for the theory's further development and testing. Described below, each design decision made in ConAg's implementation can be considered a hypothesis about the corresponding human mechanism.

This research contributes to artificial intelligence, cognitive science, and philosophy. For artificial intelligence, this research creates an agent architecture that intends to be more flexible for agent decision making. It does this by producing human-like thinking. While this architecture needs to be proven in more complex domains, it lends itself to agents that are able to find intricate solutions to problems, learn in multiple ways, and perform deliberation and synthesis. This architecture fosters the creation of complex autonomous agents. It is a new technology that should allow software agents to replace decision-making human information agents. For example, one "conscious" software agent could be intended to replace a human help-desk customer support representative.

For cognitive science and philosophy, an expandable cognitive architecture has been developed around the "consciousness" module by this research group. The resulting comprehensive model allows for cognition to be analyzed as a whole, and also for each of

the cognitive mechanisms to be studied individually. Currently, few comprehensive cognitive models exist. In addition, this research further helps in studying what machines can actually experience. "Conscious" software agents have the right mechanisms for "consciousness." It is currently unclear on how to determine if these agents are actually experiencing.

Conclusions can be drawn from this research. The functions of consciousness as specified by global workspace theory can be modeled computationally. A comprehensive cognitive architecture can be integrated around this model of consciousness. This comprehensive cognitive architecture can be implemented computationally. Many new artificial intelligence mechanisms can be used to create such a comprehensive cognitive architecture. Testable hypotheses concerning human cognition can result from such a comprehensive model.

Are Baars' Nine Functions of Consciousness Implemented In ConAg?

Chapter 3 discusses Baars' nine functions of consciousness. To some extent, each of these functions are implemented by ConAg. This is illustrated below through CMattie and IDA.

1. *Definition and context-setting.* This function occurs when one focuses on a distant tree in a forest. While multiple stimuli are present, a coherent context is able to be retrieved. In "conscious" software agents, the currently executing behavior corresponds to global workspace theory's goal context. In these

agents, the action selection mechanism ultimately decides the next behavior. This decision is greatly influenced by the contents of "consciousness." More specifically, "consciousness" codelets bring the newly perceived information to "consciousness." After the broadcast occurs, the behavior network may instantiate a new behavior stream in response to "consciousness' " contents. "Consciousness' " influence is evident if this behavior stream executes.

Global workspace theory also describes perceptual and cultural contexts. In "conscious" software agents, the perception module's understanding portion sets the perceptual context. This can be influenced by "consciousness." "Consciousness" shines on the perception module's codelets. At times, the "conscious" broadcast recruits additional perceptual codelets that help to determine the perceptual context. In CMattie, "consciousness" does not influence the cultural context. This has yet to be explored in IDA.

2. *Adaptation and learning.* This function is evident when extremely difficult material is pondered for a great deal of time while attempting to learn it. ConAg's implementation of "consciousness" provides for several forms of learning. Codelets' associations are established and strengthened when on the playing field together, and even more so when in "consciousness" together. The chunking manager broadcasts new chunks, potential items to be learned by the behavioral and perceptual learning mechanisms. After the "conscious" broadcast, the current behavior and emotions are written along with the

current perception registers into associative and episodic memory, used by the systems' learning mechanism.

3. *Editing, flagging, and debugging.* A tennis player's conscious concern over the technical details of his serve after several double faults is an example of debugging. Both in the behavior network's composition working memory and in the focus, "consciousness" codelets flag conflicts for debugging. Upon finding a conflict, these codelets work to bring the information to "consciousness." Once this information is broadcast, the different cognitive modules such as learning and metacognition can respond, adjusting their parameters accordingly. IDA's deliberation process involves the creation of scenarios. It is currently planned that "conscious" decisions are necessary for deciding the fitness of scenarios and for editing them.
4. *Recruiting and control.* An example of this function's use occurs when attempting to answer a question. While one is conscious of a question, the candidate answers to that question are recruited unconsciously and brought to consciousness. By viewing the information broadcast from "consciousness," other codelets spring into action if they understand the message and it is applicable. This is seen throughout conscious software agents. New behaviors are begun based on the contents of "consciousness," metacognition begins new system evaluations, and learning takes place.
5. *Prioritizing and access-control.* This occurs when learning a foreign language. One may wish to prioritize words which are difficult to pronounce,

giving them greater access to consciousness. When ConAg's "consciousness" mechanism broadcasts, other codelets in the system respond, increasing their activation level. This higher activation level gives them greater access to "consciousness."

ConAg's "consciousness" mechanism causes codelets to learn new associations. Over time, these associations may become strong enough for a codelet to be placed into a coalition which frequently comes to "consciousness." Therefore, this new codelet now has greater access to "consciousness." It is planned that IDA's "conscious" deliberation prioritizes scenarios before one is presented to a sailor.

6. *Decision-making or executive.* This function is useful in controlling thought and action, such as "Should I go to the mall or to the park?" As described in number 1, "consciousness" is a main but indirect reason for behaviors, corresponding to global workspace theory's goal contexts, to be chosen. It is planned that "conscious" decisions are made in IDA's deliberation. Also, in both CMattie and IDA, metacognition performs tuning based in part on "consciousness' " contents.
7. *Analogy-forming.* This function occurs when people make analogies to compare a novel experience to known ones, as seen with "Hate is the wrong road to travel." "Consciousness" plays an indirect role in both the behavioral and perceptual learning mechanisms' analogy forming. The focus' perception registers contain slots for novel words, and there are message types for both

nonsense messages and negative messages in response to CMattie's actions. After this perceptual information reaches "consciousness," behavioral and perceptual learning create new concepts, along with their underlying codelets, based on existing concepts and codelets.

8. & 9. *Metacognitive or self-monitoring, autoprogramming and self-maintenance.* The metacognitive function is evident in humans' ability to express their current feelings, and self-maintenance is seen in the desire to keep the body healthy. Both in the compositional working memory and in the focus, "consciousness" codelets work to detect internal conflicts. In addition, based in part on the contents of "consciousness," metacognition is able to monitor the system. When applicable, metacognition can then perform self-maintenance by adjusting the system's parameters. A planned self-maintenance is to regularly backup the agents' information to disk.

Hypotheses

Many design and implementation decisions are made in ConAg in order to create computational "consciousness." Each of these decisions gives rise to a hypothesis about human cognition. Given a particular decision about a portion of the agents' mechanisms, the hypothesis asserts that human cognition works in the same way (Franklin, 1997). Hopefully, these hypotheses will be confirmed or laid to rest by researchers in cognitive science and neuroscience. Twenty such hypotheses are listed below. As a reminder to readers, ConAg's codelets are intended to implement Baars' processes. As this

discussion is cognitive in nature, the term process is used over codelet. Each of these hypotheses is generated by replacing "codelet" with "process" in a true statement about codelets in "conscious" software agents.

1. There are common properties which all processes share. Five main ones are:
   A. Each process receives the "conscious" broadcast as postulated by global workspace theory.
   B. Each process carries activation level. This activation level allows processes to compete with one another for access to "consciousness."
   C. Each process carries associations with other processes. These associations are used to determine a process' coalitions.
   D. Each process carries information pertinent to its current situation. If a process becomes conscious, this information is broadcast to all processes.
   E. Each process carries identifying information. If a process becomes conscious, this information is broadcast to all processes.
2. All processes such as the coalition manager and those working to bring conflicting information to consciousness have periods in which they are not active.
3. Perceived sensory data is realized by a well-defined structure in a focal location. For this dicussion, this structure will be referred to as the perception registers. These are located in the focus.
4. This focus contains a flag, letting the perception module know that a new perception can now be placed in the perception registers.

5. A new percept can be placed in the focus only after the former percept has become conscious.
6. Upon receipt of a new perception, processes work to bring this information to consciousness.
7. While number 6 is occurring, reads from a long-term associative memory and from an intermediate-term episodic memory are performed. Processes work to bring this information to consciousness.
8. There is a single playing field where active processes are accessible by consciousness. (Recall that, in the conscious software agent architecture, the playing field contains all of the agents' active codelets). In humans, this playing field is most likely distributed broadly within the nervous system. Consciousness can potentially gather any active process' identifying information and its information pertinent to the current situation.
9. The nervous system has a coalition manager which groups the playing field's processes together.
10. This coalition manager makes its decisions based on the level of association between processes.
11. The coalition manager creates and updates associations for the processes on the playing field, each by a relatively small amount.
12. A spotlight controller chooses which coalition is conscious, based on the coalition's average activation level.

13. Short-term memory holds only previously conscious items, and stores a maximum of some seven items.
14. A chunking manager glues processes together in order to build new potential concepts. These chunks are based on associations between the processes and on which processes have previously been conscious simultaneously.
15. When a new chunk is realized, it comes to consciousness via the chunking manager. This chunk is broadcast to the entire system just as the coalitions handled by the spotlight controller are.
16. A broadcast manager picks up the information from the conscious processes, prepares it for broadcast, and sends it out to all processes in the system.
17. An action selection mechanism, after receipt of a new percept, writes the current behavior and its activation level back to the focus.
18. The emotion module, after receipt of a new percept, writes the current emotions back to the focus.
19. Upon receipt of the behavior and the emotions, the focus records both of them along with the perception registers to the long-term associative and to the intermediate-term episodic memories.
20. Specialized processes check for conflicts as the system prepares an outgoing communication. If conflicts are found, these processes work to bring this information to consciousness.

The Future

This work presents a realization of global workspace theory's "consciousness." It is only a start. Further integration followed by testing is necessary. Refinement of the techniques will no doubt occur. Even so, this research has culminated in a solid foundation for "consciousness" in software agents. ConAg is used in CMattie and in IDA, a project which if successful, will have a major impact on the United States' Naval personnel assignment. While this is several years off, already this project has proven very enjoyable and served as a significant learning experience for this author. Hopefully, this will continue for me and future participants.

BIBLIOGRAPHY

BIBLIOGRAPHY

Aho, Alfred V. & Ulman, Jeffrey D. (1992). Foundations of Computer Science. New York: Computer Science Press.

Anderson, John R. (1991). The place of cognitive architectures in a rational analysis. VanLehn, Kurt (Ed.). Architectures for Intelligence. Hillsdale, NJ: Lawrence Erlbaum Associates, 1-24.

Baars, Bernard. (1988). A cognitive theory of consciousness. New York: Cambridge University Press.

Baars, Bernard. (1997). In the theater of consciousness. New York: Oxford University Press.

Barsalou, Lawrence W. (1999). Perceptual symbol systems. Behavioral and Brain Sciences. New York: Cambridge University Press.

Basili, V., Briand, L., & Melo, W. (1996). How reuse influences productivity in object-oriented systems. Communications of the ACM, 39 (10), 104-116.

Biggerstaff, T. & Richter, C. (1987). Reusability framework, assessment, and directions. IEEE Software, 4 (2), 41-49.

Bogner, Myles. (1998). Creating a “conscious” agent. Memphis: Master’s thesis, The University of Memphis.

Bogner, Myles, Maletic, Jonathan, & Franklin, Stan. (1999). Building "consciousness" into software. Department of Mathematical Sciences Technical Report CS-99-02. The University of Memphis.

Bogner, Myles, Maletic, Jonathan, & Franklin, Stan (In Press 1999). ConAg: a reusable framework for developing "conscious" software agents. The International Journal on Artificial Intelligence Tools. River Edge, NJ: World Scientific Publishing Company.

Bogner, Myles, Ramamurthy, Uma, & Franklin, Stan. (In Press 1999). "Consciousness" and conceptual learning in a socially situated agent. Dautenhahn, Kerstin (Ed.). Human Cognition and Social Agent Technology. Amsterdam: John Benjamins Publishing Company.

Bonabeau, Eric, Henaux, Florian, Guérin, Sylvain, Snyers, Dominique, Kuntz, Pascale, & Theraulaz, Guy. (1998). Routing in telecommunications networks with "smart" ant-like agents. Santa Fe Institute Publications & Intelligent Agents for Telecommunications Applications '98.

Boone, Gary. (1998). Concept features in Re:Agent, an intelligent email agent. Proceedings of the Second International Conference on Autonomous Agents. New York: ACM Press, 141-148.

Cline, Marshall P. (1996). The pros and cons of adopting and applying design patterns in the real world. Communications of the ACM, 39 (10), 47-49.

Crosbie, Mark & Spafford, Gene (1995). Proceedings of the 18th National Information Systems Security Conference. Purdue University Technical Report 95-008.

Eckel, Bruce. (1998). Thinking in Java. Upper Saddle River, NJ: Prentice Hall Inc.

Etzkorn, Letha H. & Davis, Carl G. (1997). Automatically identifying reusable OO legacy code. IEEE Computer, 30 (10), 66-71.

Foundation for Intelligent Physical Agents. (1996-Present). http://www.fipa.org. Retrieved April 1, 1999.

Frakes, William B. & Fox, Christopher, J. (1995). Sixteen questions about software reuse. Communications of the ACM, 38 (6), 75-87.

Franklin, Stan. (1995). Artificial Minds. Cambridge, MA: The MIT Press.

Franklin, Stan. (1997). Autonomous agents as embodied ai. Cybernetics and Systems, 28 (6), 499-517.

Franklin, Stan. (Submitted). Conscious software: a computational view of mind.

Franklin, Stan and Graesser, Art. (1997). Is it an agent, or just a program?: A taxonomy for autonomous agents. Intelligent Agents III. Berlin: Springer-Verlag, 21-35.

Franklin, Stan, Kelemen, Arpad, and McCauley, Lee. (1998). IDA: a cognitive agent architecture. Proceedings of the IEEE Conference on Systems, Man and Cybernetics, 2646-2651.

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. (1995). Design Patterns. Reading, MA: Addison-Wesley.

Graesser, Arthur, Franklin, Stan, & Wiemer-Hastings, Peter. (1998). Simulating smooth tutorial dialog with pedagogical value. Proceedings of the American Association for Artificial Intelligence. Menlo Park, CA: AAAI Press, 163-167.

Graesser, A.C., & Person, N.K. (1994). Question asking during tutoring. American Educational Research Journal, 31, 104-137.

Graesser, A.C., Person, N.K., & Magliano, J.P. (1995). Collaborative dialogue patterns in naturalistic one-on-one tutoring. Applied Cognitive Psychology, 9, 359-387.

Hacker, Douglas, Bogner, Myles, Yetman, Holly, & Klettke, Bianca. (1998). The curriculum script. Curriculum Script Subgroup Progress Report, Spring 1998. The University of Memphis, *http://www.psyc.memphis.edu/trg/cur_script/Summary_4-1-1998.html*. Retrieved April 1, 1999.

Haykin, Simon. (1994). Neural Networks. Upper Saddle River, NJ: Prentice-Hall.

Hofstadter, Douglas & Mitchell, Melanie. (1994). The copycat project: A model of mental fluidity and analogy-making. Holyoak, K. & Barden, J. (Eds.). Advances In Connectionist and Neural Computation Theory, 2. Norwood, NJ: Ablex.

Holland, J. H. (1986). A mathematical framework for studying learning in classifier systems. Farmer, D., et al. (Eds.). Evolution, games and learning: Models for adaption in machine and nature. Amsterdam: North-Holland.

Jackson, John. (1987). Idea for a mind. SIGGART Newsletter, 101, 23-26.

Johnson, Ralph. (1997). Frameworks = (components + patterns). Communications of the ACM, 40 (10), 39-42.

Just, M.A., & Carpenter, P.A. (1987). The Psychology of Reading and Language Comprehension. Boston: Allyn and Bacon.

Just, Marcel Adam, Carpenter, Patricia A., & Hemphill, Darold D. (1996). Constraints on processing capacity: architectural or implementational? Steier, David & Mitchell, Tom (Ed.). Mind Matters: A Tribute to Allen Newell. Mahwah, NJ: Lawrence Erlbaum Associates, 141-178.

Kanerva, Pentti. (1988). Sparse distributed memory. Cambridge, MA: The MIT Press.

Kaspersen, Donna. (1994). For reuse, process and product both count. IEEE Software 11 (5), 12.

Kernighan, Brian W. & Ritchie, Dennis M. (1988). The C Programming Language, Second Edition. Englewood Cliffs, NJ: Prentice Hall.

Kolodner, Janet. (1993). Case-based reasoning. Morgan Kaufmann Publishers.

Kozierok, Robyn & Maes, Pattie. (1993). A learning interface agent for scheduling meetings. Proceedings of the 1993 International Workshop on Intelligent User Interfaces. Orlando, FL, 81-88.

Krueger, Charles. (1992). Software reuse. ACM Computing Surveys, 24 (2), 131-183.

Laird, John E. & Rosenbloom, Paul S. (1996). The evolution of the soar cognitive architecture. Steier, David & Mitchell, Tom (Ed.). Mind Matters: A Tribute to Allen Newell. Mahwah, NJ: Lawrence Erlbaum Associates, 1-50.

Lashkari, Yezdi, Metral, Max, & Maes, Pattie. (1994). Collaborative interface agents. Proceedings of AAAI '94 Conference. Menlo Park, CA: AAAI Press.

Maes, Pattie. (1989). How to do the right thing. Connection Science Journal, 1 (3).

McArthur, D., Stasz, C., & Zmuidzinas, M. (1990). Tutoring techniques in algebra. Cognition and Instruction, 7, 197-244.

Maturana, H. R. (1975). The organization of the living: A theory of the living organization. International Journal of Man-Machine Studies, 7, 313-32.

Maturana, H. R. & Varela, F. (1980). Autopoiesis and cognition: The realization of the living. Dordrecht, Netherlands: Reidel.

McCauley, Thomas L. & Franklin, Stan. (1998). An architecture for emtion. AAAI Fall Symposium "Emotional and Intelligent: The Tangled Knot of Cognition."

Mellor, Stephen & Johnson, Ralph. (1997). Why explore object methods, patterns, and architectures. IEEE Software, 14 (1), 27-30.

Menczer, Filippo, Belew, Richard K., and Wolfram, Willuhn. (1995). Artificial Life Applied To Adaptive Information Agents. AAAI Spring Symposium Series: Information gathering from heterogeneous, distributed environments.

Monroe, Robert T., Kompanek, Andrew, Melton, Ralph, & Garlan, David. (1997). Architectural styles, design patterns, and objects. IEEE Software, 14 (1), 43-52.

Prieto-Días, Rubén, & Freeman, Peter. (1987). Classifying software for reusability. IEEE Software, 4 (1), 6-16.

Putnam, R. T. (1987). Structuring and adjusting content for students: A study of live and simulated tutoring of addition. American Educational Research Journal, 24, 13-48.

Ramamurthy, Uma, Bogner, Myles, & Franklin, Stan. (1998). "Conscious" learning in an adaptive software agent. Proceedings of The Second Asia Pacific Conference on Simulated Evolution and Learning (SEAL 98). Canberra, Australia.

Reticular Systems, Inc. (1999). Agent construction tools, *http://www.agentbuilder.com/AgentTools/index.html*. Retrieved April 1, 1999.

Rosenbloom, Paul S., Newell, Allen, & Laird, John E. (1991). Toward the knowledge level in Soar: the role of the architecture in the use of knowledge. VanLehn, Kurt (Ed.). Architectures for Intelligence. Hillsdale, NJ: Lawrence Erlbaum Associates, 75-111.

Russel, Stuart & Norvig, Peter. (1995). Artificial Intelligence: A Modern Approach. Upper Saddle River, NJ: Prentice-Hall, Inc.

Selfridge, O. G. (1959). Pandemonium: a paradigm for learning. Proceedings of the Symposium on Mechanisation of Thought Process. National Physics Laboratory.

Software Agents Mailing List. (1994-Present). Baltimore: Laboratory for Advanced Information Technology, The University of Maryland Baltimore County, *http://www.csee.umbc.edu/agentslist/*. Retrieved April 1, 1999.

Song, Hongjun. (1998). Control structures for autonomous agents. Memphis: Doctoral dissertation, The University of Memphis.

Sycara, Katia, & Zeng, D. (1994). Visitor-hoster: Towards an intelligent electronic secretary. CIKM94 Workshop on Intelligent Information Agents. *http://www.cs.cmu.edu/afs/cs/user/katia/www/visit-host.html*. Retrieved April 1, 1999.

Tambe, Milind, Johnson, W. Lewis, Jones, Randolph M., Koss, Frank, Laird, John E., Rosenbloom, Paul S., & Schwamb, Karl. (1995) Intelligent agents for interactive simulation environments. AI Magazine, 16 (1), 15-39.

Varela, F. J., Thompson, E., & Rosch, E. (1991). The Embodied Mind. Cambridge, MA: MIT Press.

Wurman, PR, Wellman, MP, & Walsh, WE. (1998). The Michigan Internet AuctionBot: A configurable auction server for human and software agents. Proceedings of the Second International Conference on Autonomous Agents. New York: Association of Computing Machinery, 301-308

Zhang, Zhaohua, Franklin, Stan, & Dasgupta, Dipankar. (1998). Metacognition in software agents using classifier systems. Proceedings of AAAI 98, 82-88.

Zhang, Zhaohua, Franklin, Stan, Olde, Brent, Wan, Yun, & Graesser, Arthur. (1998). Natural language sensing for autonomous agents. Proceedings of the IEEE Joint Symposia on Intelligence and Systems. Rockville, Maryland, 374-81.

VITA

Myles Brandon Bogner was born in New York City on April 14, 1974. His schooling prior to high school was in Austin, TX. He attended The Westminster Schools in Atlanta, GA for high school, where he graduated Cum Laude in May, 1992. He entered Rhodes College in Memphis, TN the following August. He graduated Cum Laude from Rhodes in May, 1996 with a Bachelor of Science in Computer Science and a minor in Business Administration.

In August, 1996 Myles entered The University of Memphis, Memphis, TN as a doctorate student in Computer Science. He was a teaching assistant for Computer Literacy his first year. His second year, Myles was a research assistant working on a National Science Foundation grant to develop an intelligent tutoring system. Myles received his M.S. in May, 1998. At present Myles is at the University of Memphis, working on a Naval grant to develop an Intelligent Distribution Agent. He is a member of the University of Memphis' Institute for Intelligent Systems and the "Conscious" Software Research Group.