Modeling Micro Strategies to Detect Macro Cognitive Influences in Micro Cognition: A Computational Modeling Approach to Detecting Cognitive Structures

By: Kate Dudzik

A thesis submitted in fulfilment of the requirement for CGSC 4908 as credit towards the degree:

Bachelor of Cognitive Science with Honours

Specialization in the Biological Foundations of Cognition

Thesis Supervisor: Dr. Robert L. West

Institute of Cognitive Science

Carleton University

Ottawa, Canada

April, (2017)

Abstract

The study of microstrategies within task performance has provided information pertaining to the structure, function and existence of micro cognitive architectures. Using this same methodology, we developed a set of trials intended to analyze the potential function and influence of macro cognitive architectures on micro cognitive system productions. The effect of microstrategies in a speeded response task were analyzed across human participants, and compared to a macro cognitive SGOMS ACT-R model and an optimal ACT-R performance model. The results indicate that subjects default to an SGOMS approach but may subsequently adjust their strategy under certain conditions to obtain more optimal results.

Modeling Micro Strategies to Detect Macro Cognitive Influences in Micro Cognition: A Computational Modeling Approach to Detecting Cognitive Structures

Newell argues that understanding the functioning of methods as lower levels of cognition is an important aspect of understanding all levels of cognition involved with task performance and complex functioning of human performance (1973). In the context of this paper, methods are different strategies for using our cognitive, perceptual and motor systems to perform tasks in Cognitive Psychology experiments. Studying methods provides a more complete explanation of individual action performance that is not always addressed within high level task analysis. The concept of cognitive variability has been addressed in the past pertaining to general psychology experiments (Siegler, 2007), however it is often overlooked when results are calculated and general conclusions are drawn from experimental procedures. Lower-level processing acts as a foundation for the completion of tasks, thus any difference in method execution by individuals, task sets, or due to contextual changes would therefore effect goal completion time and process of completion (Anderson, 2002; Card, Newell, & Moran, 1983; Gray & Boehm-davis, 2000).

Methods in cognitive modelling has generally been understood in terms of small, set actions that may be repeated within the same task for higher level systems to function optimally and achieve a goal state (Card et al., 1983; West & Somers, 2011). The term has been most prominently understood when referring to GOMS (Goals, Operators, Methods, and Selection-Rules), where Methods are considered a low-level action learned and applied to achieve a specific sub-goal within a task set, specific to the context of the task (Card et al., 1983). For example, it a task requires a person to respond to a stimulus my inputting a two letter, typed code, they might perform this action by using one finger on the same hand and switching keys, one finger on each hand to press each key, or by using two fingers on the same hand. If it is the

case that the person has prior knowledge as to which keys to press, they might be pre-emptively preparing and waiting for the stimuli to occur, and they do not have to wait to see it to know what it contains and what action it requires of them. Or, the person may require the visual presentation of the stimuli to prepare the action that is required. Newell's point was that we should not average across different strategies as it produces meaningless numbers that do not accurately reflect the operation of the underlying cognitive system (1973).

The GOMS theory suggests that there are three separate, yet interactive subsystems that are operators within the cognitive processing system, including cognitive, perceptual and motor, each responsible for developing processor-specific actions, which are in turn understood as processor-specific methods (Card et al., 1983). The lowest level of GOMS refers to the operator (and operator actions), with the highest level being the unit task (Card et al., 1983). . Operators describe different actions within a task, such as move hand to mouse, move eyes to icon, move cursor to icon, and click mouse. Most often the operator specifies how much time an action will take. Methods are usually a series of operators executed in serial, or in parallel, to accomplish sub goal. Methods are specific to the interface and the task, therefore they are learned and assumed to be reused. For example, it is generally assumed that people use the same method for clicking an icon each time (parameterized to suit the specific instance by considering factors such as distance, target size, etc.). Therefore, for experienced users operating simple interfaces, there will only be a limited number of methods that could reasonably be used (Card et al., 1983, Gray & Boehm-Davis, 2000, Newell, 1990, West and Somers, 2011), The structure of the unit task was intended to act as a means of organizing components, or methods, required to complete said task (Anderson, 2002; Card et al., 1983). Multiple processing systems are required to act at a higher level to produce signals calling specific methods to occur in tandem to complete the task.

The action of multiple operators acting in parallel to accomplish goals has been explored using CPM GOMS, a variation of GOMS, often constructed in the form of PERT charts, while referring to common operator interactions and organizations as templates (Gray, John, & Atwood, 1993; John, 1990, 1996). Gray & Boehm-Davis (2000) have also explored this concept, using the term microstrategies, which appear to be similar to templates. The organization and size of method compilation is still unknown, as it has been demonstrated that the variance of task constraints has a direct impact on the architectural construction of microstrategies (Gray & Boehm-davis, 2000; Vera, Tollinger, Eng, Lewis, & Howes, 2005; John & Kieras, 1996, as cited in Vera et al., 2005). However, as Vera et al. (2005) point out, Gray & Boehm-Davis (2000) work on this concept elevates it from a descriptive tool in CPM GOMS (templates), to an actual theory of how the cognitive system interacts with the environment (micro strategies). Vera et al. (2005) also suggest using smaller units, called Architectural Process Cascades, for describing these interactions.

Our goal was to develop a simple interface to test for the presence and influence of individual cognitive microstrategy function and presence, while acknowledging but not focusing on the inner workings of methods therefore allowing for the treatment of methods and microstrategies as identical. As the structure and function of methods is believed to impact the functioning of higher level processing, our design was developed to apply the method of testing microstrategies presented by Gray & Boehm-Davis (2000) to answer questions pertaining to presence and function of macro cognition. The use of SGOMS ACT-R models within this study allows us to predict cognitive microstrategy differences at various points within a task, and to test and compare results with the performance of human participants under the same conditions.

The Macro Cognitive Architecture

The macro-architecture hypothesis proposes the addition of macro-cognitive and micro-cognitive bands to the structure of the previous system levels of Newell's theory of cognitive architecture (Newell, 1990; West & MacDougall, 2014). The term macro cognition refers to the cognition involved within complex, unpredictable, real world occurrences, and in contrast micro cognition refers to cognitive functions as visible in lab based cognitive psychology experiments (Cacciabue & Hollnagel, 1995; West, 2003; West & Nagy, 2007; West & MacDougall, 2014). The macro-cognitive and micro-cognitive bands are based upon the proposed division of cognitive science, in which micro-cognition refers to cognition as studied within controlled environments, and macro-cognition referring to cognition in real world, dynamic and complex environments (Cacciabue & Hollnagel, 1995; West & Nagy, 2007; West & MacDougall, 2014). Implementation of this theory would result in the cognitive band being renamed as the micro-cognitive band, and the insertion of the macro-cognitive band between the micro-cognitive band and the rational band (Newell, 1990; West & MacDougall, 2014). Macro cognitive architectures, in theory, act as a system for organizing the micro-cognitive architecture, so that the agent may competently address real world, unpredictable and context specific tasks in an efficient manner. It also suggests that people most often use and reuse pre-existing micro-cognitive architectures including methods and unit tasks in the same way, addressing generic issues that can reoccur across tasks (West & Nagy, 2007; West & Pronovost, 2009; West & Somers, 2011).

Macro-cognitive architectures are specifically designed to systematically organize the micro architectures (e.g., ACT-R, SOAR, EPIC, etc.) for agent performance during specific problem classes that occur in real-world environments and situations, across all types of expertise. A macro-cognitive architecture that may be used independently or implemented in conjunction with a micro-architecture, resulting in the production of realistic human constraints

and reaction times (West & Nagy, 2007, West & Provonost, 2009, West & MacDougall, 2014). An example of a macro and micro architecture relationship would be the macro-cognitive architecture of problem space search that is implemented within the SOAR micro architecture (Laird, 2012; Newell, 1990). Lebiere and Best (2009) suggest strategy evolution and discovery levels as architectural enhancements to modify micro cognitive architectures such as ACT-R to handle context demands of macro cognitive environments flexibly and efficiently. Another example, Sociotechnical GOMS (SGOMS), is a macro architecture with the purpose of modeling how experts of any domain handle and thrive within unpredictable, real-world environments that include interruptions, unexpected occurrences, re-planning, and other unpredictable agents (West, 2003; West & MacDougall, 2014; West & Nagy, 2007; West and Somers, 2011).

ACT-R and SGOMS

In order to better explore the relationship between low level processes and higher level processes in cognition, we developed a model using ACT-R (Anderson & Lebiere, 1998). In accordance with ACT-R theory and architecture, we interpret methods as a series of productions, executed in serial, directing in parallel, perceptual and motor operations. ACT-R is a micro-cognitive architecture and theory of cognition with the purpose of using a unified theory to address, explain and solve real world questions and issues in cognition using evidence and research developed from the fields of Neuroscience and Psychology (Anderson & Lebiere, 1998; Anderson et al., 2004; West & Provonost, 2009). ACT-R has been most frequently in two dominant areas of cognition research as a tool to develop models of data integration from neuroscience and neural imaging, and to model human behaviour and skill acquisition and application within a variety of complex environments and problem sets (Anderson et al., 2004). The structure of ACT-R is comprised of three main components, including a communication

system, a declarative memory system, and a procedure memory (Anderson et al., 2004; West & Nagy, 2007). The communication system is designed using chunks and buffers, chunks are stored within the declarative memory system, and the procedural memory is a pattern-matching production system (Anderson et al., 2004; West & Nagy, 2007).

Due to the restrictiveness of psychological research environments and the micro cognitive nature of the results produced, micro cognitive architectures such as ACT-R may not always provide sufficient information as to how cognition would function in macro cognitive environments and tasks (Fu et al., 2006; West & Macdougall, 2014; West & Pronovost, 2009). This is further demonstrated by the structural difficulty that ACT-R and many other architectures face with aspects of macro cognitive environments, such as interruption (West & Macdougall, 2014; West & Pronovost, 2009; West & Somers, 2011). To elevate the ACT-R architecture to better handle macro cognitive tasks, SGOMS ACT-R may be used to represent and explore the innately human abilities to function optimally under contexts that involve changes, multiple agents, interruptions, and other aspects of real world life. Like GOMS, SGOMS has been developped to model expert performance, therefore models are created with a certain level of prior knowledge assumed towards task performance and goal execution. SGOMS architecture runs using both SGOMS: ACT-R and Python ACT-R (West & Macdougall, 2014). The structure of SGOMS consists of Methods, Unit Tasks, Planning Units, and a high-level cycle of operations. Methods, as mentioned, are considered small, task-specific actions that may be reused within the same task and other, similar tasks. The lowest systematic level of SGOMS is the Unit Task, which consist of either singular or multiple specific Methods used for sub goal completion, and are highly specific to the stage of goal completion in which they are designed. Due to the size of Unit Tasks and their cognitive low-level sub goal specificity, they are task-

specific and uninterruptable. The structure of a Unit Task within the cognitive architecture acts as a means of reducing the task into smaller units, allowing for the task to be completed in a more complex context without overloading the agent system . Planning Units are interruptable control structures that contain and organize goal-specific Unit Tasks, and they may function singularly or in conjunction with other Planning Units for goal completion. Planning Units may be ordered, situationally dependent, or random, which allows them to function in macro cognitive environments with flexibility while tracking task stages and completion level (Stewart & West, 2006; West & Nagy, 2007; West & Pronovost, 2009). Ordered planning units consist of an ordered list of unit tasks, and situated planning units contain a set of unit tasks that are cued by the environment. Re-use is an important principle within GOMS, as it is assumed that stored representation of operators, methods and unit tasks are reused. As such, the SGOMS architecture as implemented within ACT-R uses the same operators, methods and unit tasks for different ordered and situated planning units. For the purposes of our experiment, our focus was regarding the two most prominent organizations of planning units in SGOMS: ordered, and situationally dependent (or *situated*). The cycle of operations that exists at the executive level depicts the compilation and order that the planning units and unit tasks are executed to complete the goal state (West & Pronovost, 2009).

The SGOMS architecture allows for fluid task interruption and resumption, thus requiring a mechanism for keeping a record of what stage one is at within the task, as well as an intelligent interruption mechanism. To create the SGOMS ACT-R model, ACT-R code describing the planning units, unit tasks and methods are placed within an SGOMS/ACT-R template that provides the functionality described above. The time of the task as performed by the SGOMS model will not produce the same rate of performance as the optimal ACT-R model. However,

without the subcomponents of the SGOMS system, our theory suggests that the optimal model in ACT-R would not function optimally within real world environments and tasks that are subject to unexpeted occurances and interruptions. This is to suggest that human cognition may only function at the same level as the optimal model under constrained, uncommon and artificial environments, and within most other conditions, they would implement an SGOMS based approach to task completion. Our hypothesis is that if it is the case that unit tasks are organized in the ways predicted by SGOMS, it will be possible to use empirically gathered reaction times to detect evidence for the structure in the SGOMS template. To test this, we developed an experimental paradigm called the Alphabet Expert Task. To compare the performance of the human participants to both the optimal model and the SGOMS model, one model of each structure performing the same task was developped in order to provide the most comprehensive and clear results.

Method

Subjects

Two of the researchers involved within the development of the experiment volunteered to perform the experiment. Both of the participants were male, graduate students completing masters degrees in Cognitive Science at Carleton University. The participants will henceforth be referred to as *NN* and *FK* respectively. The participants did not have any prior experience or knowledge pertaining to the SGOMS-ACT-R model at the time of the experiment. Due to the nature of the experimental task, supplementry information pertaining to video game playing habbits and experience was collected from the participants. It was found that participant FK had experience playing video games on a regular basis, where as participant NN had no experience playing video games in the past.

Procedure

As our goal was to analyze the functionality and presence of methods within a task performance structure, we required an extremely simple experimental task to control for cognitive variation amung methods and unit tasks. The Alphabet Expert task was designed specifically to require participants to perform the task while limiting response method variability. The purpose of the experimental design was to require clear and specific unit task structures for each trial, therefore limiting method variability. The design of the experiment required participants to provide the same structure of response with different contents based upon the type of stimuli presented upon the screen. The content of the response was either known to the participant based upon the previous stimuli, or unknown to the participant therefore requiring the participant to wait until the next prompt on the screen appeared.

The experiment was introduced in stages to the participants through practice, ensuring that they felt a level of familiarity and expertise of unit task completion prior to the introduction of more complex sequences determined by planning units. The experiment stages which involved participant practice of the unit tasks and the planning units were completed by the participants in their home enviornments using their personal laptops. During the first stage of the experiment, FK and NN were required to learn the three unit task structures (see Figure 1. for a visual representation of the structures) that would be used within each of the planning units. Within the unit task, there were one second intervals between response and prompt. Each unit task was learned seperately, and was practiced until the participants evaluated themselves as having attained their best speed and accuracy within the task.

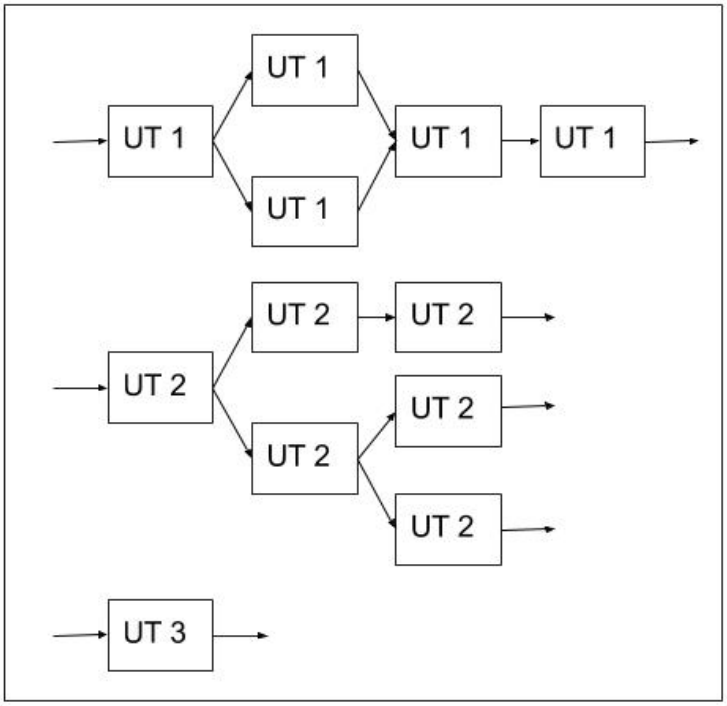

Figure 1. Structure of the three possible unit tasks. Each box represents a different response for the possible prompt.

After the participants practiced each of the three unit task structures until they felt that they had attained a level of personal expertise, they began the second stage of the experiment which involved memorizing the three planning unit structures. During practice, each planning unit was called by using a specific four-letter code inserted by the participant. The planning units each contained the three unit tasks that were learned by the participants during stage one, in different orders. Two of the three planning units were structured as ordered planning units, using each of the three unit tasks in the same order each time that was specific to the planning unit. The third was constructed as a situated planning unit that used each of the three unit tasks in a random order that changed with each commencement. Table 1 further demonstrates the structure of the three planning units used within the Alphabet Expert simulation experiment.

Planning Unit	Unit Task order
Ordered Planning Unit 1	Unit Task 1
	Unit Task 2
	Unit Task 3
Ordered Planning Unit 2	Unit Task 2
	Unit Task 3
	Unit Task 1
Situated Planning Unit 3	Unit Task ?
	Unit Task ?
	Unit Task ?

Table 1. Planning unit structures.

The third stage required the participants to practice the Alphabet Expert task in it's entirety, combining the tasks together with one second intervals between each trial commencement and removing the one second intervals between trials. Under the condition of an ordered planning unit being presented, the simulation would provide the participants with a planning unit specific code that signalled which planning unit was starting. Due to the practicing of the planning units, the participants had prior knowledge of what unit task would be first within the planning unit as well as what response to input based upon the code perceived. The next step in the process required the input of the code that corresponded with the first unit task after the planning unit prompt, and the unit task simulation would begin. After successful completion, the simulation presented the planning unit specific code in the same fashion as it had at the beginning of the planning unit, requiring the participant to type in the response associated with

the second, repeating the process again until successful completion of the third unit task in the ordered planning unit. The process was the same within the situated planning unit, except that the situated planning unit presented a random order of the three unit tasks, thus requiring the participants to wait until they perceive the code corresponding to the unit task that would begin, and respond accordingly. After participants practiced the full version of the Alphabet Expert task to the level they felt was most efficient and accurate, the data of their subsequent simulations of the task were collected for analysis.

The Models

The SGOMS model was developped using Python ACT-R to create an agent capable of performing the identical task defined above within the human procedure. The code was written specifically for the unit tasks, planning units, and perceptual/motor methods required for successful completion of the task and fitted to a developped template tailored to the SGOMS-ACT-R architecture (see Appendix A for the SGOMS model code developped). Within the structure of the unit tasks, the productions fire in the same fashion that they occur within the optimal model, as they also have no overhead function management. The macro structure of the architecture is developped through the organization of the planning unit systems and unit task organization.

The optimal model code was structured in keeping with a micro architecture, in which the planning unit was stored within the working memory buffer. The production for code selection was dependent upon the information provided by the working memory buffer, and the visual buffer, which held the current prompt code perceived. When the information was matched to a production, it would fire with the correct response to the stimuli. The optimal model was developped to have the same perceptual/motor system as the SGOMS model.

Results

The trials within the experiment were divided into different categories, in order to allow measurement of the response times of the human participants in comparison to the models of the experiment. The response categories are represented in Table 2. During instances of known response conditions the perceptual/motor method was assumed to be singular in human participants, suggesting that the goal of entering the next response as fast as possible would rely on knowledge of past prompt and predicted prompt, ommitting the need to apply extra productions to treat the predicted prompt as unknown.

Term	Description
Unknown Mid UT	A response in the middle of a unit task that is not known until the code is perceived
Known Mid UT	A response in the middle of a unit task that is determined by the previous response.
Unknown PU-O Start	The response to the code to begin an ordered planning unit. This response cannot be determined by the previous response.
Known PU-O Mid	The response to the code to begin the second or third unit task in an ordered planning unit. This response can be determined by the previous response.
Known PU-S Mid	The response to the code to begin the second or third unit task in an unordered planning unit. This response can be determined by the previous response.
Known First UT	The response to the first unit task in an ordered planning unit. This response can be determined by the previous response.
Unknown First UT	The response to the first unit task in a situated planning unit. This response cannot be determined by the previous response.
Unknown PU-S Start	The response to the code to begin a situated planning unit. This response cannot be determined by the previous response.

Table 2. Response category descriptions.

Within the context of unknown response conditions, it was assumed that the participant would require two perceptual/motor methods, as they would need to both identify the code (as it is random, therefore unpredictable), and then enter the corresponding code. For example, within response category Unknown Unit Task Middle (Unknown Mid UT), Unknown refers to the conditions that the participant was required to see the prompt prior to inserting the corresponding response, and within Known Unit Task Middle (Known Mid UT), Known refers to the condition where the participant was aware of the next prompt response based upon the previous prompt and response. The methods assumed in participants were subsequently designed and integrated into the model structure of both the optimal model and SGOMS model. The design of the methods inside the models did not differ. Unknown and Known analysis points during trial performances were subsequently used to form the baseline to fit the results from the experiment.

During the analysis of results, trials with errors and outlier response times that were more than two standard deviations from the mean were removed from the data. Results were then equalized across participant scores by subtracting the differences between mean response times under both conditions. External method variation between participants included FK using one finger on each hand to input responses, and having had significant experience and practice playing video games of a similar nature prior to the experiment, versus NN having no experience with video game play. The correction was applied to all trials under both Known and Unknown conditions. The SGOMS and optimal model were developped in accordance with the condition results. The resulting data suggests the reaction times withhin the two conditions, Known and Unknown, were consistent across trials and participant performance. The perceptual/motor time estimates specific to condition in all trial setting and response points were integrated into the model performance. The data implemented into the model structure produced identical methods

within both models, ensuring that the sole difference between the SGOMS model structure and the optimal model structure was the organization of the productions (the macro cognitive structure) and the amount of productions as determined prior to experiment commencement. Figure 2 depicts the results of the experiment with 0.05 confidence intervals for the collected data.

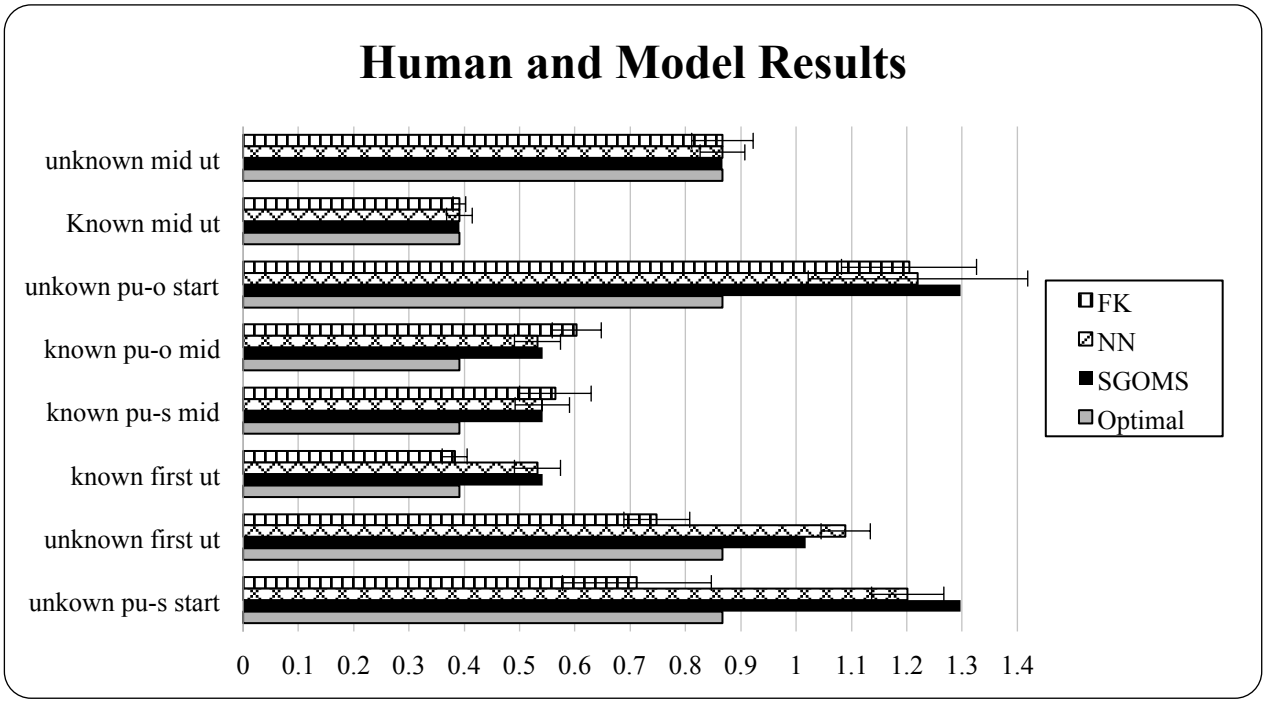

Figure 2. Human and model results.

Within response categories pertaining to ordered planning units, the results suggest that the performance of the SGOMS model, participant FK and participant NN were extremely similar in response times, differing only under the response category of Known First Unit Task (Known First UT). Within this condition, the first unit task within the ordered planning unit was consistently the same, and predictable by the subject performing the task. Human participant NN matched the SGOMS model under this condition, and human participant FK matched the optimal model under this condition. Within response categories pertaining to situated planning units

(containing a random order of the three unit tasks), the human participants differed at a greater degree. In most cases, NN performed at a similar rate to the SGOMS model, and depending on response category, FK performed similarly to either the SGOMS model and the optimal model. Participants reported experimenting with different method strategies to learn how to perform the trials of both conditions with the utmost efficiency, with special focus placed upon the situated planning unit. FK reported experimenting with different strategies in order to increase the speed of response times, which produced the effects of faster responses for the first prompt of the situated planning unit (Known PU-S start). The focus participants placed upon reorganizing their understanding of the situated planning units resulted in significant difference in times. The differences in participant response times under the situated planning unit conditions suggest that the structure of the situated planning unit may stand out more, causing participants to focus more on the optimization of performance under these conditions.

Results obtained suggest that it is less common and effective to assume the optimal model performance as the default method of human cognitive task performance. As such, our results support the SGOMS macro cognitive architecture as a more realistic default cognitive approach to task completion. This is not to say that people cannot learn the optimal way to perform a task, as we recognize cognitive variance at different levels of cognition within individual cognitive task structure and execution. Our argument, supplemented by the results suggests that the optimal method of task performance may not be the most effective or common method of approaching and completing real world tasks, therefore should not necessarily be the default starting point within the development of cognitive models.

The results of the human-model comparison show clear support for the SGOMS macro architecture, as it was comparatively a better fit for the majority of human response times. The

SGOMS model fit six out of six trial response times for participant NN, and three out of six trial response times for participant FK. Within the instances of the SGOMS model not matching participant FK's responses times, the optimal model was a reasonable match to account for the differences. The results support our primary research hypothesis that the SGOMS macro architecture as a default system within cognitive structure of task performance. The secondary research hypothesis is also supported by the experimental results, suggesting that the optimal model might only be converted to if the task involves no interruptions, and if there is sufficient motivation for critically analyzing and practicing the task with the purposes of optimizing performance to the greatest degree.

Discussion

Models are intended to act as representations that may be used to analyze and understand the true entity of their design, with representation abstracting from unnecessary components while retaining the essential features(Cooper & Fox, 2002). This means that cognitive models are intended to act as representations of the mind, in it's truest form, to the most complete and faithful degree possible (Anderson & Lebiere, 1998; Anderson et al., 2004; Cooper & Fox, 2002). Given that it is most often the case that human cognition occurs in non-laboratory, uncontrolled macro cognitive settings, it is important to develop models that are designed to represent cognition in a way that best represents strategies and methods that are used in naturalistic settings (Cooper & Fox, 2002; Lebiere & Best, 2009; West & Pronovost, 2009; West & Somers, 2011). The micro cognitive experimental methods developed by Gray & Boehm-Davis to detect microstrategies (2000) were successfully implemented within our experiment, demonstrating the usefulness of studying the method level to detect the presence and function of a macro cognitive architecture.

The SGOMS macro cognitive architecture is designed to describe how expert knowledge is used and executed in real world environments, which is most often full of interruptions, replanning and unexpected occurrences. We argue this to be a more realistic representation of human cognition, therefore conclude that the SGOMS model is more likely to be the default model of cognition in expert task performance in macro cognitive environments than the optimal model. The data patterns collected and analyzed within our experiment suggest that the default system is primarily used, until modified to suit the contextual and task demands. We suggest that the optimal model may be converted to as a method of task execution and completion after the initial SGOMS structure approach limited to micro cognitive environments. This is consistent with the argument that macro cognitive structures act to organize and control micro cognitive structures to better navigate real world environments (Lebiere & Best, 2009; West & Macdougall, 2014; West & Somers, 2011).

Though our study produced statistically significant results supporting the function and influence of a macro cognitive architecture upon a micro cognitive architecture, our findings are limited due to the number of participants and the subsequent amount of human participant data collected. Future studies might consider replicating the experimental methodology on a greater sample size to better generalize data pertaining to human performance and model performance comparison. Due to the level of participant unfamiliarity prior to the practice phase required and level of practice necessary, future research might also consider developing an appealing game task that many participants would enjoy performing at length on a platform that is extremely accessible, such as a mobile phone. This would encourage participants to extend the practice phase to a level of expertise, facilitate the participation of a greater sample size, as well as ensure model replicability to the same level as produced within our experiment.

References

Anderson, J. R. (2002). Spanning seven orders of magnitude: A challenge for cognitive modeling. *Cognitive Science*, *26*(1), 85–112. https://doi.org/10.1016/S0364-0213(01)00062-3

Anderson, J. R., Bothell, D., Byrne, M., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, *111*(4), 1036–1060. https://doi.org/10.1037/0033-295X.111.4.1036

Card, S. K., Newell, A., & Moran, T. P. (1983). The psychology of human-computer interaction. https://doi.org/10.1016/0003-6870(84)90205-9

Cooper, R. P., & Fox, J. (2002). *Modelling High-Level Cognitive Processes*, Erlbaum.

Fu, W. T., Bothell, D., Douglass, S., Haimson, C., Sohn, M. H., & Anderson, J. (2006). Toward a real-time model-based training system. *Interacting with Computers*, *18*(6), 1215–1241. https://doi.org/10.1016/j.intcom.2006.07.011

Gray, W. D., & Boehm-Davis, D. A. (2000). Milliseconds Matter: An Introduction to Microstrategies and to Their Use in Describing and Predicting Interactive Behavior. *Journal of Experimental Psychology*, *6*(4), 322–335. https://doi.org/10.1037//1076-898X.6.4.322

Lebiere, C., & Best, B. J. (2009). From microcognition to macrocognition: Archtectural support for adversarial behavior. *Ournal of Cognitive Engineering and Decision Making*, *3*(2), 176–193. https://doi.org/10.1518/155534309X441844.

Meyer, D. E., & Kieras, D. E. (1997). A computational theory of executive control processes and human multiple-task processes and human multiple-task performance: Part 1. Basic Mechanisms. Psychological Review, 104, 3-65.

Newell, A. (1973). You can't play 20 questions with nature and win: Projective comments on the

papers of this symposium. *Visual Information Processing*. Academic Press.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Siegler, R. S. (2007). Cognitive variability. *Developmental Science*, *10*(1), 104–109. https://doi.org/10.1111/j.1467-7687.2007.00571.x

Stewart, T. C., & West, R. L. (2006). Deconstructing ACT-R. *Proceedings of the Seventh International Conference on Cognitive Modeling*, *1*(2), 298–303. Retrieved from http://act-r.psy.cmu.edu/papers/641/stewartPaper.pdf

Vera, A., Tollinger, I., Eng, K., Lewis, R., & Howes, A. (2005). Architectural building blocks as the locus of adaptive behavior selection. *Proceedings of the 27th Annual Meeting of the Cognitive Science Society*, 2295–2301.

West, R. (2013). The Macro Architecture Hypothesis: A Theoretical Framework for Integrated Cognition. *In Proceedings of AAAI Fall Symposium Series,* North America.

West, R. L., & Macdougall, K. (2014). The macro-architecture hypothesis: Modifying Newell's system levels to include macro-cognition. *Biologically Inspired Cognitive Architectures*, *8*, 138–147. https://doi.org/10.1016/j.bica.2014.03.009

West, R. L., & Nagy, G. (2007). Using GOMS for Modeling Routine Tasks Within Complex Sociotechnical Systems: Connecting Macrocognitive Models to Microcognition. *Journal of Cognitive Engineering and Decision Making*, *1*(2), 186–211. https://doi.org/10.1518/155534307X232848

West, R. L., & Pronovost, S. (2009). Modeling SGOMS in ACT-R:Linking Macro- and Microcognition. *Journal of Cognitive Engineering and Decision Making*, *3*(2), 194–207. https://doi.org/10.1518/155534309X441853

West, R. L., & Somers, S. (2011). Scaling up from Micro Cognition to Macro Cognition : Using

SGOMS to build Macro Cognitive Models of Sociotechnical Work in ACT-R. *Proceedings of the 33rd Annual Cognitive Science Society*, 1788–1793. Retrieved from http://palm.mindmodeling.org/cogsci2011/papers/0397/paper0397.pdf

Appendix A

SGOMS model code written in Python ACT-R: Alphabet Expert

```
import sys
import ccm
from random import randrange, uniform
log = ccm.log()
log=ccm.log(html=True)
from ccm.lib.actr import *



# Create Environment


class MyEnvironment(ccm.Model):


    response = ccm.Model(isa='response', state='none',
salience=0.99)
    display = ccm.Model(isa='display', state='structura',
salience=0.99)
    response_entered = ccm.Model(isa='response_entered',
state='no', salience=0.99)


    motor_finst = ccm.Model(isa='motor_finst',
state='re_set')
```

```
# Create a Motor Module

class MotorModule(ccm.Model):  ### defines actions on the
environment

# change_state is a generic action that changes the state
slot of any object
# disadvantages (1) yield #time is always the same (2) cannot
use for parallel actions

    def change_state_slow(self, env_object, slot_value):
        yield 1
        x = eval('self.parent.parent.' + env_object)
        x.state = slot_value
        print env_object
        print slot_value
        self.parent.parent.motor_finst.state =
'change_state_slow'

    def change_state_fast(self, env_object, slot_value):
        yield 3
        x = eval('self.parent.parent.' + env_object)
        x.state = slot_value
```

```
            print env_object
            print slot_value
            self.parent.parent.motor_finst.state =
    'change_state_fast'

        def vision_slow(self):
            yield 5
            print 'target identified'
            self.parent.parent.motor_finst.state = 'vision_slow'
            self.parent.visual_buffer = 'spotted'

        def vision_fast(self):
            yield 2
            print 'target spotted'
            self.parent.parent.motor_finst.state = 'vision_fast'



        def motor_finst_reset(self):
            self.parent.parent.motor_finst.state = 're_set'

    class MyAgent(ACTR):

    # BUFFERS
```

```
    # goal system buffers
    b_context = Buffer()
    b_plan_unit = Buffer()
    b_unit_task = Buffer()
    b_method = Buffer()
    b_operator = Buffer()

    b_emotion = Buffer()

    # module buffers
    b_DM = Buffer()
    b_motor = Buffer()
    visual_buffer=Buffer()
    b_image = Buffer()
    b_focus = Buffer()

    # initial buffer contents
    b_context.set('status:unoccupied have_plan:no
planning_unit:none')
    b_emotion.set('threat:ok')
    b_plan_unit.set('planning_unit:P cuelag:P cue:P
unit_task:P state:P ptype:P')
```

```
# MODULES (import modules into agent, connect to buffers, and
add initial content)

    # vision module - from CCM suite
    vision_module=SOSVision(visual_buffer,delay=.085)

    # motor module - defined above
    motor = MotorModule(b_motor)

    # declarative memory module - from CCM suite
    DM = Memory(b_DM)

    # initial memory contents
    DM.add('planning_unit:structura         cuelag:none
cue:start           unit_task:primus')
    DM.add('planning_unit:structura         cuelag:start
cue:primus          unit_task:astructus')
    DM.add('planning_unit:structura         cuelag:primus
cue:astructus       unit_task:situs')
    DM.add('planning_unit:structura         cuelag:astructus
cue:situs           unit_task:finished')
```

```
    DM.add('planning_unit:structura2        cuelag:none
cue:start          unit_task:astructus')
    DM.add('planning_unit:structura2        cuelag:start
cue:astructus      unit_task:situs')
    DM.add('planning_unit:structura2        cuelag:astructus
cue:situs          unit_task:lectus')
    DM.add('planning_unit:structura2        cuelag:situs
cue:lectus         unit_task:finished')

    DM.add('planning_unit:dominus         cuelag:none
cue:start          unit_task:lectus')
    DM.add('planning_unit:dominus         cuelag:start
cue:lectus         unit_task:fiat')
    DM.add('planning_unit:dominus         cuelag:lectus
cue:fiat           unit_task:finished')


    ########### create productions for choosing planning
units ###########

    ## these productions are the highest level of SGOMS and
fire off the context buffer
```

```
    ## they can take any ACT-R form (one production or more)
but must eventually call a planning unit and update the
context buffer


    def run_dominus(b_context='status:unoccupied
planning_unit:none'):

b_plan_unit.modify(planning_unit='dominus',cuelag='none',cue=
'start',unit_task='lectus',state='begin_sequence',ptype='orde
red')
        b_context.modify(status='occupied')
        print
'DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDD'


    def run_structura(b_context='status:unoccupied
planning_unit:structura'):

b_plan_unit.modify(planning_unit='structura',cuelag='none',cu
e='start',unit_task='primus',state='begin_sequence',ptype='or
dered')
        b_context.modify(status='occupied')
```

```
        print
'1111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111'

    def run_structura2(b_context='status:unoccupied
planning_unit:structura2'):

b_plan_unit.modify(planning_unit='structura2',cuelag='none',c
ue='start',unit_task='primus',state='begin_sequence',ptype='o
rdered')
        b_context.modify(status='occupied')
        print
'2222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222'



#########################################################
########## unit task management productions ###########
#########################################################

######################### these set up whether it will be an
ordered or a situated planning unit
```

```
    def
setup_situated_planning_unit(b_plan_unit='planning_unit:?plan
ning_unit state:begin_situated ptype:unordered'):
        b_unit_task.set('state:start type:unordered')
        b_plan_unit.modify(state='running')
        print 'begin situated planning unit = ',
planning_unit

    def
setup_ordered_planning_unit(b_plan_unit='planning_unit:?plann
ing_unit cuelag:?cuelag cue:?cue unit_task:?unit_task
state:begin_sequence ptype:ordered'):
        b_unit_task.set('unit_task:?unit_task state:start
type:ordered')
        b_plan_unit.modify(state='running')
        print 'begin orderdered planning unit = ',
planning_unit

######################### these manage the sequence if it is
an ordered planning unit

    def
request_next_unit_task(b_plan_unit='planning_unit:?planning_u
```

```
nit cuelag:?cuelag cue:?cue unit_task:?unit_task
state:running',


b_unit_task='unit_task:?unit_task state:finished
type:ordered'):
        DM.request('planning_unit:?planning_unit
cue:?unit_task unit_task:? cuelag:?cue')
        b_plan_unit.modify(state='retrieve')
        print ' finished unit task = ', unit_task


    def retrieve_next_unit_task(b_plan_unit='state:retrieve',

b_DM='planning_unit:?planning_unit cuelag:?cuelag
cue:?cue!finished unit_task:?unit_task'):


b_plan_unit.modify(planning_unit=planning_unit,cuelag=cuelag,
cue=cue,unit_task=unit_task,state='running')
        b_unit_task.set('unit_task:?unit_task state:start
type:ordered')
        print ' unit_task = ', unit_task


########################## these manage planning units that
are finished ################## PROBLEM HERE, WANT TO DO
DOMINUS
```

```
    def
last_unit_task_ordered_plan(b_context='have_plan:yes',

b_plan_unit='planning_unit:?planning_unit',

b_unit_task='unit_task:finished state:start type:ordered',
                                    utility=5):  # high
priority if a plan was generated in the planning unit
        print 'finished planning unit=',planning_unit
        b_unit_task.set('stop')
        b_context.modify(status='unoccupied', have_plan='no')
# have plan always needs to be re-set to no

    def
last_unit_task_ordered_noplan(b_plan_unit='planning_unit:?pla
nning_unit',

b_unit_task='unit_task:finished state:start type:ordered',
                                      utility=1):  # by
default no plan is generated
        print 'finished planning unit=',planning_unit
        b_unit_task.set('stop')
```

```
        b_context.modify(status='unoccupied',
planning_unit='none', have_plan='no') # have plan always
needs to be re-set to no

    def
interupted_unit_task(b_plan_unit='planning_unit:?planning_uni
t',

b_unit_task='unit_task:interupted state:interupted
type:?type'):
        print 'interupting planning unit=',planning_unit
        print planning_unit
        b_unit_task.set('state:end')
        b_context.modify(status='interupted')



##########################################################
##################### unit task productions ###########
##########################################################

## unit task fiat

 ## add condition to fire to this production
```

```
    def fiat_unordered(b_unit_task='unit_task:? state:start
type:unordered',
                       display='state:zzz'):
        b_unit_task.set('unit_task:fiat state:begin
type:unordered')                    #### competes with other
unit tasks to fire
        print 'start unit task fiat unordered'

    def fiat_ordered(b_unit_task='unit_task:fiat state:start
type:ordered'): ### this unit task is chosen to fire by
planning unit
        b_unit_task.modify(state='begin')
        print 'start unit task fiat ordered'

    ## the first production in the unit task must begin in
this way
    def fiat_start(b_unit_task='unit_task:fiat state:begin
type:?type'):
        b_unit_task.set('unit_task:fiat state:running
type:?type')
        b_focus.set('start')
        print 'starting unit task fiat now'

    ## body of the unit task
```

```
    def fiat_known_response(b_unit_task='unit_task:fiat
state:running type:?type',
                            b_focus='start'):
        b_context.modify(planning_unit='structura',
have_plan='yes')
        b_focus.set('done')
        b_method.set('state:finished') # this is to finish
the body but its not so good, finishing the mehtod should be
moved into the body
        b_unit_task.modify(state='end')  ## this line ends
the unit task
        print 'I known which planning unit to use -
structura'


    def fiat_known_response2(b_unit_task='unit_task:fiat
state:running type:?type',
                            b_focus='start'):
        b_context.modify(planning_unit='structura2',
have_plan='yes')
        b_focus.set('done')
        b_method.set('state:finished') # this is to finish
the body but its not so good, finishing the mehtod should be
moved into the body
```

```
        b_unit_task.modify(state='end')  ## this line ends
the unit task
        print 'I known which planning unit to use -
structura2'


    ## finishing the unit task
    def fiat_finished_ordered(b_method='state:finished',  ##
this line assumes waiting for the last method to finish
                              b_unit_task='unit_task:fiat
state:end type:ordered',
                              b_emotion='threat:ok'):
        print 'finished unit task fiat - ordered'
        b_unit_task.set('unit_task:fiat state:finished
type:ordered')


    def fiat_finished_unordered(b_method='state:finished',
                                b_unit_task='unit_task:X
state:end type:unordered',
                                b_plan_unit='ptype:unordered',
                                b_emotion='threat:ok'):
        print 'finished unit task fiat - unordered'
        b_unit_task.set('unit_task:fiat state:start
type:unordered')
```

```
    def
fiat_interupt_planning_unit(b_method='state:finished',
                                    b_unit_task='unit_task:fiat
state:end type:?type',
                                    b_emotion='threat:high'):
        print 'finished unit task fiat - interupting planning
unit'
        b_unit_task.set('unit_task:interupted
state:interupted type:?type')


################## UNIT TASK: PRIMUS
    ##### PRIMUS CONDITIONS #####
    ## add condition to fire to this production
    ## Primus is an "ORDERED UNIT TASK" - not to be confused
with the
    ## ordered stated below. (should change?)
    def primus_unordered(b_unit_task='unit_task:? state:start
type:unordered',
                     display='state:zzz'):
        b_unit_task.set('unit_task:primus state:begin
type:unordered')                    #### competes with other
unit tasks to fire
        print 'start unit task primus unordered'
```

```
    def primus_ordered(b_unit_task='unit_task:primus
state:start type:ordered'): ### this unit task is chosen to
fire by planning unit
        b_unit_task.modify(state='begin')
        print 'start unit task primus ordered'
    ##### PRIMUS START #####
    ### STEP 1:
    ## the first production in the unit task MUST begin in
this way
    def primus_start(b_unit_task='unit_task:primus
state:begin type:?type'):
        b_unit_task.set('unit_task:primus state:running
type:?type')
        b_focus.set('start')
        print 'starting unit task primus now'
    ##### PRIMUS BODY #####
    ### STEP 2:
    ## body of the unit task
    def primus_known_response(b_unit_task='unit_task:primus
state:running type:?type',
                    b_focus='start'):
        b_method.set('method:known_response target:response
content:primus1 state:start')############## here
        b_focus.set('done')
```

```
        b_unit_task.modify(state='end')  ## this line ends
the unit task
        print 'I know the response'
              # change name here - maybe include ANS state
print?
    ##### PRIMUS END #####
    ### STEP 3:
    ## finishing the unit task; the end state depends on the
    ## init state
    def primus_finished_ordered(b_method='state:finished',
## this line assumes waiting for the last method to finish
                            b_unit_task='unit_task:primus
state:end type:ordered',
                            b_emotion='threat:ok'):
        print 'finished unit task primus - ordered'
        b_unit_task.set('unit_task:primus state:finished
type:ordered')

    def primus_finished_unordered(b_method='state:finished',
                              b_unit_task='unit_task:X
state:end type:unordered',
                              b_plan_unit='ptype:unordered',
                              b_emotion='threat:ok'):
        print 'finished unit task primus - unordered'
```

```
        b_unit_task.set('unit_task:primus state:start
type:unordered')
    ##### PRIMUS INTERUPT #####
    ### IF STEP:
    ## this will run if Primus faces an interuption while
running
    def
primus_interupt_planning_unit(b_method='state:finished',

b_unit_task='unit_task:primus state:end type:?type',
                                  b_emotion='threat:high'):
        print 'finished unit task primus - interupting
planning unit'
        b_unit_task.set('unit_task:interupted
state:interupted type:?type')



################ UNIT TASK: LECTUS
    ## UT Lectus is implemented in all cases where there are
    ## two possible responses. This required that UT Lectus
is
    ## able to account for:
    ##      - perceiving the code (in order to know which
```

```
    ##      response is appropriate, it must first "get" the
code)
    ##      - passing the correct response based upon the
code perceived
    ##      to the method that requests the information

    ##### LECTUS CONDITIONS #####
    ## add condition to fire to this production
    def lectus_unordered(b_unit_task='unit_task:? state:start
type:unordered',
                         display='state:zzz'):
        b_unit_task.set('unit_task:lectus state:begin
type:unordered')                    #### competes with other
unit tasks to fire
        print 'start unit task lectus unordered'

    def lectus_ordered(b_unit_task='unit_task:lectus
state:start type:ordered'): ### this unit task is chosen to
fire by planning unit
        b_unit_task.modify(state='begin')
        print 'start unit task lectus ordered'
    ##### LECTUS START: #####
    ### STEP 1:
```

```
    ## the first production in the unit task must begin in
this way
    def lectus_start(b_unit_task='unit_task:lectus
state:begin type:?type'):
        b_unit_task.set('unit_task:lectus state:running
type:?type')
        b_focus.set('start')
        print 'starting unit task lectus now'
    ##### LECTUS BODY: #####
    ### STEP 2: PERCEIVE CODE
    ## body of the unit task - this initializes the methods
of fast and slow
    ## code retrieval (see methods)
    def lectus_get_code(b_unit_task='unit_task:lectus
state:running type:?type',
                        b_focus='start'):
        b_method.set('method:get_code target:response
content:lectus_1 state:start')
        b_focus.set('get_code')
        print 'getting the code'
    ##### LECTUS CHOICE: #####
    ### STEP 3: USE INFORMATION
    ## This is where Lectus uses the information from the
perceptual
```

```
    ## method to select the appropriate response for the
stimuli
    # TESTING: response 1 and 2 are chosen randomly here but
they would really be chosen based on the percieved code
    def lectus_response_1a(b_unit_task='unit_task:lectus
state:running type:?type',
                           b_focus='code:identified'):
        b_method.set('method:response target:response
content:lectus_1a state:start')
        b_focus.set('done')
        b_unit_task.modify(state='end')  ## this line ends
the unit task
        print 'entering the code'


    def lectus_response_1b(b_unit_task='unit_task:lectus
state:running type:?type',
                           b_focus='code:identified'):
        b_method.set('method:response target:response
content:lectus_1b state:start')
        b_focus.set('done')
        b_unit_task.modify(state='end')  ## this line ends
the unit task
        print 'entering the code'
    ##### LECTUS FINISH #####
```

```
    ### STEP 4:
    ## finishing the unit task
    def lectus_finished_ordered(b_method='state:finished',
## this line assumes waiting for the last method to finish
                                b_unit_task='unit_task:lectus
state:end type:ordered',
                                b_emotion='threat:ok'):
        print 'finished unit task lectus - ordered'
        b_unit_task.set('unit_task:lectus state:finished
type:ordered')

    def lectus_finished_unordered(b_method='state:finished',

b_unit_task='unit_task:lectus state:end type:unordered',

b_plan_unit='ptype:unordered',
                                  b_emotion='threat:ok'):
        print 'finished unit task lectus - unordered'
        b_unit_task.set('unit_task:lectus state:start
type:unordered')

    ##### LECTUS INTERUPT #####
    ### IF STEP:
```

```
    ## this will run if Lectus faces an interuption while
running
    def
lectus_interupt_planning_unit(b_method='state:finished',

b_unit_task='unit_task:X state:end type:?type',

b_emotion='threat:high'):
        print 'finished unit task lectus - interupting
planning unit'
        b_unit_task.set('unit_task:interupted
state:interupted type:?type')

################# THE LEMONS
################# AKA: UNIT TASK: ASTRUCTUS
    ## UT astructus is the first Complex Unit task, involving
multiple, sequenced
    ## actions, like cutting a lemon. The order of the unit
task is as follows:
    ##      - 1st: ONLY 1 ANS, known
    ##          - ANS =(FAST)
    ##      - 2nd: TWO POSSIBLE, known
    ##          - IF 2A:
    ##          - ANS = (LAG)(FAST)
```

```
    ##          - IF 2B:
    ##          - ANS = (LAG)(FAST)
    ##      - 3rd: IF
    ##          - 2A: ONLY 1 ANS
    ##          - ANS = (FAST)
    ##
    ##          - 2B: TWO POSSIBLE, known
    ##              - IF 2B1, 2B2:
    ##              - ANS = (LAG)(FAST)
    ## NOTE: Should set stimuli up in PU for know-when-to-
call S->T->P OR T->P->S
    ##### ASTRUCTUS CONDITIONS #####
    ## add condition to fire to this production
    ## astructus is a sequenced planning unit - it is always
ordered,
    ## with some uncertainty inside; but it always has the
same order
    def astructus_unordered(b_unit_task='unit_task:astructus
state:start type:unordered',
                            display='zzz'): ### this unit task
is chosen to fire by planning unit
        b_unit_task.set('unit_task:astructus state:begin
type:unordered')
        print 'selected Astructus'
```

```
        print 'start unit task astructus unordered'


    def astructus_ordered(b_unit_task='unit_task:astructus
state:start type:ordered'): ### this unit task is chosen to
fire by planning unit
       b_unit_task.modify(state='begin')
       print 'selected Astructus'
       print 'start unit task astructus ordered'
    ##### ASTRUCTUS START: #####
    ### STEP 1:
    ## the first production in the unit task must begin in
this way
    def astructus_begin1(b_unit_task='unit_task:astructus
state:begin type:?type'):
        b_unit_task.set('unit_task:astructus state:running
type:?type')
        b_focus.set('astart')
        print 'starting unit task astructus now'


    ##### ASTRUCTUS BODY: #####


    ##### ASTRUCTUS PROMPT 1 #####
    ### ROUND 1 -
    ### BANG BANG
```

```
    ### PROMPT 1 - KNOWN, FAST
    def astructus_prompt_1(b_unit_task='unit_task:astructus
state:running type:?type',
                    b_focus='astart'):
        b_method.set('method:known_response target:response
content:astructus1 state:start')
                     # method FAST vision/motor call here
        b_focus.set('astructus_step2')
        print 'entering response for Astructus Prompt 1'

        ## Prompt 1 = running perfect.

    ##### ASTRUCTUS PROMPT 2 #####
    ### IDENTIFY -> RESPOND
    ### ROUND 2 - TWO POSSIBLE, KNOWN, LAG
    ### IDENTIFY:
    def astructus_identify2(b_unit_task='unit_task:astructus
state:running type:?type',
                            b_focus='astructus_step2'):
        b_method.set('method:get_code target:response
content:astructus2 state:start')
        b_focus.set('get_code')
        b_focus.set('astructus2prompt')
        print 'waiting to see if 2A or 2B'
```

```
        print 'getting the code for second prompt...'


    #### RESPOND 2A:
    def astructus_prompt_2A(b_unit_task='unit_task:astructus
state:running type:?type',
                            b_focus='astructus2prompt'):
        b_method.set('method:response target:response
content:astructus2A state:start')
        b_focus.set('astructusA1')
        print 'entering the code for prompt 2A'
        print '2A means I know the next prompt is A1'


    ### RESPOND 2B:
    def astructus_prompt_2B(b_unit_task='unit_task:astructus
state:running type:?type',
                          b_focus='astructus2prompt'):
        b_method.set('method:response target:response
content:astructus2B state:start')
        b_focus.set('astructusB')
        print 'entering the code for prompt 2B'
        print '2B means the next prompt will be B1 or B2'
        ### RUN GET CODE METH


    ##### ASTRUCTUS PROMPT 3 #####
```

```
    ### ROUND 3: A1
    ### BANG BANG
    def
astructus_response_A1(b_unit_task='unit_task:astructus
state:running type:?type',
                  b_focus='astructusA1'):
        b_method.set('method:known_response target:response
content:astructusA1 state:start')
        ### FOCUS SET TO END
        b_focus.set('astructus_done')
        b_unit_task.modify(state='end')  ## this line ends
the unit task
        print 'Entering code for A1'


    ### ROUND 3: B1 or B2
    ### IDENTIFY->RESPOND
    ### IDENTIFY B1 or B2:
    def astructus_identify(b_unit_task='unit_task:astructus
state:running type:?type',
                         b_focus='astructusB'):
        b_method.set('method:get_code target:response
content:astructusB state:start')
        b_focus.set('get_code')
        print 'getting the code for prompt...'
```

```
    ### RESPOND:
    ### B1:
    def
astructus_response_B1(b_unit_task='unit_task:astructus
state:running type:?type',
                          b_focus='code:identified'):
        b_method.set('method:response target:response
content:astructusB1 state:start')
        b_focus.set('astructus_done')
        b_unit_task.modify(state='end')  ## this line ends
the unit task
        print 'entering the code for B1'


    ### RESPOND:
    ### B2:
    def
astructus_response_B2(b_unit_task='unit_task:astructus
state:running type:?type',
                          b_focus='code:identified'):
        b_method.set('method:response target:response
content:astructusB2 state:start')
        b_focus.set('astructus_done')
        b_unit_task.modify(state='end')  ## this line ends
the unit task
```

```
        print 'entering the code for B2'


    ##### ASTRUCTUS FINISH #####
    ### Final step:
    ## Finishing the unit task
    def astructus_finished_ordered(b_method='state:finished',
                                   ## this line assumes waiting
for the last method to finish
                                   b_focus='astructus_done',

b_unit_task='unit_task:astructus state:end type:ordered',
                                   b_plan_unit='ptype:ordered',
                                   b_emotion='threat:ok'):
        print 'finished unit task astructus(ordered)'
        b_unit_task.set('unit_task:astructus state:finished
type:ordered')

    def
astructus_finished_unordered(b_method='state:finished',
                                   b_focus='astructus_done',

b_unit_task='unit_task:astructus state:end type:unordered',

b_plan_unit='ptype:unordered',
```

```
                                    b_emotion='threat:ok'):
        print 'finished unit task astructus (unordered)'
        b_unit_task.set('unit_task:astructus state:start
type:unordered')

    ##### ASTRUCTUS INTERUPT #####
    ### IF STEP:
    ## this will run if Situs faces an interuption while
running
    def
astructus_interupt_planning_unit(b_method='state:finished',

b_unit_task='unit_task:X state:end type:?type',

b_emotion='threat:high'):
        print 'finished unit task astructus - interupting
planning unit'
        b_unit_task.set('unit_task:interupted
state:interupted type:?type')

################# THE LIMES
################# AKA: UNIT TASK: SITUS
    ## UT situs is the second Complex Unit task, involving
multiple, sequenced
```

```
    ## actions, now equated to cutting a lime - like cutting
a lemon, with
    ## a right/wrong, step sequence. The order of the unit
task is as follows:
    ##      - 1st: ONLY 1 ANS, known
    ##          - ANS =(FAST)
    ##      - 2nd: TWO POSSIBLE, known
    ##          - IF 2A:
    ##          - ANS = (LAG)(FAST)
    ##          - IF 2B:
    ##          - ANS = (LAG)(FAST)
    ##      - 3rd: ONLY 1 ANS, known
    ##          - ANS =(FAST)
    ##      - 4th: ONLY 1 ANS, known
    ##          - ANS =(FAST)
    ##      - 5th: TWO POSSIBLE, known
    ##          - IF 5A:
    ##          - ANS = (LAG)(FAST)
    ##          - IF 5B:
    ##          - ANS = (LAG)(FAST)
    ## NOTE: Should set stimuli up in PU for know-when-to-
call S->T->P OR T->P->S
    ##### SITUS CONDITIONS #####
    ## add condition to fire to this production
```

```
    ## Situs is a sequenced planning unit - it is always
ordered,
    ## with some uncertainty inside; but it always has the
same order

    def situs_unordered(b_unit_task='unit_task:situs
state:start type:unordered',
                            display='zzz'): ### this unit task
is chosen to fire by planning unit
        b_unit_task.set('unit_task:situs state:begin
type:unordered')
        print 'selected Situs'
        print 'beginning unit task situs unordered'

    def situs_ordered(b_unit_task='unit_task:situs
state:start type:ordered'): ### this unit task is chosen to
fire by planning unit
       b_unit_task.modify(state='begin')
       print 'selected Situs'
       print 'beginning unit task situs ordered'

    ##### SITUS START: #####
    ### STEP 1:
```

```
    ## the first production in the unit task must begin in
this way
    def situs_start(b_unit_task='unit_task:situs state:begin
type:?type'):
        b_unit_task.set('unit_task:situs state:running
type:?type')
        b_focus.set('situs_start')
        print 'starting unit task situs now'

    ##### SITUS PROMPT 1 #####
    ### ROUND 1 -
    ### BANG BANG
    ### PROMPT 1 - KNOWN, FAST
    def situs_step1(b_unit_task='unit_task:situs
state:running type:?type',
                    b_focus='situs_start'):
        b_method.set('method:known_response target:response
content:situs1 state:start')
        b_focus.set('situs_step2')
        print 'Getting the code for Situs Prompt 1'

    ##### SITUS PROMPT 2 #####
    ### IDENTIFY -> RESPOND
    ### ROUND 2 - TWO POSSIBLE, KNOWN, LAG
```

```
    ### IDENTIFY:
    def situs_step2(b_unit_task='unit_task:situs
state:running type:?type',
                        b_focus='situs_step2'):
        b_method.set('method:get_code target:response
content:situs2 state:start')
        b_focus.set('get_code')
        b_focus.set('situs_prompt2')
        print 'Check if 2A or 2B'
        print 'getting the code for second prompt...'

    #### RESPOND 2A:
    def situs_prompt_2A(b_unit_task='unit_task:situs
state:running type:?type',
                        b_focus='code:identified'):
        b_method.set('method:response target:response
content:situs2A state:start')
        b_focus.set('situs_prompt3')
        print 'entering the code for prompt 2A'

        ### RUN GET CODE METH
    ### RESPOND 2B:
    def astructus_prompt_2B(b_unit_task='unit_task:situs
state:running type:?type',
```

```
                        b_focus='situs_prompt2'):
        b_method.set('method:response target:response
content:situs2B state:start')
        b_focus.set('situs_prompt3')
        print 'entering the code for prompt 2B'


    ##### SITUS PROMPT 3 #####
    ### ROUND 3 -
    ### BANG BANG
    ### PROMPT 3 - KNOWN, FAST
    def situs_step3(b_unit_task='unit_task:situs
state:running type:?type',
                    b_focus='situs_prompt3'):
        b_method.set('method:known_response target:response
content:situs3 state:start')
        b_focus.set('get_code')
        b_focus.set('situs_prompt4')
        print 'Getting the code for Situs Prompt 3'
        print 'I know that the next prompt will be Situs 4'


    ##### SITUS PROMPT 4 #####
    ### ROUND 4 -
    ### BANG BANG
    ### PROMPT 3 - KNOWN, FAST
```

```
    def situs_prompt4(b_unit_task='unit_task:situs
state:running type:?type',
                      b_focus='situs_prompt4'):
        b_method.set('method:known_response target:response
content:situs4 state:start')
        b_focus.set('get_code')
        b_focus.set('situs_step5')
        print 'Getting the code for Situs Prompt 4'
        print 'I know that the next prompt will be either 5A
or 5B'


    ##### SITUS PROMPT 5 #####
    ### IDENTIFY -> RESPOND
    ### ROUND 5 - TWO POSSIBLE, KNOWN, LAG
    ### IDENTIFY:
    def situs_step5(b_unit_task='unit_task:situs
state:running type:?type',
                          b_focus='situs_step5'):
        b_method.set('method:get_code target:response
content:situs5 state:start')
        b_focus.set('get_code')
        b_focus.set('situs_prompt5')
        # Running issue with get code for PROMPTS
        print 'Identify if 5A or 5B'
```

```
        print 'getting the code for fifth prompt...'


    #### RESPOND 5A:
    def situs_prompt5A(b_unit_task='unit_task:situs
state:running type:?type',
                       b_focus='situs_prompt5'):
        b_method.set('method:response target:response
content:situs5A state:start')
        b_focus.set('situs_done')
        b_unit_task.modify(state='end')
        print 'entering the code for prompt 5A'


    ### RESPOND 5B:
    def situs_prompt5B(b_unit_task='unit_task:situs
state:running type:?type',
                       b_focus='situs_prompt5'):
        b_method.set('method:response target:response
content:situs5B state:start')
        b_focus.set('situs_done')
        b_unit_task.modify(state='end')
        print 'entering the code for prompt 5B'


    ##### SITUS FINISH #####
    ### Final step:
```

```
    ## Finishing the unit task
    def situs_finished_ordered(b_method='state:finished',  ##
this line assumes waiting for the last method to finish
                               b_unit_task='unit_task:situs
state:end type:ordered',
                               b_emotion='threat:ok'):
        print 'finished unit task Situs (ordered)'
        b_unit_task.set('unit_task:situs state:finished
type:ordered')

    def situs_finished_unordered(b_method='state:finished',

b_unit_task='unit_task:situs state:end type:unordered',

b_plan_unit='ptype:unordered',
                                 b_emotion='threat:ok'):
        print 'finished unit task Situs (unordered)'
        b_unit_task.set('unit_task:situs state:start
type:unordered')

    ##### SITUS INTERUPT #####
    ### IF STEP:
    ## this will run if Situs faces an interuption while
running
```

```
    def
situs_interupt_planning_unit(b_method='state:finished',

b_unit_task='unit_task:X state:end type:?type',

b_emotion='threat:high'):
        print 'finished unit task Situs - interupting
planning unit'
        b_unit_task.set('unit_task:interupted
state:interupted type:?type')

################ methods ######################

### known response method ########################

    ### RESPONSE TYPE 1: BANG BANG
    ### known response method ########################
    # the response is assumed to be passed down by the
production that called this method
    # there is a quick visual check that the code has changed
then there is a response
    # possibly someone could train themselves to ignore the
visual altogether
```

```
    # here it is assumed that seeing the code change still
plays a role as subjects were trained initially to wait for
it to change
    def known_response_vision(b_method='method:known_response
target:?target content:?content state:start'):  # target is
the chunk to be altered
        motor.vision_fast()
        b_method.modify(state='running')
        b_focus.set('target:looking')
        print 'known_responding'

    def vision_fast_finished(motor_finst='state:vision_fast',

b_method='method:known_response',
                             b_focus='target:looking'):
        motor.motor_finst_reset()
        b_focus.set('target:spotted')
        print 'I have spotted the target, the new code is
there'

    def enter_response_fast(b_focus='target:spotted',
                            b_method='method:known_response
target:?target content:?content state:running'):
        motor.change_state_fast(target, content)
```

```
        b_method.modify(state='running')
        b_focus.set('response_entered')
        print 'entering response'
        print content
        print 'target object = ', target

    def response_entered(b_method='method:?method
target:?target state:running',

motor_finst='state:change_state_fast',
                        b_focus='response_entered'):
        b_method.modify(state='finished')
        motor.motor_finst_reset()
        print 'I have altered', target


    ### RESPONSE TYPE 2: IDENTIFY->RESPOND
    ### get_code method ################################
(get_code)
    # in the case where the next response depends on the code
the agent must first read the code
    # AKA - this is the instance where the agent is not
predicting the next response
```

```
    # but reading->chosing (not 'bang bang' but 'identify
bang')
    # The different pace times are accounting for the lag -
LOW
    # This method is inseperable, and ordered
    ### PART A: IDENTIFY

    def get_code_vision(b_method='method:get_code
target:?target content:?content state:start'):  # target is
the chunk to be altered
        motor.vision_slow()
        b_method.modify(state='running')
        print 'getting code'

    def
vision_slow_finished(motor_finst='state:vision_slow'):
        motor.motor_finst_reset()
        b_method.modify(state='finished')
        b_focus.set('code:identified')
        print 'I have spotted the target, I have the new
code'
    ### PART B: RESPOND
    # in this case the vision component took place already
using the get_code method so this is only motor
```

```
    def response(b_method='method:response target:?target
content:?content state:start'):  # target is the chunk to be
altered
        motor.change_state_fast(target, content)
        b_method.modify(state='running')
        b_focus.set('response_entered')
        print 'entering response'
        print 'target object = ', target

    def response_entered2(b_method='method:?method
target:?target state:running',

motor_finst='state:change_state_fast',
                          b_focus='response_entered'):
        b_method.modify(state='finished')
        motor.motor_finst_reset()
        print 'I have altered', target



############## run model #############

aeva = MyAgent()  # name the agent
lab = MyEnvironment()  # name the environment
lab.agent = aeva  # put the agent in the environment
```

```
ccm.log_everything(lab)  # print out what happens in the
environment
lab.run()  # run the environment
ccm.finished()  # stop the environment
```