Understanding Micro Cognition and Macro Cognition using SGOMS/ACT-R predictions of an SGOMS-based Mobile Application

by

Elisabeth M. Reid

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Cognitive Science

in

Master of Cognitive Science

Carleton University
Ottawa, Ontario

© 2023, Elisabeth M. Reid
Abstract

This research continues to develop SGOMS, a general theory of how all experts organize routine expert task knowledge. Previous predictions of SGOMS/ACT-R motivated experimental and model design improvements to predict all expert players (expected of a general theory of expertise). In this study, the micro-strategies predictions of two models (Internal and External SGOMS/ACT-R models) are compared to the micro-strategies used by individual participants. A visual cue was added to the experimental game played in this study to encourage the same SGOMS game strategy. It was assumed that participants would use the color cue to help them follow the Internal model (a previously proposed model), but surprisingly, all participants better matched the External model. The external model assumes the visual system learns and handles higher gameplay knowledge. This provided evidence that high-level expert task information can and is more readily offloaded to the visual system.
Acknowledgements

I express my deepest gratitude to my professor and committee chair, Dr. Rob West. You are a mentor who truly cares about your students and a wellspring of knowledge. I am fortunate to have you as my supervisor. I am also extremely grateful to my defense committee and chair, who generously provided expert feedback and who were there when times were busy. And to the Department of Cognitive Science Administration, thank you for your support and understanding.

I am also grateful to my participants, who put much time into training and learning the Four Button Game. This research would not have existed without you. I would also like to thank my family, especially my Husband, Mother and Mother-in-law, and sister. They sacrificed their time to give me the time I needed to complete my work.

Lastly, I'd like to acknowledge my daughter. You have been with me through this degree for your entire life thus far and taught me the amazing endless love, patience, and encouragement a child can give.
# Table of Contents

Abstract................................................................................................................... i
Acknowledgements................................................................................................... ii
Table of Contents...................................................................................................... iii
List of Tables.............................................................................................................. v
List of Illustrations.................................................................................................... vi
List of Appendices..................................................................................................... vii
Introduction............................................................................................................... 1
  Micro Strategies...................................................................................................... 2
  ACT-R....................................................................................................................... 3
  SGOMS................................................................................................................... 4
  SGOMS/ACT-R....................................................................................................... 8
Applying SGOMS to a Game.................................................................................... 9
  Four Button Expert Game..................................................................................... 12
Experimental Conditions......................................................................................... 17
Study 1....................................................................................................................... 19
  Methodology.......................................................................................................... 19
  The model................................................................................................................ 20
  Results and Findings.............................................................................................. 20
Study 2....................................................................................................................... 27
Methods ..................................................................................................................29

Four Button Expert Training ..................................................................................31

Reaction Time Recording and Analysis ..................................................................33

Results and Findings .............................................................................................35

Model Comparisons and Breakdown .....................................................................41

Internal vs. External Model ....................................................................................43

Conclusions .............................................................................................................46

Appendix ..................................................................................................................48

References .............................................................................................................99
List of Tables

Table 1    Participant E and B counts an accuracy for each SGOMS level ...................... 36
Table 2    Participant E’s Descriptive Statistics RT for all experimental conditions....... 37
Table 3    Participant B’s Descriptive Statistics RT for all experimental conditions ....... 37
### List of Illustrations

<table>
<thead>
<tr>
<th>Illustration</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SGOMS control flow in Chapter 1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Operators: Super Mario Controls</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Methods: Super Mario Controls</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Unit Task: Mario Maker Gameplay Screenshot</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>Planning Unit: Mario Maker screenshot of a full-level layout</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Screenshot of the FBG screen where the game is played</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>FBG gameplay screen displaying an rp method cue</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>A visual depiction of the three planning units in the FBG</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>Known and unknown methods condition visual depiction</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>Unit task first method visual depiction</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>Planning unit first methods visual depiction</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>Participant S mean RT compared to the Internal Model Predictions</td>
<td>18</td>
</tr>
<tr>
<td>13</td>
<td>Participant S ranked RT distribution for each condition</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>Participant M mean RT compared to the Internal Model Predictions</td>
<td>23</td>
</tr>
<tr>
<td>15</td>
<td>Participant M ranked RT distribution for each condition</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>Participant EM 1 mean RT compared to the Internal Model Predictions</td>
<td>24</td>
</tr>
<tr>
<td>17</td>
<td>Participant EM ranked RT distribution for each condition</td>
<td>25</td>
</tr>
<tr>
<td>18</td>
<td>Participant EM 1 ranked RT distribution for each condition</td>
<td>25</td>
</tr>
<tr>
<td>19</td>
<td>Participant EM 1 ranked RT distribution for each condition zoomed out.</td>
<td>25</td>
</tr>
<tr>
<td>20</td>
<td>Participant 2 mean RT compared to the Internal Model Predictions</td>
<td>26</td>
</tr>
</tbody>
</table>
Illustration 21  Participant EM 2 ranked RT distribution for each condition ................. 26
Illustration 22  FBG main menu screenshot indicating the training modes. ............... 31
Illustration 23  Box-Cox cut-off algorithm part 1....................................................... 35
Illustration 24  Box-Cox cut-off algorithm part 2....................................................... 35
Illustration 25  E mean RT times compared to the External Model predictions............. 39
Illustration 26  Participant E ranked RT distribution for each condition..................... 39
Illustration 27  Participant B mean RT compared to model predictions for all conditions.
.......................................................................................................................................... 40
Illustration 28  Participant B ranked RT distribution for each condition. ....................... 40
Illustration 29  Picture of the three nested control loops in a SGOMS/ACT-R model. .. 42
Illustration 30  Comparison of Internal/External Models predicted times of the.. .......... 45
Illustration 31  Bar Graph of all participants and conditions mean RT (Study 1 & 2). ... 45
List of Appendices

A: Verbal Reports ............................................................................................................ 48
B: RT Frequency Distribution ....................................................................................... 50
C: Box-Cox Transform and Cut Results ........................................................................ 57
D: Source Code and Links ............................................................................................. 63
The motivation behind a series of previous research and this research is to develop a general theory of how all experts organize their routine expert tasks. Previous studies show some validity for SGOMS (MacDougall et al., 2014; West, 2013; West & MacDougall, 2014; West & Nagy, 2007; West & Pronovost, 2009), a general theory of how experts organize their task knowledge. SGOMS is meant to guide model design and can be tested by creating agents to predict human behavior. This research continues developing SGOMS to explore exactly how it is implemented in human expertise and what sort of experimental design is best to shed light on it. To do this we studied the micro strategies used by participants when playing an SGOMS based game. This research explores micro strategies by comparing two computational SGOMS/ACT-R models to predict participant performance under different conditions (for previous research on this, see: Dudzik et al., 2018; Greve et al., 2021; Greve et al., 2020).

Usually, in research involving cognitive tasks, many participants are observed, and their results are averaged together to address noise and random error. However, averaging across participants can be disadvantageous if it inadvertently averages across information that can tell us what strategy an individual is using in a cognitive task (Daily et al., 2001; Gobet & Ritter, 2002; Gray & Boehm-Davis, 2000; Heathcote et al., 2000; Newell, 1973; Shiffrin & Cousineau, 2004). Cognitive microstrategies occur at the millisecond scale, can exhibit variations even for simple cognitive tasks (Gray and Boehm-Davis, 2000), and provide fundamental clues regarding low-level cognitive mechanisms.
Micro Strategies

In this research, a micro strategy is a low-level control structure that processes basic information to do a cognitive task (Newell 1973), such as perceptual inputs, task information, or procedural information (Gray & Boehm-Davis, 2000). To further elaborate, functional components and information that make up a cognitive processor can be applied differently to complete the same task; the resulting possible structure arrangements are micro strategies. Micro strategies are observable, and research indicates humans are sensitive to the time cost and savings different micro strategies offer (Gray & Boehm-Davis, 2000). Micro strategies may vary or be the same across individuals, and how they develop is not well understood, but they appear to become stable once a task has been well learned and practiced. Like many cognitive mechanisms, micro strategies are not directly observable but can be investigated by examining performance measures such as reaction time (Gobet & Ritter, 2002; Shiffrin & Cousineau, 2004). Individuals' varying reaction times can indicate different micro strategies. However, if reaction times are averaged among individuals, then the micro-strategy information of each individual is lost. Studies where we are interested in lower-level cognitive mechanisms, such as in cognitive modeling, require individual information to be analyzed even to begin detecting low-level mechanisms (Chuderski et al., 2007; Shiffrin & Cousineau, 2004). However, noise in data still needs to be accounted for. One approach that can be taken to compensate for noise in data without averaging is training participants to an expert level while strictly controlling task conditions and collecting large amounts of data per individual so we can be confident with what micro strategies individuals are using.
ACT-R

ACT-R (Anderson & Lebiere, 1998) stands for “Adaptive control of thought–rational,” it is a cognitive architecture meant to unify and model the functional mechanisms of human cognition. A cognitive architecture is a theory of integrating the mind to produce intelligent cognitive behavior (Newell 1990). ACT-R is written as a programming language that allows for a clear modeling theory to be defined for a model to compile and run, resulting in well-defined and testable theories of human behavior. There are multiple implementations of ACT-R: CMU, Python, Etc. This shows ACT-R theory is not bound by the computer language it is implemented in (Stewart & West, 2007).

In ACT-R theory, functional areas of the mind are modular and represented by modules corresponding to physical brain regions (Anderson et al., 2004). Modules operate in parallel to process information, and each module has access to its dedicated buffer space to update by placing, clearing, or editing a limited number of chunks of information. Contents within modules are not accessible outside the module, but their associated buffer contents are available to the central production module. Information is stored in chunks and production rules. Chunks of information are contained within buffers and declarative memory. Buffer access and pattern matching are how modules communicate to process information and fire productions to guide action.

All system buffers' contents form the system's current ‘context.’ Traditionally, ACT-R models have included a goal, retrieval, and visual buffers (Ritter et al., 2019). The context of the buffers directs which productions are matched and fired by the central production module. Buffers have a limited storage capacity and have been defined as one
chunk of information at a time; chunks are pieces of information that store 7±2 slots
where small pieces of information can be stored (Anderson & Lebiere, 1998).

The components of ACT-R form a system, and at the heart of that system is the
central production module. This module monitors all buffer contents, keeps track of an
action stack, and matches patterns of information in buffers (production rules) to
determine if production can be fired; this occurs at the rate of 1 production fired in a
50ms cycle.

Production rules follow the form of an IF → THEN rule. The condition ‘IF’ is a
defined pattern matched to the system's context. The ‘THEN’ part fires a production,
which fires a production. A fired production can request another module to offload work
or/and update a buffer available to the system.

Memory in ACT-R is modeled as procedural, working, and declarative. Memory,
such as working memory, is represented as active buffers. Procedural memory is stored in
production rules and compiled productions. The declarative memory module handles
declarative memory and has access to stored chunks of information. In addition, other
cognitive mechanisms of ACT-R, such as learning mechanisms and forgetting, are not
used in the model described in this paper.

**SGOMS**

SGOMS is a type of GOMS model. GOMS (Card et al., 1983) is a method of
modeling human interaction with a computer system; its methodology defines the
components that a researcher will use to construct a model of how a human performs an
expert routine task. The usable product of a GOMS model is a time prediction to execute a task. GOMS has been applied to predicting training times, product usability, and product design. GOMS has been rigorously tested/validated and works well if applied within its usage constraints in situations that involve well-learned tasks (John, 1995). Although it has been used to analyze computer system interaction in human-computer interaction research (HCI), GOMS-based models are a cognitive theory of how humans process and execute well-learned routine tasks (Card et al., 1983; West et al., 2022).

There are many different versions of GOMS, but they all follow the same general structure (John, 1995). GOMS stands for Goals, Operators, Methods, and Selection rules, which are the four cognitive components a GOMS model is divided into (Card et al., 1983; John, 1995). Goals are the goals or steps needed to do a job/task, with primary goals comprising sub-goals. Operators are the simplest perceptual, motor, or cognitive acts that produce a specific output, such as moving hand to location, button press, or looking at location x. Methods are defined sequences of operators, the stored knowledge used to complete a task, and are strung together to complete sub-goals. Selection Rules choose between available methods within a goal in a simple, smooth way without the need for problem-solving; these rules are of the form “if condition x is true in the current task situation, then use method M.” In addition to the above components, there are Unit Tasks, a group of methods an individual uses to execute a sub-goal.

Determining the different components that make up a GOMS model is done through a task analysis – observing, questioning, and recording the behavior and reasoning of a real human expert within the task domain of question. During task analysis, the grain size (Card et al., 1983) is also determined, which is the level of detail
the model includes, starting from the unit tasks down to the methods. Higher detail results in more behavior reaction times to be measured (more complex task analysis) but allows for more flexibility and reusability in constructing different models. GOMS models may be constructed as charts on paper, programmed models, etc.

GOMS only produces time estimates for error-free behavior, so in situations where errors occur, time estimates are not accounted for. Errors still occur in expert routine cognitive tasks even when errors are not routine (Card et al., 1983). The detection and correction of errors is neither part of the GOMS model nor the ability to problem-solve in unseen scenarios outside the routine task. A complete dynamic description of behavior, measured at the level of goals, methods, and operators, is capable with GOMS but only to its highest level of control structure, the unit task.

SGOMS is a GOMS-based theory that allows for modeling in complex Sociotechnical systems (West & Nagy, 2007). A sociotechnical situation involves a social aspect where activity includes decision-making, sensemaking, planning, adaption/replanning, problem detection, interrupt handling, and coordination. The SGOMS theory is meant to be a template used to guide a cognitive modeler on constructing a model meant to model an agent at human expert performance. The claim made by SGOMS is that all experts follow the same way of organizing the knowledge they know to perform the task they are experts in, and how they use and structure this information allows them to perform in a sociotechnical environment. There are two main additions to GOMS to define SGOMS:

1) Create plan mechanism: The create plan mechanism deals with learning and problem-solving. The create plan component can be modeled in ACT-R or Soar as they are compatible with GOMS, but the components can also be treated as a
black box. The SGOMS system can be modeled using theorized output from the create plan component (West & Nagy, 2007).

2) Unit task size limit: The second addition to the GOMS theory is applying a size limit to the unit tasks so they are unlikely to be interrupted. The size limit is determined by the task during a model learning phase (how the size is determined is not part of the SGOMS theory) but is inferred through task analysis of a human expert (West & Nagy, 2007).

Figure 1 shows a cycle of how an agent would use SGOMS knowledge structures to work through a routine task, beginning with choosing the planning unit corresponding to that task.

![Figure 1. SGOMS control flow. Note some parts are greyed out as they are not in the scope of this paper.](image)

The social aspect of the theory, which allows for interactions between individuals, is achieved simply by every agent having the same SGOMS template, and thus, communication occurs across the environment between individuals (MacDougall et al., 2014; West & Nagy, 2007).
An SGOMS/ACT-R model is the application product of using SGOMS to guide how an ACT-R model is constructed to do a routine task at expert performance. SGOMS/ACT-R theory assumes ACT-R and SGOMS are true, and this assumption can be tested by testing SGOMS/ACT-R model predictions against human expert data.

When constructing an SGOMS model using ACT-R, expert task knowledge must be broken down into SGOMS components identified through task analysis. The implementation of individual SGOMS components is reviewed in West & Somers (2011) and summarised here.

Operators ➔ Operators in ACT-R can hand off an action to a module. For example, a button press is offloaded to the motor module. Because these actions are processed in parallel with the central production system, they are faster.

Methods ➔ Rapidly fired productions that occur relatively quickly and in the same order after being compiled. For example, four button presses consistently paired in the same order become compiled and can be ballistically fired by the motor module as a single method.

Unit tasks ➔ A set of productions for accomplishing a specific task component by firing methods. However, unlike methods, unit tasks can match the conditions to behave differently under different conditions.

Planning units ➔ “…represented planning units as sequentially chained chunks in declarative memory. Each chunk represents a unit task and the unit task that should
follow it.” (West & Somers, 2011 p3). A complete representation of planning units in ACT-R is still being developed.

**Applying SGOMS to a Game**

Let us take a familiar example to understand how SGOMS can be applied to a computer game. There are many different Mario games, but they can all be broken down into SGOMS and divided into the following structures:

**Operators** are the smallest most basic actions necessary to interact with a task. For a Mario player, this is pressing a button (*Figure 2*).

![Figure 2. Super Mario Controls (From Nintendo Mario Odyssey) used as an example of operators.](image)

**Methods** are a learned string of chained operators which do a distinct (and meaningful) action in the game, such as jump, kick, jump kick, throw, and so on (*Figure 3*). Multiple ordered operators execute a ground pound, but jump is just one. The key is that operators are chained together for fast execution when a method is needed. For
example, shell jump, which involves pressing the correct buttons (motor actions) to pick up a shell to jump further/higher, is a method. The method for a shell jump would be

$$\text{Run}(L) \rightarrow \text{Pick up}(Y \text{ hold}) \rightarrow \text{Run}(L) \rightarrow \text{Throw}(Y \text{ release}) \rightarrow \text{Run}(L) \rightarrow \text{Jump}(A).$$

Unit tasks for an expert Mario player would be any variation of ordered methods that completes a sub-goal, such as avoiding a specific obstacle in a course or navigating through a series of enemies. Unit tasks cannot handle interruptions and would remain as short.

Figure 3. Super Mario Controls (Nintendo Mario 64) used as an example of operators. A combination of compiled button presses and stick controls are used to perform a method.

Figure 4. Mario Maker Gameplay Screenshot. A player would use a combination of methods to make it to the otherside of the lava.
segments. The methods needed to get past certain enemies and obstacles in the course are organized in the unit task control structure. For example, in Figure 4, the player’s sequence timing and jumps to get over the lava would be a unit task, and multiple unit tasks could achieve it.

![Figure 5. Mario Maker screenshot shows an example of a full level layout. A player’s knowledge of this level is managed by a planning unit.](image)

A **Planning Unit** is a control structure that mediates the entire knowledge to complete a course by grouping and stringing unit tasks together. This equates to knowing what order and when the enemies/obstacles in the whole course (Figure 5) appear to choose the correct unit task. We can be experts in more than one video game in a more enormous scope of things. Different video games require different controls and inputs, each mediated by their planning units.
Four Button Expert Game

The Four-button Expert game (FBG) is a mobile application game designed to test the SGOM theory. The game's basis is a stimulus button pressing response game, but the stimulus presentation order follows the SGOMS hierarchy of methods, unit tasks, and planning units. Because the Four Button App follows the SGOMS structure, it is a computerized experiment within an Android smartphone, representing a structure that is theorized to be the base structure of all expert routine tasks.

The FBG was the game participants trained and played for a previous study (study 1) and this study (study 2). FBG is a stimulus, touch screen button game played on an Android mobile smartphone. Reaction time data was recorded and sent to researchers within the app. The app was programmed using Android Studio v2021.2.1. It consisted of six different interface screens accessible to players: (1) username entry screen, (2) main menu, (3) buttons screen, (4) data view screen, (5) settings screen, and (6) instructions screen. The source code link and further documentation are available in the Appendix D.

Figure 6. Screenshot of the FBG screen where the game is played. This shows the screen before the game has started. Once the game stats a method cue appears where the “Press Start” text is.
Gameplay and data collecting occurred in the Buttons screen in Figure 6. The buttons display the 2x2 number grid used for entering codes, the code prompt, game score, game hints, data collecting check box, and training options checkboxes. The hint and training checkboxes were only visible to players while in training levels of the game; otherwise, the button screens were the same across gameplay and training.

The data view screen, seen on the right of Figure 7, was accessible through the progress button in the main menu and was there for the player to monitor their accuracy and average time for each level of the game. The data view screen was also where users sent their data to the researcher’s email via the send data button in the top right of the screen (Figure 7).

Figure 7. A screen shot of main menu (left) and the data viewer screen (right).
Game Play

Operators are the most essential actions necessary to do a task, such as a single motor operation to press a button. In the FBG, an operator is a single button tap. Operators are rapidly fired during method execution.

Methods are learned sequences of operators. There are eight methods in the FBG. Each method is a combination of four operators (four button taps), which are uniquely matched to a method cue. In the game, method cues are the stimuli presented in the yellow box seen in Figure 8, and the response participants give is the corresponding method.
Unit tasks dictate what methods to use in what order. A unit task is realized in the FBG by methods planned to appear in the same order, ranging from a sequence of three to five methods in a row. The start of a unit task was indicated by the player noticing the first method at the start of that unit task.

The planning unit directs task flow and is essentially the knowledge of the unit tasks required to complete a task. In the FBA, a planning unit is a group of three ordered unit tasks. There are three unit tasks in the FBG, each using the same unit tasks in different orders (see Figure 9). In Study 1, there was no color indication of the game's planning unit, and players had to keep track by recognizing what method was given at the start of a new planning unit. This proved to be difficult, so the single gameplay change of adding a background color cue was made for study 2. The background color changed to the corresponding planning unit at the beginning of each planning unit, and the cue would decrease in saturation as each unit task in the planning unit was entered.

Figure 9. A visual depiction of the three planning units in the FBG. Methods are in yellow, unit tasks in grey, and planning units in their corresponding color cue boxes.
The SGOMS structure is realized in the Four Button Game by the order in which a method cue is presented, as the order in which a cue is presented is determined by an SGOMS structure programmed within the game. When the main game is played, the game will choose a random planning unit out of the three planning units programmed into the game to display the first method cue in the yellow cue bar and the corresponding planning unit color cue in the background (See Figure 9 above for the start cue of the blue planning unit). When the player finishes entering the method cue, they will receive feedback for accuracy, and the following method will be displayed. The accuracy feedback for the previous cue disappears as soon as the user enters the buttons for the new cue.

Once a player responds to all methods in the current planning unit, the game chooses a new planning unit at random; this cycle repeats until the game is closed via the back or end game buttons.

In a gameplay situation, when a player knows what method will be displayed next, the player does not need to look at the method cue before entering a method; this is a known method condition. In the known method condition, it is assumed a player executes the fastest SGOMS strategy, entering the known method code without looking. In the game, some methods appear randomly, resulting in a need to look, such as in a split situation that occurs in the RP and HW unit tasks; these are unknown conditions. In Figure 10, an example of one of the planning units in the game shows known and unknown methods.
Experimental Conditions

Based on the planning units programmed into the game, the conditions in which the model predicts reaction times total five different conditions to be analyzed and compared to the SGOMS/ACT-R model predictions:

**Known Conditions** (when the player is theorized to know a method was coming and did not look at the cue to enter the correct method code):

1) Known Method Condition – The methods within unit tasks that are known to come next. These are methods that will always appear in the same order.

2) Unit Task (UT) First Condition – The first method in a unit task, other than the first one starting a planning unit (*Figure 11*).

![Planning Unit](image1)

*Figure 10.* An image displaying methods in the green planning unit. Known methods are circled (10 in this example) while unknown methods darkened (6 in this example).

![Planning Unit](image2)

*Figure 11.* An image displaying methods in the green planning unit. Unit task first methods are darkened (2 in this example).

**Unknown Conditions** (when players cannot predict the method and must look at the cue because the method is chosen at random)
3) Planning Unit (PU) First Condition – The first method of a planning unit (planning units are chosen at random during final gameplay; see Figure 12).

4) Two-split Condition – The method at the start of a randomly chosen split of two possibilities. It only occurs in unit tasks that have a two-split (RP unit task).

5) Three Split Condition – The method at the start of a randomly chosen split of three possibilities. It only occurs in unit tasks that have a three-split (HW unit task).

![Planning Unit Diagram](image)

*Figure 12.* An image displaying methods in the green planning unit. Planning unit first methods are darkened (1 in each unit task).

If the SGOMS theory is plausible, we should see faster reaction times for known conditions than unknown conditions because the human processor can immediately retrieve and fire a known motor response (button code entered) without processing the visual stimulus first. However, this is impossible without tracking where you are in the game. This means tracking what planning unit and what unit task you are in. There is a time cost for this, which can be estimated using an ACT-R/SGOMS model of the game.
Study 1

The author of this study was involved in an early publication (Greve et al., 2020) and programmed the game used in Study 1. Since Study 1 has already been published (Greve, 2021), this thesis focuses on Study 2. However, since Study 2 is a follow-up on Study 1, it is essential to understand the results of Study 1 in detail. Study 1 and 2 use the 4 Button Expert game to gather data.

Methodology

The game in Study 1 was based on an SGOMS/ACT-R model (Greve, 2021) developed based on a previous pilot study. The new model assumed buffer storage for the planning unit existed and that planning unit information was updated and held in this buffer throughout planning unit execution. Since the model relied on storing the planning unit knowledge, the game was changed to test this assumption by removing the explicit visual cues for the planning unit start. The new game design meant planning unit cues were only given at the start of a planning unit and disappeared after entering the PU first method, forcing participants to store the planning unit information, which should not have been a problem if planning units have their own, dedicated, storage buffer. Here, we refer to this model as the Internal SGOMS-ACT-R Model.

For a complete account of the methodology used in this study, see Greve (2021) and Greve, Reid & West (2020). Essentially, subjects practiced the game extensively until they attained an expert level of play as measured by reaction time. Their results were then compared to the model. Here, the data and the model have been re-created and re-analyzed to contextualize better and motivate Study 2.
The model

Both studies compared the results to ACT-R/SGOMS models of the game. The ACT-R/SGOMS model uses SGOMS to structure how the ACT-R agent plays the game. It is possible to base the model on different microstrategies. We used the fastest strategy as our default model. The model was created in Python ACT-R (Stewart & West, 2007; source code) and played a simulated version of the phone game.

Results and Findings

The data was filtered for correct responses, and significant outliers were removed using the Kneed script (Arvai, 2019). This method looks for a distinct upward trend, or Knee, in the ranked distribution of results and cuts this out. Note Participant EM in this study was initially referred to as E in Study 1 but was changed to EM to distinguish between Participant E in Study 2. Mean RT times for each participant's conditions were compared to the SGOMS/ACT-R model. Only the filtered data from Greve, Reid & West (2020) and Greve (2021) was available.

To estimate the time for motor actions, pressing buttons on the phone, we took the known methods condition and subtracted 50 milliseconds from it. This is because the model claims it takes 50 milliseconds to make the decision or one production, and the rest would be the motor time. We then used this motor time in all other conditions. To estimate the time to perceive the code, we took the two split conditions minus the motor time and minus 100 milliseconds, or two productions, which the model claims are required to initiate motor and perceptual actions. We then use this perceptual time in all other conditions. The bar graphs (Figures 13, 15, 17, and 20) show the Internal
SGOMS/ACT-R model predictions and the actual mean times for each individual participants’ conditions. The model times are broken down into action (hit) perception (look) and cognition or production rules (productions). The ranked distribution graphs are included in addition to the bar graphs.

For subject S (Figure 13), we can see that the model predictions are pretty accurate. Figure 14 shows S is consistently lowering their UT first condition times. However, UT first method is not as fast as the model predicted. This can be accounted for by a mixed strategy where sometimes S can remember what comes next, and sometimes S needs to check the code. This could be assumed to occur if S had not fully learned this or if it was too difficult for S to learn. S also spent more time in the PU first method than predicted by the model. This can be accounted for by assuming S rehearsed during the planning unit. These results indicate that S found it challenging to use the knowledge from the planning unit to predict the next unit tasks. Looking at the two split conditions in S’s ranked RT distributions (Figure 14) we can see the splits times are sharply curved at the beginning and end indicating the split conditions were not as stable. This could have been due to guessing when response times were rapid and some hesitation or back-tracking on a guess may be attributed to when responses were high.

In contrast, subject M (Figure 15) matched the planning unit (PU) first results predicted by the model with stable ranked response time distributions (Figure 16). However, M did not match the unit task (UT) first condition. Instead, M seems to have pursued an alternative micro strategy, as M's results are consistent with M constantly
checking the code in the unit task (UT) first method. These results indicate that M did as instructed and noted which planning unit they were in but did not use this information to speed up the UT first method and instead just looked at the code. In this case, the UT first method should match the two-split and the three-split, which is what we found.

Looking at the bar graph, subject EM (*Figure 17*) appears to have used the same micro strategy as subject M, which constantly checks the code. However, when we examined the ranked RT distribution, the results were inconsistent with the bar graph. The ranked distribution results indicate that EM was close to the model predictions in the UT first condition. One possible explanation is that removing outliers was less effective in this condition, causing a higher mean. *Figure 19* shows a zoomed out view of the EM’s ranked RT distribution for part 1, which supports this conjecture.

Based on the bar graph, EM tried to play again and lower her UT first RT score. This was coded as EM2. However, the UT's first score was already correct, and EM could not reduce it any further and essentially got the same results, with the planning unit becoming shorter. However, the amount of noise increased due to increased guessing, which shows up as increased speed in the distribution because the incorrect guesses were thrown out and the correct guesses were kept.
Participant S

Figure 13. Participant S mean RT compared to the Internal SGOMS/ACT-R Model Predictions. Bars measure to the time in MS on the x-axis.

Figure 14. Participant S ranked RT distribution for each condition. We can see UT first condition is a bit faster than the look conditions. Ranked data distribution shown here is done by all response times being ordered by lowest to highest and then plotted in order.
Participant M

**Figure 15. M Results and Method Predictions** - Participant M mean RT compared to the Internal SGOMS/ACT-R Model Predictions. Bars measure time in MS on the x-axis.

**Figure 16.** Participant M ranked RT distribution for each condition. We can see that the UT first condition is the same as the look conditions. Ranked data distribution shown here is done by all response times being ordered by lowest to highest and then plotted in order.
Participant EM Part 1

Figure 17. Participant EM part 1 mean RT compared to the Internal SGOMS/ACT-R Model Predictions. Bars measure time in MS on the x-axis.

Figure 18. Participant EM ranked RT distribution for each condition. We can see UT is faster here unlike in the model comparison. Ranked data distribution shown here is done by all response times being ordered by lowest to highest and then plotted in order.

Figure 19. Participant EM ranked RT distribution for each condition zoomed out. We can see UT has not been cut-offs soon enough.
Participant EM Part 2

**Figure 20.** Participant EM part 2 mean RT compared to the Internal SGOMS/ACT-R Model Predictions. Bars measure time in MS on the x-axis.

**Figure 21.** EM part 2 ranked RT distribution for each condition in EM part 2. We can see UT is faster and the PU condition has been somewhat altered along with 2 and 3 splits. Ranked data distribution shown here is done by all response times being ordered by lowest to highest and then plotted in order.
Study 2

In Study 1, the Internal Model did not consistently match the planning unit first method condition and unit task first condition. This was because participants were able to adjust their micro strategies. Overall, the results indicated that they did so in response to the difficulty in holding the planning unit information in their head. This was reflected in the response times for both the planning unit and unit task first conditions. All other conditions were well predicted so it was concluded very little changes would need to be made to get participants to use the same micro strategies. The External Model was created to model the game with the new changes.

Overall, four significant changes were made from Study 1. In Study 1, RT results and self-reports indicated it was difficult for participants to keep track of or use planning unit (PU) information (the order in which the unit tasks appeared) to speed up the unit task first condition. This provided an opportunity to investigate the effects of external memory aids. This was implemented by adding a visual in-game color cue indicating which of the three unit tasks the player was in by color and their position in the planning unit by the darkness of the color (background planning unit color lightened for each completed unit task). The Four Button app was re-implemented to achieve this game change in Android Studio, maintaining the same visual interface and gameplay but with the added background color cue.

The second addition was the development of a new method to remove outliers. Ranking the reaction time data in ascending order resulted in a flat line showing that most reaction times were close, with a steep increase to high RT values detected and cut-offs by the previous method. However, this method did not consider removing small response
times to the left of the ranked RT distribution caused by guessing. A new data analysis method for removing outliers where the data was first transformed into a normal distribution using the Boc-Cox method (Box & Cox, 1964). The Box-Cox outlier removal method does not rely on the ranked RT distribution, but we can use the ranked RT graph to assess its quality visually (see Appendix C for examples).

The third change was a method to detect Planning Unit (PU) learning by splitting the UT first condition into the first and fourth quarter gameplay data and comparing this to the model predictions. The fourth edition was updating the SGOM/ACT-R Model. There are two possible outcomes to how players respond to the PU color cue. Participants could either be assisted by the color cue to help achieve Internal Model performance (so no change in the SGOMS/ACT-R Model) or match the new model prediction.

The new model, the External Model, claims that players will completely externalize the Planning unit management. This is modeled by having the model cue planning unit information directly from the visual buffer. This new model prediction incorporates the idea of Extended Cognition (Clark & Chalmers, 1998) by the assumption that the External Model uses the external game color cue as a support in processing the unit task order contained in the planning unit. The change in the model equates to the PU first condition reducing to a look time (same as 2 and 3 split conditions) while the UT first condition remains the same. This is a drastic claim because it says no additional time is needed to process the PU to speed up UT's first conditions.
Methods

Three young adults between 18 and 30 and one adult between 30 and 50 were recruited for four participants (3 female and one male). Of the four participants recruited, only two of the four participants’ data were included in the final analysis. Of the two participants data excluded, one participant dropped out early in the training phase due to time limitations. At the same time, the other had not finished training by the time data collection had ended. The young adult participants had a mean age of 27 (SD = 2). The remaining participants who completed training had a mean age of 28 (SD = 1.41) (1 male and one female). The remaining participants trained for a total of 3-4 months. The research ethics board approved the study at Carleton University CUREB-B Clearance # 106484. As a token of appreciation, participants were given refreshments during the interview sessions (if in person). No other compensation was provided.

All participants confirmed to have some level of mobile gameplay experience and proficiency using their mobile devices. All participants had a self-reported experience playing mobile applications. Three of four participants (all young adults) self-reported good video game exposure and skills, while the fourth reported some exposure. All participants had access to the personal Android smartphone they used throughout the study. All participants were native English speakers with normal vision.

Participants were trained to use a simple mobile game at an “expert” level. To train, participants aimed to complete multiple 5-to-10-minute training sessions every week for multiple weeks until they reached expert performance at the final game level. The number of weeks needed for participants to reach expert performance depended on individual participants' progress/learning speed between 8 and 16 weeks. As participants
progressed through training, they met with the researcher when they felt ready to move on to the next level to monitor progress and give feedback. Meetings occurred in a mutually convenient, safe location, over the telephone, or online. Participant training took place in a quiet, distraction-free location of the participants choosing. Some practice sessions would occur during meetings with the researcher, but otherwise, most sessions would occur without the researcher. Digitally recorded gameplay data (reaction times) were collected through the app and sent to the research lab via email using the app's data-sending feature at the participants choosing.

Upon first meeting with the researcher, participants were given a consent form outlining the study requirements. Participants then downloaded and played the Four Button Android app via a Google Play store download link provided by the researcher following signed consent to participate in the study. Next, participants were instructed to open the Four Button App and enter a username assigned by the researcher, read the game instructions in the app, and then begin methods training practice with guidance from the researcher. Participants were asked to play the app with one finger (index finger) of their dominant hand and hold their phone comfortably with the other. Consecutive meetings with the researcher continued as the participants felt ready to move to the next training level. Participants were told not to move on to the next training level until the researcher approved.


Four Button Expert Training

Training participants to play the mobile app was broken into three training phases: one training phase for the top three SGOMS levels (method, unit task, and planning unit). Each training level consisted of multiple ongoing practice 5-10 min training sessions ranging from 2-5 times a week, depending on a participant’s available time. The total number of sessions for each training level depended on an individual’s developed performance. Performance required to move forward to higher training levels (methods, unit task, and planning unit training) was measured by a dynamic combination of accuracy, time, and self-assessment since accuracy and performance on one level were observed to improve upon the practice of a higher level inherently. Thus, lenience in performance was accounted for. Final expert performance was measured by individual speed, reaching a minimum individual reaction time, participants' reaction was generally automatic (required little thought), and accuracy meeting 75-80% for the final game level. The different levels were accessible from the app's main screen, as seen in Figure 22 above. To help assess their progress, participants had access to their progress in the app data viewer, which displayed their accuracy and average time for each method, unit task, and planning unit.

Figure 22. Main menu with a green circle indicating the different training modes in game.
It is worth noting that when participants were well-trained, they did not have to recall coded numbers consciously.

Methods training: This is the first training level participants mastered. Participants were trained to press number sequences on a 2x2 number grid ranging from 1-4 (see Figure 8 on page 14 for a visual) upon a randomly given cue represented by two alphabetic letters. To train, the participants were required to memorize eight different four-button codes and respective cues. The eight method types were broken down into three sets, and a participant could choose to practice each set individually, all together, or two at a time; in any combination; this was to make it less overwhelming for participants to learn the methods and speed up method level learning time.

Unit task training: This is the second training level. Participants were trained to learn three different unit tasks on the same grid and two-letter cue interface as in the methods training. One change from the method's interface was the training options were for the unit tasks rather than the method training groups. Each unit task consisted of a specific grouping of methods and could be practiced individually, in combination, or separately, much like at the methods level.

Planning unit training: This was the final game level participants mastered, and in the mobile game, this section was labeled as a game. There were three planning units. The interface was the same as the previous two training levels; however, there were no training options in this level, and all three planning units would appear during play/training at random. Each planning unit consisted of a specific sequence of three unit tasks, and each of the three planning units was a different sequence. Participants trained in this level by turning off data collecting in the game while they practiced.
Reaction Time Recording and Analysis

Gameplay data were collected as participants trained in the methods and unit task training level and played Game1 in the mobile application. Data was collected for each button press (MS), each completion of a method (MS), unit task (MS), and planning unit (MS) and their corresponding total times (MS), user ID, session number, session type (training type and Game1) mobile device date/time, and accuracy saved into a database (.db) file stored in the game’s files. The Four Button Game mobile application sent the database file to the researcher’s email.

Once data was obtained, it was extracted from the database file using SQLite software and exported as a .csv file. The .csv file was opened, inspected, and processed using Python 3.10 with Pandas, Numpy, Matplotlib, and Scikitlearn Python libraries.

The following steps were applied:

1) Review the accuracy and total counts of each method.

2) Filtered for correct results.

3) Separated data into each experimental condition and stored them in their own panda's data frame for individual outlier removal.

4) Checked the distribution of each condition. This included mean, median, mode, skew, and kurtosis measurements. Frequency charts and numbers can be found in the Appendix section C. Unit task first conditions were selected for first and fourth-quarter gameplay.
5) Applied Box-Cox to each experimental condition. Box-cox transform used was from the third-party package Scipy (link).

6a) Filter outliers for each experimental condition by filtering for data between two cut-offs values A: Cut-offs were calculated by first converting reaction time data for each condition using the box-cox transform, finding the box-cox mean, standard deviation, and retrieving lambda the resulting lambda value from the transform algorithm. Cut-off values in the box-cox transform were 1.75 SD from the mean to calculate a cut-off. See the code snippet in Figures 23 and 24.

6b) Filter outliers by filtering for data between two cut-off values. Part B: Apply cut-offs by using reverse Box-Cox to obtain upper and lower cut-offs. Cut-offs were then transformed back into reaction time numbers using the inverse transform (scipy.stats.boxcox) and the lambda value of that transform. This removes times when participants unexpectedly get interrupted, distracted, zone out, and too fast, such as twitches and guessing.

7) Box-Cox cuts were evaluated visually by plotting the ranked RT distribution for each condition and marking the cut-off point to see how well the plateau (flat, stable line) was selected for in the distribution. Box-Cox evaluations can be viewed in the Appendix C.
Results and Findings

Reaction time data at the methods level for both participants E and B’s final game level (Game1) were used for statistical analysis, and initial raw Game1 data was

```python
def applyBoxCoxCutoff(data):
    ogData = data['TOTAL_METHOD_TIME_MS'].copy()
    bc = boxcox_transform_rt_dict(data)
    data = bc['transformedRTData']
    upperCut = bc['upperCut']
    lowerCut = bc['lowerCut']
    print(lowerCut, upperCut)
    print(bc 'lambda', bc['lambda'])
    print('new skew: ', data.skew(), ' new kurtosis: ', data.kurtosis())
    return ogData.loc[lambda x : (x > lowerCut) & (x < upperCut)]
```

Figure 23. Box-Cox cut-off algorithm part 1. This code snippet applies the cut to the original rt data.

```python
def boxcox_transform_rt_dict(data):
    boxcox_out = stats.boxcox(data['TOTAL_METHOD_TIME_MS'])
    data_boxcox = pd.Series(boxcox_out[0])
    mean, sd, median = data_boxcox.mean(), data_boxcox.std(), data_boxcox.median()
    x = 1.75
    upperCut, lowerCut = (mean + (sd*x)), (mean - (sd*x))
    return {
        "originalRTData": data['TOTAL_METHOD_TIME_MS'],
        "transformedRTData": data_boxcox,
        "meanTransformed": mean,
        "medianTransformed": median,
        "sdTransformed": sd,
        "mean": inv_boxcox(mean, boxcox_out[1]),
        "median": inv_boxcox(median, boxcox_out[1]),
        "sd": inv_boxcox(sd, boxcox_out[1]),
        "upperCutTransformed": upperCut,
        "lowerCutTransformed": lowerCut,
        "upperCut": inv_boxcox(upperCut, boxcox_out[1]),
        "lowerCut": inv_boxcox(lowerCut, boxcox_out[1]),
        "lambda": boxcox_out[1]
    }
```

Figure 24. Box-Cox cut-off algorithm part 2. This code snippet applies the Box-Cox transformation, calculates statistics, and converts values back from the transformation space into rt data.
checked for entry counts, accuracy, and distributions. Participant E had a total of 4281 method entries with an accuracy of 4150/4281 (96.94%), a total of 1046 unit tasks entries with an accuracy of 941/1046 (89.96%), and a total of 326 planning unit entries with an accuracy of 240/326 (73.61%).

Participant B had a total of 2324 method entries with an accuracy of 2255/2324 (97.03%), a total of 573 unit task entries with an accuracy of 522/573 (91.1%), and a total of 182 planning unit entries with an accuracy of 139/182 (73.67%).

Specific methods, unit tasks, planning unit entries, and accuracy for participants E and B can be seen in Table 1 below. Note that the total methods recorded are further divided into unit tasks and planning units; hence, the unit and planning unit counts decrease by their increasing level in the SGOMS theory since planning units are made of unit tasks and unit tasks are made of methods.

Table 1. Participant E and B counts an accuracy for each SGOMS level.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Participant E</th>
<th>Participant B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Known Method</td>
<td>2421/2466 correct (98.18%)</td>
<td>1309/1344 correct (97.4%)</td>
</tr>
<tr>
<td>Two Split</td>
<td>328/365 correct (89.86%)</td>
<td>184/194 correct (94.85%)</td>
</tr>
<tr>
<td>Three Split</td>
<td>325/349 correct (93.12%)</td>
<td>186/195 correct (95.38%)</td>
</tr>
<tr>
<td>Unit Task First</td>
<td>679/701 correct (96.86%)</td>
<td>373/386 correct (96.63%)</td>
</tr>
<tr>
<td>Planning Unit First</td>
<td>397/400 correct (99.25%)</td>
<td>203/205 correct (99.02%)</td>
</tr>
</tbody>
</table>

The results of participants E and B are compared to the External SGOMS/ACT-R model only since participants in study 2 were consistently faster than the PU first condition predicted by the Internal SGOMS/ACT-R model. Results for the PU condition...
indicate participants used a look condition, as the external SGOMS/ACT-R model predicted (see Figures 25 and 27).

**Table 2.** Participant E’s Descriptive Statistics RT for all experimental conditions.

<table>
<thead>
<tr>
<th>Condition</th>
<th>known_methods</th>
<th>two_split</th>
<th>three_split</th>
<th>PUFirst</th>
<th>First Quarter UFirst</th>
<th>Last Quarter UFirst</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>2298.00</td>
<td>291.00</td>
<td>297.00</td>
<td>360.00</td>
<td>147.00</td>
<td>157.00</td>
</tr>
<tr>
<td>mean (MS)</td>
<td>1008.97</td>
<td>1453.09</td>
<td>1528.21</td>
<td>1450.12</td>
<td>1441.11</td>
<td>1190.65</td>
</tr>
<tr>
<td>std</td>
<td>210.40</td>
<td>185.61</td>
<td>201.12</td>
<td>194.80</td>
<td>230.36</td>
<td>232.99</td>
</tr>
<tr>
<td>min (MS)</td>
<td>766.00</td>
<td>1092.00</td>
<td>1241.00</td>
<td>1193.00</td>
<td>1010.00</td>
<td>810.00</td>
</tr>
<tr>
<td>0.25</td>
<td>851.00</td>
<td>1327.00</td>
<td>1384.00</td>
<td>1319.00</td>
<td>1318.00</td>
<td>984.00</td>
</tr>
<tr>
<td>0.50</td>
<td>927.00</td>
<td>1425.00</td>
<td>1476.00</td>
<td>1401.00</td>
<td>1411.00</td>
<td>1194.00</td>
</tr>
<tr>
<td>0.75</td>
<td>1118.00</td>
<td>1535.50</td>
<td>1635.00</td>
<td>1512.50</td>
<td>1552.00</td>
<td>1351.00</td>
</tr>
<tr>
<td>max (MS)</td>
<td>1701.00</td>
<td>2061.00</td>
<td>2177.00</td>
<td>2178.00</td>
<td>2344.00</td>
<td>1845.00</td>
</tr>
<tr>
<td>Conf.</td>
<td>8.64</td>
<td>18.46</td>
<td>24.96</td>
<td>37.24</td>
<td>36.44</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.** Participant B’s Descriptive Statistics RT for all experimental conditions.

<table>
<thead>
<tr>
<th>Condition</th>
<th>known_method</th>
<th>two_split</th>
<th>three_split</th>
<th>PUFirst</th>
<th>First Quarter UFirst</th>
<th>Last Quarter UFirst</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>1195.00</td>
<td>167.00</td>
<td>167.00</td>
<td>186.00</td>
<td>84.00</td>
<td>93.00</td>
</tr>
<tr>
<td>mean (MS)</td>
<td>771.16</td>
<td>1340.01</td>
<td>1271.49</td>
<td>1254.86</td>
<td>1229.40</td>
<td>1107.48</td>
</tr>
<tr>
<td>std</td>
<td>144.71</td>
<td>186.30</td>
<td>168.29</td>
<td>141.87</td>
<td>212.09</td>
<td>231.16</td>
</tr>
<tr>
<td>min (MS)</td>
<td>605.00</td>
<td>1091.00</td>
<td>1035.00</td>
<td>1038.00</td>
<td>880.00</td>
<td>672.00</td>
</tr>
<tr>
<td>0.25</td>
<td>679.00</td>
<td>1212.00</td>
<td>1159.50</td>
<td>1146.75</td>
<td>1109.75</td>
<td>943.00</td>
</tr>
<tr>
<td>0.50</td>
<td>726.00</td>
<td>1276.00</td>
<td>1237.00</td>
<td>1226.50</td>
<td>1185.00</td>
<td>1119.00</td>
</tr>
<tr>
<td>0.75</td>
<td>799.00</td>
<td>1413.50</td>
<td>1346.00</td>
<td>1331.50</td>
<td>1303.25</td>
<td>1231.00</td>
</tr>
<tr>
<td>max (MS)</td>
<td>1286.00</td>
<td>1991.00</td>
<td>1836.00</td>
<td>1757.00</td>
<td>1905.00</td>
<td>2097.00</td>
</tr>
<tr>
<td>Conf.</td>
<td>8.30</td>
<td>22.32</td>
<td>32.24</td>
<td>29.48</td>
<td>45.35</td>
<td>46.98</td>
</tr>
</tbody>
</table>

Look conditions were not precisely predicted for the 2-split and 3-split conditions. The resulting differences between the split conditions were similar but differing by 75-70ms for E and B, respectively. The different split conditions were observed in EM 2 and S model comparison graphs (Figures 13 and 20). An important observation can be seen comparing Participant B’s average split times as evidence the two-split is not inherently faster than the three-split condition. Study 1 showed Hick’s law (Hick, 1952) did not apply to the 2 and 3 split conditions and explained this was probably because possible
split outcomes were not stored in declarative memory as previously molded (Schneider and Anderson, 2011). However, before Participant B’s results, all differences observed in participants thus showed the two-split as faster than the three-split condition. Participant B’s results show the two-split condition is not inherently faster than the two-split but suggest the faster split is just the one that has been better learned (this is why we used the faster split to calculate a participant's look time).

The unit task (UT) first condition of Participant B was not predicted by either the internal or external model (both models predicted the same at times). However, B’s mean UT condition method time was still faster than a look condition, indicating that B was possibly using different strategies than the SGOM/ACT-R UT first condition or a look condition prediction. In verbal reports (Appendix A), Participant E said they covered up the unit task code with their finger to unlearn looking at the end of a unit task. It was also possible participants were using a mix of strategies as they practiced playing the game, and learning to not look in the UT first condition took some time.

The Four button Game (FBG) did not include a training level specific to the planning units (see Figure 22), and participants were asked to practiced planning units in the game with data recording turned off (this was done via a check box in the top right during gameplay, seen in Figure 6). However, this did not guarantee the data collection was turned off as intended, so the UT first method condition RTs for E and B possibly contained training data. The resulting implication of no distinct PU training level meant the PU first condition RT may have been contaminated with training data (RT data did not contain expert data only). So, further analysis of the unit task condition was done by
sorting the data by entry number, dividing the UT first RT data into four parts, applying the Box-Cox cut method, and then calculating means for the first and last quarter.

Participant E

Figure 25. E mean RT times compared to the External SGOMS/ACT-R Model predictions. Bars measure time in MS on the x-axis.

Figure 26. Participant E ranked RT distribution for each condition. We can see PU condition is pulled down to the same speeds as in the splits. This data is raw RT and does not include the box-cox filtered data. Ranked data distribution shown here is done by all response times being ordered by lowest to highest and then plotted in order.
Participant B

**Figure 27.** Participant B mean RT compared to model predictions for all conditions. Bars measure time in MS on the x-axis.

**Figure 28.** Participant B ranked RT distribution for each condition. We can see UT is faster here than splits and PU first is as fast as splits. This data is raw RT and does not include the box-cox filtered data. Ranked data distribution shown here is done by all response times being ordered by lowest to highest and then plotted in order.
The additional UT First analysis for learning evidence showed that participants E and B used the same strategy as in the split conditions in the first half of the PU first quarter, as seen in Figures 25 and 27. In the UT first condition, it appears the external model does a good job predicting E’s method reaction times for all experimental conditions (Figure 25) after learning to the planning unit level. B’s unit task analysis indicates that a looking strategy was used in early gameplay, as their UT Method First Quarter matched the split conditions (Figure 27).

**Model Comparisons and Breakdown**

A high-level overview of the SGOMS/ACT-R code (See Appendix D) can be broken down into three control loops. The three distinct control loops are the methods, unit task, and planning unit control loops. The control loops are nested in nature meaning they occur inside each other; in this case they are two levels deep with a total of the three levels. We can also see the three loops in the SGOMS control flow diagram in Figure 1, note the “end” check in this diagram.

When the ACT-R SGOMS model code begins to run (the Four Button Game begins) it enters the planning unit loop. In the planning unit loop the first planning unit is chosen, and from that planning unit the unit task orders are retrieved and positions in the ordered unit tasks saved, then the model enters the unit task loop.

Next, the unit task control loop is started and loops through the unit tasks retrieved in the planning unit, but the unit task methods are retrieved, and using those, the method loop is entered. The method loop runs through the unit tasks methods in order,
looking only when needed, and finishes with an end cue. End cues mark the end unit tasks and planning units and are internally managed (they don’t appear in the game).

It takes computation time to jump out of a loop to the next higher level and to enter back into a loop at a lower level, and only one jump either above or below can happen at a time. In ACT-R the productions manage moving control loops, handling end conditions, recalculating positions, and firing methods. See Figure 29 for a visual representation of jumping out of a control loop.

Figure 29. A visual representation of the three nested control loops within an SGOMS/ACT-R model. Colored arrows indicate exiting a loop. Moving back into a loop (not included in this diagram) would look like arrows moving inward to the next most center loop, after adjusting for the next item in that loop.
Internal vs. External Model

When a loop hits an end marker it jumps out of the current loop to continue to the next item in the loop one level above. For example, if the model is running a method loop and encounters an end method marker it jumps out of that loop to retrieve the next unit task and begins a new method loop with the methods in that unit task. If the unit task end code is retrieved instead, the model moves to the planning unit level to begin the next task from the order saved in the planning unit.

The first difference between the internal and external SGOMS/ACT-R model is the way in which the next unit is retrieved in the unit task first method condition (UT first). The end conditions are handled the same way by handling the method end code and jumping out of the method loop. Then, the next unit task is retrieved in the unit task loop, and the current unit task is updated. The difference in the External and Internal Models is how the next unit task is retrieved. In the Internal Model used in Study 1, the next unit task is retrieved from the planning unit buffer, and in the External Model, the next unit task is retrieved from the visual buffer.

The second difference is how the models handle the planning unit first method (PU first) condition. In the Internal Model, the planning unit control loop is broken out to set up the new planning unit order (order of unit tasks to execute) which costs three productions to exit each loop and three productions to work back into the loop to start the first method of the new planning unit. In the External Model, the planning unit is never exited because the order of the planning units is managed by the visual system, which learns to pair the unit task orders with external color cues, and the model treats the PU first condition as a look.
The two models in this research predict all the conditions in this experiment to be the same response time, except for the planning unit first (PU first) method conditions. The Internal Model predicts 400ms of production fire time while the External Model predicts 200ms.

Model prediction times for the unit task first condition are the same for the internal and external models. UT first-time estimation is calculated using individual hit time (MS), one production for the hit (50ms), and the other 3 productions (150ms). Productions other than the motor response included in the calculation are 1) production to handle the end of the last method entered, production to handle the end of the current unit task, and a production to retrieve the next planning unit task. The internal model retrieves the next unit task from the planning unit buffer. In the external model, the next UT is retrieved from the visual color cue info stored in the visual buffer (See Models in Appendix D).

Model predictions for the planning (PU) unit first condition derived from the Internal model include calculated hit time (MS), calculated look time (MS), one production for the hit (50ms), one production for looking (50ms), and six other productions (300ms). The 300ms of production time includes 3 productions for handling end conditions, and 3 productions for setting up the start of each loop (first method, first unit task, and the planning unit). The External Model’s PU first condition is estimated using a calculated look time (MS), one production for the hit (50ms), and another for the
look (50ms). See Figure 30 to visually compare the Internal and External SGOM/ACT-R predictions for each experimental condition.

**Figure 30.** A comparison of the predicted times between the Internal and External SGOMS/ACT-R Models. Notice how the only difference is the time estimation between the planning unit (PU) first method condition, with the Internal model giving a longer time estimate due to more productions required. The times for the hit and look conditions have been filled in with 950 MS and 600 MS respectively, as filler example.

**Figure 31.** All participants mean RT for all conditions. Conditions are labeled by color with the X-axis time in MS. Participants have varying base response times which makes comparisons difficult, but noting the known method across participants helps to visually compare.
Conclusion

Overall, results show the experimental design change (adding a color cue to the planning unit level) was well warranted. Changes were evident as the response time for the planning unit first method condition was dramatically reduced in Study 2 compared to Study 1 participants (see Figure 31). In addition, the new statistical method for removing outliers seemed to work. Additionally, the Box-Cox outlier removal method improved outlier removal by removing both guessing (small values) from the RT data and high RT values.

The results also indicate that learning seems to move toward the External SGOMS/ACT-R Model. Early gameplay of both E and B showed a match to a look micro strategy for the UT first method, and E employed the strategy predicted by the model in late gameplay. Participant B moved in the direction of the model prediction. Previous GOMS models can provide learning time estimates, but here, we show a potential to provide time estimates for control structures outside of the GOMS level of analysis. It would be possible to measure individual PU learning time with the look (or equivalent) condition as a starting point and use an individual’s learning curves to predict the time needed to learn a cue in a task.

How does our evidence indicate the need to remove the planning unit buffer? The internal model uses a planning unit buffer to store the order of the unit tasks, which is updated right before the first method of a planning unit is executed. Study 1 concluded keeping track of the planning unit was difficult for participants. Since the change to the model and game in study 1 relied on the assumption of a planning unit buffer, we theorized this was because there was no dedicated planning unit buffer and so
participants had to come up with other micro strategies to hold the planning unit information (e.g., through rehearsal).

In Study 2 it was possible that the participants would follow the Internal Model and use the color cue to help them maintain or keep track of the planning unit information. However, the results show that participants matched the External Model. The External Model assumes the visual system learns and handles the order of the unit tasks in the planning unit, which is present in the visual buffer with the color cue. This means that planning unit information is offloaded to the visual system. This helps define the relationship between Extended Cognition and Micro Strategies because the External Model is an example of how external cognition can more quickly elicit the required information. This suggests that computation offloading to external representations is preferred when the ability to do so is present.
Appendices

A: Verbal Reports

Verbal data collection occurred when participants actively gave feedback on their own Four Button App (FBA) Experiences or were asked directly how their playing and learning experiences were with the FBA.

Participant E:

- “The best way to not look when needed is to play and try not to look when practicing. Only look when I have to.”

- “I practiced my unit tasks all together at random, and when I moved to game one, it felt like I had to unlearn to look; I think I formed the habit of looking every time I finished a unit task. I should have practiced one unit task at a time and not all together presented randomly.”

- “I formed the habit of exiting the game every time I make a mistake.”

- “Short HW unit task seems harder to do. Feels too short, and the look condition at the end makes me look at the next unit tasks method even if I know what's coming next.”

- “I wish there was a planning unit practice before game one so I could practice the planning units individually.”

- “As I go faster in game 1, I make more mistakes.”

- “I found I learned a habit of looking while training in the unit task because when I trained in the unit task, they were at random, so I always looked every time I finished a
unit task. Probably would have been easier to practice only one unit task at a time, then I would know it was always next.”

Participant B:

- “Rhythm helped keep the motor movement from getting too fast. Otherwise, it was too easy to get off sync.”

- Noted the need for a very distraction-free environment, low stimulus. The participants said they were “easily thrown off.”

- “AK unit tasks speed/movement is swift.”

- Noted that when learning planning units, found colors changing helped to know when a unit task was over, or a new one was starting.

There are two pu problems: green and purple pu has a possible hw to rp, and when it has su, it messes me up immediately to another su.

- was advised to practice unit tasks one at a time…
Appendix B: RT Frequency Distributions

Participant E

E known_methods Condition Distribution

E three_split Condition Distribution
E PU First Condition Distribution

- Mean 1576 ms
- Median 1417 ms
- Mode 1251 ms

E Reaction Time (ms) KDE

```
UTFirst_bocCox_cut['TOTAL_METHOD_TIME_MS'].plot.kde(bw_method=0.2, title = "E Reaction Time (ms) KDE")
```
Participant B
UTFirst_bocCox_cut["TOTAL_METHOD_TIME_MS"].plot.hist(bins=50, alpha=0.5, title = "B Reaction Time (ms) Frequency Histogram")

UTFirst_bocCox_cut["TOTAL_METHOD_TIME_MS"].plot.kde(bw_method=0.2, title = "B Reaction Time (ms) KDE")
### Section C: Box-Cox Transformation and Cut Results

#### E Box-Cox Cut

<table>
<thead>
<tr>
<th>Condition</th>
<th>known_methods</th>
<th>UTFirst Q1</th>
<th>UTFirst Q4</th>
<th>two_split</th>
<th>three_split</th>
<th>PUFist</th>
</tr>
</thead>
<tbody>
<tr>
<td>min cut</td>
<td>765.621</td>
<td>974.644</td>
<td>802.830</td>
<td>1002.343</td>
<td>1238.381</td>
<td>1185.470</td>
</tr>
<tr>
<td>max cut</td>
<td>1703.719</td>
<td>2382.739</td>
<td>1894.618</td>
<td>2084.079</td>
<td>2256.493</td>
<td>2235.075</td>
</tr>
<tr>
<td>lambda</td>
<td>-2.513</td>
<td>-0.959</td>
<td>-0.871</td>
<td>-0.139</td>
<td>-2.757</td>
<td>-2.912</td>
</tr>
<tr>
<td><strong>Skew</strong></td>
<td>0.000</td>
<td>-0.14716</td>
<td>-0.011</td>
<td>-0.022</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td><strong>Kurtosis</strong></td>
<td>0.000</td>
<td>1.642</td>
<td>-0.091</td>
<td>2.290</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

#### B Box-Cox Cut

<table>
<thead>
<tr>
<th>Condition</th>
<th>known_methods</th>
<th>UTFirst Q1</th>
<th>two_split</th>
<th>three_split</th>
<th>PUFist</th>
</tr>
</thead>
<tbody>
<tr>
<td>min cut</td>
<td>603.518</td>
<td>873.064</td>
<td>1084.388</td>
<td>1031.493</td>
<td>1035.029</td>
</tr>
<tr>
<td>max cut</td>
<td>1287.958</td>
<td>1921.552</td>
<td>1995.058</td>
<td>1848.851</td>
<td>1769.985</td>
</tr>
<tr>
<td>lambda</td>
<td>-2.63832</td>
<td>1.126586</td>
<td>-2.88171</td>
<td>-2.70114</td>
<td>-2.82257</td>
</tr>
<tr>
<td><strong>Skew</strong></td>
<td>0.000</td>
<td>-0.097</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td><strong>Kurtosis</strong></td>
<td>0.000</td>
<td>1.422</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
E’s Data Box-Cox Cuts (displayed in RT ranked distribution)
B’s Data Box-Cox Cuts (displayed in RT ranked distribution)
Section D: Source Code and Links

Source Code Links

Four Button Game Android Play Store link: https://github.com/EliReid/FourButtonApp_Version2.git

Four Button Game source code link: https://github.com/rlwest/GamePlayer.git

SGOM/ACT-R source code: https://github.com/rlwest/GamePlayer.git

Internal SGOMS/ACT-R Model Code

import sys

import python_actr

from EmilyMotorModule import *
from RTModule import *
from python_actr.actr import *
from random import randrange, uniform

class MyAgent(ACTR):

    # BUFFERS
    focus=Buffer()
    b_context = Buffer()
    b_plan_unit = Buffer()
    b_plan_unit_order = Buffer()
    b_unit_task = Buffer()
    b_method = Buffer()
    b_operator = Buffer()
    b_DM = Buffer()
    b_motor = Buffer()
    b_visual = Buffer()

    motor = EmilyMotorModule(b_motor)
    DM = Memory(b_DM)

    # initial buffer contents
b_context.set('status:unoccupied planning_unit:none')

b_plan_unit.set('planning_unit:P unit_task:P state:P type:P')

b_visual.set('00')

b_plan_unit_order.set('counter:oo first:oo second:oo third:oo fourth:oo')

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

def START_start(b_context='status:unoccupied planning_unit:none'):
    b_unit_task.set('unit_task:START state:running')
    b_context.modify(status='starting_game')
    print ('Look at code to determin new planning unit')
    motor.see_code()

def START_AK(b_context='status:starting_game planning_unit:none',
             b_unit_task='unit_task:START state:running',
             b_method='state:finished',
             b_visual='AK'): 
b_plan_unit.modify(planning_unit='AK', unit_task='AK', state='begin_sequence', type='ordered')

b_context.modify(status='occupied', planning_unit='AK')

print ('run_AK_PU')

b_plan_unit_order.set('counter:one first:AK second:HW third:RP fourth:finished')

############ new buffer

def START_RP(b_context='status:starting_game planning_unit:none',
             b_unit_task='unit_task:START state:running',
             b_method='state:finished',
             b_visual='RP'):

    b_plan_unit.modify(planning_unit='RP', unit_task='RP', state='begin_sequence', type='ordered')

    b_context.modify(status='occupied', planning_unit='RP')

    print ('run_RP_PU')

    b_plan_unit_order.set('counter:one first:RP second:HW third:AK fourth:finished')

    ############ new buffer
def START_HW(b_context='status:starting_game planning_unit:none',
              b_unit_task='unit_task:START state:running',
              b_method='state:finished',
              b_visual='HW'):

    b_plan_unit.modify(planning_unit='HW', unit_task='HW', state='begin_sequence', type='ordered')
    b_plan_unit_order.set('counter:one first:HW second:RP third:AK fourth:finished')

    b_context.modify(status='occupied', planning_unit='HW')

    print('run_HW_PU')
    print

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

    ## manage the sequence if it is an ordered planning unit stored in DM
    ## not used in this model (removed for simplicity)

    ## these manage the sequence if it is an ordered planning unit stored in buffer
def setup_first_unit_task(b_plan_unit='unit_task:?unit_task state:begin_sequence
type:ordered'):

    b_unit_task.set('unit_task:?unit_task state:start type:ordered')

    b_plan_unit.modify(state='running')

    print ('fast - start first unit task')

def request_second_unit_task(b_plan_unit='state:running',
                             b_unit_task='unit_task:?unit_task state:finished type:ordered',
                             b_plan_unit_order='counter:one first:?first second:?second third:?third
                                         fourth:?fourth'):

    b_unit_task.set('unit_task:?unit_task state:finished type:ordered',
                    b_plan_unit_order='counter:one first:?first second:?second third:?third
                                         fourth:?fourth')

    b_plan_unit.modify(counter='two')

    print ('fast - start second unit task')

def request_third_unit_task(b_plan_unit='state:running',
                             b_unit_task='unit_task:?unit_task state:finished type:ordered',
                             b_plan_unit_order='counter:two first:?first second:?second third:?third
                                         fourth:?fourth',):

    b_unit_task.set('unit_task:?unit_task state:finished type:ordered',
                    b_plan_unit_order='counter:two first:?first second:?second third:?third
                                         fourth:?fourth')

    b_plan_unit.modify(counter='three')

    print ('fast - start third unit task')

def request_fourth_unit_task(b_plan_unit='state:running',
                             b_unit_task='unit_task:?unit_task state:finished type:ordered',
                             b_plan_unit_order='counter:three first:?first second:?second third:?third
                                         fourth:?fourth',)

    b_unit_task.set('unit_task:?unit_task state:finished type:ordered',
                    b_plan_unit_order='counter:three first:?first second:?second third:?third
                                         fourth:?fourth')

    b_plan_unit.modify(counter='three')

    print ('fast - start fourth unit task')
b_unit_task='unit_task:finished state:finished type:ordered',

b_plan_unit_order='counter:three first:second second:third third:fourth'

b_plan_unit_order.modify(counter='four')

b_unit_task.set('fourth state:start type:ordered')

print ('fast - start fourth unit task')

## these manage planning units that are finished #

```python
def last_unit_task_ordered_plan(b_plan_unit='planning_unit:planning_unit',
                               b_unit_task='finished state:finished type:ordered'):
    print ('finished planning unit =');
    print (planning_unit)
    b_unit_task.set('stop')
    b_context.modify(status='unoccupied', planning_unit='none')

choices = ['AK','RP','HW']
x=random.choice(choices)
motor.referee_action('display', 'state', x)
```

# referee
# AK unit task AK-WM-SU-ZB-FJ

## add condition to fire this production

```python
def AK_ordered(b_unit_task='unit_task:AK state:start type:ordered'):
    b_unit_task.modify(state='begin')
    print('start unit task AK')

def AK_start(b_unit_task='unit_task:AK state:begin'):
    b_unit_task.set('unit_task:AK state:running')
    focus.set('AK')
    target='response'
    content='AK-AK-1234'
    motor.enter_response(target, content)

def AK_WM(b_unit_task='unit_task:AK state:running',
          vision_finst='state:finished',
          focus='AK'):
    focus.set('WM')
```
target='response'
content='AK-WM-1432'
motor.enter_response(target, content)

def AK_SU(b_unit_task='unit_task:AK state:running',
    vision_finst='state:finished',
    focus='WM'):
    focus.set('SU')
target='response'
content='AK-SU-4123'
motor.enter_response(target, content)

def AK_ZB(b_unit_task='unit_task:AK state:running',
    vision_finst='state:finished',
    focus='SU'):
    focus.set('ZB')
target='response'
content='AK-ZB-2143'
motor.enter_response(target, content)

def AK_FJ(b_unit_task='unit_task:AK state:running',
    vision_finst='state:finished',
    focus='ZB'):
focus.set('done')

target='responce'

content='AK-FJ-3214'

motor.enter_response(target, content)

def AK_finished_ordered(b_unit_task='unit_task:AK state:running', vision_finst='state:finished', focus='done'):
    print ('finished unit task AK(ordered)')
    b_unit_task.set('unit_task:AK state:finished type:ordered')

#
# YP-FJ
#
# RP unit task RP-SU<
#
# ZB-WM

def RP_ordered(b_unit_task='unit_task:RP state:start type:ordered'):
    b_unit_task.modify(state='begin')
print ('start unit task RP')

def RP_start(b_unit_task='unit_task:RP state:begin'):
    b_unit_task.set('unit_task:RP state:running')
    focus.set('RP')
    target='responce'
    content='RP-RP-4321'
    motor.enter_response(target, content)

def RP_SU(b_unit_task='unit_task:RP state:running',
            vision_finst='state:finished',
            focus='RP'):
    focus.set('SU')
    target='responce'
    content='RP-SU-4123'
    motor.enter_response(target, content)

### Unknown code

def RP_identify2(b_unit_task='unit_task:RP state:running',
                 vision_finst='state:finished',
                 focus='SU'):

    ################################### referee
choices = ['YP','ZB']
x=random.choice(choices)
motor.referee_action('display', 'state', x)

def RP_YP(b_unit_task='unit_task:RP state:running',
          vision_finst='state:finished',
          focus='code_seen',
          b_visual='YP'):
    focus.set('YP')
target='responce'
content='RP-YP-3412'
motor.enter_response(target, content)

def RP_FJ(b_unit_task='unit_task:RP state:running',
          vision_finst='state:finished',
          focus='YP'):
    focus.set('done')
target='responce'
content='RP-FJ-3214'
motor.enter_response(target, content)

def RP_ZB(b_unit_task='unit_task:RP state:running',
         vision_finst='state:finished',
         focus='code_seen',
         b_visual='ZB'):
    focus.set('ZB')
    target='response'
    content='RP-ZB-2143'
    motor.enter_response(target, content)

def RP_WM(b_unit_task='unit_task:RP state:running',
         vision_finst='state:finished',
         focus='ZB'):
    focus.set('done')
    target='response'
    content='RP-WM-1432'
    motor.enter_response(target, content)

def RP_finished_ordered(b_unit_task='unit_task:RP state:running',
                        vision_finst='state:finished',
                        focus='done'):
    print ('finished unit task RP(ordered)')
b_unit_task.set('unit_task:RP state:finished type:ordered')

################################################################

##### HW Unit Task ######

################################################################

# / FJ
# HW unit task HW-YP--- ZB
# \ SU

def HW_ordered(b_unit_task='unit_task:HW state:start type:ordered'):
    b_unit_task.modify(state='begin')
    print ('start unit task HW')

    ## the first production in the unit task must begin this way

def HW_start(b_unit_task='unit_task:HW state:begin'):
    b_unit_task.set('unit_task:HW state:running')
    focus.set('HW')
    target='responce'
    content='HW-HW-2341'
    motor.enter_response(target, content)

def HW_YP(b_unit_task='unit_task:HW state:running',
vision_finst='state:finished',

focus='HW'):

focus.set('YP')

target='response'

content='HW-YP-3412'

motor.enter_response(target, content)

### Unknown code

def HW_identify3(b_unit_task='unit_task:HW state:running',

    vision_finst='state:finished',

    focus='YP'):

    #################################### referee

    choices = ['FJ','SU','ZB']

    x=random.choice(choices)

    motor.referee_action('display', 'state', x)

    #################################### referee

    motor.see_code()

    focus.set('code_seen')

    print ('waiting to see if FJ, SU, or ZB')

    #### FJ or SU or ZB then end
def HW_FJ(b_unit_task='unit_task:HW state:running',
    vision_finst='state:finished',
    focus='code_seen',
    b_visual='FJ'):
    focus.set('done')
    target='responce'
    content='HW-FJ-3214'
    motor.enter_response(target, content)

def HW_SU(b_unit_task='unit_task:HW state:running',
    vision_finst='state:finished',
    focus='code_seen',
    b_visual='SU'):
    focus.set('done')
    target='responce'
    content='HW-SU-4123'
    motor.enter_response(target, content)

def HW_ZB(b_unit_task='unit_task:HW state:running',
    vision_finst='state:finished',
    focus='code_seen',
    b_visual='ZB'):
focus='code_seen',

b_visual='ZB'):

    focus.set('done')
    target='responce'
    content='HW-ZB-2143'
    motor.enter_response(target, content)

def HW_finished_ordered(b_unit_task='unit_task:HW state:running',

    vision_finst='state:finished',
    focus='done'):

    print ('finished unit task HW(ordered)')
    b_unit_task.set('unit_task:HW state:finished type:ordered')

import sys

import python_actr

from python_actr.actr import *

from random import randrange, uniform


import sys

import python_actr

from python_actr.actr import *

from random import randrange, uniform


### MOTOR MODULE ####

class EmilyMotorModule(python_actr.Model): # defines actions in the environment
def referee_action(self, env_object, slot_name, slot_value):
    x = self.parent.parent[env_object]
    setattr(x, slot_name, slot_value)
    print('[referee]')
    print('object=', env_object)
    print('slot=', slot_name)
    print('value=', slot_value)

def see_code(self):
    self.parent.parent.vision_finst.state = 'busy'  # register that the vision system is busy
    yield 0.47
    print('[vision - looking]')
    code = self.parent.parent.display.state  # get the code from the state slot of the display object
    self.parent.b_visual.set(code)  # put code into visual buffer
    self.parent.b_method.set('state:finished')
    print('[vision - I see the code is..', code, ']')
    self.parent.parent.vision_finst.state = 'finished'
def enter_response(self, env_object, slot_value):
    self.parent.parent.vision_finst.state = 'busy'
    yield 0.63
    x = eval('self.parent.parent.' + env_object)
    x.state = slot_value
    print (env_object);
    print ('[motor entering',slot_value, ']')
    self.parent.parent.vision_finst.state = 'finished'

#### This resets the finst state indicating the action is finished
#### Currently using the vision finst for all actions (so no interleaving or parallel)

def vision_finst_reset(self):
    self.parent.parent.vision_finst.state = 're_set' # reset the vision_finst
    print('['motor module] vision_finst reset')

---

**External SGOMS/ACT-R Model Code**

import sys
import python_actr
from ExternalMotorModule import *

from RTModule import *

from python_actr.actr import *

from random import randrange, uniform

class MyAgent(ACTR):

    # BUFFERS
    focus = Buffer()
    b_context = Buffer()
    b_unit_task = Buffer()
    b_method = Buffer()
    b_operator = Buffer()
    b_DM = Buffer()
    b_motor = Buffer()
    b_visual = Buffer()

    motor = ExternalMotorModule(b_motor)
    DM = Memory(b_DM)

    # initial buffer contents
    b_context.set('status:unoccupied')
    b_visual.set('code:00')
def START_start(b_context='status:unoccupied'):
    b_unit_task.set('unit_task:START state:running')
    b_context.modify(status='planning_unit_start')
    print ('Look at code to determin new planning unit and unit task')

# Planning unit start choices contain a color cue, represents prosesed visual info
choices = ['code:AK color:purple',
            'code:RP color:blue',
            'code:HW color:green']
x=random.choice(choices)
motor.referee_action('display', 'state', x)

motor.see_code()
b_method='state:finished',

b_visual='code:?code color:?color'):

b_context.modify(status='occupied')

b_unit_task.set('unit_task:?code state:start type:ordered position:start')

print ("Starting " + color + " planning unit with " + code + " unit task")

# Planning unit set up has been removed

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

def request_second_unit_task(b_unit_task='unit_task:?unit_task state:finished type:ordered position:start',
                              b_visual='second:?second'):

    b_unit_task.set('unit_task:?second state:start type:ordered position:second')

    print ('fast - start second unit task ' + second)

def request_third_unit_task(b_unit_task='unit_task:?unit_task state:finished type:ordered position:second',
                            b_visual='third:?third'):

    b_unit_task.set('unit_task:?third state:start type:ordered position:third')

    print ('fast - start third unit task ' + third)
def request_fourth_unit_task(b_unit_task='unit_task:?unit_task state:finished type:ordered position:third',
                            b_visual='fourth:?fourth'):
    b_unit_task.set('unit_task:?fourth state:start type:ordered')
    print ('fast - start fourth unit task ' + fourth)

    ## manage the sequence if it is an ordered planning unit stored in DM
    ## not used in this model (removed for simplicity)

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

    ## these manage planning units that are finished ###################################

    #TODO: Look part of PU first condition - use as before. Check for color at ut change

    def last_unit_task_ordered_plan(b_unit_task='unit_task:finished state:start type:ordered',
                                    b_visual='color:?color'):

        print ('finished planning unit =');
        print (color)

        b_unit_task.set('unit_task:START state:running')
b_context.modify(status='planning_unit_start')
print ('Look at code to determine new planning unit and unit task')

####################################
# Planning unit start choices contain a color cue, represents prosesed visual info
choices = ['code:AK color:purple',
           'code:RP color:blue',
           'code:HW color:green']
x=random.choice(choices)
motor.referee_action('display', 'state', x)
#################################### referee
b_visual.clear() # Done here, simulating what visual buffer would do before seeing code
motor.see_code()

####################################
##### AK UT #####
####################################

# AK unit task AK-WM-SU-ZB-FJ

## add condition to fire this production

def AK_ordered(b_unit_task='unit_task:AK state:start type:ordered'):
b_unit_task.modify(state='begin')

print ('start unit task AK')

def AK_start(b_unit_task='unit_task:AK state:begin position:?position'):
    b_unit_task.set('unit_task:AK state:running position:?position')
    focus.set('AK')
    target='response'
    content='AK-AK-1234'
    motor.enter_response(target, content)

def AK_WM(b_unit_task='unit_task:AK state:running',
         vision_finse='state:finished',
         focus='AK'):
    focus.set('WM')
    target='response'
    content='AK-WM-1432'
    motor.enter_response(target, content)

def AK_SU(b_unit_task='unit_task:AK state:running',
         vision_finse='state:finished',
         focus='WM'):
    focus.set('SU')
    target='response'
content='AK-SU-4123'

motor.enter_response(target, content)

def AK_ZB(b_unit_task='unit_task:AK state:running',
       vision_finst='state:finished',
       focus='SU'):
    focus.set('ZB')
    target='responce'
    content='AK-ZB-2143'
    motor.enter_response(target, content)

def AK_FJ(b_unit_task='unit_task:AK state:running',
       vision_finst='state:finished',
       focus='ZB'):
    focus.set('done')
    target='responce'
    content='AK-FJ-3214'
    motor.enter_response(target, content)

def AK_finished_ordered(b_unit_task='unit_task:AK state:running position:?position',
                        vision_finst='state:finished',
                        focus='done'):
    print ('finished unit task AK(ordered)')
b_unit_task.set('unit_task:AK state:finished type:ordered position:?position')

def RP_ordered(b_unit_task='unit_task:RP state:start type:ordered'):
    b_unit_task.modify(state='begin')
    print ('start unit task RP')

def RP_start(b_unit_task='unit_task:RP state:begin position:?position'):
    b_unit_task.set('unit_task:RP state:running position:?position')
    focus.set('RP')
    target='responce'
    content='RP-RP-4321'
    motor.enter_response(target, content)
def RP_SU(b_unit_task='unit_task:RP state:running',
    vision_finst='state:finished',
    focus='RP'):
    focus.set('SU')
    target='response'
    content='RP-SU-4123'
    motor.enter_response(target, content)

### Unknown code

def RP_identify2(b_unit_task='unit_task:RP state:running',
    vision_finst='state:finished',
    focus='SU',
    b_visual='color:?color'):

    choices = ['code:YP', 'code:ZB']
    x = random.choice(choices)
    motor.referee_action('display', 'state', x + ' color:' + color)

    motor.see_code()
    focus.set('code_seen')
    print('waiting to see if YP or ZB')
def RP_YP(b_unit_task='unit_task:RP state:running',
    vision_finst='state:finished',
    focus='code_seen',
    b_visual='code:YP'):
    focus.set('YP')
    target='responce'
    content='RP-YP-3412'
    motor.enter_response(target, content)

def RP_FJ(b_unit_task='unit_task:RP state:running',
    vision_finst='state:finished',
    focus='YP',
    b_visual='code:YP'):
    focus.set('done')
    target='responce'
    content='RP-FJ-3214'
    motor.enter_response(target, content)

def RP_ZB(b_unit_task='unit_task:RP state:running',
    vision_finst='state:finished',
    focus='code_seen',
    b_visual='code:ZB'):
    focus.set('ZB')
    target='responce'
content='RP-ZB-2143'

motor.enter_response(target, content)

def RP_WM(b_unit_task='unit_task:RP state:running',
        vision_finst='state:finished',
        focus='ZB'):
    focus.set('done')
target='response'
content='RP-WM-1432'
motor.enter_response(target, content)

def RP_finished_ordered(b_unit_task='unit_task:RP state:running position:?position',
                        vision_finst='state:finished',
                        focus='done'):
    print ('finished unit task RP(ordered)')
    b_unit_task.set('unit_task:RP state:finished type:ordered position:?position')

#############################
##### HW Unit Task ######
#############################
#
# / FJ
#
# HW unit task HW-YP--- ZB
def HW_ordered(b_unit_task='unit_task:HW state:start type:ordered'):
    b_unit_task.modify(state='begin')
    print ('start unit task HW')

## the first production in the unit task must begin this way
def HW_start(b_unit_task='unit_task:HW state:begin position:?position'):
    b_unit_task.set('unit_task:HW state:running position:?position')
    focus.set('HW')
    target='responce'
    content='HW-HW-2341'
    motor.enter_response(target, content)

def HW_YP(b_unit_task='unit_task:HW state:running',
          vision_finst='state:finished',
          focus='HW'):
    focus.set('YP')
    target='responce'
    content='HW-YP-3412'
    motor.enter_response(target, content)

### Unknown code
def HW_identify3(b_unit_task='unit_task:HW state:running',
    vision_finst='state:finished',
    focus='YP',
    b_visual='color:?color'):
    ################################### referee
    choices = ['code:FJ','code:SU','code:ZB']
    x=random.choice(choices)
    motor.referee_action('display', 'state', x+' color:'+color)
    ################################### referee
    motor.see_code()
    focus.set('code_seen')
    print ('waiting to see if FJ, SU, or ZB')

    #### FJ or SU or ZB then end

def HW_FJ(b_unit_task='unit_task:HW state:running',
    vision_finst='state:finished',
    focus='code_seen',
    b_visual='code:FJ'):
    focus.set('done')
    target='response'
content='HW-FJ-3214'
motor.enter_response(target, content)

def HW_SU(b_unit_task='unit_task:HW state:running', vision_finst='state:finished', focus='code_seen', b_visual='code:SU'):
    focus.set('done')
target='response'
content='HW-SU-4123'
motor.enter_response(target, content)

def HW_ZB(b_unit_task='unit_task:HW state:running', vision_finst='state:finished', focus='code_seen', b_visual='code:ZB'):
    focus.set('done')
target='response'
content='HW-ZB-2143'
motor.enter_response(target, content)
def HW_finished_ordered(b_unit_task='unit_task:HW state:running position:?position',
    vision_finst='state:finished',
    focus='done'):

    print ('finished unit task HW(ordered)')
    b_unit_task.set('unit_task:HW state:finished type:ordered position:?position')

import sys
import python_actr
from python_actr.actr import *
from random import randrange, uniform

class ExternalMotorModule(python_actr.Model): # defines actions in the environment

    def referee_action(self, env_object, slot_name, slot_value):
        x = self.parent.parent[env_object]
        setattr(x, slot_name, slot_value)
print('[referee]')

print('object=','env_object')

print('slot=','slot_name')

print('value=','slot_value')

### This sees the code, which is a value in the state slot of the display object

def see_code(self):
    self.parent.parent.vision_finst.state = 'busy' # register that the vision system is busy
    yield 0.47
    print('[vision - looking]')
    code = self.parent.parent.display.state # get the code from the state slot of the display object

    # process state retrieved from display
    visual_memory = {'purple':'first:AK second:HW third:RP fourth:finished',
                     'blue':'first:RP second:HW third:AK fourth:finished',
                     'green':'first:HW second:RP third:AK fourth:finished'}
    retrieved_visual_memory = '+visual_memory[code.split()[1].split(':')][1]]
    print('[vision - I see the code is..','code,']')
    code = code + retrieved_visual_memory

    # put code into visual buffer code slot
    self.parent.b_visual.set(code)
self.parent.b_method.set('state:finished')

self.parent.parent.vision_finst.state = 'finished'

##### This enters the code

def enter_response(self, env_object, slot_value):
    self.parent.parent.vision_finst.state = 'busy'
    yield 0.63
    x = eval('self.parent.parent.' + env_object)
    x.state = slot_value
    print ('[motor-entering', slot_value, ']')
    self.parent.parent.vision_finst.state = 'finished'

##### This resets the finst state indicating the action is finished

##### Currently using the vision finst for all actions (so no interleaving or parallel)

def vision_finst_reset(self):
    self.parent.parent.vision_finst.state = 're_set'  # reset the vision_finst
    print('[motor module] vision_finst reset')
References

http://act-r.psy.cmu.edu/


An ACT-R Process Model of the Signal Duration Phenomenon of Vigilance. (n.d.).

https://doi.org/10.1177/1541931214581191


https://doi.org/10.1037/0033-295X.111.4.1036


https://doi.org/10.1016/j.cogsys.2007.06.001


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


West, R. L., & Macdougall, K. (2014). The macro-architecture hypothesis: Modifying Newell’s system levels to include macro-cognition. Biologically Inspired Cognitive Architectures, 8, 140-149. doi:10.1016/j.bica.2014.03.009


