# Using ACT-R and SGOMS to Predict Micro Strategies Used by Experts During Routine Tasks

by

Emily Greve

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

Cognitive Science

Carleton University
Ottawa, Ontario

## Abstract

Micro strategies refer to fast, low level, unconscious strategies that constitute a control structure for information processing within a specific task (Newell, 1973). Micro strategies determine how we process low level information, such as perceptual inputs (Shiffrin & Cousineau, 2004) or procedural information (Gray & Boehm-Davis, 2000). Using an SGOMS/ACT-R model I was able to predict expert, individual performance on a stimulus-response reaction time game involving a memorized hierarchical structure. I found that slightly different micro strategies were used by the participants, indicating that micro strategies have a very real influence on individual performance. These findings represent an understudied area of research that has larger implications for many Psychology and Cognitive Science studies.

**Acknowledgements**

I have so many thanks to give. First and foremost, Thank you to Dr. Rob West. Without your support and guidance this thesis would not have been possible. I have learned so much from you and these lessons will be with me all throughout my life.

I would like to give special thanks to Elizabeth Reid, without her wealth of Python knowledge I would still be stuck on ‘hello world’.

I need to give a HUGE thank you to my participants. It was hard finding people willing to give up so much of their time to a weird mobile game, but you took to this project without hesitation. I am eternally grateful that you stuck through it to the end.

I have too many wonderful friends and family to fit naming them all here, but to each and every one of you thank you. I only got this far because of all of the endless support you all have given me.

## Table of Contents

## List of Tables

## List of Figures

## List of Appendices

# Chapter 1: Introduction

Often in studies on cognitive tasks many participants are used, and their results are averaged together to deal with variation in the results. This variation is generally interpreted as noise. However, Newell (1973) argued that it could also represent the result of different micro strategies. Micro strategies are low level strategies that take place at the millisecond scale and can vary during simple cognitive tasks (Gray & Boehm-Davis, 2000). Newell (1973) originally referred to micro strategies as *Methods*, but the term *Methods* was later used to refer to a unit of action within a GOMS model (Card et al., 1983) which is more limited in scope. I will use the term *micro strategies* and reserve the term *Methods* for use in my model (a type of GOMS model built in ACT-R).

The concept micro strategies, as referred to in this paper, is specific to the study of cognitive architectures (Vera et al., 2004). Micro strategies refers to very fast, low level, unconscious differences between individuals. Micro strategies are ways of creating a control structure for processing basic information from a task (Newell, 1973) as well as low level information such as basic perceptual inputs (Shiffrin & Cousineau, 2004) or procedural information (Gray & Boehm-Davis, 2000). However, Micro strategies are not what researchers normally consider to be strategic 'choice'. In ACT-R micro strategies are implemented as specific patterns of productions that call modules such as memory or motor action in a strategic way to accomplish a task (Anderson, 2009).

The issue is, if people use different micro strategies, then you cannot use the averaged data (e.g., reaction times) to gain insight into the internal cognitive steps involved in the task. For example, in a task where a participant must respond to a series of stimuli with specific

corresponding responses, if one specific stimulus/response pattern is always followed by another specific stimulus/response pattern, then to respond the second stimuli the participant need only see that a stimulus appears: If they have memorized the order they do not have to wait to fully recognize it before responding. However, some people might avoid the work of memorization and instead wait to fully register the identity of the stimulus before responding. Newell's point was that we should not average across different strategies (memorizing or waiting in this case) as it produces meaningless numbers that do not accurately reflect the operation of the underlying cognitive system.

Using individuals as the unit of analysis is an approach that is seen more often in psychophysics research (see West et al., 2000), but has also been used in other, more cognitive tasks (Gray & Boehm-Davis, 2000; Shiffrin & Cousineau, 2004). The key feature of these studies are (1) train the participants to high level of expertise to avoid including the process of exploration and learning in the data, (2) collect large amounts of data from each participant, and (3) analyze the data of each participant separately. The current study builds on a previous study by West et al.(2018). In the previous study two participants were extensively trained to be high performing experts in a speeded memory game. The results showed that two participants matched each other within milliseconds of accuracy under specific conditions, but not under others. There was also a very close match to an ACT-R model (Anderson & Lebiere, 1998) built according to SGOMS (West, 2013; West & MacDougall, 2014; West & Nagy, 2007; West, & Pronovost, 2009; West and Somers, 2011) on these conditions.

Gray and Boehm-Davis (2000) suggest that certain features of a task may constrain the micro strategies that can be adopted, but do not present a computational theory that explains it. Both Gray and Boehm-Davis (2000), and Shiffrin and Cousineau (2004) built their models to fit

the data of individuals in their studies. In contrast, I used SGOMS theory (West, 2013; West & MacDougall, 2014; West & Nagy, 2007; West, & Pronovost, 2009; West and Somers, 2011) to constrain the model to predict what the optimal strategy could be. So, in this study the model was created before the data was gathered. Therefore, the current study was based on a predicted model rather than fitting the model to data.

To further the research started by West et al. (2018), I decided to follow a similar experimental design and developed a version of the game without the conditions where the two subjects varied, to see if this part of the game causes participants to use the same micro strategies. In addition, although the essential elements of the game were retained, the stimulus response patterns and the input output interface were altered to see if this makes a difference. Finally, I used an improved ACT-R/SGOMS model with a better, more conservative way of fitting it to the data.

For my study I made an SGOMS model built in ACT-R. SGOMS is a member of the GOMS (Card et al., 1983) family and is a system for modeling human expertise. SGOMS constrains ACT-R to use an optimal strategy for hierarchical routine expertise (West, 2013; West & MacDougall, 2014; West & Nagy, 2007; West, & Pronovost, 2009; West and Somers, 2011). I implemented the game from the study into Python 3 so the model could play it. The model had to play the game just as the human players did.

## Chapter 2: The Game

The new game, called 4 Button Expert was built using MIT app inventor 2 (https://appinventor.mit.edu/), and is run as an app on mobile devices. 4 Button Expert levels follow the SGOMS structure and explicitly uses a hierarchy structure composed of operators

(individual button presses), Methods (fixed series of button presses), Unit Tasks (patterns of Methods), and Planning Units (ordered lists of Unit Tasks). 4 Button Expert can be best explained by comparing gameplay to that of a First-Person Shooter video game, such as Dead Space. A game in which a player must navigate through different areas of a wrecked spaceship. Throughout the levels players encounter aliens which they must fight using different combinations of moves and weapons.

## 2.1 The Methods Level

The Methods level is equivalent to knowing which buttons correspond to which actions of the character. Buttons on a game controller such as X, O, and R2 correspond to actions such as Jump, duck, and shoot. In the Four Button Expert the Methods take the form of a two-letter prompt and a corresponding four-digit response. Players must enter the four numbers when prompted by the appearance of the two letters at the top of the screen (see Table 1). Expert players should have the four-number sequence proceduralized and be able to enter it immediately when they know which Method is required.

| Methods | |
|---|---|
| AK | 1234 |
| SU | 4123 |
| ZB | 2143 |
| RP | 4321 |
| FJ | 3214 |
| HW | 2341 |
| YP | 3412 |
| WM | 1432 |

Table 1. The eight, two letter Methods and their corresponding four-digit responses.

## 2.2 The Unit Task Level

The Unit Task level is equivalent to knowing the different action sequences you must use to fight an enemy. For example if there is an alien type that you can shoot twice before he

executes an attack on you, upon which you duck, then resuming shooting, your button sequence would be R2, R2, O, R2. Compare this to another alien type where you must shoot, jump to avoid his attack type, shoot and then duck, in this case your button sequence would be R2, X, R2, O. Players must use a different sequence of the same actions in the different conditions. In our game the two-letter prompts are organized in specific and consistent sequences (see Table 2). These are the Unit Tasks. Expert players recognize that specific Methods signify the beginning of a Unit Task and know which related sequence of numbers are needed to complete the Unit Task. In two of the Unit Tasks (RP and HW) there are Splits that occur, where one of two or one of three Methods could be required. The Splits always occur at the same place in each Unit Task. This is the equivalent of some of the aliens having two or three different possible attack types that they employ at random at a certain point in the sequence. In these cases, the player cannot ballistically execute the responses. Instead, they must first identify the attack, then respond appropriately.

| Unit Tasks |
| --- |
| Unit Task 1 (AK UT) |
| AK-WM-SU-ZB-FJ |
| Unit Task 2 (RP UT) |
| RP-SU< ZB-WM / YP-FJ |
| Unit Task 3 (HW UT) |
| HW-YP < FJ / ZB / SU |

Table 2. The order of the Methods within the Unit Tasks.

### 2.3 The Planning Unit Level

The Planning Unit level can be compared to a full level of our hypothetical video game. During a level, different aliens would be activated at different points throughout the level. An

expert at this game would know that on level 1, Alien type 1 appears, followed by Alien type 2 and followed by Alien Type 3. Whereas on Level 2 the Alien order is Type 3, 1, then 2. Therefore, expert Players would know exactly which order the Aliens appear in and which Unit Task to use. My game follows this structure as well, where each Planning Unit holds the same Unit Tasks in different orders (see Table 3). Expert players are able to recognize which Planning Unit they are in by looking at the first Method code of the Planning Unit.

| The Planning Units |
|---|
| Planning Unit 1 |
| (AK UT)-(HW UT)-(RP UT) |
| Planning Unit 2 |
| (RP UT)-(HW UT)-(AK UT) |
| Planning Unit 3 |
| (HW UT)-(RP UT)-(AK UT) |

Table 3. The order of the Unit Tasks within the three Planning Units.

## 2.4 The Conditions

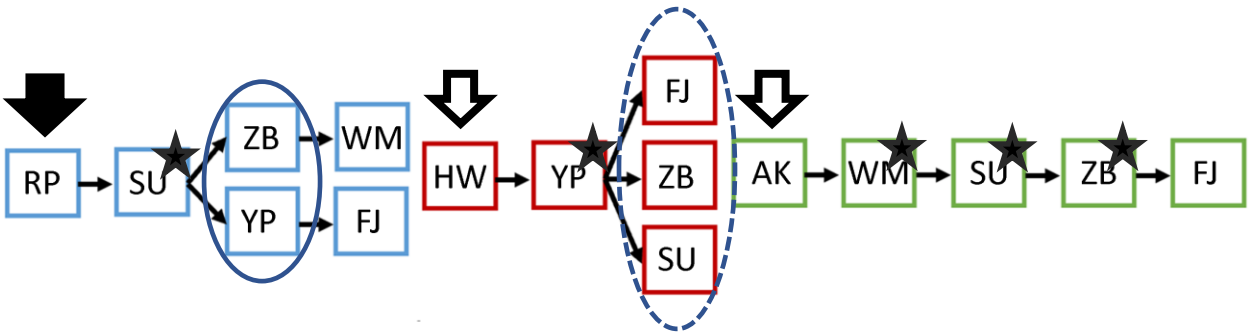


Figure 1. An example of a full Planning Unit in the game. The Known Methods condition is identified by stars, the Two Split by a solid circle, the Three Split by a dashed circle, the Unit Task First by empty arrows, and the Planning Unit First by a solid arrow.

The image displayed in Figure 1 is an example of one out of three full Planning Units in the game. Each square represents a single method within the game, the separate-colored

segments represent the separate Unit Tasks, and the full segment represents the entire Planning Unit. In the other two Planning Units in the game, the exact Unit Tasks are also used but in different orders. The stars, arrows, and circles indicate different conditions of interest and will be explained in detail below.

The first condition of the study is the 'Known Methods Condition', this includes Methods that are internal to the unit-tasks – so neither the beginning or the end – and never change. They are always in the same spot and therefore do not need to look at the code to know the Method (see the stared methods).

The next condition is the 'Two Split Condition', this condition occurs in the in RP Unit Task. At this point the Method is going to be one of two and cannot be predicted. The player must look at the code to see which of the two Methods is used before responding (see solid circled section).

Similar to the 'Two Split Condition' we also have a 'Three Split Condition', this condition occurs in the HW Unit Task. In this case the Method is going to be one of three and cannot be predicted. Like with the Two Split the player must look at the code to see which of the three Methods is used before responding (see dashed circled section).

The 'Unit Task First Condition' refers to the first Method of a Unit Task, specifically the Unit Tasks which are internal to the Planning Unit. In this condition expert players should be able to predict which Unit Tasks are within the Planning Unit and therefore do not need to look at the code before responding (see arrows with no fill).

Finally, the last condition is the 'Planning Unit First Condition', this is the First Method of the first Unit Task of a Planning Unit. This Method cannot be predicted and the player must look at the code before responding (see solid filled arrow).

Note. The West et al. (2018) study had different conditions that did not produce easily interpretable results, so I redesigned the game to improve the experimental design. I also added a visual and motor module to fully account for all cognitive processes.

## 2.5 App/ Game Details

A demo of the app is available on the google play store under the title ‘4 Button Expert’, The demo only differs from the full game in that the demo app does not collect any user data. When someone is going to play the game, the first page that they see in the app is the title page (Figure 2). This page is where they either select Methods Training, Unit Task Training, or Full Game play styles. Other menu options include the game instructions, setting options, and a High Score feature. The game instructions page is where the response to each Method is provided, as well as a visual representation of the Unit tasks and Planning Units are laid out. The app was play tested by volunteers and went through multiple iterations before the full experiment was run.


Figure 2. Screen Capture of Main Menu Screen on ‘4 Button Expert ’

At the beginning of training participants are instructed to start with the Methods Training level. Once selecting this option, the player is brought to the Methods Training Screen. There the player can select on or offline mode. In offline mode data would not be collected – this mode was used primarily during play testing. The player also gets the option to control which Methods are in circulation at the time. This feature was implemented when play testers suggested it would be easier to learn only a few Methods at a time. Before any participants are given permission to move to the next level, they must report they are able to play with all 8 Methods in circulation at once. I also checked their Method data to ensure that all 8 Methods can be played with the same fastest time. Once the start button is pressed by the participant, the 'press start' message on the yellow bar (see Figure 3) is replaced by which Method is the current stimuli – this is true for all levels of the game. Also true for all levels of the game is an audio feature: When the participant gets the Method correct there is a pleasant tone as well as a green visual above the yellow bar that says 'correct', and when a mistake is made there is an unpleasant tone and a red visual saying 'incorrect'.

Once players have moved past the Methods training, they go to the Unit Task training. The Unit Task training has most of the same features as the Methods training. The player can control which Unit Tasks are in circulation, thus giving them the option to learn the Unit Tasks one at a time. The yellow only indicates the current Method, so it is up to the participant to learn the order that the Methods come in for each Unit Task or reference the instructions page of the app. Participants are deemed 'finished 'this level of training when they self-report that they can typically play all 3 Unit tasks without making mistakes.

The full game page (Figure 3) is very similar to the previous two pages discussed. The main difference being that now the player has no control over Methods or Unit Tasks in rotation.

At this level it is up to the participant to track which Planning Unit they are in, the order of the Unit Tasks within that, and the order of the Methods within each Unit Task. The only feedback they are given is the same tone and visual indicator that informs them if they entered the correct Method response. While the game does not technically have an end to it – it can be played forever – there is a point where a message pops up on the top of the screen saying, 'Game Complete'. I asked participants to play until they saw that message for each session that they played in, which would take roughly 5 minutes of game play. Participants were asked to continue to play at this level of the game until they self reported they could play through a full run of the game with relatively few mistakes – and the researcher confirmed there was enough data by volume.

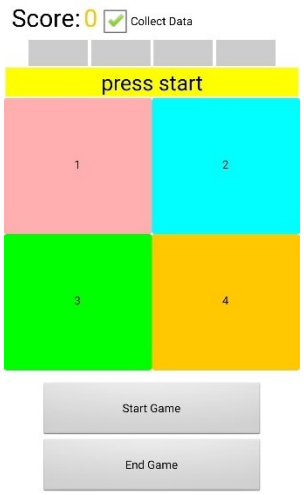


Figure 3. Screen Capture of the Full Game Screen.

## Chapter 3: Methodology

For my project I used 3 participants. They were all females aged 19 – 24. I looked for people who were videogame players. I felt that videogame players would be able to understand the structure of the game I designed faster than a non-videogame player. My participants

downloaded the app onto a mobile device. At the time, the app was only available on android phones so to be eligible to participate, the participant must have an android phone.

The game app was downloaded onto the participant's phone. The participant learned the game, starting at the Methods level. They moved up to the Unit Task level after they could confidently play each Method and their timing was consistent across all 8 Methods. Once they could play all three Unit Tasks they were moved up to the final level, the full game or Planning Unit level. Reaction time was based on how fast it takes for the participant to enter the corresponding four-digit code from when the two-letter code first appears on the screen. There is no delay between entering the four-digit code and the feedback of being correct or not, and no delay between receiving the feedback and the next trial starting.

### 3.1 Data Cleaning

The data was cleaned by first removing any trials where the players made an error in response. My research focus was on conditions where the participants were playing their best, because I was interested in proceduralized action – that is, automatic, correct responses. Secondly, I started sorting the data into the specific conditions I was looking at; the known Methods, the Two Split, the Three Split, the Unit Task first Method, and the Planning Unit first Method. Once the data was separated into the conditions, it was sorted from smallest to largest time (in milliseconds). Outliers were cut off by detecting a knee in the data. The Kneed script (Arvai, 2019) was applied. The knee of a curve is found when data is sorted from highest to lowest score (or vise verse) and visibly bends, the point of maximum curvature (Satopaa et al., 2011). I used this as my 'cut off point' because my interest in the data was when it was most stable – when the player was playing their best. A participant who gets distracted while playing

or interrupted would result in an abnormally long time for that method. By sorting and cutting of the knee, I was removing the trials where the players time was unreasonably long.

## Chapter 4: The Model

The model I built assumes that both ACT-R and SGOMS accurately describe human cognition. The model represents the optimal way to play the game with these assumptions. The model predicts that different conditions will take different amounts of time. From fastest to slowest these are the following. Known Methods are fastest, they require a motor action and one production (50ms) for firing the motor action. The next fastest is the Unit Tasks First because it requires a motor action and 4 productions (200ms), three productions are minimally required for the overhead of tracking which Unit Task is beginning and one production is needed for the motor actions. The Two Split and Three split come next in terms of speed, they require a visual action, a motor action and 2 productions (100ms), one for firing the visual action and one for the motor action. Finally, the slowest is the Planning Unit First, this requires a visual action, a motor action, and minimally 8 productions to keep track of which planning unit is beginning, which unit task is beginning, and which method to use.

It is possible for people to play in sub optimal ways that could also be modeled using SGOMS and ACT-R. The SGOMS ACT-R model can also be systematically altered to account for non-optimal strategies that fit with the SGOMS ACT-R assumptions. I anticipated that everyone would play according to the model in the optimal way. However, if this were not the case, I anticipated that a non-optimal SGOMS ACT-R micro strategy could fit the data. In other words, there were two levels of prediction, the first is optimal SGOMS ACT-R play and the

second is non optimal SGOMS ACT-R play. It is also possible that people could play in ways that could not be modeled by SGOMS and ACT-R at all.

The SGOMS ACT-R game model (see Appendix A) that I built is similar to the model used by West et al., (2018). The reference model used separate estimates for the action of responding and the action of looking then responding. In the model used for this study I estimated the action of responding and the action of looking and combined them in the model to create an estimate of looking then responding. To do this, I first fit the model to the Known Method condition. In this condition the model predicts that the player will respond without looking. The model predicts that one production (50 milliseconds) is needed to trigger the response. I estimated the average time to make the motor movements for the response by taking the average human response time in this condition and subtracting 50 milliseconds. Following this, I fit the model to the Two Split condition. For this condition, the model predicts two productions (50 milliseconds each) to trigger first a look then a motor action. To estimate the look time I took the average human time in this condition and subtracted 100 milliseconds for the productions and the estimate for the motor movements. Thus, the model assumes that motor and perception times are constant and additive.

For the conditions with multiple possible responses (2 Split and 3 Split), I predicted that having to choose between two options or three options would make no difference (see table 2). This is an ACT-R prediction that is in contradiction to Hick's law, which states that reaction times increase as the number of stimulus–response alternatives increase (Hick, 1952). However, Schneider and Anderson (2011) have argued that Hick's law applies only when the choice is stored in declarative memory, and the model relies only on the ACT-R Procedural Memory. However, this assumption has never been directly tested.

The model assumed that players will minimize their response times by using the sequential dependencies inherent in the game to speed up responses. Specifically, the model assumed players will respond without taking the time to perceive the signal in cases where they can predict the signal. In order to do this the model assumed that players keep track of where they are in the game hierarchy. This incurs a time cost. It was assumed that players would incur this cost but would do it in the most efficient way, given the constraints of the ACT-R architecture. That is, the minimum number of production rules (where production rules fire one at a time and take 50 milliseconds each) would be used to keep track of the game. The model kept track of the game by temporarily storing the current Method, Unit Task, and Planning Unit in the ACT-R buffer system (which represents working memory). Compared to the West et al., (2018) model, I found a way to reduce the number of productions needed for the Planning Unit condition, so that the prediction for this condition was faster.

### 4.1 Code Examples for each Condition

In this section I will be showing parts of the code that specifically tie to the different conditions within the study. It is important to note that none of this code functions completely in isolation and requires communication with other parts of the model (see Appendix A) to run.

The following section of code shows an example from the model where it responds to the one of the Methods in the Known Methods condition. Specifically in this section of code the model is responding to the WM Method within the AK Unit Task. In the model this condition only fires a motor action and 1 production.

```
def AK_WM(b_unit_task='unit_task:AK state:running',
          vision_finst='state:finished',
          focus='AK'):
```

```
    focus.set('WM')

    target='responce'

    content='AK-WM-1432'

    motor.enter_response(target, content)
```

The following section of code shows an example from the model where it responds to the Two Split condition within the RP Unit Task. In the model this condition fires a visual action, motor action, and 2 productions.

```
    ### Unkown 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)

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

        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='YP'):
    focus.set('done')
    target='responce'
    content='RP-YP-3412'
    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('done')
    target='responce'
    content='RP-ZB-2143'
    motor.enter_response(target, content)
```

The following section of code shows an example from the model where it responds to the Three Split condition within the HW Unit Task. In the model this condition fires a visual action, motor action, and 2 productions.

```
### Unkown 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.set('done')

target='responce'

content='HW-ZB-2143'

motor.enter_response(target, content)
```

The following section of code shows an example from the model that begins a Unit Task. In the model this condition fires a motor action, and 2 production. This condition has more productions that fire to account for the overhead, these productions are found elsewhere within the model (see Appendix A).

```
# 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'):

      b_unit_task.set('unit_task:AK state:running')

      focus.set('AK')

      target='responce'

      content='AK-AK-1234'

      motor.enter_response(target, content)
```

The following section of code shows an example from the model beginning a Planning Unit, specifically the AK Planning Unit. In the model this condition fires a visual action, a motor action, and 2 productions. This condition has more productions that fire to account for the overhead, these productions are found elsewhere within the model (see Appendix A).

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

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

   def 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')
```

```
print ('run_AK_PU')

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

## Chapter 5: Results

Figure 4 shows the model fitted to the conditions in the reference game that match the conditions in this game and the resulting predictions. The results show that the model accurately predicts the Planning Unit condition and the Unit Task condition. The faster time for the Planning Unit prediction is a better fit than found in West et al., (2018), showing that the improvements to the model were warranted.

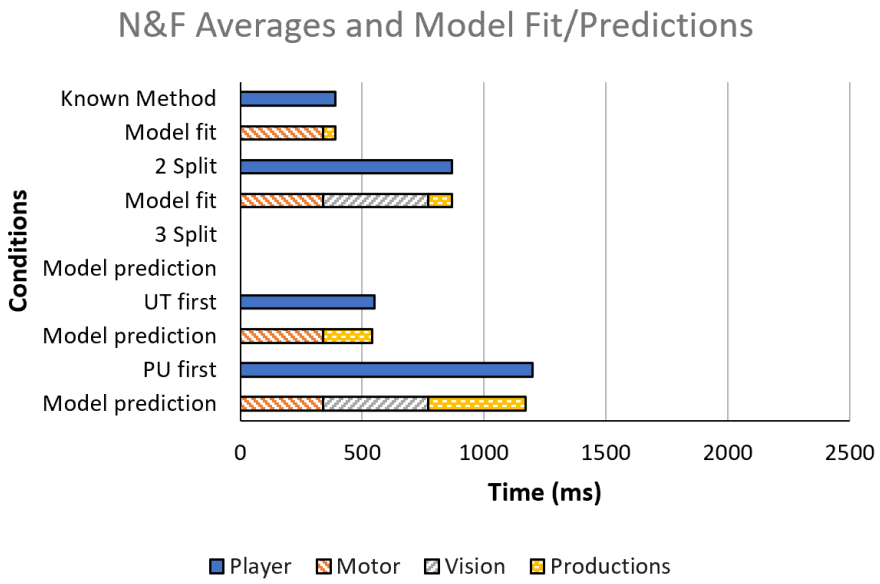


Figure 4. This graph shows the combined averages of Nathan and Fraydon from the West et al. (2018) study compared to the model predictions for all the conditions that match the conditions of the current study.

### 5.1 Player 1

Turning to the new game created for this study, Figure 5 (1a) shows the results for player 1. The model predicted the Three Split condition but player 1 was considerably slower than the model for the Unit Task condition. In fact, the player 1's Unit Task time was approximately the same as the Two Split and Three Split conditions, which is what the model would predict if player 1 was visually retrieving the Unit Task code, rather than remembering which code comes next. Player 1 also took significantly longer for the Planning Unit task than what was predicted by the model.

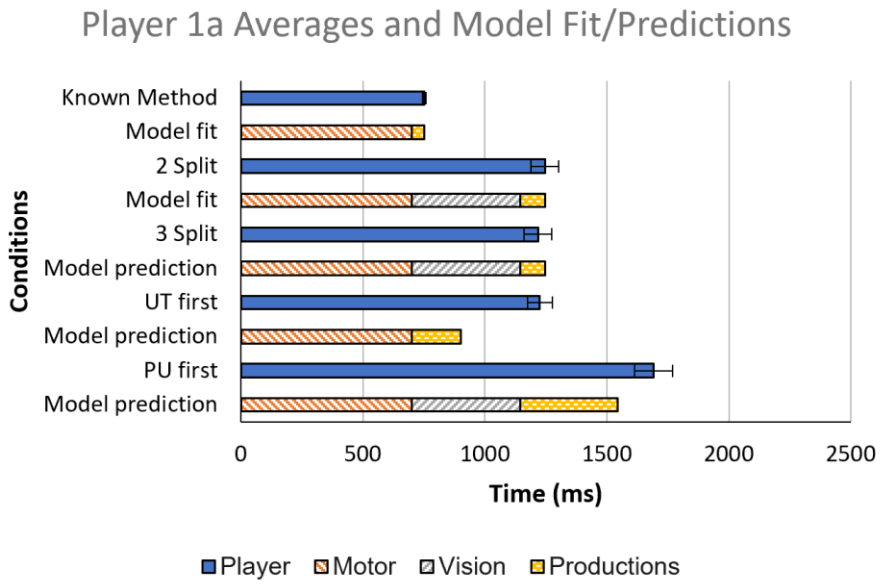


Figure 5. The average times for Player 1a in each condition compared to Model fit and predictions. Error bars are based on a 0.05 confidence interval.

To test the interpretation that player 1 was visually acquiring the code in the Unit Task condition, rather than remembering it, another set of games was played by player 1, with the instructions not to look in the Unit Task condition and to try to go faster on the Planning Unit condition. Figure 6 (1b) displays the results. This time player 1 took longer on the Three Split than the Two Split condition. The Unit Task condition was faster, but still not as fast as the model prediction. As well, the Planning Unit condition was now faster than the model.

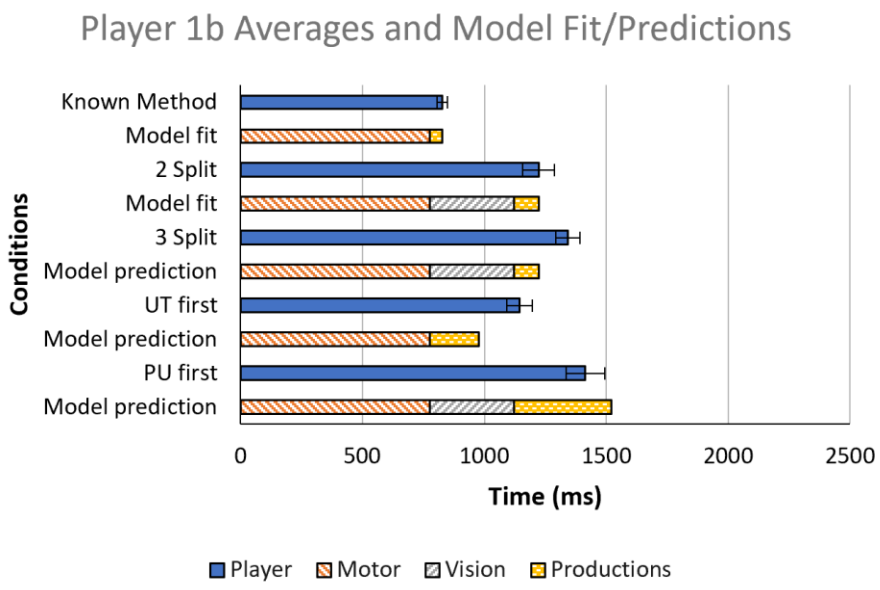


Figure 6. The average times for Player 1b in each condition compared to Model fit and predictions. Error bars are based on a 0.05 confidence interval.

One reason why the Two Split condition could be faster than the Three Split condition was guessing. Although the players are not supposed to guess, guessing would be a more effective strategy under the Two Split condition (50% chance of guessing right) than in the Three Split condition (33.333...% chance of guessing right). Given that guessing is faster and that I discarded incorrect guesses, this meant that the Two Split condition would be faster if guessing occurred on some trials. To examine this we looked at the distribution of reaction times for the player 1a and 1b results.

Figure 7 displays the player 1(a and b) reaction time distributions by condition. This was very revealing. My suspicions about the effect of guessing were confirmed. The Two Split condition for games a and b produced approximately the same reaction times except at the extreme end, where the two choice condition produced reaction times that where faster than possible if a visual check occurred. Another issue was that the cut-offs for high reaction times were uneven across conditions.

Overall, disregarding the extreme ends of the distribution, games a and b appeared to be the same, except for the Planning Unit condition. The instructions in game b to use memory rather than vision for the Unit Task condition appeared to have had no effect, probably because player 1 was already using memory in both games. In Figure 7 the Unit Task distributions are reliably 200 to 300 milliseconds faster than the Two Split and Three Split conditions (which are approximately the same). This is approximately what the model predicts (see Figure 6). However, the instruction to speed up the Planning Unit had a large effect, producing a high range of variability in the Planning Unit condition and, at the low end, reaction times the same RT as the Split conditions, indicating that no extra processing was occurring beyond look and respond. This indicates that a mixture of strategies were used in this condition. It is also possible that the other conditions had mixed strategies. Specifically, the results indicate that a guessing strategy was mixed into the choice conditions.

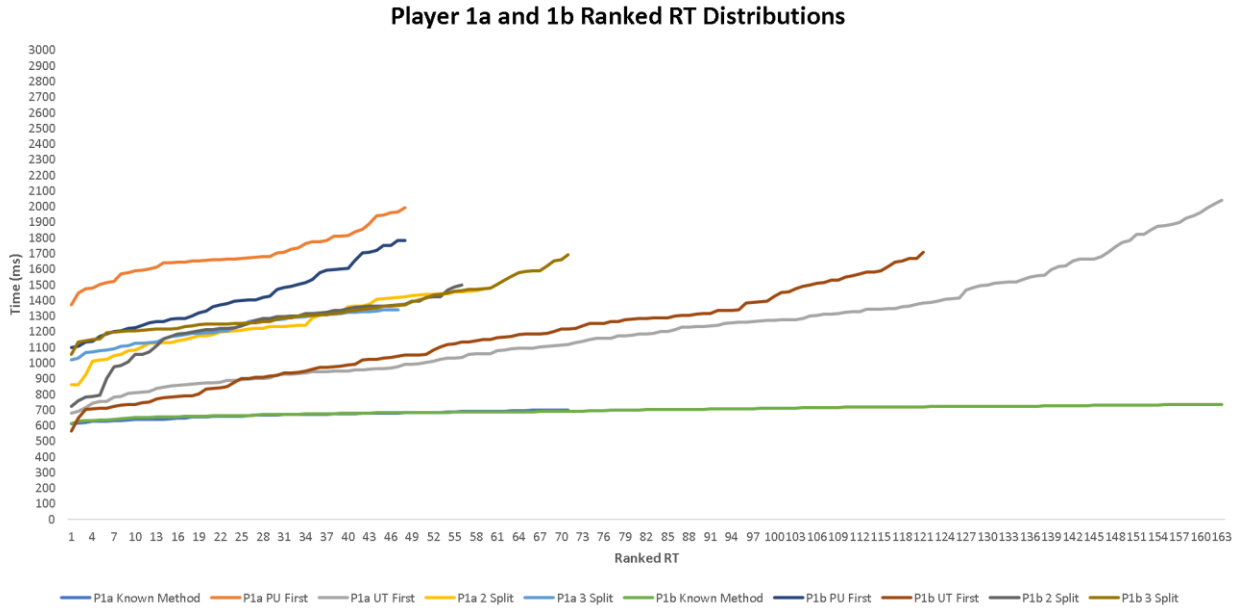


Figure 7. This Graph shows the Ranked RT distribution from player 1a and b for each condition from fastest to slowest. The x-axis is the Ranked RT from fastest to slowest, and the y-axis is the time in milliseconds.

## 5.2 Player 2

Player 2's results are displayed in Figures 8 and 9. Like Player 1a, the model fit suggested that player 2 was visually acquiring the code in the Unit Task condition, rather than relying on procedural memory (Figure 8). In this case the RT distribution supported this interpretation. Figure 9 shows that the RT distribution for the Unit Task condition is essentially the same as the RT distribution for the Two Split and Three Split conditions. Player 2 also displayed less noise in the RT distributions than Player 1. Player 2's RT distributions show relatively flat areas, indicating a stable, automatized process. The mean scores reported in Figure 8 all correspond to center areas of the flat parts of the distributions in Figure 9, indicating that the mean was a good measure for Player 2. If it is assumed that player 2 used this alternative strategy, then the fit of the model would be quite good.

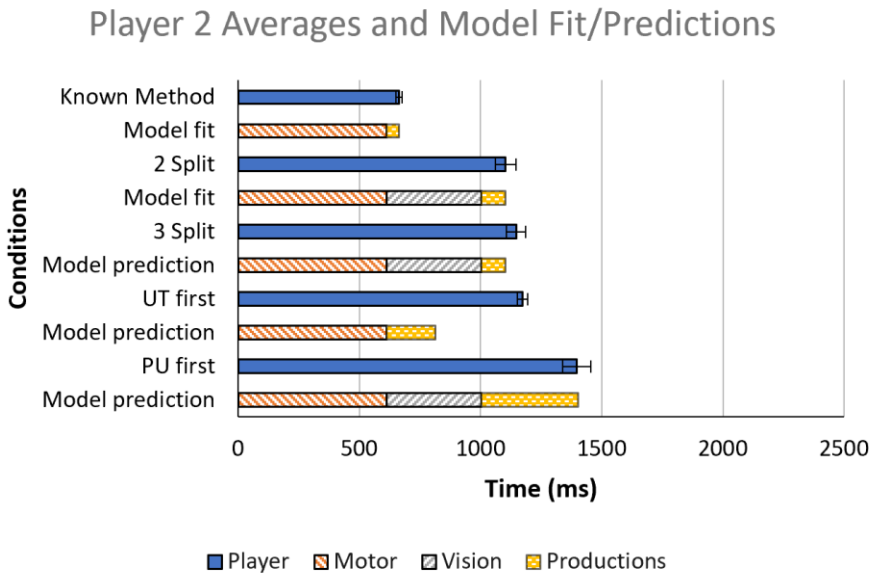


Figure 8. The average times for Player 2 in each condition compared to Model fit and predictions. Error bars are based on a 0.05 confidence interval.

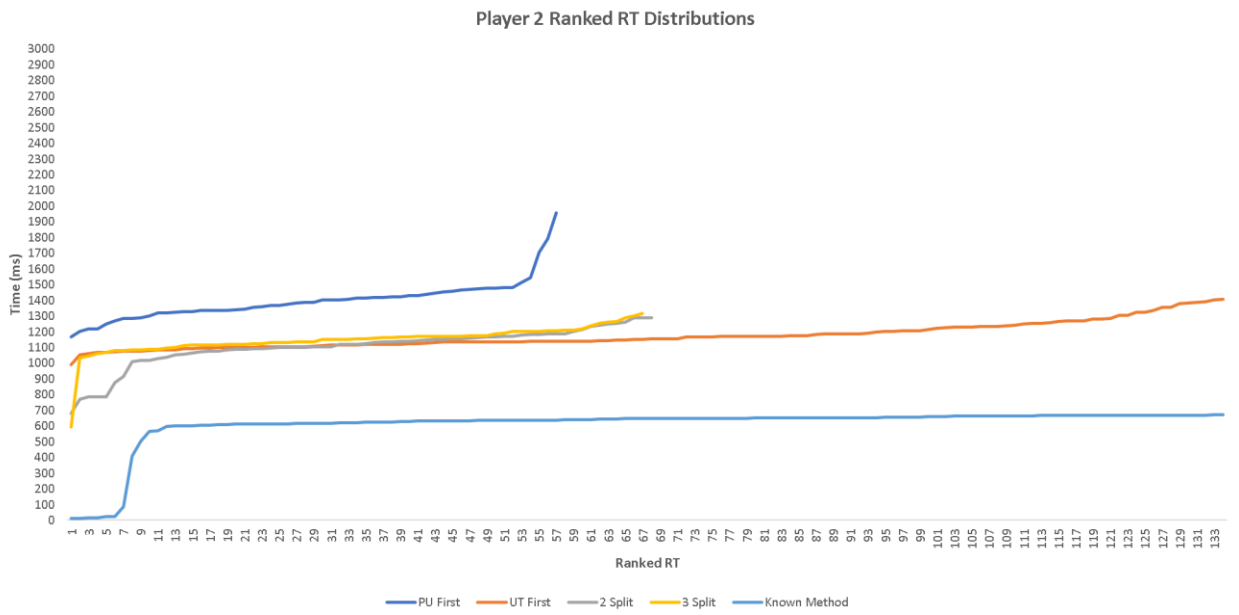


Figure 9. This Graph shows the Ranked RT distribution from player 2 for each condition from fastest to slowest. The x-axis is the Ranked RT from fastest to slowest, and the y-axis is the time in milliseconds.

### 5.3 Player 3

Player 3's results are displayed in Figures 10 and 11. Figure 10 shows that while Player 3 was faster in the Unit Task condition than in the Split conditions, Player 3 was not as fast as the model predicted. The RT distributions (Figure 11) shows that the Unit Task condition was reliably faster than the Two Split or Three Split conditions, but it also shows that the Two Split condition was reliably faster than the Three Split condition. As well, the Planning Unit condition was considerably slower than the model predicted. Regarding the difference between the Two Split and Three Split condition, I argue that it should not be taken as evidence for Hick's law as it did not occur in the other players (a law should apply to all players). Instead, Player 3's results suggest a systematic but less than optimal micro strategy, where they keep track of whether it was a 2 split or a 3 split. at a cost of one production, even though it produced no value.

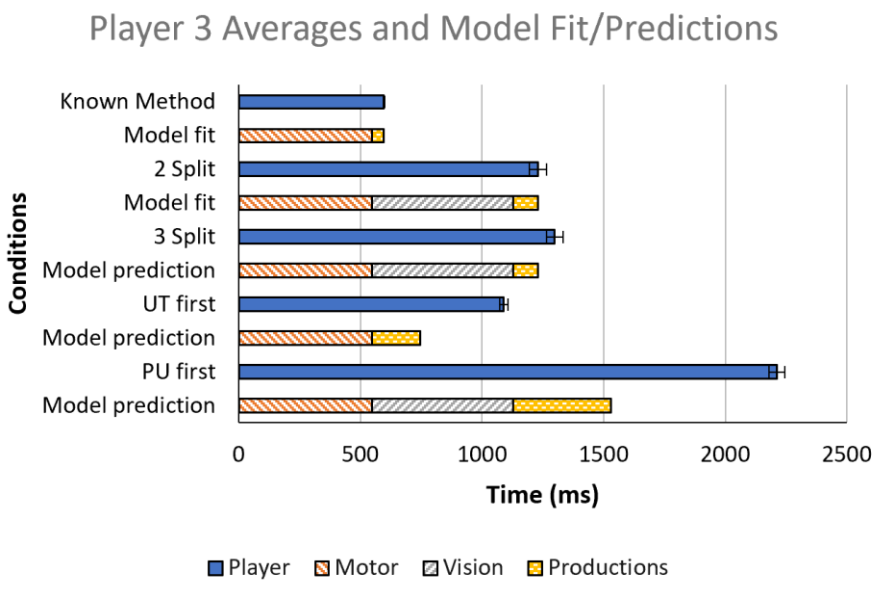


Figure 10. The average times for Player 3 in each condition compared to Model fit and predictions. Error bars are based on a 0.05 confidence interval.

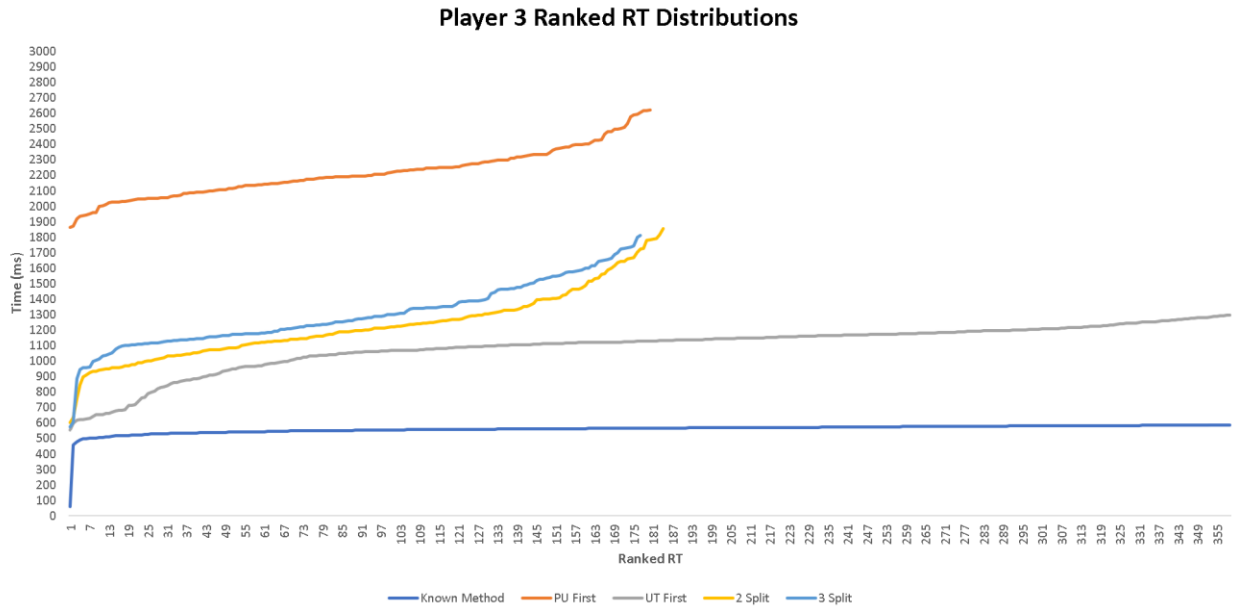


Figure 11. This Graph shows the Ranked RT distribution from player 3 for each condition from fastest to slowest. The x-axis is the Ranked RT from fastest to slowest, and the y-axis is the time in milliseconds.

## Chapter 6: Discussion

### 6.1 Known Method Condition

The Known Method Condition was the most straight forward – this condition was based on the methods internal to the Unit Tasks that never changed, players who are experts would be able to respond to these methods automatically, without looking. The model accordingly predicts one production (50ms) and a motor action. At this level of the game, all the participants appear to have mastered this. This condition was used for a model fit condition, the model predicts that there is only one production at this level and the rest of the timing is motor output. If we look at the distribution of the Known Methods for all three participants (figure 7, 9, & 11) we can see that all three of them are consistently flat. This suggests that I am correct in my prediction and that it is a ballistic and proceduralized condition.

### 6.2 Two and Three Split condition

The Two Split and the Three Split conditions occur in two of the Unit Tasks (see Table 2 and Figure 1). In these conditions the player must wait to see if either one of two or one of three Methods respectively. In this condition the model predicts additively; two productions (100ms), a motor action, and a look. Schneider & Anderson (2011) argue that Hick's Law will not have an effect on proceduralized responses. My results were consistent with this for player 1 and player 2. However, player 3 showed a consistently longer response time in a Three Split condition compared to the Two Split. A Law should apply universally, and in our overall results it does not. Thus Hick's "law" was not supported for our data. An alternative explanation for Player 3 is that she was firing an extra production. Further research would be needed to verify, but potentially when player 3 was playing the game, she developed a strategy where every time she sees that she is at a Three Split, she fires an extra production to keep track of where she is, this is a specific example of a non optimal SGOMS ACT-R strategy.

### 6.3 Unit Task First Method condition

The Unit Task First Method condition is the first Method of a Unit Task that does not begin the Planning Unit. The model predicts that because the order of the Unit Task First Method is determined by the current Planning Unit, an optimal strategy would not include a look. According to the model, the Unit Task First Method minimally requires 4 productions (200ms) accompanied by a motor action. For this condition, no participants matched the model's predictions. Based on the distribution analysis it seems that player 2 was consistently doing a look, this is a specific example of a non optimal SGOMS ACT-R strategy. This is shown in Figure 10, where the Unit Task First line maps almost exactly onto the Two and Three Split conditions, where a look is necessary. In contrast, Player 1 and Player 3 were reliably faster for the Unit Task First condition (Figures 6 and 10) indicating that they were able to respond from memory, but not as fast as the model.

### 6.4 Planning Unit First Method condition

The Planning Unit First Method condition is the first method of a Planning Unit, it is an unknown condition where the player must look before responding. The ACT-R model represents the fastest way to play the game and keep track of where you are. Overall, the results indicate that players in this study deviated from the optimal strategy, whereas the players from West et al., (2018) did not. However, in simplifying the game we also seem to have made it harder. Specifically, in West et al., (2018) the players were given a unique code to signal the beginning of a Planning Unit, whereas in this version the players had to keep track of where they were to know that a new Planning Unit had started. This suggests the possible influence of working memory interference. Specifically, the West et al., (2018) players had a unique code for the planning unit while the players in this study had to use the same code for the method, the unit

task, and the planning unit. Overall, it appears that this may have created interference and/or overloaded working memory.

Based on the results from Player 1a and b, where Player 1 was instructed to go faster on the Planning Unit, we can see that the Planning Unit level was cognitively penetrable, whereas the Methods and Unit Task level did not appear to be. This is consistent with SGOMS theory, where the Planning Unit is seen as something that can be adjusted, based on strategic thinking. However, the results of Player 1b also indicate that changing a Planning Unit takes practice and possibly experimentation. Without time for that, Player 1b's results indicate an unstable, highly variable process. In contrast, Player 3 produced stable results but was very slow in the Planning Unit condition, possibly reflecting the use of rehearsal to stabilize the identity of the Planning Unit in working memory, this is a specific example of a non optimal SGOMS ACT-R strategy. Player 2 appears to use a strategy where she does not use the Planning Unit information, this is demonstrated in the way that she consistently looks during the Unit Task First Method Condition. However, she does seem to track which Planning Unit she is in, as indicated by the fact that this condition is still slower than the Unit Task condition but does not use this information to avoid having to look for the Unit Task First Method.

### 6.5 Final General Conclusion.

Micro strategies are very real, not well understood, and occur during very simple and controlled tasks. Newell first acknowledged their significance in 1973. Since then, very little research has acknowledged their effect and even less research has been done to understand them. This research as well as that of Gray and Boehm-Davis (2000), and Shiffrin and Cousineau (2004) indisputably show just how much of an effect they can have. It is imperative that micro strategies need to be taken into consideration when preforming psychology and cognitive studies

involving averaging. Moving forward, more work needs to be done to understand different micro strategies and how they are learned.

## Appendix A

**Main.**

```
import sys
import ccm

from random import randrange, uniform
from Emily import *
#       ^ here you can point the code to the agent you want, the player will then call its own motor module



log = ccm.log()

class hyrule (ccm.Model):

### objects for task preformance
    response = ccm.Model(isa='response', state='state')
    display = ccm.Model(isa='diplay', state='RP')
    response_entered = ccm.Model(isa='response_entered', state='no')
    vision_finst = ccm.Model(isa='motor_finst', state='re_set')

######## run model #########
link = MyAgent()        # name the agent
env = hyrule()          # name the environment
env.agent = link        # put the agent in the environment
ccm.log_everything(env)  # print out what happens in the environment
env.run()               # run the environment
```

```
ccm.finished()          # stop the environment
```

**The Agent.**

```
import sys
import ccm
from EmilyMotorModule import *
from RTModule import *
from ccm.lib.actr import *
from random import randrange, uniform

######################
##### The Agent #####
######################

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()

    # visual = Buffer()

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

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

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

    # reation time module - used to record the reaction time of the agent
    RT = RTModule()

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

    # initial memory contents

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

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

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


    def 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')
        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')
        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')
######## new buffer
        b_context.modify(status='occupied')
        print ('run_HW_PU')
        print


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

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

## removed

########################## 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:?second state:start type:ordered')
        b_plan_unit_order.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:?third state:start type:ordered')
        b_plan_unit_order.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_plan_unit_order.modify(counter='four')
        b_unit_task.set('unit_task:?fourth state:start type:ordered')
        print ('fast - start fourth unit task')
```

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

    def last_unit_task_ordered_plan(b_plan_unit='planning_unit:?planning_unit',
                                    b_unit_task='unit_task:finished state:start type:ordered'):
        print ('finished planning unit =');
        print (planning_unit)
        b_unit_task.set('stop')
        b_context.modify(status='unoccupied')
        ################################## referee
        choices = ['AK','RP','HW']
        x=random.choice(choices)
        motor.referee_action('display', 'state', x)
        ################################## referee


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

#########################
##### RP Unit Task #####
#########################

#                           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)

### Unkown 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)
```

```
    ################################# referee
    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='YP'):
    focus.set('done')
    target='responce'
    content='RP-YP-3412'
    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('done')
    target='responce'
    content='RP-ZB-2143'
    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_WM(b_unit_task='unit_task:RP state:running',
              vision_finst='state:finished',
              focus='ZB'):
        focus.set('done')
        target='responce'
        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='responce'
    content='HW-YP-3412'
    motor.enter_response(target, content)

### Unkown 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.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')
```

**The Motor Module**

```
import sys
import ccm
from ccm.lib.actr import *
from random import randrange, uniform

#########################
##### MOTOR MODULE #####
#########################

class EmilyMotorModule(ccm.Model): # defines actions in the environment
```

```
##### This instantly causes changes in the environment
##### It is not a proper part of the agent
    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 4
        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'

##### This enters the code
    def enter_response(self, env_object, slot_value):
        self.parent.parent.vision_finst.state = 'busy'
        #yield 3
##        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 parallal)

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

**RTModule**

```
import ccm
import time
import os
from datetime import datetime



class RTModule(ccm.Model):
    last_time = time.time()
    last_sim_time = 0

    def __init__(self, file_name=None, *args):

        if file_name == None:
            self._file_name = os.path.join(os.curdir, "rtData.csv")
```

```
    else:
        self._file_name = file_name

    # Set up header for .csv file format
    if not os.path.exists(self._file_name):
        print("Writing new file")
        self.fh = ShallowFileHandle(self._file_name)

        csv_header = "agent,time_stamp,rt(seconds),simTime_stamp,rt(sim)"

        for x in args:
            entry = entry + "," + '"' + str(x) + '"'

        self.fh.write(csv_header + '\n')

    else:
        self.fh = ShallowFileHandle(self._file_name)

    self.last_time     = time.time()
    self.flush_time    = time.time() - self.last_time

def recordRT(self, *args):
    rt      = (time.time() - self.last_time  + self.flush_time)
    simTime = self.get_sim_time()
    rt_sim  = '%.3f'%(simTime - self.last_sim_time)

    # Create the row entry string with rt time and extra args
```

```
        entry = "agent," + "" + str(datetime.fromtimestamp(time.time())) + "" + "," + str(rt) + "," +
str(simTime) + "," + rt_sim

        for x in args:
            entry = entry + "," + "" + str(x) + ""

        self.last_time = time.time()
        self.fh.write(entry + "\n")
        self.flush_time = time.time() - self.last_time
        self.last_sim_time = self.get_sim_time()

    def get_sim_time(self):
        return self.log.time._log.time


class ShallowFileHandle():

    def __init__(self, file_name):
        self._file_name = os.path.join(os.curdir,file_name)
        self.file_handle = open(self._file_name, "a")

    def write(self, x):
        self.file_handle.write(x)
        self.file_handle.flush()

        return
```

```
def __deepcopy__(self, memo):
    return self
```

## References


Anderson, J. R. (2009). *How can the human mind occur in the physical universe?*. Oxford University Press.

Anderson, J. R., & Lebiere, C. (1998). The atomic components of thought. Mahwah, NJ: Erlbaum.

Arvai, K. (2019) Kneed (version 0.5.1) [Python 3]. https://pypi.org/project/kneed/

Card, S., Moran, T., & Newell, A. (1983). *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Gray, W. D., & Boehm-Davis, D. A. (2000). Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of experimental psychology: Applied*, *6*(4), 322.

Hick WE. (1952) On the rate of gain of information. Quarterly Journal of Experimental Psychology. 4:11–26.

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

Satopaa, V., Albrecht, J., Irwin, D., & Raghavan, B. (2011, June). Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops* (pp. 166-171). IEEE.

Schneider, D. W., & Anderson, J. R. (2011). A memory-based model of Hick's law. *Cognitive psychology*, *62*(3), 193–222.

Shiffrin, R., & Cousineau, D. (2004). Termination of a visual search with large display size effects. *Spatial vision*, *17*(4), 327-352.

Vera, A., Howes, A., McCurdy, M., & Lewis, R. L. (2004, April). A constraint satisfaction approach to predicting skilled interactive cognition. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 121-128).

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

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

West, R. L., & Nagy, G. (2007). Using GOMS for modeling routine tasks within complex sociotechnical systems: Connecting macrocognitive models to microcognition. *Journal of Cognitive Engineering and Decision Making*, *1*(2), 186-211.

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

West, R., Ward, L., Dudzik, K., Nagy, N., & Karimi, F. (2018, July). Micro and Macro Predictions: Using SGOMS to Predict Phone App Game Playing and Emergency Operations Centre Responses. In *International Conference on Engineering Psychology and Cognitive Ergonomics* (pp. 501-519). Springer, Cham.

West, R. L., Ward, L. M., & Khosla, R. (2000). Constrained scaling: The effect of learned psychophysical scales on idiosyncratic response bias. *Perception & Psychophysics*, *62*(1), 137-151.