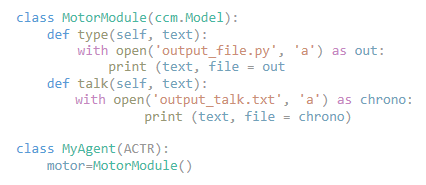
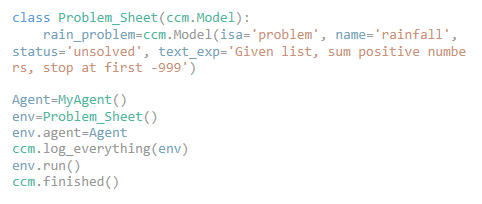
General model components, common to all models.

Motor Module



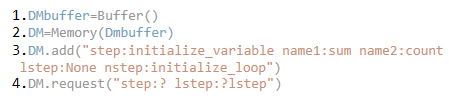
This is the motor module common to all models. The motor module can type and talk. When using the type function, the motor module adds code to its Python output file. When using the talk function, the motor module logs its retrieved goals and Python code to a transcript file.

Environment



These are the expressions that define the environment. Problem\_Sheet is the name of the environment in which problem solving occurs, and the rainfall problem text is defined within the Problem\_Sheet. MyAgent() initializes the problem solving productions (often referred to as the agent) within the environment.

Declarative Memory Module



The declarative memory module is accessed by the problem solving productions through the DMbuffer. Information can be either added to the declarative memory using the command DM.add(“”) or retrieved from the declarative memory using the command DM.request(“”).

Below are annotated code samples of the models built. Models were built in Python ACT-R (version CCMSuite3)

Can be found at: https://github.com/CarletonCognitiveModelingLab/CCMSuite3

Model 1: Proceduralized



Figure D.1 Problem Solving Productions of the Proceduralized Model

The proceduralized model has the algorithm of the problem embedded within the problem-solving productions of the model (productions are defined using the def statements). The first part of the model (Figure D.1 line 1) specifies its buffers and the modules they communicate with (if any). If the buffer does not specify any outside modules it is connected to, it is only used within the problem-solving portion of the model (like the focus buffer see Figure D.1 line 1a). The problem-solving process starts with the model being initialized into the problem-solving environment (see Figure D.1 line 2) and the focus buffer is set to start the model’s problem solving (Figure D.1 line 3). Then the next production (start\_problem) fires based on the conditions that a) the focus buffer contains the chunk “start” and b) the rainfall problem in the environment has its status set to unsolved (see Figure D.1 line 4). The model then logs the problem goal using the talk function of the motor module, accessing the motor module via the motor buffer (Figure D.1 line 5); the focus buffer is set to the first step in the algorithm, "initialize sum"(Figure D.1 line 6). Based on the content of the focus buffer, the next production fires, ini\_sum, which has as its condition the current contents of the focus buffer (Figure D.1 line 7) and the content of the focus buffer acts as the only condition for the production firing. The production accesses the motor module to implement the step of typing "sum = 0" that will initialize the sum in the Python program in addition to including it as a step in the log (Figure D.1 lines 8 and 9). Finally, the production sets the contents of focus buffer to contain the chunk that will act as the condition for the next step, in this case "initialize count", as can be seen in Figure D.1 line 10.

The setting of the focus buffer in the prior production triggers the firing of the next production in the algorithm ini\_count (Figure D.1 line 11). This production initializes the variable count within its Python script and adds that completed step to its log, following that is sets the focus buffer to the next step in the algorithm “iterate loop”. The ini\_count production uses the exact same process as the ini\_sum production (D.1 lines 12-14). The next production in the algorithm, looper, fires once the focus buffer contains “iterate loop” (Figure D.1 line 15) and similarly to the last two productions it fulfils its step (initializing the loop that will iterate through the list), records the step within its log file and changes the focus buffer to trigger the next production “stop loop at -999”. This process continues until the full solution is produced, at which point the problem solving stops and the model turns off.

Model 2: Algorithm Retrieval



Figure D.2 Sample of Goals used to Initialize the DM of the *Algorithm-Retrieval* Model

The *algorithm-retrieval* model stores the entire algorithm for the rainfall problem in its declarative memory. A fragment of the algorithm is shown in Figure D.2, with each line representing a goal, and the slots tracking: (a) the step, which resolves the goal of the declarative memory line; (b) the lstep, which is the previous step; (c) the nstep, which is the step that follows the current target step; and (c) the variables necessary for the target step (var1-var4). The model is shown in Figure D.3.

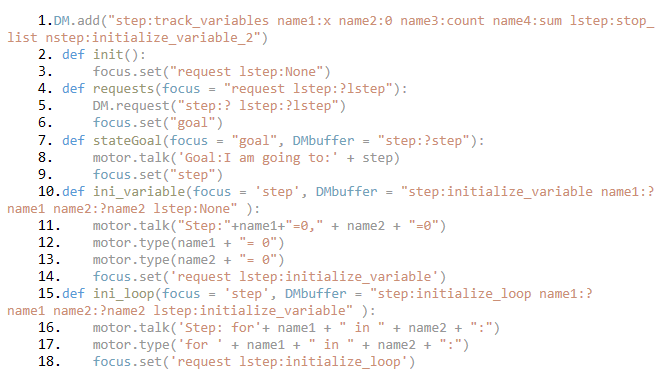


Figure D.3 Sample of the *Algorithm-Retrieval* Model’s Problem Solving Productions

Problem solving begins when the model fires the *requests* production (Figure D.3 line 4) to retrieve the first goal to be implemented from the declarative memory (Figure D.3 line 5), the first goal being initializing the variables. Additionally, the *requests* production sets the focus buffer of the model to "goal", which is the condition for the production state\_goal (Figure D.3 line 7). The state\_goal production is what allows the model to log its goals and add them to the chronotranscript. After this the focus buffer is set to “Step” so that the model will locate a production whose precondition matches both the contents of the focus buffer of the model (step) and the contents of the DMbuffer (the goal). An example of that is illustrated by the production ini\_variable (Figure D.3 line 10) that fires when the focus buffer is set to step and the content of the DMbuffer contain the goal to implement the step “initialize variables”. Then the appropriate production (in this case ini\_variable) implements the step that resolves the current goal. The production has the motor module write the step to the model’s output file, and adds it to its chronotranscript (Figure D.3 lines 11-13). Then ini\_variable sets the focus to “request lstep: “ and sets the chunk lstep to the name of the step just implemented (ini\_variable) (Figure D.3 line 14). This sets the focus buffer to contain the conditions for firing the requests production again, so that the requests production retrieves the next goal from declarative memory, in this case it would be to iterate through the list using the ini\_loop production (Figure D.3 lines 15-18). The cycle repeats until all of the goals in the algorithm are resolved, and all of the steps implemented.

Model 3: Algorithm Generation



Figure D.4 Initialization of *Algorithm-Generation* Model and Productions that Process the Problem Statement

The *algorithm-generation* model generates the goals of the algorithm in its declarative memory and then implements it. It begins by reading through the problem statement and translating it into the goals of the algorithm. To accomplish the translation step the model relies on a very rudimentary keyword search of the problem statement. Specifically, the model is initialized with chunks in its declarative memory representing keyword-goal associations. To start, the production (text\_parse) converts the problem statement in the model's environment into a list of words that the model can iterate through to check for keywords (Figure D.4 lines 5-7). This is not analogous to human cognitive performance (humans do not tend to make lists composed of the words we are reading), but as our focus is not on the NLP aspects, I have simplified the processing in this way. Currently, the productions that manage this process are very specific to the rainfall problem, and thus the model would have difficulty generating an algorithm for a different problem statement even if the python expressions needed were the same.

Once the text\_parse production converts the problem statement into a list of the words of the problem statement, the read\_list production checks each word in the list. If the current word in the list is not a keyword, or the declarative memory fails to retrieve it, the no\_id function fires to redirect the focus back to the read\_list production to check the next word in the list (Figure D.4 line 17). If the word a keyword the appropriate production fires based on the focus buffer contents and the retrieved keyword – goal chunk from memory.

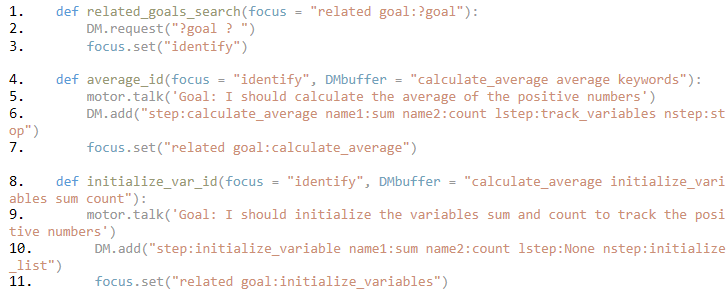


Figure D.5 Sample of the *Algorithm-Generation* Model’s Productions that Control Goal Identification and Step Implementation

When read\_list successfully retrieves a keyword association from declarative memory, a goal-specific production fires that adds the appropriate goal into the declarative memory. For example, when the keyword “average” is encountered, the keyword-goal association stored in the model’s declarative memory (see Figure D.4 line 2 for the model being initialized with that fact) is retrieved and stored in the DMBuffer by the read\_list production. Once the focus buffer contains the chunk “identify” (letting us know the model will need to identify the goal) and the DMBuffer contains the calculate\_average keyword association, the average\_id function fires (Figure D.5 line 4). The average\_id function then adds the goal to calculate the average to its declarative memory (Figure D.5 line 6) and adds the calculating the average goal to its log (Figure D.5 line 5). The goal added to the declarative memory is the same as calculate average goal used in the algorithm by model 2, and uses the same slot values.

Since not all goals are directly readable in the problem statement the model is also capable of searching its declarative memory for goal – goal association. When the focus buffer is set to search for related goals, the related\_goals\_search production makes a request to the declarative memory to check whether it has a goal – goal association (Figure D.5 line 2) and sets the focus buffer to “identify”. If there is no goal – goal association, the no-id production fires (just as in the case that there is no keyword association) and the model is redirected to the next word in the list. If there is a goal – goal association in memory (as there is for calculating average and initializing the variable) then the ID production for the non-keyword goal fires. In the example, the inititialize\_var\_ID production fires, as its conditions are “identify” in the focus buffer and the calculate\_average – initialize\_variables association in the DMBuffer (Figure D.5 line 8). It then adds the goal to its declarative memory (Figure D.5 line 10) and logs its goal (Figure D.5 line 9). When all the words in the list have been processed, the read\_list production sets the focus buffer to trigger the problem solving process (Figure D.4 line 16). The problem solving process, which translates goals to steps, then proceeds identically to the *algorithm-retrieval* model, as the algorithm the *algorithm-generation* model generates is identical to the one the *algorithm-retrieval* model is initialized with, and the step implementation productions are identical as well.

Model 4: Goal Expansion

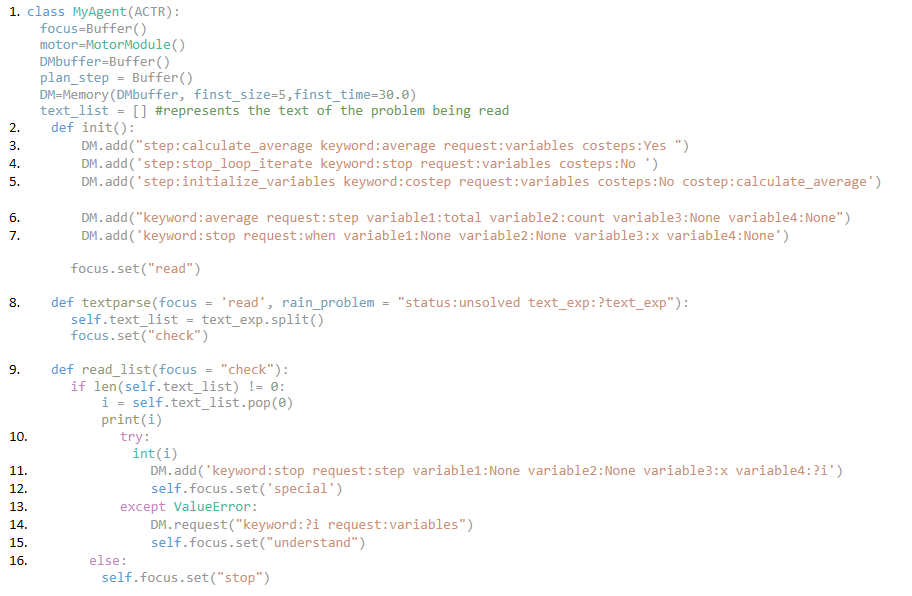


Figure D.6 The Initialization of the *Goal-Expansion* Model and its Productions Which Process the Problem Statement

The Goal Expansion model is initialized with keyword associations for goals (Figure D.6 lines 3-5) and associated variables (Figure D.6 lines 6 and 7), as was the case for model 3. Keyword associations for the goals provide some information through the following slots: (a) step - is the step which would resolve the goal chunk; (b) keyword - the keyword associated with the goal; (c) request - identifies the information the goal needs to implement the step; (d) costeps - states if the goal requires the resolution of any other goals; (e) costep - states the goal that requires the current goal to resolve. If the keyword is costep, that means it is not a focal goal identified from keyword association with the problem text, but one of the goals which is expanded around it; such as in the case of initialize\_variables which is required for the resolution of calculate\_average, so its keyword is “costep” and it has calculate\_average listed as its costep (Figure D.6 line 5).

Additionally, the model is initialized with keyword associations for variables which would be used by a given step. This is done to keep track of the variables and conditions of the problem and which ones are relevant to which goal-step combo; the variables and conditions are treated this way to allow for expansion of the model for non-rainfall problems. The following information is tracked by the following slots: (a) keyword – the associated keyword for the variables, which is used when later productions request the variables to implement the step; (b) request - identifies what the variables need to make sense (will always be step, as only a goal-step combo can request variables currently); and (c) variable(x) - tracks the necessary variables for implementing the step-goal associated with the keyword.

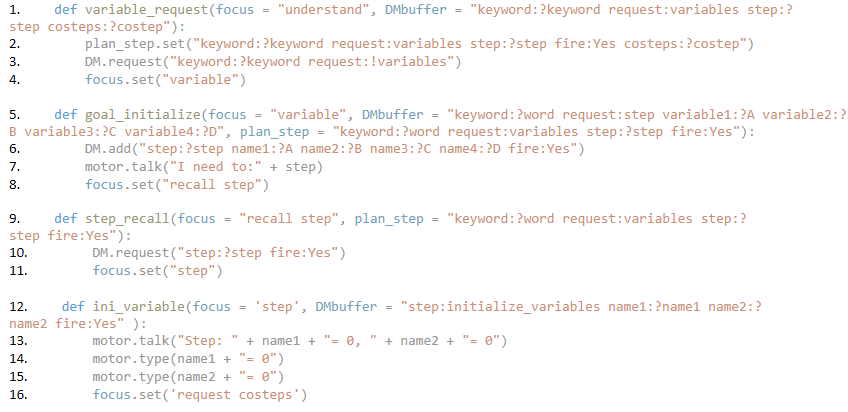


Figure D.7 *Goal-Expansion* Model’s Productions Which Generate Goals and a Step Implementing Production

Language processing occurs in the same way as the *algorithm-generation* model, using text\_parse and read\_list, with text\_parse generating a list of words of the problem statement and read\_list checking each word to identify whether or not it is a keyword. Similarly to model 3, the read\_list production adds the stop signal value (-999) to the declarative memory (Figure D.6 line 11), and if the list item is not an integer, the production sends a request to the declarative memory to check if it is a keyword.

If a word in the list made by text\_parse and read by read\_list matches a keyword - goal association then the variable\_request production fires, to request any of the variables or conditions needed by the goal to implement the step (Figure D.7 line 1). This production moves the goal retrieved by the keyword – goal association in the read\_list production, from the DMBuffer to the plan\_goal buffer, so that the model tracks current goal (Figure D.7 line 2). The plan\_goal buffer also tracks whether or not the current goal should fire using the slot “fire”; the slot should be set to yes, until the goal is resolved and the step implemented. The variable\_request production then queries the declarative memory for the variables associated with the step (using !variables in the request, because variables in the DM request step, while goals in the DM request variables) (Figure D.7 line 3) and focus is set to go to “variable” (Figure D.7 line 4) to trigger the production which combines the variables and the goal to trigger the step.

The focus buffer containing the chunk “variable”, the plan step buffer containing the current goal – keyword association, and the DMBuffer containing the variable names associated with the keyword in the model’s declarative memory are the conditions which trigger the firing of the goal\_initialize production (Figure D.7 line 5). This production connects the goal – keyword association and the keyword – variable associations into a single actionable goal which it then adds to the declarative memory (Figure 5.10 line 6). It then logs its goal (Figure D.7 line 7) and sets its focus buffer to “recall step” to trigger the production which will request the complete goal from the declarative memory (step\_recall). Step\_recall requests the goal in the declarative memory that is ready to fire (slot fire set to yes) (Figure D.7 line 10) and sets the focus buffer to “step” (Figure D.7 line 11) to trigger the appropriate step to fire.

If for example the goal which had been added and retrieved from memory was the initialize variables goal (as you would need to initialize variables to solve the rainfall problem), it would then trigger the firing of the ini\_variable production (Figure D.7 line 12). Step implementing productions (like ini\_variable, calc\_average and track\_variable) then implement the step by writing the step to its Python file (Figure D.7 lines 14 and 15) and log that step (Figure D.7 line 13). Since piecemeal modelling models program expansion, the focus buffer is set to “request costeps” to trigger a production that will check if initialize variables requires other steps to be implemented to resolve.

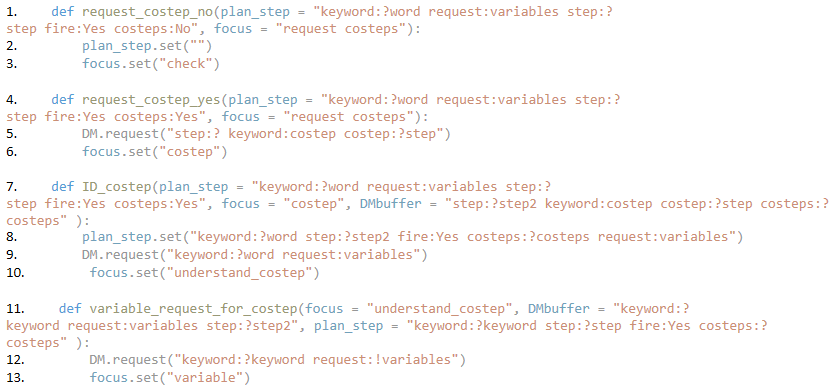


Figure D.8 Productions that Manage Goal Expansion in the *Goal-Expansion* Model

Costeps are tracked in the declarative memory goal – keyword associations (Figure D.6 lines 3-5), and the plan step buffer stores a given goal – keyword association until it is checked for costeps. If the plan step buffer does not have a costep in it, the request\_costep\_no production fires (Figure D.8 line 1), which clears the plan step buffer (Figure D.8 line 2) and sets the focus buffer to “check” which triggers the read\_list production to look at the next word in the problem text list. This is what happens after the variables are initialized, as it was itself a costep for the calculate average goal. If the plan step buffer contains a goal – keyword association that includes a costep (which the goal in the plan step buffer needs for its own resolution) then the request\_costep\_yes production fires (Figure D.8 line 4) and the declarative memory is queried for a costep for the goal just implemented (Figure D.8 line 5) and the focus buffer is set to “costep”. This then triggers the firing of the ID\_costep production (Figure D.8 line 7), which sets the plan step buffer to the goal to implement the costep (Figure D.8 line 8) and requests the variables for the costep from declarative memory (Figure D.8 line 9). Once the production sets the focus buffer to “understand\_costep” (Figure D.8 line 10), the variable\_request\_for\_costep production fire, to request the necessary variables for the new goal from the declarative memory (Figure D.8 line 12). This is different from the earlier variable\_request production (Figure D.7 lines 1-4) as it does not set the plan step buffer, which has already been set by the ID\_costep production. Once the variable\_request\_for\_costep production sets the focus buffer to “variable” (Figure D.8 line 13) the goal with its variables get added to the declarative memory and logged using the earlier described goal\_initialize production (Figure D.7 lines 5-8), and implementation proceeds the same. The goal to initialize variables gets added and implemented in this way, as a program expansion from the step to calculate average.

Goals are identified from the word list and via costep search until the model is done reading the list. At which point the Python program is considered finished.

Model 5: SGOMS



Figure D.9 Initialization of *SGOMS* Model and the Planning Units into its Declarative Memory

The SGOMS model uses an additional set of goal buffers (alongside the focus buffer) to track and shuttle information between productions. These buffers include a context buffer (see Figure D.9 line 1d), which tracks where the model is in the problem solving process (the focus buffer has a slightly different role in this model) and is initialized with the following slots: (a) finished – which contains the most recently finished unit task, and is initialized with none (as no unit tasks have been finished); (b) status – which tracks whether or not the model is currently occupied with a unit task and is initialized as unoccupied (as it does not start mid unit task); (c) conditions and variables – these track the conditions and variables as needed by the model, and are modified by different planning units (which will rely on different variables and conditions), storing them in the context buffer allows them to be accessible for the entire planning unit, and not just a single unit task (see Figure D.9 line 7).

The model’s declarative memory is initialized with the planning unit chunks which represent the unit tasks necessary to complete the "calculate average" goal of the rainfall problem (calculate the average of the positive numbers in the list) (see Figure D.9 lines 3-3d). The slots represent the following information: (a) planning\_unit - the name of the planning unit declarative memory chunk is part of; (b) cuelag - prior cue for unit task the model completed in planning unit; (c) cue – model’s cue for the unit task (also the last unit task completed); (d) calling – which tracks which planning unit requires the current planning unit to resolve and the (e) unit task - the unit task to be implemented by the model. , The planning unit buffer (Figure D.9 line 1e) stores the current line of the planning unit being implemented by the model, the line being any one of the lines initialized in the declarative memory (as in Figure D.9 lines 3-3d). The unit task buffer (Figure D.9 line 1f) tracks the progress of the model’s current unit task (whether it is complete or not). The focus buffer is only used to track the model’s process through a unit task, if there are multiple productions which must be implemented (See Figure D.10 lines 16 and 17).

Productions start a planning unit based on the contents of the context buffer. The production that initiates the calculate average planning unit (run\_calc\_ave; see Figure D.10 line 1) is able to fire as the initialized context buffer can act as a precondition of its fire (since the calculate average planning unit organizes many other planning units in hierarchical order below it, it must be able to fire after many different completed unit\_tasks, and may therefore fire under multiple contexts). It then logs its goal to calculate the average (Figure D.10 line 2), sets the planning unit buffer to reflect the first planning unit line in the declarative memory (Figure D.10 line 4), and unit task buffer is set to reflect the first unit task to be completed as part of the overall planning unit (line 3). Additionally, the context buffer is modified to reflect that no unit tasks have been completed for this planning unit thus far (as the planning unit is just starting) (Figure D.10 line 5). Once a unit task is loaded in the unit task buffer, the first unit task in the planning unit will fire.

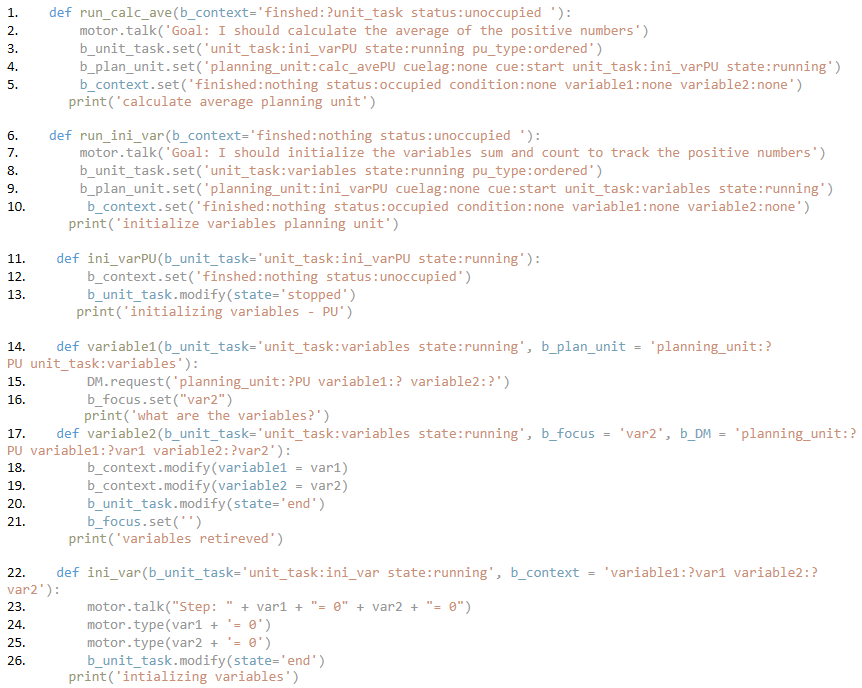


Figure D.10 Sample of *SGOMS* Model’s Productions that Control Planning Units and Implement Unit Tasks

According to the planning unit for calculating the average stored in the declarative memory (Figure D.9 line 3) the first unit task is to complete the initialize variables planning unit (ini\_varPU), therefore the first unit task which fires is the ini\_varPU unit task (Figure D.10 line 11), which changes the context buffer (Figure D.10 line 12) to the context which will act as the precondition to the production which starts the initialize variables planning unit (run\_ini\_var) (Figure D.10 line 6). From Figure 4.2 we know that that first unit task for the initialize variables planning unit is to retrieve the variables to initialize from the declarative memory. Those variables are sum and count (Figure D.9 line 5) and are retrieved by the unit task variables, which the run\_ini\_var sets its first unit task to in the unit task and plan unit buffer (Figure D.10 lines 8 and 9).

Variables are retrieved with the variable1 and variable2 productions. Variable1 fires when the unit\_task is set to variables in the unit task and planning unit buffer (Figure D.10 line 14). It then sends a request to the declarative memory to retrieve the variables needed by the current planning unit (Figure D.10 line 15), and the focus buffer is set to “var2” (Figure D.10 line 16). This is done to ensure the next production (variable2) fires (Figure D.10 line 17) as it is the production which then adds the variables to the context buffer so that they are accessible to the next unit task. From there the unit task is set to implement the step of initializing the variables in the python program (Figure D.10 line 22). It states its step to the chronotranscript (Figure D.10 line 23), then types the code necessary to initialize the variables into its Python file (Figure D.10 lines 24 and 25). Since this completes the initialize variables planning unit, the model returns to its calling planning unit, calculate average, and the next unit task in calculate average is implemented.

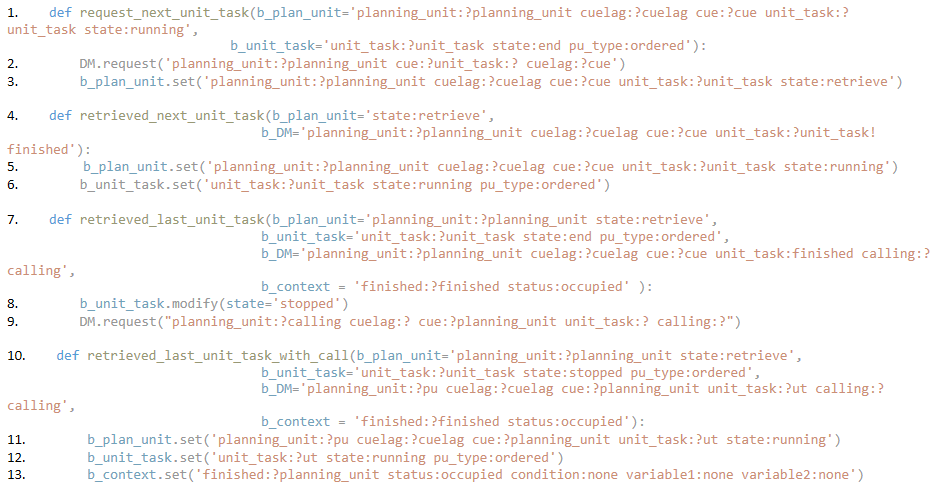


Figure D.11 Sample of *SGOMS* Model’s Production that Control Unit Tasks and Ending Planning Units

While I have given a general overview of how planning units and unit tasks are managed, there are separate productions that manage the retrieval of the unit tasks from within the planning unit and fire whenever a unit task has been completed. The request\_next\_unit\_task production (Figure D.11 line 1) fires based on both the plan unit buffer and unit task buffer storing within them a planning unit with a completed unit task (state = end). Tasks are set as ended by unit task productions when they complete their unit tasks (see Figure D.11 lines 13, 20 and 26). With the request\_next\_unit\_task production the model can then request the next unit task from memory. The next unit task is retrieved by the declarative memory buffer, and will contain the next line in the planning unit (Figure D.11 line 2). A reminder that the planning unit lines are stored in declarative memory (Figure 5.13 lines 3-3d). The planning unit state is also set to retrieve, to let the model know that it will be retrieving and beginning the next unit task (Figure 5.15 line 3). Once the planning unit state is set to retrieve, the retrieve\_next\_unit\_task is fires. The retrieve\_next\_unit\_task production’s conditions are for the planning unit to have the state set to retrieve and the declarative memory buffer to have the next planning unit line (Figure 5.15 line 4). The production then sets the planning unit (Figure 5.15 line 5) and unit task buffer (Figure 5.15 line 6) with the next unit task in the planning unit, which is then executed.

When the last unit task in a planning unit is completed, the next unit task requested by request\_next\_unit\_task will be “finished”. When that occurs instead of retrieve\_next\_unit\_task firing, the retrieved\_last\_unit\_task will fire (Figure 5.15 line 7). The production sets the unit task state to stopped (Figure 5.15 line 8), to track that the unit tasks of the planning unit have ended. Reminder that the planning units in declarative memory also store the calling planning unit, that is the planning unit which needs the current planning unit to be resolved to resolve itself (Figure D.9 line 3-3d). The retrieved\_last\_unit\_task production then requests the next step of the calling planning unit, that is the calling planning unit, has the current planning unit as one of its unit task and this production requests the next unit task in calling planning unit. Once the DMbuffer has the next unit task of the calling unit the retrieved\_last\_unit\_task\_with\_call production fires (Figure D.11 line 10). This production then sets the plan unit buffer to being a line within the calling planning unit (Figure D.11 line 11), the unit task buffer to reflect the next unit task in the calling planning unit (Figure D.11 line 12) and the context is cleared of the all the variables and conditions used by the last planning unit. For example, this set of productions fires when the initialize variables planning unit is completed, and the model returns to address the next unit task in the calculate average planning unit (which is the calling planning unit for initialize variables).