Carleton Cognitive Model

ACT-R

A Guide to employing Python ACT-R in Cognitive Modelling

Steve Highstead

2021

DRAFT

Abstract

This guide provides an outline for the various ACT-R modules used to create cognitive models. Python ACT-R has five modules. These include a procedural module, a declarative module, an imaginal module, a vision module and a motor module. The cognitive model developed using these modules is placed into an environment that is created and defined for the use by the cognitive model. This guide describes each module, and how it is used in a cognitive model

This is a DRAFT of the guide

Table of Contents

Introduction

Overview of the CCM ACT-R Architecture

General Description of the CCM ACT-R Architecture

There are a variety of software methods used to emulate human cognition. Approaches such as Deep Mind's machine based artificial intelligence (AI) used to play games such as GO and Starcraft have been developed, as well as other modelling software such as EPIC and SOAR used in robotics. ACT-R (Adaptive Control of Thought – Rational) was developed by John

ACT-R was, and still is, being developed by John R. Anderson at Carnegie-Mellon University in response to a question Allen Newell (also late of Carnegie-Mellon) posed, "How can the human mind occur in the physical universe?" This is also the title of Anderson's book (Anderson, 2007) which is the core description of ACT-R. The original version of ACT-R was developed using the LISP programming language. There has since been a couple of variants developed such as JACTR (using Java), etc. The variant of concern for this guide is written in Python, ported from the canonical ACT-R by Terrence C. Stewart and Robert L. West in 2007 at Carleton University, and is referred to as CCM (Carleton Cognitive Model)

This CCM ACT-R guide is intended to provide the cognitive modeller with the necessary tools to develop a cognitive model. The guide discusses the main components of the architecture, and how they are related. The first topic of discussion is what is a cognitive architecture, and what is it attempting to accomplish? This is followed with an overview of CCM-ACT-R and its components. A short example is provided at a high level to explain how the components fit together. Each of the components (modules) are then discussed in detail. The guide concludes with a model development life-cycle, complete with a discussion of how to approach the development and testing of a cognitive model.

Introduction

What is an architecture and its relation to a model

An architecture is methodology that provides a framework to relate a structure to its function. In the realm of building construction, architecture is a profession where building designers design a building space that conform to the specifications for the user of the building. A factory's function is very different from the function of a residential home. The building designers employ architectural practices to convert building specifications into a building design that meets the function described in the specifications. So, and architecture can be viewed as a way to turn a description of something into a form that performs the function in the description.

Specification → Architecture → Design → Construction → Building

ACT-R is an architecture since it provides a framework to develop a cognitive specification into a cognitive model. So, ACT-R can be understood as an architecture and the resulting computer program that adheres to this architecture is a cognitive model.

Cognitive Specification → ACT-R Architecture → Design → Development → Cognitive model

In cognitive modelling, the intention is to abstract the structure of the human brain by developing a cognitive model in a software language that performs functions reflective of the human brain's function. An architecture provides the link between the structure of a human brain and the structure of a cognitive model.

Human Brain → ACT-R Architecture → Cognitive Model

Anderson summarizes this concept as, "A *cognitive architecture* is a specification of the structure of the brain at a level of abstraction that explains how it achieves the function of the brain" (Anderson, 2007). The concept is that the structure of the brain is the foundation for the human mind; how a mind functions from the operations of a brain.

One way of understanding the function of a cognitive model is to refer to what is known as the cognitive sandwich:

Introduction

Sense → Think → Act

A cognitive model, essentially is a software program that performs this behaviour.

This guide describes the Python ACT-R Architecture that can be employed in developing cognitive models.

Overview of ACT-R

While this overview applies in general to canonical ACT-R, the specific references will be to CCM ACT-R

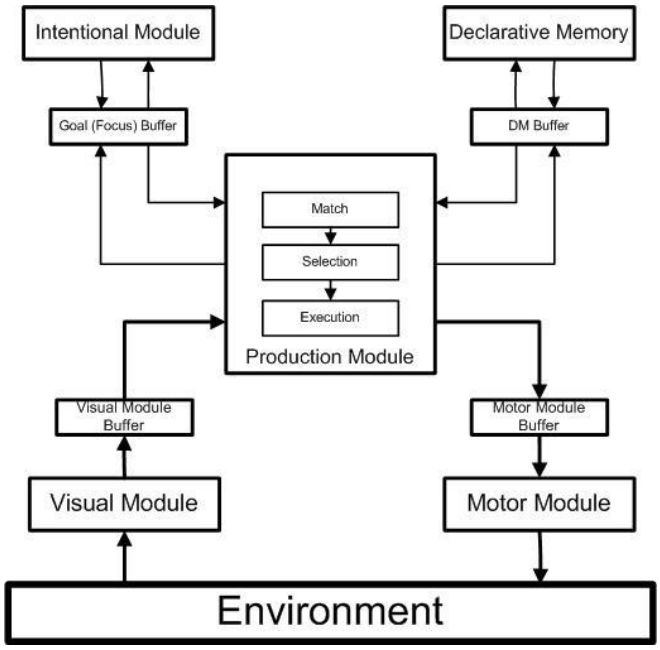

Figure 1.0 ACT-R Architecture

The two main components of ACT-R are the Declarative Memory and the Procedural Memory. The Declarative Memory contains knowledge of the world. The procedural memory contains knowledge of our behaviours; how we accomplish tasks. The declarative memory stores chunks of information, while the procedural memory contain productions: the basic building blocks of any ACT-R model. There are additional modules that are used to interface with the world as well as support model cognition.

The ACT-R architectures includes six modules that all work together. These are: the Production Module, the Declarative Memory module, the Imaginal Module (sometimes

referred to as the Intentional module), the Motor Module, the Visual Module, and the Environment Module. Only two modules are necessary to create a limited cognitive model: the Production Module and the Declarative Memory Module. Communication between the modules is through buffers, temporary storage, which pass data between modules.

ACT-R theory is grounded in the understanding that the human brain is organized by functional modules, and the ACT-R modules correspond with these functional areas. The production module, the core of an ACT-R model, corresponds to the procedural memory. The buffers correspond to working memory. The motor module corresponds to the cerebral motor cortex, and the vision module corresponds to the visual cortex. The imaginal module is a type of memory module that that is employed to temporarily store the intention of the cognitive model. The intentional buffer is sometimes referred to as the ‘goal’ buffer, or the ‘focus’ buffer. The imaginal buffer stores the initial purpose for the agent to take action. For example, the imaginal module buffer can contain a feature of a search. Once the conditions of the imaginal buffer are met, then the search ends.

From a Python perspective, each of these modules are a class that inherit modules from CCM and which constitute the ACT-R architecture and are instantiated when the model is run. The configuration of the modules used in a model are determined by the cognitive modeller.

For example, consider a model that represents the operation of reading text from a piece of paper and entering it into a keyboard. First, the production module will have method that reads text from a piece of paper, then enters that text into a keyboard of a computer. To make this work, there needs to be an environment that contains a keyboard and a sheet of paper. The text needs to be read which employs the visual module. Before the text can be entered into the keyboard it first has to be stored in the model's memory so as not to be

forgotten. The text is then recalled and the motor module is employed to enter the text into the keyboard.

ACT-R modules and their relation to brain architecture

General – Model Elements

This section introduces each of the ACT-R modules in general terms. Each module will have its own section, where a more in depth discussion provides an explanation how the software for the module functions.

Environment

The Environment is the virtual world wherein an agent operates. Generally, every model requires an environment in order to function. The environment contains the objects and affordances that an agent can use to succeed in meeting its goals. It is also possible for an agent to perform functions without any objects to manipulate, in which case an environment is not required. An example would be an *a priori* such as 2 + 2. There is no requirement for an environment to perform this cognitive task.

In CCM ACT-R, the environment is created as a class that inherits from the CCM class, Model. In brief,

class MyEnvironment(ccm.Model):

When the model is run, this class is instantiated as the model's environment, the quasi world wherein an agent can operate.

Items in the environment are stored following the chunk pattern used in buffers. An example of the code pattern is,

<object>=ccm.Model(isa= 'thing', location= 'location, ...)

The environment module will be presented in detail in the environment section.

Buffers

Buffers are collectively equivalent to working memory. Only items currently necessary for the operation of the model are held in the buffers.

One of the important elements in ACT-R to understand are the Buffers (refer to Figure 1). All the modules with the exception of the environment module, transfer data using a buffer. If the production module transfers data to declarative memory, it places the data into the declarative memory buffer. If the production module requests data from the declarative memory, the declarative memory places the data into a declarative memory buffer before a production can use it. For the vision and motor module the direction of data transfer is one way. For the vision module, data is passed in the vision buffer from the vision module to the production module. For the motor module, data is passed from the production module to the motor module.

The data is passed as "chunks." The form of a chunk is a string <'type: data, type: data, ...'>. Each 'type is a slot in the chunk. There is no limit to the number to the number of slots in a chunk in CCM ACT-R, but the customary practice is to limit the number of slots to 7±2 (Miller, 1956).

Production Module

The production module performs the function of a procedural memory system. It contains the steps to accomplish tasks during the operation of the cognitive model. This is where the model's ability to execute tasks occurs: the 'how to' part of memory.

The production module is a Python class that contains a set of methods that represent the productions used by an agent in its thinking processes.

((The production module is identified as a class in Python and is the core module in the model. Once the model is run, this module is instantiated as a Python object.))

Since the production module is seen as providing the agency for the model, the class is often referenced as an agent. The cognitive agent is instantiated as the production class's object. The production module inherits from the ACTR modules, which is the source of the module's cognitive functions. The agent is invoked from the production module class as follows:

```
class MyAgent(ACTR):
```

Each production is designed by the model designer. The format for a production is to first identify the conditions in which the production will fire. If the production fires, then tasks identified in the production can be executed. The CCM code for this is:

```
def production_name( focus= 'entry_requirement', <<other conditions>>):
    <production instructions>
```

Productions are conditional methods having an *if-then* format. If the production's entry requirements are met, the production 'fires' and the production method is executed. The productions do not 'fire' in sequence. Any one of the productions can fire if the entry conditions are met.

There is no limit to the number of productions that can be included in a production module other than efficiency concerns for the cognitive model.

The production module can be understood as the central thinking control for the model, and it is often compared to the brain's basal ganglion for this reason.

Declarative Memory Module

Declarative memory is the long term memory for the model. It is invoked as a part of the production module, and provides a memory bank for the model's agent. To add something to memory, there first needs to be a buffer. The data is added to memory as chunks. There is no limit to the number of chunks that can be added to the declarative memory. Stored chunks are understood cognitively to be 'symbols'.

There are a number of conditions (subsymbolic functions) placed on memory chunks such decay, activation, noise, salience, etc. which are discussed later.

The code pattern to create a memory for a production module agent is:

```
DMBuffer= Buffer()
DM = Memory(DMBuffer, <sub-symbolic functions as desired>)
```

The declarative memory is commonly referred to as DM. The first thing needed is a buffer for the memory. The DM variable is assigned the module Memory giving it a buffer to hold chunks that pass to and from the production module. To add a chunk to memory, the command is,

```
DM.add(chunk)
```

To recall something from memory the command is,

```
DM.request(chunk)
```

These commands reside within a production method.

The declarative memory is an extensive theoretical area in ACT-R with numerous symbolic and sub-symbolic functions associated with it. These will be discussed in detail in the declarative memory section.

Imaginal Module

This is been described as a temporary memory for the goal or focus buffer. The imaginal module holds the cognitive agent's intention. It could be used as a temporary store for a feature for an image of a search item. When the search item is found then the feature can be checked against what is stored in the imaginal module's buffer. The imaginal buffer is declared the same way as the focus buffer,

```
focus_buffer = Buffer()
imaginal_buffer  = Buffer()
```

Vision Module

The vision module is one of the input modules for the agent. The second is the aural module which is not implemented in CCM ACT-R as yet. There are two vision modules that can be implemented within an agent: the vision.py module and the sosvision.py (sos := simple operating system) module.

As with the declarative memory, the vision module is declared within in the production (agent) class. It is identified by:

vision_buffer = Buffer()

vision = SOSVision(vision_buffer)

The vision module does not "see", but performs the function of supplying the agent with a fully formed chunk. For the vision module to "see" something, that something must exist in the model's environment. Once there is a chunk in the vision buffer, it can be used by a production method

Motor Module

The motor module is the only output module for CCM ACT-R. It is used to change environment object's chunks. It is declared as a class which is called by the agent as required. The form the code takes is:

```
class MotorModule(ccm.Model):
    def do_something(self, <var>)
    yield <int>
        self.parent.parent.chunk_name.chunk_slot='<new_value>
```

def do_something(self, <var>) This line identifies the name of the method (function). It contains two parameters: "self", and "<var>".

The third line of the motor class is the instruction to change the environment chunk. Self refers to the function, the first 'parent' refers to the production module, and the second

'parent' refers to the environment module. The chunk_name is the name of the chunk in the environment while the chunk.slot is the slot for the chunk_name and <new_value> refers to the new contents for the environment's chunk.slot.

The <var> is optional and only required when a <var> is used for a <new_item>

Yield refers to the amount of time it takes for the motor method to execute.

The motor module is invoked within the production (agent) class. If there is parameter passing between the production module and the motor module, then a buffer is created for the motor methods.

Instantiation of an ACT-R Model

In Python, classes are instantiated as objects by assigning them to variables. For an ACT-R model, the first class to be instantiated is the environment. Next, the agent is instantiated. The motor module it is also a class, is instantiated when the production (agent) module is instantiated. Next, the agent has to be placed within the environment. The code to do this is,

```
env  = Environment()              # name the environment
agentA = MyAgent()                # identify the agent
env.agent = agentA                 # put agentA into the environment
env.run(<number of runs(optional))  # run the environment
ccm.finished()                    # stop the model
```

This code creates the ACT-R model and brings all the inherited classes into the model in order for the model to function.

Next, each of the modules introduced above are presented in detail.

CCM ACT-R Buffers

General Description

A Buffer in CCM ACT-R is an inheritable class that performs a very specific function. It is found under ccm/buffer.py as a part of the CCMSuite3. Collectively, all the buffers in a model constitute the context for the model. They are considered to be the model's working memory. A buffer holds a chunk, a fundamental element in an ACT-R model, which contains knowledge and facts that can be used in cognition.

CCM Buffer Structure

The CCM Buffer is designed to hold a Chunk, which contain a number of slots. Chunks are stored as strings in CCM. There is no software limit to the number of slots a chunk can hold, but it is generally limited to 7±2 for cognitive theoretical reasons. Each slot contains one item called a symbol. These symbols can be just about any object, including other chunks. The convention in canonical ACT-R is to label each slot, but this is not strictly necessary in CCM ACT-R. For clarity and readability, however, it is recommended that slot labels are employed.

A buffer's chunk takes the form of a python dictionary employing key:value pairs. In CCM, this is deep in the model code for buffers (buffer.py), and not immediately relevant to understanding CCM.

Chunks Slots

As an example, a chunk can contain the properties of an animal such as a dog. If the dog is a Labrador, that is large, hairy, and friendly, then the chunk could be expressed as,

'isa:dog breed:Labrador coat:long manner:friendly'

In practice, 'isa' is the first slot and provides the chunk with an identification label. The next slots can be just about anything that describe the properties for the chunk. Each of the properties in a chunk's slot are considered and referred to as symbols.

The whole chunk is stored as one string of key:value pairs.

Buffer Assignment

Buffers are assigned to modules, which provides them with specific uses in a model. There really is no limit to the number of buffers employed in a model. For example, a vision buffer may hold a chunk that can be sensed from the environment. A declarative memory holds chunks that are facts that are at the disposal of the production system. As well, chunks can be passed to the motor module that can modify the model's environment.

Buffers can assigned to modules for their use. The most common example is the memory module. The contents of declarative memory are stored as chunks using buffers. For this to occur, the declarative memory is assigned a buffer as follows,

```
# Memory Module assignment
   DMbuffer = Buffer()
   DM = Memory(DMbuffer)
```

Similarly for a vison module,

```
# Vision Module assignment
vision_buffer = Buffer()
vision = SOSVision(vision_buffer)
```

There is no limit to the number of buffers a model can have. They do not necessarily need to be associated with a specific module, but can be used independently of cognitive modules.

There are other parameters that are required for these modules which will be discussed in the relevant sections.

Buffer Commands

The module, buffer.py, contains a number of methods. Buffer contents are mutable, and can be overwritten by setting the buffer to a new chunk. For example, a buffer can be 'set' to a value. There is also a 'clear' instruction, as well as a 'modify' instruction. For example,

```
vision_buffer.clear()          #This clears the current contents of the buffer.
focus_buffer.set(',value>')    #This replaces a chunk in the buffer with a new chunk
DMbuffer.modify('<value>')  #This can modify a chunk in the buffer.
```

It needs to be noted that the buffer contents are current states, and do not change the chunks within their assigned modules. As noted earlier, the buffers collectively provide the context for the model, and are considered the models working memory.

Since a Buffer() contains only one chunk at any given time, the chunks are accessible directly. For example, the current contents of a declarative memory buffer can be accessed as follows.

```
print(DMbuffer.chunk)
```

A specific slot within the chunk can be accessed, using key:value pairs[1]. For a chunk that contains the following,

'isa:dog name:George disposition:friendly'

The name of the dog can be accessed by,

```
print(DMbuffer.chunk['name']
```

A production system can use buffers for temporary storage, and need not be assigned to a particular module.

[1] The slots of a buffer's chunk do not require a key in CCM ACT-R. They can be accessed as list index.

Python ACT-R Environment

General Description

In ACT-R, the environment is a Python class which contains objects and affordances for use in the cognitive model. Agent(s) are placed within the environment and can make use of the objects within the environment, if the modeller so chooses. An ACT-R environment can have any number of objects from none to as many as the modeller decides to have. The modeller decides which objects are placed into the environment. Once an object has been placed in the environment, it can be seen and modified by the agent's actions.

There can be only one environment per cognitive model.

The reason for an environment

ACT-R is designed so that an agent, in order to function, must be able to sense and manipulate objects. These objects are located external to an agent and in the case of ACT-R, in a virtual environment. The sense → think → act paradigm, the so called cognitive sandwich (Dawson, 2013) requires an environment. This means, for cognition to occur at all, there must necessarily be an environment (virtual or real) which contains objects that can be sensed and manipulated.

Also, for an ACT-R model to run, an environment is a necessity because it runs an environment with one or more agents in it.

General Format for an Environment

In Python ACT-R, the environment module is declared as a Python class. As with any Python class declaration, it can occur anywhere within the program model. Customarily, though, it is the first class to be declared. The syntax used is,

class <Environment_Name>(ccm.Model):

Environment Module

The environment class naming convention is to use camel case starting with a capital letter. The environment calls the ACT-R ccm Model module. This is a module that provides the environment class with the required cognitive model capabilities.

If the class does not contain any objects and is not used in the cognitive model, then the Python keyword "pass" can be used.

```
class MyEnvironment(ccm.Model):
    pass
```

Even though the environment is not used by the agent in the cognitive model, it still must be declared. As with all Python class declarations, the statement ends with a colon (:).

Cognitive models can be developed without an environment class. However, the concept of an agent operating within an environment supports cognition, and from a conceptual perspective would appear to be necessary. However, it is not technically required.

To identify an object in the environment, the syntax used is,

<object>=ccm.Model(isa= 'thing', location= 'location', contents= None ...)

The <object> is an item to be found within the environment. The <object> is assigned a value that is defined by a ccm.Model class that has the parameters of the object passed to it.. The contents are stored in 'slots'; each slot containing one item. If a chunk has an empty slot then the Python keyword "None" is used. Otherwise, the <contents> could be 'full', or 'half_empty', 'grapes', etc. Anything the modeller decides. The chunk items are stored in their slots as strings.

{Unlike a declarative memory chunk's slot which has the syntax isa:'Thing, the environment slot is written as isa='Thing'.}

Example 1 of an Environment Class

An example of an environment used in the example model is,

```
###################### The Environment ############################
```

Environment Module

```
class Desk_Environment(ccm.Model):
    """
    This model reads a message from a piece of paper
    then inputs the message via a keyboard
    to be displayed on a computer monitor
    The agent then reads the computer screen, and remembers the message.
    The message is recalled and displayed.
    kbd == keyboard
    vdu == video display unit (computer monitor)
    note == a piece of paper with something written on it.
    """
    kbd = ccm.Model(isa='KBD', location='desk', message=None)
    vdu = ccm.Model(isa='VDU', location='desk', screen=None, salience=0.9)
    note= ccm.Model(isa='note', location='desk', message='Hello World', salience =0.9)
                        #note: salience= 0.5 the lowest for SOSVision to pick up
```

In this case, the environment is named <Desk_Environment>. There are three items of note for the agent, a keyboard, a computer screen and a piece of paper. The keyboard has nothing typed into it, therefore << message = None >>. The computer screen is not displaying anything therefore << screen=None >>. The piece of paper has a message written on it so the note is << message='Hello World' >>. Note that the slot descriptions are all Python strings.

The parameter <salience=0.9> describes how significant the object is to the vision module. If the salience is to low, when the vision module is requested to read the note, it may not notice it. The minimum salience in this instance is 0.5 (found by trial and error), so setting the salience to 0.9, makes it more likely to be spotted by the vision module.

Example 2 of an Environment Class

The following example comes from the ACT-R course taught at Carleton University Cognitive Science Department. It is the environment for the classic training model, "Ham and Cheese Sandwich.".

```
class Subway(ccm.Model):
    bread=ccm.Model(isa='bread', location='on_counter')
    cheese=ccm.Model(isa='cheese', location='on_counter')
    ham=ccm.Model(isa='ham', location='on_counter')
    bread_top=ccm.Model(isa='bread_top', location='on_counter')
```

In this example, the elements needed for constructing a ham and cheese sandwich are made available for the production module agent.

Example 2 of an Environment Class

This example is found in ccm tutorials/environments as 'stopping.py.'

```
class ForcedChoiceEnvironment(ccm.Model):
 # this is an action that can be taken by the agent in the environment
        def press(self,letter):     # 'self' refers to the thing we are currently
                                    #  of defining.  In this case, the environment
        log.action=letter           # here we record what letter was pressed
        if letter=='A':
                self.reward=1    # if it was 'A', we set the reward to one.
        else:
                self.reward=0     # otherwise, set it to zero.
self.score=self.score+self.reward
```

```
log.score=self.score         # record the score in the log
self.trials=self.trials+1    # increase the number of trials
log.trials=self.trials       # record the number of trials
if self.trials==20:          # if the number of trials is 20
self.stop()                  #  stop the simulation
```

While this interesting, what it does is adjust the utility of the function if the agent presses letter 'A'. The simulation runs for 20 trials before stopping. The result is the following,

```
0.000 action A
    0.000 score 1
    0.000 trials 1
    1.000 action A
    1.000 score 2
    1.000 trials 2
    ...<<continues for another 18 trials>>
```

It is presented here as an example how an environment can look very different than the ones presented earlier. Just about anything can be placed into the model's environment.

Environment Characteristics

Action taken by an agent on the environment can modify any of the chunks of an object. For example, if an agent types the message from the piece of paper into the keyboard, then the message chunk for <kbd> can be changed to << message='Hello World' >> through the agent's action within a production method.

The items in the environment can 'seen' as well as 'modified' by the agent. This is done by the vision and motor modules which are called from the relevant methods in the agent's production module.

Environment Module

Since the environment does not perform any cognitive functions, it can be used to create the conditions for experimentation. For example, the environment can contain code which turns a light on an off at a set period of time, or randomly. The cognitive agent productions can then be designed to respond by turning the light off when the agent notices the light is on. The timing of the agent's response can then be measured, and compared to empirical results of similar tests of human beings.

CCM ACT-R Production Module

General Description

A production module holds the 'thinking' or cognitive part of an ACT-R model. It is the procedural memory for an ACT-R model. The procedural memory contains steps, called productions in ACT- R, that occur sequentially, and in no particular order, to accomplish a cognitive task. The production module is also referred to as the "Agent", since the production module receives sensations from and acts on an environment.

A production (also referred to as a production rule) is designed as a conditional proposition which is essentially an *if-then* rule. A conditional proposition is of the form, if <antecedent> then <consequence>. An example is, "If John is an unmarried man then, classify John as a bachelor." The *if* or Right-Hand-Side (RHS) contains the antecedent in the form of a pattern for a matching condition, which in the example is 'John is unmarried'. If the matching condition is true and there is a match, then the Left-Hand-Side (LHS) or the consequence of the expression is executed, which is to 'classify John as a bachelor.'

Procedural memory in ACT-R can learn over a period of time, as with human procedural memory. Practicing a particular exercise, improves one's ability to complete the work. This is can also be set in CCM ACT-R.

Production Module

The production module is established in a model as a python class as follows,

```
class MyAgent(ACTR):
    <class timing attributes>
    <class utility attributes>
```

Both class timing and utility attributes are presented below. Neither of these attributes need to be explicitly set, since the production modules have default values. The decision to

use them is a decision made by the cognitive modeller based on what the modeller is attempting to accomplish

General Format for a Production Rule

Buffers contain the conditions for matching to occur. If, for example, a memory buffer provided the correct information to match the RHS of a production rule, then the LHS of the production rule would be executed.

```
def prod_name(RHS):
    <LHS>
```

For example, such a production rule could look like,

```
def  prod_name(buffer = 'matching_condition'):
    <production instructions>
```

More specifically,

```
def  prod_name(memory_buffer = 'isa:John' status:unmarried'):
    print('John is a bachelor')
    DMbuffer.add('isa:John' 'status:bachelor')
```

In this example, the memory_buffer is compared to the matching condition <'isa:John' status:unmarried'>. If the memory_buffer contents are the same as the matching condition, the production rule will 'fire' and execute the two instructions <print()>, and update the declarative memory buffer, <DMbuffer.add()>.

While the matching condition looks like an assignment (=), the buffer contents are not changed. It is the CCM ACT-R syntax used for pattern matching in the RHS of the rule.

If no match occurs, i.e. 'John is married', then the rule will not fire.

Production Rule Usage

All the rules (methods) take the conditional form in a production module. There is no theoretical limit to the number of rules in a production system. It depends on the cognitive modeller, and what is trying to be accomplished.

To provide some structure to production rule organization, a production rule can have intentionality. This means, there can be an additional condition that is necessary before the production rule will fire. In a sense this gives the production rule a reason for firing. In the above example, such a reason could be to check the marital status of an individual. The intentionality of a rule uses a goal buffer; the goal or reason for the rule's existence. The goal is also referred to as the rule's focus, implemented as the focus buffer. In this context, both terms mean the same thing. The following is an example of how this would work.

```
def get_person(focus = 'request_person'):
        DM.request('isa:John status:?status)
        focus.set('check_status')
```

The 'get_person' production rule recalls John's status from the declarative memory module and places it in the declarative memory buffer. Using '?status' indicates that the buffer's chunk slot will contain whatever the memory module's status for 'John' is holding. The <?status> is an ACT-R variable that identifies the chunk's slot position as a variable.

It should be remembered that there is only one buffer per module. In this case, the declarative memory could have 100 people with their status, but only 'John' is requested and placed into the DMbuffer. This is how knowledge is transferred from the declarative memory to the production system

```
def  status_check(focus = 'check_status',
                  DMbuffer = 'isa:John status:unmarried'):
        print('John is a bachelor')
```

DM.add('isa:John status:bachelor')

In practice, it is a little more involved. For example, there has to be a memory datum that contains the information to be manipulated. That means it has to be added to memory. This is done with the DM.add() instruction.

To start the process, the first production rule initializes the system with the rule def init(). This performs the same purpose and function as __init__(self) in Python. While it is not strictly necessary to use an init() production in CCM; the attribute assignments can be done outside this production. However, if it is used, it is the first production executed, and a good place to initialize buffers and attributes, such as setting the memory location and providing a starting intention for the focus buffer.

The next rule requests the status of 'John' from memory. This request places the chunk containing the data into the memory buffer using the <?status> variable. Once the status is placed into the memory buffer, <DMbuffer>, it can be used to check the memory chunk.

After confirming the match to the memory contents 'John' as 'unmarried', then the memory can be updated to bachelor.

To confirm that the memory has been updated, a production rule is fired to retrieve the memory contents. Once the memory buffer contains the chunk from the memory, then it can be checked for a match.

Appendix A contains sample code for this little *a priori* exercise

S**ub-symbolic Production Parameters**

This explanation presents the surface symbolic level for productions. There is also subsymbolic parameters to be taken into account, mostly that deal with how the procedural memory module functions but also how the productions system learns how to perform during the execution of the rules

Production Timing

The default amount of time for a production rule to fire is 50 milliseconds (0.05). This can be adjusted as part of the attributes for the production class, and can be explicitly set with the following instruction,

```
class Model(ACTR):
    production_time = 0.5
    production_sd = 0.0
```

The production time <production_time=0.5> is shown as the 0.5 value, the default 50 millisecond value. However this can be adjusted as required by the cognitive modeller.

The standard deviation <production_sd=0.0> for the time can also be set. This allows for a range of timing about the 50 millisecond median. The default value is 0, but sometimes the value of 0.01 has been used.

Production Utility

All the production rules are checked in parallel to determine which should fire. If two production rules have the same matching criteria, and the same utility then the production that fires will be selected randomly. If one production has a higher threshold than the second production, it will be selected over the production with the lower threshold.

Productions rules have a "utility" or a measure of how likely a production will fire. The default threshold for utility is 0.0. This can be made explicit setting the threshold attribute for the production class as follows,

```
class Model(ACTR):
    production_threshold = 0.0
```

An example of how this is employed is by ensuring the productions will always fire by setting the threshold low, such as,

```
production_threshold=-20.0
```

This is a subsymbolic support for each production rule. As a model is run, and the production rules fire, the utility of those production rules increase.

A production can have its utility increased by providing the production with a reward directly. The reward is included in the LHS of the production. The code snippet is,

```
def production_name(<matching conditions>):
    <production_instructions>
    self.reward(0.1)
```

The term self refers to the production rule. The reward term is the instruction found in the production modules subsymbolic methods (ccm.lib.actr.production.py). The default for reward = 0.0. By increasing the productions reward, the productions threshold is also increased making it more favourable than similar productions with the same matching conditions to be selected to be fired. Values that are positive are good while values that are negative are bad for the threshold setting.

The learning systems require a way of determining success or failure. This can be done explicitly using the following instructions,

```
self.success()
self.fail()
```

The utility value can also be changed by employing subsymbolic learning functions.

Utility Learning

Each production can have its threshold changed, depending how often it is fired. The more times it fires, the threshold changes so that the probability the production will fire increases. This is equivalent to learning. The more a cognitive function is performed, the

more likely it will be remembered. The value for the utility for the production is defined by the following function,

$U_i = P_iG_i – C_i$

Where i identifies the production

U_i is the production rule utility. The default value is 0.9

P_i is the probability that production i will fire. The default value for the P = 1.

C_i is the cost, in time, for the production to complete. The default value is 0.05 seconds

G_i is the value of the objective measured in time. This a global value with a default of 20 seconds

While this equation provides the expected utility, there is also noise (ε) to be considered. The equation is adjusted to reflect this as,

$U_i = P_iG_i – C_i + \varepsilon$

In Python ACT-R the noise is set by the PGC learning function presented below as PMPGC, the production module's probability, cost, and cost calculations.

PMPGC: the old PG-C learning rule

This is the original learning rule. This is set as a parameter for the production module as,

pm=PMPGC(goal=20)

The probability P is arrived at from,

P = Successes/Successes + Failures

where Successes and Failures are actual successes and failures of productions. Similarly for the cost of learning (C)

C= Efforts/Successes + Failures

where Efforts is the accumulated time over all the successes and failures.

Default values are, Efforts = 0.05, Successes = 1.0, and Failures = 0.0.

PMPGCNoise

The noise parameter is used to cloud the utility of a production rule. The production rule's utility must be high enough to overcome the noise in order to fire.

pmnoise = PMNoise(noise=0, baseNoise=0.0)

Defaults are: noise = 0.0, BaseNoise = 0.0.

PMPGCSuccessWeighted: the success-weighted PG-C rule

This is success weighted expression for that attribute. ((Success-weighted PG-C (see Gray, Schoelles, & Sims, 2005))

This is set as a parameter for the production module as,

pm=PMPGCSuccessWeighted(goal=20)

C = production_time / production_successes

U=P*G-C

PMPGCMixedWeighted: the mixed-weighted PG-C rule

This is the mixed weighted version of the PMPGC parameter. ((mixed-weighted PG-C (see Gray, Schoelles, & Sims, 2005)))

pm=PMPGCMixedWeighted(goal=20)

P = production_successes/production successes+production_failures

C = production_time / production_successes

U=P*G-C

PMQLearn: standard Q-learning

This a variant of production utility calculation using q learning

pm=PMQLearn(alpha=0.2,gamma=0.9,initial=0)

PMTD: TD-Learning (Fu & Anderson, 2004)

Production Module

This is Time delay learning, (((Temporal Difference Learning (see Fu & Anderson, 2004))

pm=PMQLearn(alpha=0.1,discount=1,cost=0.05)

PMNew: the new ACT-R 6 standard learning rule

This the currently popular learning function. ((The new TD-inspired utility learning system)

pm=PMNew(alpha=0.2)

Appendix - A Syllogism Model

```
import ccm                          #Import the ccm modelling modules
from ccm.lib.actr import *          # Import the ACT-R modelling modules

class DM_Update(ACTR):

    """ Create Buffers and Modules """
    focus = Buffer()          # Assign focus buffer
    DMbuffer = Buffer()               # Assign Memory buffer
    DM = Memory(DMbuffer)  # Invoke Memory module

    """ Initialize the model """
    def init():
        DM.add('isa:John status:unmarried')
        print('start')
        focus.set('request_person')

    """Get the memory status of a person"""
    def get_person(focus = 'request_person'): # IF the focus condition is matched
        DM.request('isa:John status:?status')     # THEN do this
        focus.set('check_status')

    """If the status matches then update the memory"""
    def status_check(focus = 'check_status',
                     DMbuffer = 'isa:John status:unmarried'): # IF these conditions are met
        print('John is unmarried is a match')                    # THEN execute these instructions
        print('Update John to be a bachelor')
        DM.add('isa:John status:bachelor')
        DMbuffer.clear()
        focus.set('request_status')

    """Request current memory status"""
    def request_status(focus = 'request_status'):

        DM.request('isa:John status:?status')
        print('Get updated status')
        focus.set('report_status')

    """Report on the current memory status"""
    def report(focus = 'report_status',
               DMbuffer = 'isa:?isa status:?status'):
        print(f'The status of {isa} is {status}')
        self.stop()

""" Instantiate model and run """
model = DM_Update()
ccm.log_everything(model)
model.run()
ccm.finished()
```

Appendix B – Syllogism Model Log

The log output for the code in Appendix A

```
start
   0.000 production_match_delay 0
   0.000 production_threshold None
   0.000 production_time 0.05
   0.000 production_time_sd None
   0.000 DM.error False
   0.000 DM.busy False
   0.000 DM.latency 0.05
   0.000 DM.threshold 0
   0.000 DM.maximum_time 10.0
   0.000 DM.record_all_chunks False
   0.000 DMbuffer.chunk None
   0.050 production None
   0.050 DM.busy True
   0.050 focus.chunk check_status
   0.100 DMbuffer.chunk isa:John status:unmarried
   0.100 DM.busy False
   0.100 production status_check
   0.150 production None
John is unmarried is a match
Update John to be a bachelor
   0.150 DMbuffer.chunk None
   0.150 focus.chunk request_status
   0.150 production request_status
   0.200 production None
   0.200 DM.busy True
Get updated status
   0.200 focus.chunk report_status
   0.250 DMbuffer.chunk isa:John status:bachelor
   0.250 DM.busy False
   0.250 production report
   0.300 production None
The status of John is bachelor
end...
```

Appendix C

Project notes

From production.py regarding the use of the "magic variable" 'top.'

#TODO: rethink this. top should refer to the highest level in the tree.

    # but it would be nice if we had something to refer to the highest level

    # that could cause a matching change, to optimize the yield._top.changes

    # call. This was the original idea and may impact the logic calculating

    # top above. However, any fix to this should make sure not to break the

    # rock paper scissors tutorials.

Example from production.py

if 'top' in keys: top=m

m = m.parent

And

context['self']=self

context['top']=top

self._top=top

self._context=context

{{{no idea what this means}}}

Python ACT-R Declarative Memory

General Description

The declarative memory is used to store facts in the form of "chunks" that the model finds useful. With the procedural memory (production module), the declarative memory (memory module) is employed in cognitive models. The two memory systems communicate with each other using buffers. The procedural memory can request information from the declarative memory, and the information can be stored into the procedural memory by the procedural memory.

Information is passed in the form of chunks, which contain slots of key:value pairs. While there is no limit to the number of slots in a chunk, for cognitive and pragmatic reasons, the limit is set at 7±2.

An example of a chunk having three slots for a dog name George that is friendly,

chunk = 'isa:dog' 'name:George' 'disposition:friendly'

Once a chunk is stored into memory it cannot be changed, or modified. While our memories change over time, and we see changes in the state of some our information, the original form the information took remains. For example, if we know that John is an unmarried man, we also know John is a bachelor. If I store 'name:John' 'status:unmarried' into memory; knowing that unmarried men are bachelors, I can also store 'name:John' 'status:bachelor' into memory. The second store action does not overwrite the first chunk of information, but adds a second chunk of information to memory.

If two identical chunks are added, then they are merged into a single chunk.

Declarative Memory Module

The declarative memory module is declared within a production module. The syntax is,

class MyModel (ACTR)

```
<class timing attributes>

<class utility attributes>

# Create a declarative memory

DMbuffer = Buffer()

DMmemory =Memory(DMbuffer)
```

The first step is to identify a buffer for use by the declarative memory. This done by assigning a buffer to a variable (DMbuffer = Buffer()). By doing this, the DMbuffer has all the methods and attributes from the Buffer() module. The next step is to create a memory for use in the production module. This done by assigning the Memory() module to a variable, (DMmemory = Memory(DMbuffer). The Memory module inherits from the Buffer module through the inheritance assignment Memory(DMbuffer). Since DMbuffer and DMmemory are variable, they can carry any label such a "buf" for the buffer and "mem" for the declarative memory module. For example,

```
buf = Buffer()

mem = Memory(buf)
```

While this code is being used for a declarative memory, it should be noted that it can be any memory that the modeller sees fit for their model. More than one memory can be created for a model, but standard practice is to have only one fact storage memory for a model.

This does not take into account the sub-symbolic functions that can affect the behaviour of the declarative memory such as decay, recollection threshold etc. These will be resented later.

General Format for a Declarative Memory Module

The chunks passed to the declarative memory are stored as list, a list of chunks. Since each chunk is a string, it is stored as an object in the memory's list.

To add a chunk to the DMmemory use the Memory.add(chunk) method.

DMmemory.add('isa:dog name:George disposition:friendly')

This instruction will add a chunk to the declarative memory. The chunk is first placed into the DMbuffer, then the memory module copies the chunk into the memory list.

To retrieve a chunk from declarative memory use the Memory.request() method.

DMmemory.request('isa:dog name:George disposition:friendly')

The chunk is placed into the DMbuffer then the DMmemory responds to the request by placing the contents from the DMmemory into the buffer. This is accomplished by matching the pattern of the request with the contents of the DMmemory module. Once there is a match, the relevant chunk is placed into the DMmemory's buffer.

Variables can also be employed to retrieve chunks. For example, if the disposition is not known for the dog named "George" the request can take the form,

DMmemory.request('name:George disposition:?disposition')

Note two things: first, the request does not require the whole chunk, the match occurs on the name "George." If there is more than one "George" then the chunk is randomly selected. The second thing to notice is that the disposition is identified as a variable. In CCM, variables are preceded by a "?." The value for the key "disposition" will be returned, and subsequently can be used in eth production as the variable "disposition."

The same process is followed. The request is placed into the DMbuffer. When the DMmemory matches the request held in the buffer, it replaces the contents of the DMbuffer with the contents form the DMmemory, which can be used by the production requesting the information.

Due to the timing of production rules, it takes two productions to retrieve and use the DMmemory information.

def retrieve_Chunk(focus='get_chunk'):

```
        DMmemory.request('name:George disposition:?disposition')
        focus.set('report)
    def report_chunk(focus='report',
                     DMbuffer='name:George disposition:?disposition'):
        print(f 'The dog\'s name is {George} and his disposition is {disposition}.')
        focus.set('next_rule')
```

Returns,

> The dog's name is George and his disposition is friendly.

Once the model (or MyModel) class is instantiated as an object, the initial chunks can be added to the declarative memory. This can be done as instructions in the model's def init() or as part of the production run code block at the end or the program.

```
mod = MyModel()
mod.DM.add('chunk') # adds a chunk to the declarative memory
mod.run()
```

Adding Chunks to Memory

There are numerous ways of adding information to memory. It depends what the modeller wants to accomplish. All these are examples of the memory.add(chunk) method. Here are a few examples.

Explicit contents

When the information is known, the string containing key:value pairs is created as the arguments for the memory.add() method. Taking the earlier example,

```
DMmemory.add('isa:dog name:George disposition:friendly')
```

each slot has clearly identified contents. Two things of note. The first is that the listing of slots is one string type with a quotation and the beginning and at the end of the slots. The second thing to notice is that each slot is given a value for each key.

Non-explicit slot contents

There will be occasions where the slot contents are being defined by a variable. For example, if the production run has two independent buffers, some of the contents of which are variables, and these variables need to be stored, the chunk formation must take this into account. For example, if a chunk has been stored into the imaginal buffer, and it is to be transferred to declarative memory, but it is not clear what the imaginal buffer is holding, then the following instructions can be used,

ImaginalBuffer1('name:George dispostion:?dispostion isa:?isa')

DM.add('name:George diposition:?disposition isa:?isa')

The contents of the imaginal buffer's chunk, whatever they may be, are stored into the declarative memory

Combining specific slots from different buffers into memory

If there ae two imaginal buffers, and the modeller wants to store specific slots from one buffer and different slots from a second buffer into one memory chunk, the following instructions can be used,

```
def deduce(focus='deduce',
        IM1buffer ='predicate:man subject:?subject',
        IM2buffer ='subject:man predicate:?predicate'):
    print(f'The conclusion is: {subject} is {predicate}')
    #Add the conclusion to declarative memory
    DM.add('premise:conclusion subject:?subject predicate:?predicate')
    focus.set('stop')
```

Note, there are populated slots in the IM?buffers that are not required for the rule to fire, therefore they do not need to be explicitly identified. If the predicate of IM1buffer equals

the subject of IM2buffer, then the production rule fires. The IM1buffer ?subject, whatever it may be, and the IM2buffer ?predicate are placed into the chunk to be stored into memory.

Another way to load the chunk from two different buffers is as fallows,

```
IM1buffer = 'predicate:man'
IM2buffer  = 'subject:man'
DM.add({'premise'    : 'conclusion',
            'subject'     : IM1buffer.chunk['subject'],
            'predicate'  : IM2buffer.chunk['predicate']})
```

While this not really acceptable for an ACT-r perspective, it does display what a chunk looks like. What is of interest in this approach, it the chunk is explicitly define by its dictionary type of key:value pairs denoted by { ... }. As well, the IM?buffers do not require all the slots identified. Another observation is that the buffer chunk' slots are pulled from the buffer as, IM1buffer.chunk['subject'] and IMbuffer.chunk['predicate']. This indicates that a buffer's chunk is a list of slots.

Declarative Memory Inspection

Memory() stores chunks in a list called dm = []. DM is the declarative memory object for Memory(), and dm is a list in Memory() that contains the chunks that have been added to the declarative memory.

If the contents of the declarative memory are desired the declarative memory contents can be reported after the run command.

```
mod = MyModel()
mod.run()
print('\n\rReport the contents of the declarative memory')
```

```
for chunk in mod.DM.dm:
print(chunk)
ccm.finished()
```

Specific chunks can also be accessed by using the list index for the chunk in the declarative memory list.

```
print( DM.dm[0]) # prints the first chunk in the declarative memory list.
```

The following instructions will provide the indices of a chunks in the memory list. There are two approaches. Both will work.

```
i=0
while i < len(sophy.DM.dm):
  print('dm index:',i,
sophy.DM.dm[i] )
  i+=1
```

```
i=0
for chunk in mod.DM.dm:
  print('Index:, i, chunk)
  i+=1
```

Sub-symbolic Memory Parameters

The chunks that contain strings of information are considered the symbolic level of memory. There is a second dimension to memory chunks which determines whether or not chunks will be retrieved when requested. This dimension is referred to as the sub-symbolic level since it does not explicitly deal with the symbols in the chunks. Associated with each chunk stored in declarative memory is a numerical value referred to as its activation. The activation value must meet or exceed the memory threshold which determines the chunk's availability for retrieval. The higher the activation, the more likely a particular chunk will be retrieved.

For the activation and thresholds to be effective, they must be turned on using instructions in the cognitive in the models. If the subsymbolic processing is not turned on

then the activation defaults to 0.0, and the activation value has no effect on chunk processing. The subsymbolic processing for the chunks activation is calculated every time a chunk is stored or retrieved from memory, determining the chunk's activation each time it is accessed. The activation for a chunk is calculated in the subsymbolic processing system by,

$$A = B + \varepsilon,$$

where A is the chunks activation level, B is the base level activation and ε is the mental noise component.

As with human memories, memories fade over time. This is reflected in the Memory() module as memory decay. The more frequently a chunk is recalled, the higher the activation level, and hence more likely the chunk will be retrieved when requested.

Sub-symbolic properties

The subsymbolic memory attributes are:

threshold: this the level that a chunk's activation must meet or exceed before it can be retrieved. The default value is 0.0. threshold = 0.0

latency: this the time it takes for a chunk to be retrieved. It is the relationship between the activation and how long it takes to retrieve a chunk. The default latency factor is 50 milliseconds (latency=0.50). The time it takes to retrieve a requested chunk is,

$$T = Fe^{-A}$$

where T is the time it takes to retrieve the chunk from declarative memory, F is the chunks latency factor, and A is the activation level for the chunk.

If T exceeds the maximum_time for retrieval, then T is equal to maximum_time.

maximum_time: this is the maximum amount for a retrieval can take from the start of the request for a chunk.

finst_size: Finst is derived from FINST, or finger instantiation. This allows for chunks to be recalled that have not been recalled for period time. The value for the finst_size controls

how many chunks have been recently called and hold the attention of the cognitive model. The default is 4, finst_size=4

finst_time: There is a limit to the amount of time a chunk can be attended to, which decays over time. The finst time for a chunk is refreshed each time it is retrieved. The default is 3.0 minutes. When the decay reaches zero, the chunk is dropped from the finst system. The default is 3 minutes: finst_time = 3.0

The instruction to create a memory module in the model using default parameters,

```
dmbuffer = Buffer()
DM = Memory(dmbuffer,
            latency = 0.05,
            threshold = 0,
            maximum_time = 10.0,
            finst_size = 4,
            finst_time = 3.0)
```

All these parameters are accessible during a production run. However, they will only change if the modeller provides instructions to do so.

For example, if the modeller wanted to increase number of chunks from 4 to 10 for the finst_size then the following instructions could be placed in a production rule,

```
DM.finst.size = 10
```

Memory Parameter Inspection

The final state of the memory parameters can be reported at the end of the production run as follows,

```
model.run()
print('\n\rReport on Memory Attributes and contents\n\r')
print(f'Latency:',float(model.DM.latency))
```

```
print(f'Threshold:', int(model.DM.threshold))
print(f'Maximum Time:', int(model.DM.maximum_time))
print(f'FINST size', int(model.DM.finst.size), 'FINST time:',
int(model.DM.finst.time))
```

Declarative Memory Subsymbolic Components

For the activation levels for each chunk to change, the sub-symbolic processing components need to be turned on. When these are activated, they provide a context in which the memory operates, a set of circumstances that affect the availability and ability for memory chunks to be retrieved.

If no subsymbolic components are turned on, then the activation of all chunks is always zero and components like noise or time of past requests do not affect memory performance. This a full list of the components included in the declarative memory module included in the CCM ccm.actr package. Each component is presented in sufficient detail so that if the modeller wishes to use the component, they understand the impact it has on their model's memory.

Each component calculates its specific effect on a memory chunk, and the results are tracked and updated internally by the relevant component. These values are accessed by Memory when chunks are requested, which is affected by the values in each subsymbolic component. The subsymbolic components included in the declarative memory module are:

1. Base Level Activation
2. Noise
3. FINST
4. Spreading Activation
5. Inhibition
6. Salience

7. Spacing
8. Fixed Activation
9. Partial Activation
10. Associate
11. Blending Memory

Base Level Activation

The base level component records the time when and how often a memory chunk is accessed. The activation value reduces (decays) due the time between each request controlled by the decay value (normally set to the default of 0.50), and the limit for the component. To turn on the base level component, the following is included in the model.

dm_bl = DMBaseLevel(DM, decay=0.50, limit=None)

The default parameters are:

DM, the instance of the declarative memory

decay = 0.50

limit = None

Limit is used to optimize the behaviour of the base level activation. Ideally, each time a chunk is accessed, the time should be recorded. In small entry level models, this is not an issue, but in larger models the calculations can be come burdensome. To ameliorate this, an optimization algorithm is implemented, which is tied to the limit of the number of time are recorded. If the value is zero, then no times are recorded and the fully optimized equation is used. If the value is None, then all times are stored. If the value is a positive integer, then the number of access times is limited to the value of the integer.

$$B = \ln\left(\frac{n}{1-d}\right) - d * ln(L)$$

Noise Level

Noise can be introduce to cloud the ability of memory to retrieve a chunk. This performs the function of providing distracting thoughts that may interrupt chunk retrieval. To turn on noise for memory include this instruction inside the model.

dm_n = DMNoise(DM, noise=0.0, baseNoise=0.0)

DM, the instance of the declarative memory

noise = 0.0, Noise level set by the modeller, default is 0.0.

baseNoise = 0.0, the base noise for the model. Set by the modeller. Default is 0.0.

FINST

The FINST settings are included in the declarative memory. The FINST holds the number of chunks that are being attended, remaining within the models attention allowing them to be recalled, while the time determines how long they can be held. The default values are

finst_size = 4

finst_time = 3.0

The default finst size is 4 chunks, while the finst time's defaut is 3 minutes.

If for any reason the modeller wishes to change these values, the following instructions can be included in a production rule.

DM.finst.size = 10

This changes the finsts size to finst_size = 10

DM.finst.time = 6.0

This changes the finst time to finst_time = 6.0

A FINST is an attentional marker and is derived from Finger INSTatiation, first developed for visual attentional markers from the work of Zeno Pylyshyn.

Spreading Activation

Spreading activation is used when the modeller wants a buffered chunk to influence other chunks in declarative memory. The concept is that if a buffered chunk has slot values that occur in other chunks in memory, then by turning on spreading activation, the activation values in the buffered chunk influence the memory chunks with corresponding slots.

To turn on spreading activation the following code is added to the model,

dm_spread=DMSpreading(DM,focus)

This turns on spreading activation for DM (Memory). The spreading activation source is the focus buffer. If a chunk in the focus buffer contains slot contents that are used in other chunks in memory, then the activation levels in the memory chunks get a boost.

dm_spread.strength=1.5

The strength of activation for the buffer's slots affecting related chunks. Default = 1.0

dm_spread.weight[focus]=0.5

Sets the weight to adjust for how many slots in the buffer are affected. This is usually the strength divided by number of slots

The activation value, A_j , not only includes the base level activation, B_j , and noise, ε, but also the spreading activation component.

Spreading Activation = $\sum_j W_j S_{IJ}$

W represents the amount of activation from the source element , j, in the goal buffer

S is the strength between the elements in the goal buffer, j, an the slots i of the chunks in memory.

When this put together with the activation calculation noted earlier, the equation becomes.

$$A_j = B_j + \sum_j W_j \; S_{ij} + \epsilon$$

Spreading activation can be understood as the activation of concepts in memory from an association with the concept that is top of mind. These associations of ideas, as they are activated is how spreading activation works. For example, if one has the concept of a red ball, this can trigger thoughts of beach balls, or red bowling balls etc.

Partial Activation

Partial matching allows for a matching process whereby chunk can be retrieved that is not an exact match to the request specification. This subsymbolic process works in conjunction with spreading activation since the spreading activation increases the affected chunk's activation levels.

The following instructions are used to activate partial matching, and set that parameters to what slots are affected by the partial matching.

partial=Partial(DM, strength=1.0, limit=-1.0)

The next instructions identify the affected slots and the strength of the partial matching between them. In this case, we are matching different customers, and determining the strength of matching.

partial.similarity('customer1','customer2',-0.1)

This sets the similarity between customer1 and customer2. The third parameter sets the value of similarity. The higher the number, the more similar that slots.

More than one partial matching similarity can be employed in the model. In this case, changing the similarity to be less

partial.similarity('customer1','customer3',-0.9)

The calculation for partial matching is as follows,

$$\sum_{l} P\ M_{li}$$

Where,

P is the weighting given to slot l

M_{li} is the similarity between the value l in the retrieval request and the corresponding slot of chunk i.

The full activation equation with the addition of partial matching becomes,

$$A_j = B_j + \sum_j W_j \, S_{ij} + \sum_l PM_{li} + \epsilon$$

This equation is probably the most common level of usage for subsymbolic processing as it takes into account base level activation, spreading activation as well as partial matching.

Inhibition

The inhibition method prevents chunks from being retrieved from declarative memory.

dm_I=DMInhibition(DM, decayScale=5.0, timeScale=0.05)

The decayScale value should be close to 0.0 or negative, otherwise it can prevent chunk retrieval from memory.

The timeScale value cannot be negative or it throws a divide by zero error.

Salience

Salience is a function that originates from the vision module. If an object in the environment has a relatively higher salience value it increases the probability that the vision module will notice it and put into its vision buffer. This is as a result of the object being unique and therefore noticeable.

Similarly for a chunk in declarative memory, a salience can be associated with a memory chunk, which makes it more noticeable if there is more than one chunk vying for retrieval.

To turn on salience, the following instruction is used,

dm_sal = DMSalience(DM)

The salience value will then be included in the calculation for the activation level for each chunk when it is requested.

A more blunt way to impact the salience of a chunk is to include a salience value when the chunk is add to declarative memory as follows,

DM.add('isa:person name:John occupation:barrista', salience=1.3)

The higher the salience value the higher the activation for the chunk becomes.

Spacing

Not sure the purpose of this subsymbolic function. It is invoke with the following,

DMSpace = DMSpacing(DM, decayScale=0.0,decayIntercept=0.5)

Fixed Activation

Not clear what the purpose of this subsymbolic function. It is invoked with,

fixed = DMFixed(DM, default = 0)

Association

Not clear what the purpose of this subsymbolic function. It is invoked by.

assoc = DMAssociate(DM, DMBuffer, weight=1, decay=0.5, limit=None)

Blending Memory

Not clear on the purpose of this subsymbolic function. It is involked with,

Blend = BlendingMemory(DM)

Implementing Subsymbolic Activation

```
retrieval=Buffer()
memory=Memory(retrieval,latency=0.05,threshold=0,maximum_time=10.0,finst_size
=0,finst_time=3.0)
```

If no sub-modules are used, then the activation level of all chunks is always zero. The following systems adjust the activation in various ways. They are shown along with the default values for their parameters.

```
 # standard random noise
 dm_n=DMNoise(memory,noise=0.3,baseNoise=0.0)
  # the standard base level (bl) learning system
 #  if limit is set to a number, then the hybrid optimized equation
 #  from (Petrov, 2006) is used.
 dm_bl=DMBaseLevel(memory,decay=0.5,limit=None)
 # the spacing effect system from (Pavlik and Anderson, 2005)
 dm_space=DMSpacing(memory,decayScale=0.0,decayIntercept=0.5)
  # the standard fan-effect spreading activation system
 dm_spread=DMSpreading(memory,goal)  # specify the buffer(s) to spread from
   # other parameters are configured like this:
   #   dm_spread.strength=1
   #   dm_spread.weight[goal]=0.3
```

Python ACT-R Vision Module

General Description

The vision module is the only built in module that provides an input from sources external to the cognitive model. It is used to provide the cognitive model with data and information from the environment. This is the sense function of the cognitive sandwich: sense → think → act. While the sense function normally references the five senses (sight, hearing, smell, taste, touch) the only sense available for the CCM ACT-R is specifically identified as vision. However, this module can be used for the other senses as well such as hearing.

The module, as it is employed in CCM ACT-R, can only access fully formed objects (chunks) in the environment module. The vision module is not designed to formulate concepts or ideas from raw input vision data. So, the only source of data for the vision module is the environment module.

There are two vision modules in CCM ACT-R: the sosvision.py, and vision.py. The module most often employed at the novice level is the sosvision.py module (SOS is an acronym for Simple Operation System).

The SOSVision Module

The SOSVision module is found is found is the actr directory (ccm\lib\actr) as sosvision.py. Similar to the declarative memory, the vision module requires a buffer in which to transfer data from the vision module to the production module. This is accomplished with the following instruction,

```
vision_buffer = Buffer()
```

The SOSVision module is declared within the Production module using the following syntax,

```
vision = SOSVision(vision_buffer,
                   delay = 0.0,
```

delay_sd=None)

The vision instantiation requires the vision buffer to be identified. The delay value is the time that can be set for the vision module to acquire the searched object. The default is 0.0. The third parameter is delay_sd (the delay standard deviation) which can provide range around the delay value that can occur. This is calculated randomly but does not exceed the value of the standard deviation. This generally is not used and therefore set to None (delay_sd=None).

Searching the Environment

Before the vision module can return a chunk from the environment, the environment must first contain chunks the vision module can find.

The instruction to search for an object that meets a specific pattern is,

vision.request(chunk)

For example, to look for a red striped sock the instruction would be,

vision.request('isa:sock colour:red feature:stripe')

For this sock to be found, the environment module must contain a red striped sock.

class Sock_Env(ccm.Model):

sock1 = ccm.Model(isa='sock', colour='red', feature='stripe')

sock2 = ccm.Model(isa='sock', colour='blue', feature='stripe')

Salience

A salience value can be added to an environmental chunk, so that it stands out in the environment. This ensures the vision module will not miss noticing it.

Python ACT-R Motor Module

General Overview

The motor module is a Python class that provides the ACT-R agent with the ability to make changes to the environment. The motor module class functions are written to perform changes to chunks of an environment object. The function within the motor module can change the value of one of the chunks in an environment object.

The motor module is a class that is declared along with the environment and production modules. When declared, it can be invoked within the production (agent) module.

General Format of the Motor Module

The syntax for the motor module class is,

```
class MotorModule(ccm.Model):
        def do_something(self, <var>)
                yield <int>
                self.parent.parent.chunk_name.chunk_item='<new_value>'
```

The motor module inherits from the ccm.Model, model.py,

The def do_something(self, <var>) statement. This motor method is the instruction to change an environment chunk. The <var> is optional, but a variable can be passed to the motor function via the motor module buffer. The <var> is optional and only required if a <var> is required to modify a <new_item> variable.

yield <int> Yield refers to the time it takes for the motor method to complete its function. A typical value is 2 seconds (<yield 2>), but zero (<yield 0>) can also be used. However, the yield value can change depending on the estimate for the motor function to effect a change in the virtual environment.

self.parent.parent.chunk_name.chunk_item='<new_value>

- <self> refers to the instance of the motor module class

- the first parent refers to the instance of the production (agent) class

- the second parent refers to an instance of the environment module class

- the <chunk_name> is the name of the chunk object in the environment

- the <chunk_item> is one of the chunks for the <chunk_name>

- <new_value> refers to the new content for the environment's <chunk_item>. This is optional, and a constant value can be used.

The motor module is invoked in the production (agent) module. The code for this is is,

```
motorBuffer = Buffer()
motor = MotorModule(motorBuffer)
```

The motor module is called from within a production module's method. This takes the form,

```
motor.do_somthing(<var>)
```

The motor module is useful to the model's agent allowing the agent to make changes to the model's virtual environment. If the motor method is self-contained there is no requirement to pass a variable.

Example 1

In this example, the motor function changes bread location from on counter to a plate.

Beginning state for the environment variable:

```
bread=ccm.Model(isa='bread', location='on_counter')
```

The motor method that changes the environment object <bread>

```
def do_bread_bottom(self):
    yield 2
    print("done the bread")
    self.parent.parent.bread.location='on_plate'
```

This method prints a message. It also changes the value of the bread location from 'on_counter' to 'on_plate'. The result is that the environment object <bread> has been changed to,

bread=ccm.Model(isa='bread', location='on_plate')

In this case the motor module method is using a constant for changing the environment variable for the <bread>

Example 2

In this example a variable is passed from the production module to the motor module.

Beginning state for the environment variable:

```
kbd = ccm.Model(isa='KBD', location='desk', message=None)
```

In this case <message=None> which means this chunk does not contain a value.

The motor module method to change this chunk is,

```
def do_KBD(self, msg):
        yield 0
       print("do_KBD")
        self.parent.parent.kbd.message= msg
```

This method is expecting a parameter <msg>.

When the motor module method is called in a production module method, the parameter is passed from the production module to the motor module. The code for this looks like,

motor.do_KBD(msg)

The parameter is stored in a motor module buffer created in the production (agent) class when motor module attribute was assigned. The code for this is,

```
# Motor Module assignment
   motorBuffer = Buffer()
```

motor = MotorModule(motorBuffer)

The content of the <motorBuffer> can be anything that has been created in the production module. In this case the parameter is,

msg = 'Hello World'

When the <do_KBD> method is executed in the Motor Module, the chunk in the environment affected by the <do_KBD> method is changed. The resulting environment object is,

kbd = ccm.Model(isa='KBD', location='desk', message='Hello World')

Optional Code

While these methods are inheriting methods, etc from ccm.lib, it is still possible to include Python code within the method. A second environment object can be modified within the same method. For example, if there is a parameter in the motor buffer is a value other than <None>, then a second environmental object can be modified. The following code makes this change to the video screen in Example 2,

```
def do_KBD(self, msg):
    yield 0
    print("do_KBD")
    self.parent.parent.kbd.message= msg
    if msg != None:                          # If the kbd has information
       print("Update VDU")                    # then display it on the screen
       self.parent.parent.vdu.screen= msg
```

By adding the conditional statement,

```
    if msg != None:
       print("Update VDU")
       self.parent.parent.vdu.screen= msg
```

Here the video screen is also updated, providing that there is something entered into the keyboard.