

Unit 2: Perception and Motor Actions in ACT-R

2.1 ACT-R Interacting with Experimental Software

This unit will discuss how ACT-R can interact with experimental software. This is made possible with the addition of ACT-R/PM (the PM stands for perceptual-motor), which was developed by Mike Byrne. It is a set of modules, which act in parallel with the ACT-R production system for controlling vision, motor, audition, and speech. It also contains the code necessary to allow those modules to interact with the computer i.e. see what is on the screen, press keys, and move and click the mouse. It is quite elaborate and we will not be describing the entire system in the tutorial. For more complete information you can check out the ACT-R/PM web site at <http://chil.rice.edu/byrne/RPM/index.html>.

2.2 The First Experiment

The **demo2** model contains Lisp code to present a very simple experiment and a model that can perform the task. A single letter is presented to the participant and the participant is to press that key. Then, when a key is pressed the display is cleared. When you first open the model ACT-R is set to be the participant, but you can change it so that you are the participant. To do so, you need to change the line in the **command** window that looks like this:

```
(setf *actr-enabled-p* t)
```

to this:

```
(setf *actr-enabled-p* nil)
```

and press the **Reload** button. ***actr-enabled-p*** is a global variable that is used to set whether or not the model is to perform the task. Details of its use and a discussion of the code that runs the experiment, both with and without the model, are contained in the “Unit2 Experiment Code” unit. The units from now on will have two parts. The main unit text will discuss the ACT-R theory and the models that perform the task and an optional “Experiment Code” text will discuss the experiment Lisp code and the experiment generation tools built into ACT-R 5.0.

To run the experiment, you need to call the **do-experiment** function. To do that, type

```
(do-experiment)
```

in the listener window. A window will appear with a letter (the window may be obscured by the listener or other windows and you will have to arrange them to see everything you want). If it is set to run you as a participant, when you press a key (while the experiment window is the active window) the experiment window will clear and that is the end of the experiment. The letter you typed will be returned by the **do-experiment** function.

If you run the experiment with ***actr-enabled-p*** set to t (if you changed it to run yourself as a participant you need to change it back and reload) you can see ACT-R perform the same experiment. That will produce the following trace:

```

> (do-experiment)

Time 0.000: Vision found loc178
Time 0.000: Find-Unattended-Letter Selected
Time 0.050: Find-Unattended-Letter Fired
Time 0.050: Module :vision running command find-location
Time 0.050: Vision found loc178
Time 0.050: Attend-Letter Selected
Time 0.100: Attend-Letter Fired
Time 0.100: Module :vision running command move-attention
Time 0.185: Module :vision running command encoding-complete
Time 0.185: Vision sees text177
Time 0.185: Encode-Letter Selected
Time 0.235: Encode-Letter Fired
Time 0.235: Respond Selected
Time 0.285: Respond Fired
Time 0.285: Module :motor running command press-key
Time 0.535: Module :motor running command preparation-complete
Time 0.585: Module :motor running command initiation-complete
Time 0.685: Device running command output-key

<< Window "Letter recognition" got key #\v at time 685 >>

Time 0.770: Module :vision running command encoding-complete
Time 0.835: Module :motor running command finish-movement
Time 0.835: Checking for silent events.
Time 0.835: * Nothing to run: No productions, no events.
"v"

```

Here we see production firing being intermixed with various vision and motor actions as the system encodes the stimulus and issues a response. If you watch the window while the model is performing the task you will see a red circle. That is a debugging aid and indicates the vision module's current location of attention. It can be turned off if you do not want to see it, and how to do that will be discussed in the parameters section below.

In the following sections we will look at how to have ACT-R "read" the screen being presented, how to have it issue responses, and briefly discuss parameters in ACT-R.

2.3 The Visual Interface

Much of the interaction with the software involves interacting with the visual system. ACT-R communicates with the vision module through three buffers. There is a **visual** buffer that can hold a chunk that represents an object in the visual scene, a **visual-location** buffer that holds a chunk which represents the location of an object in the visual scene, and a **visual-state** buffer which holds a chunk that represents the internal state of the vision module. All three are illustrated in the two productions **find-unattended-letter** and **attend-letter**. The first production applies whenever the goal is in the start state:

```

(P find-unattended-letter
  =goal>
  ISA      read-letters
  state    start
  ==>

```

```

+visual-location>
  ISA      visual-location
  attended nil
=goal>
  state    find-location
)

```

It requests that a new visual location be placed in the **visual-location** buffer, and it changes the goal state to find-location. In this action, we are requesting the location of an object on the screen that has not yet been looked at by ACT-R (specified by attended nil). The following portion of the trace reflects the operation of this production:

```

Time 0.000: Find-Unattended-Letter Selected
Time 0.050: Find-Unattended-Letter Fired
Time 0.050: Module :vision running command find-location
Time 0.050: Vision found loc178

```

Ignore the first line of the trace that looks like this:

```
Time 0.000: Vision found loc178
```

for now. That is the result of a mechanism which we will not discuss until the next unit.

The **+visual-location** request results in the vision module performing a find-location command to find the location of an object on the screen that meets the requirements, build a chunk to represent the location of that object if one exists, and place that chunk in the **visual-location** buffer.

You can inspect the contents of the vision module's buffers by using the **Buffer viewer** button on the **Control Panel**. Of importance here is the **visual-location** buffer. If you select it in the list on the left, the display on the right will show the name of the chunk in the **visual-location** buffer and the details of that chunk. In the current experiment, if you inspect the chunk in the **visual-location** buffer after the first production fires it will look like the following (most likely with a different name):

```

Loc178
  isa VISUAL-LOCATION
  screen-x 130
  screen-y 160
  attended New

```

There will be a few other slots listed as well, but those are not important for now, and can be ignored. The ones of interest are **screen-x**, **screen-y**, and **attended**. The exact coordinates of the object in the window are encoded in the **screen-x** and **screen-y** slots. The upper-left corner of the window is **screen-x** 0 and **screen-y** 0. The x coordinates increase from left to right, and the y coordinates increase from top to bottom. In general, the specific values are not that important for the model, and do not need to be specified when making a request for a location. There is a set of descriptive specifiers that can be used for requests on those slots, like lowest or highest, but again those details will not be discussed until unit 3. The other important slot is **attended**. It encodes whether ACT-R has looked at (or attended to) the object found at that location. If it is **nil**, then ACT-R has not looked at it, and if it is **t**, then ACT-R has looked at it. If you notice above

however, it has the value **new**. This means that not only has ACT-R not looked at it, but the object there has been added to the screen recently. If you request a visual location that is attended **nil**, as was done in the find-unattended-location production, the chunk that is placed in the **visual-location** buffer may have a value of either **new** or **nil** in the attended slot. However, if you request a location with attended **new**, the chunk placed in the **visual-location** buffer (if there is a location that matches that request) will always have the value **new** in the attended slot. That does not usually cause a problem however because you typically do not need to test the value of the attended slot of the **visual-location** buffer. You almost always know what it will be because it was specified in the request that was made in a previous production.

The next production applies when the goal state is find-location, there is a visual-location in the **visual-location** buffer, and the vision module is not currently active:

```
(P attend-letter
  =goal>
    ISA      read-letters
    state    find-location
  =visual-location>
    ISA      visual-location
  =visual-state>
    ISA      module-state
    modality free
==>
  +visual>
    ISA      visual-object
    screen-pos =visual-location
  =goal>
    state    attend
)
```

It requests a visual object be placed in the **visual** buffer and changes the goal state to attend.

On the LHS of this production are two buffer tests that have not been seen before. The first is a test of the **visual-location** buffer. Notice that the only test on the buffer is the **isa** slot. All that is necessary is to make sure that there is a chunk of type **visual-location** in the buffer. The details of its slot values do not matter. Then, we test the **visual-state** buffer. This buffer always holds a chunk of type **module-state**, which describes whether or not the vision module is currently available. In general, each of the modules (vision, audition, motor, and speech) has a state buffer associated with it that holds a **module-state** chunk. The **module-state** chunk-type has several slots that indicate which portions of a module are currently in use. The possible values for those slots are **busy** or **free**. We will most often be concerned whether or not any part of the module is currently in use, and the **modality** slot tests that – if any part of the module is **busy** then modality is **busy** and if all of the module’s components are **free** then **modality** is **free**. We will discuss why we need to do this test after we describe the RHS request.

The RHS of this production has one action that we have not seen before. It requests that the visual object whose **screen-pos**[ition] is the visual location in the **visual-location** buffer be placed into the **visual** buffer. This request causes the vision module to move its attention to the specified

location, encode the object that is there, and place that object into the **visual** buffer. The following portion of the trace reflects this operation:

```
Time 0.050: Attend-Letter Selected
Time 0.100: Attend-Letter Fired
Time 0.100: Module :vision running command move-attention
Time 0.185: Module :vision running command encoding-complete
Time 0.185: Vision sees text177
```

Note that the request to move-attention is made at time 0.100 seconds but that the encoding does not complete until 0.185 seconds. That 85 ms represents the time to shift attention and create the visual object. Altogether, counting the two production firings (one to request the location and one to request the attention shift) and the 85 ms it takes to execute the attention shift and object encoding it takes 185 ms to create the chunk that encodes the letter on the screen.

You can see what is in the **visual** buffer by using the **Buffer viewer**. In the **demo2** model, if you inspect the visual buffer after the encoding occurs you will see a chunk similar to the following:

```
Text177
  isa TEXT
  screen-pos Loc178
  value "h"
  status nil
  color Black
  height 10
  width 7
```

The chunk is of type **text** because it is a letter that was encoded from the screen. The **screen-pos** slot holds the location chunk for that object, which matches the chunk in the **visual-location** buffer as was specified in the request. The **value** slot holds a string that contains the text encoded from the screen, in this case a single letter. The **status** slot is empty, and is essentially a free slot which can be used by the model to encode additional information in that chunk. The **color**, **height**, and **width** slots hold information about the visual features of the item attended.

If you now inspect the chunk in the **visual-location** buffer you will see that the attended slot has changed to **t** to indicate that the object at this location has now been attended:

```
Loc178
  isa VISUAL-LOCATION
  screen-x 130
  screen-y 160
  attended t
```

Now, we will go back to the need for the **visual-state** test in this production. It is there because of the request to the vision module made on the RHS. The vision module is only able to execute one move-attention at a time. Since the vision module and the production system operate in parallel and it takes 85 ms to complete the move-attention it would be possible for a second production to request a move-attention before the first one completed. This is called “jamming” the module. When a module is jammed, it will output an error message in the trace to let you know what has happened. It is possible to write productions such that no requests will be made that can jam any of the modules and thus no tests of the module-state chunks would be necessary

(in fact the **demo2** model has this property). However, it is always a good idea to include those tests in every production that makes a request that could jam a module even if you know that it will not happen because of the structure of the other productions. That makes it clear to anyone else who may read the model and you may decide to later apply that model to a different task where that assumption no longer holds.

After a visual object has been placed in the **visual** buffer, it can be harvested by a production like this one:

```
(P encode-letter
  =goal>
    ISA      read-letters
    state    attend
  =visual>
    ISA      text
    value    =letter
==>
  =goal>
    letter   =letter
    state    respond
)
```

which stores the letter that was read in the **letter** slot of the goal and sets the **state** slot to respond.

This process of seeking the location of an object in one production, switching attention to the object in a second production, and harvesting the object in a third production is a common style in ACT-R models. One important thing to appreciate is that this is one way in which ACT-R can acquire new declarative chunks. Initially the chunks will be in the **visual** buffer, but they will remain around in declarative memory as a permanent chunk encoding what has been perceived in that location even after they have left the **visual** buffer. The visual location chunk will be changed the next time an object occurs in that location, but the visual object chunk will serve as a permanent encoding of the visual object and the location where it occurred.

There is another line in the trace of the model that shows the vision system doing something which needs to be addressed:

```
Time 0.770: Module :vision running command encoding-complete
```

At time .770 seconds there is an encoding that was not the result of a request made by a production. This is a result of the screen being cleared after the key press. When the screen is updated, if the vision module is currently attending to a location it will automatically re-encode that location to encode any changes that may have occurred there. In this case, the item at that location has been removed, and to represent that the **visual** buffer is cleared to indicate that there is no object there. This automatic re-encoding process of the visual system requires that you be careful when writing models that process changing displays for two reasons. The first is that you cannot be guaranteed that the chunk in the **visual** buffer will not change in response to a change in the visual display. The other is because while the re-encoding is occurring, the vision module is busy and cannot handle a new attention shift. So, you will have to make sure to test the **visual-state** buffer before all visual requests to avoid jamming the vision module.

2.4 The Motor Module

When we speak of motor actions in ACT-R/PM we are only concerned with hand movements. It is possible to extend the motor module to other modes of action, but the default mechanism is built around controlling a pair of hands. In this unit we will only be concerned with finger presses at a keyboard but the fingers can also be used to press other devices and the hand can be used to move a mouse or other device. Information about these features or extending the motor module is available in the ACT-R/PM manual.

Like the vision module, there are multiple buffers associated with the motor module. They are the **manual** buffer and the **manual-state** buffer. The **manual-state** buffer holds the **module-state** chunk for the motor module, and it works the same as the one described for the vision module – keeping track of which parts of the motor module are **busy** and which are **free**. The **manual** buffer is used to request actions be performed by the hands. As with the vision module, you should always check to make sure that the motor module is **free** before making any requests to avoid jamming it. There are multiple stages in the motor module, and it is possible to make a new request before the previous one has completed by testing the individual stages. However we will not be discussing that, and will only test on **modality** i.e. whether the entire module is **free** or **busy**. The **respond** production from the **demo2** model shows both buffers in use:

```
(P respond
  =goal>
    ISA      read-letters
    letter   =letter
    state    respond
  =manual-state>
    ISA      module-state
    modality free
==>
  +manual>
    ISA      press-key
    key      =letter
  =goal>
    state    stop
)
```

This production fires when the goal state is respond and the **manual-state** buffer indicates that the hands are not currently in action. Then a request is made to press the key corresponding to the letter from the **letter** slot of the goal and the state is changed to stop. The type of action requested of the hands is specified in the isa slot of the **manual** buffer request. The **press-key** request assumes that the hands are located over the home row and the fingers will be returned there after the key has been pressed. There are many other requests that can be made, and they are described in the ACT-R/PM manual. The trace of the actions that result from this production are shown here:

```
Time 0.235: Respond Selected
Time 0.285: Respond Fired
Time 0.285: Module :motor running command press-key
Time 0.535: Module :motor running command preparation-complete
Time 0.585: Module :motor running command initiation-complete
Time 0.685: Device running command output-key
Time 0.835: Module :motor running command finish-movement
```

When the production is fired a request is made to press the key. However, it takes 250 ms to prepare the features of the movement (preparation-complete), 50 ms to initiate the action (initiation-complete), another 100 ms for the key to be struck (output-key), and finally it takes another 150 ms for the finger to return to the home row (finish-movement). Thus the time of the key press is at .685 seconds, however the motor module is still busy until time .835 seconds. The **press-key** request obviously does not model the typing skills of an expert typist, but is a sufficient mechanism for many tasks.

2.5 ACT-R Parameters

ACT-R and ACT-R/PM both have many parameters that can be set. These parameters are used to modify the behavior of models, control the output that gets displayed in the trace, and to control general characteristics of how the system operates. These parameters are set using the functions **sgp** and **pm-set-params**. **Sgp** (set global parameters) is used to set the ACT-R parameters and **pm-set-params** is used to set the ACT-R/PM parameters. These will show up in the **command** window of the environment when the model is opened. As the tutorial progresses the units will discuss the parameters that relate to the material being covered. A complete list of the ACT-R parameters is available on the tutorials page of the ACT-R website (<http://act-r.psy.cmu.edu/tutorials/>) and the ACT-R/PM parameters can be found in the ACT-R/PM manual. The current values of the parameters can be seen by calling **sgp** without any parameters specified and by calling **pm-show-params** respectively.

In the **demo2** model the parameters are set like this:

```
(sgp :v t)
(pm-set-params :real-time t :show-focus t)
```

The **v** (verbose) parameter for ACT-R is a flag that controls whether or not the trace is printed when the model is run. If a flag parameter is set to **t**, then the option is enabled and if it is set to **nil** it is disabled. Thus, when **v** is set to **t**, as it is in the **demo2** model, the trace is printed. If it were set to **nil**, then the trace would not be displayed. Models run much faster when the trace is not printed, but when debugging a model it is very useful to be able to see the trace.

The real-time and show-focus parameters are also flags for the perceptual motor system. Real-time sets whether or not the model should run in real time (1 second for the model takes 1 second of real time) or in simulated time (the time to run 1 second of the model depends on the speed of the machine being used, but is almost always much faster than real time). The **demo2** model is set to run in real time. It is often useful to turn real-time on when debugging a model so that you can watch what it is doing because in simulated time things may happen too fast for you to see (like attention shifts or mouse movements). Show-focus controls whether or not the red visual attention ring is displayed in the experiment window when the model is performing the task. It is a useful debugging tool, but for some displays you may not want to see it.

2.6 Assignment 2

Your assignment is to extend the capacity of the model in **demo2** to do a more complex experiment. The new experiment presents three letters. Two of those letters will be the same. The participant's task is to indicate by a keystroke the one that is different from the other two. Everything except the productions (which you will have to write) is contained in the model **assignment2**. By default, that model has `*actr-enabled-p*` set to nil and which means that it will run you as the participant.

To run the experiment, call the function **do-experiment**. When you press a key the function will return correct if you pressed the right key and nil if you pressed the wrong key. This shows what happens when the right key was pressed:

```
> (do-experiment)
CORRECT
```

and this shows the result when the wrong key was pressed:

```
> (do-experiment)
NIL
```

Your task is to write a model that always responds correctly. In doing this you should take the model in **demo2** as a guide. It reflects the way to interact with the vision and motor modules and the productions it contains are similar to the productions you will need to write. You will also need to write additional productions to read the other letters and decide which key to press.

You are provided with a chunk-type for the goal, and an initial chunk in the goal buffer. This chunk-type is similar to the one used in the **demo2** model, but has additional slots for encoding the other letters:

```
(chunk-type read-letters letter1 letter2 letter3 state)
```

The initial goal provided looks just like the one used in **demo2**:

```
(goal isa read-letters state find)
```

These are what we used in our solution of the task, but you do not have to use them. If you have a different representation you would like to use feel free to do so. There is no one “right” model for the task. In later units we will consider fitting models to data from real experiments. Then, how well the model fits the data can be used as a way to decide between different representations and models, but that is not the only way to decide. Plausibility is another important factor when modeling human performance – you want the model to do the task like a person does the task. A model that fits the data perfectly using a method completely unlike a person is probably not a very good model of the task.

Remember, to have ACT-R perform the experiment rather than you, you will need to set `*actr-enabled-p*` to `t` in the **command** window like this:

```
(setf *actr-enabled-p* t)
```