Unit 2: Perception and Motor Actions in ACT-R

2.1 ACT-R Interacting with the World

This unit will introduce some of the mechanisms which allow ACT-R to interact with the world, which for the purposes of the tutorial will be experiments presented via the computer. This is made possible with the addition of perceptual and motor modules which were developed by Mike Byrne, and which were previously referred to as ACT-R/PM but are now an integrated part of the system. It is a set of modules for ACT-R which provides a model with visual, motor, auditory, and vocal capabilities as well as the mechanisms for interfacing those modules to the world. The default mechanisms which we will use allow the model to interact with the computer i.e. process the visual items presented, press keys and move and click the mouse. Other more advanced interfaces can be developed, but that is beyond the scope of the tutorial.

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. The experiment consists of a window in which a single letter is presented. The participant's task is to press that key. When a key is pressed, the display is cleared and the experiment ends.

After you load the model you can perform the task yourself if you are running the ACT-R Environment or you are using a Lisp with a GUI for which there is an existing ACT-R interface (currently Clozure Common Lisp for Macs, Allegro Common Lisp for Windows, and LispWorks). To run the experiment with a human participant instead of the ACT-R model you need to call the **do-demo2** function and pass it the symbol human. Thus you would enter this:

```
(do-demo2 'human)
```

at the Lisp prompt.

A window will appear with a letter (the window may be obscured by your editor or other windows so you may have to arrange things to ensure you can see everything you want). 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-demo2** function.

If you call the **do-demo2** function without including the symbol human then the ACT-R model will be run through the experiment instead of waiting for a person to do the experiment. That will produce the following trace:

```
0.000   GOAL          SET-BUFFER-CHUNK GOAL GOAL REQUESTED NIL
0.000   VISION        SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0 REQUESTED NIL
0.000   PROCEDURAL    CONFLICT-RESOLUTION
```

```
0.000   PROCEDURAL    PRODUCTION-SELECTED FIND-UNATTENDED-LETTER
0.000   PROCEDURAL    BUFFER-READ-ACTION GOAL
0.050   PROCEDURAL    PRODUCTION-FIRED FIND-UNATTENDED-LETTER
0.050   PROCEDURAL    MOD-BUFFER-CHUNK GOAL
0.050   PROCEDURAL    MODULE-REQUEST VISUAL-LOCATION
0.050   PROCEDURAL    CLEAR-BUFFER VISUAL-LOCATION
0.050   VISION        Find-location
0.050   VISION        SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0
0.050   PROCEDURAL    CONFLICT-RESOLUTION
0.050   PROCEDURAL    PRODUCTION-SELECTED ATTEND-LETTER
0.050   PROCEDURAL    BUFFER-READ-ACTION GOAL
0.050   PROCEDURAL    BUFFER-READ-ACTION VISUAL-LOCATION
0.050   PROCEDURAL    QUERY-BUFFER-ACTION VISUAL
0.100   PROCEDURAL    PRODUCTION-FIRED ATTEND-LETTER
0.100   PROCEDURAL    MOD-BUFFER-CHUNK GOAL
0.100   PROCEDURAL    MODULE-REQUEST VISUAL
0.100   PROCEDURAL    CLEAR-BUFFER VISUAL-LOCATION
0.100   PROCEDURAL    CLEAR-BUFFER VISUAL
0.100   VISION        Move-attention VISUAL-LOCATION0-0-1 NIL
0.100   PROCEDURAL    CONFLICT-RESOLUTION
0.185   VISION        Encoding-complete VISUAL-LOCATION0-0-1 NIL
0.185   VISION        SET-BUFFER-CHUNK VISUAL TEXT0
0.185   PROCEDURAL    CONFLICT-RESOLUTION
0.185   PROCEDURAL    PRODUCTION-SELECTED ENCODE-LETTER
0.185   PROCEDURAL    BUFFER-READ-ACTION GOAL
0.185   PROCEDURAL    BUFFER-READ-ACTION VISUAL
0.235   PROCEDURAL    PRODUCTION-FIRED ENCODE-LETTER
0.235   PROCEDURAL    MOD-BUFFER-CHUNK GOAL
0.235   PROCEDURAL    MODULE-REQUEST IMAGINAL
0.235   PROCEDURAL    CLEAR-BUFFER VISUAL
0.235   PROCEDURAL    CLEAR-BUFFER IMAGINAL
0.235   PROCEDURAL    CONFLICT-RESOLUTION
0.435   IMAGINAL      CREATE-NEW-BUFFER-CHUNK IMAGINAL ISA ARRAY
0.435   IMAGINAL      SET-BUFFER-CHUNK IMAGINAL ARRAY0
0.435   PROCEDURAL    CONFLICT-RESOLUTION
0.435   PROCEDURAL    PRODUCTION-SELECTED RESPOND
0.435   PROCEDURAL    BUFFER-READ-ACTION GOAL
0.435   PROCEDURAL    BUFFER-READ-ACTION IMAGINAL
0.435   PROCEDURAL    QUERY-BUFFER-ACTION MANUAL
0.485   PROCEDURAL    PRODUCTION-FIRED RESPOND
0.485   PROCEDURAL    MOD-BUFFER-CHUNK GOAL
0.485   PROCEDURAL    MODULE-REQUEST MANUAL
0.485   PROCEDURAL    CLEAR-BUFFER IMAGINAL
0.485   PROCEDURAL    CLEAR-BUFFER MANUAL
0.485   MOTOR         PRESS-KEY v
0.485   PROCEDURAL    CONFLICT-RESOLUTION
0.735   MOTOR         PREPARATION-COMPLETE
0.735   PROCEDURAL    CONFLICT-RESOLUTION
0.785   MOTOR         INITIATION-COMPLETE
0.785   PROCEDURAL    CONFLICT-RESOLUTION
0.885   MOTOR         OUTPUT-KEY #(4 5)
0.885   PROCEDURAL    CONFLICT-RESOLUTION
0.970   VISION        Encoding-complete VISUAL-LOCATION0-0-1 NIL
0.970   VISION        No visual-object found
0.970   PROCEDURAL    CONFLICT-RESOLUTION
1.035   MOTOR         FINISH-MOVEMENT
1.035   PROCEDURAL    CONFLICT-RESOLUTION
1.035   ------        Stopped because no events left to process
```

Here we see production firing being intermixed with actions of the vision, imaginal, and motor modules as the model encodes the stimulus and issues a response. If you watch the window while

the model is performing the task you will also see a red circle drawn. That is a debugging aid which indicates the model's current point of visual attention. It can be turned off if you do not want to see it. How that is done will be discussed in the parameters section below. You may also notice that the task always presents the letter "V". That is also due to a parameter setting in the model and is done so that it always generates the same trace. You can also change that if you would like to see how the model performs the task for different letters, and that will also be described below.

In the following sections we will look at how the model perceives the letter being presented, how it issues a response, and briefly discuss some parameters in ACT-R.

One thing to note is that from this point on in the tutorial all of the models will be interacting with an experiment of some form. Thus you will always have to call the appropriate function from the Lisp prompt to run the experiment. That experiment function will run the model as needed. So, from this point on in the tutorial you will typically not be using the run command directly to run the models as was done in unit 1.

2.3 Control and Representation

Before looking at the details of the new buffers and modules, however, there is something different about this model relative to the models that were used in unit 1 which needs to be addressed. There are two chunk types created for this model:

```
(chunk-type read-letters state)
(chunk-type array letter)
```

The chunk type read-letters has one slot which is called state and will be used to track the current task state for the model. The other chunk type, array, also has only one slot, which is called letter, and will hold a representation of the letter which is seen by the model.

In unit 1, we saw that the chunk placed into the **goal** buffer had slots which held all of the information relevant to the task – one buffer held all of the information. That represents how things have typically been done with ACT-R models in the past, but with ACT-R 6.0, a more distributed representation of the model's "state" is the preferred means of modeling. Now, we will use two buffers to hold the information. The **goal** buffer will be used to hold control state information – the internal representation of what the model is doing and where it is in the task. In this model the **goal** buffer will hold chunks of type read-letters. A different buffer, the **imaginal** buffer, will hold the chunk which contains the problem state information, and in this model that will be a chunk of type array.

2.3.1 The State Slot

In this model, the state slot of the chunk in the **goal** buffer will maintain information about what the model is doing. It can then be used to explicitly indicate which productions are appropriate at any time. This is often done when writing ACT-R models because it is easy to specify and makes

them easier to follow. It is however not always necessary to do so, and there are other means by which the same control flow can be accomplished. In fact, as we will see in a later unit there are consequences to keeping extra information in the goal chunk. However, because it does make the production sequencing in a model clearer you will see state slots in many of the models in the tutorial even if they are not always necessary. As an additional challenge for this unit, you can try to modify the **demo2** model so that it works without needing to maintain an explicit state and thus not even use the **goal** buffer at all.

2.4 The Imaginal Module

The first new module we will describe in this unit is the imaginal module. This module has a buffer called **imaginal** which is used to create new chunks. These chunks will be the model's internal representation of information – its internal image (thus the name imaginal module). Like any buffer, the chunk in the **imaginal** buffer can be modified by the productions to build that representation using RHS modification actions as shown in unit 1.

The important thing about the **imaginal** buffer is how the chunk first gets into the buffer. Unlike the **goal** buffer's chunk which we have been creating and placing there in advance of the model starting, the imaginal module will create the chunk for the **imaginal** buffer in response to a request from a production.

All requests to the imaginal module through the **imaginal** buffer are requests to create a new chunk. The imaginal module will create a new chunk using the chunk-type and any initial slot values provided in the request and place that chunk into the **imaginal** buffer. An example of this is shown in the encode-letter production:

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

We will explain the details of how the text chunk gets into the **visual** buffer in the next section. For now, we are interested in this request on the RHS:

```
+imaginal>
    isa          array
    letter       =letter
```

This request of the **imaginal** buffer is asking the imaginal module to create a chunk of type array and which has the value of the variable =letter in its letter slot. We see the request and its results in these lines of the trace:

```
0.235   PROCEDURAL      PRODUCTION-FIRED ENCODE-LETTER
…
0.235   PROCEDURAL      MODULE-REQUEST IMAGINAL
…
0.235   PROCEDURAL      CLEAR-BUFFER IMAGINAL
…
0.435   IMAGINAL        CREATE-NEW-BUFFER-CHUNK IMAGINAL ISA ARRAY
0.435   IMAGINAL        SET-BUFFER-CHUNK IMAGINAL ARRAY0
```

The production makes the request and automatically clears the buffer at that time as happens for all buffer requests. Then, we see that the imaginal module reports that it is creating a new chunk and that chunk is then placed into the buffer.

An important detail of the request to the imaginal module is that the chunk is not immediately placed into the buffer as a result of the request. It took .2 seconds before the chunk was made available. This is an important aspect of the imaginal module – it takes time to build a representation. The amount of time that it takes the imaginal module to create a chunk is a fixed cost, and the default time is .2 seconds (though that can be changed with a parameter). In addition to the time cost, the imaginal module is only able to create one new chunk at a time. That does not impact this model because it is only creating the one new chunk in the **imaginal** buffer, but there are times where that can matter. In such situations one may want to verify that the module is available to create a new chunk and how one does that is described later in the unit.

Thus in this model, the **imaginal** buffer will hold a chunk which contains a representation of the letter which the model reads from the screen. For this simple task, that representation is not strictly necessary because the model could use the information directly from the vision module to do the task, but for most tasks there will be more information which must be maintained thus requiring such a chunk to be created. In particular, for this unit's assignment the model will need to read multiple letters which must be considered before responding.

2.5 The Vision Module

Many tasks involve interacting with visible stimuli and the vision module provides a model with a means for acquiring visual information. It is designed as a system for modeling visual attention.

It assumes that there are lower-level perceptual processes that generate the representations with which it operates, but it does not model those perceptual processes in detail. It includes some default mechanisms for parsing text and other simple visual features from a window and provides an interface that one can use to extend it when necessary.

The vision module has two buffers. There is a **visual** buffer that can hold a chunk that represents an object in the visual scene and a **visual-location** buffer that holds a chunk which represents the location of an object in the visual scene. As with all modules, it also responds to queries of the buffers about the state of the module. It can also respond to more detailed queries which will not be covered in this unit. Visual interaction is shown in the demo2 model in the two productions **find-unattended-letter** and **attend-letter**.

2.5.1 Visual-Location buffer

The **find-unattended-letter** production applies whenever the **goal** buffer's chunk has a state of start (which is how the chunk is initially created):

```
(P find-unattended-letter
   =goal>
      ISA         read-letters
      state       start
 ==>
   +visual-location>
      ISA         visual-location
      :attended   nil
   =goal>
      state       find-location
)
```

It makes a request of the **visual-location** buffer and it changes the goal state to find-location. The following portion of the trace reflects the actions of this production:

```
0.050   PROCEDURAL    PRODUCTION-FIRED FIND-UNATTENDED-LETTER
0.050   PROCEDURAL    MOD-BUFFER-CHUNK GOAL
0.050   PROCEDURAL    MODULE-REQUEST VISUAL-LOCATION
0.050   PROCEDURAL    CLEAR-BUFFER VISUAL-LOCATION
0.050   VISION        Find-location
0.050   VISION        SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0
```

You should ignore the earlier line of the trace related to the vision module that looks like this:

```
0.000   VISION        SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0 REQUESTED NIL
```

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

The **visual-location** request asks the vision module to find the location of an object in its visual scene (which for this model is the current experiment's window) that meets the specified

requirements, build a chunk to represent the location of that object if one exists, and place that chunk in the **visual-location** buffer.

Looking at the trace, these events are a result of that request:

```
0.050   PROCEDURAL      MODULE-REQUEST VISUAL-LOCATION
0.050   PROCEDURAL      CLEAR-BUFFER VISUAL-LOCATION
0.050   VISION          Find-location
0.050   VISION          SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0
```

We see the notice of the request and then the automatic clearing of the buffer due to the request being made by the procedural module. Then the vision module reports that it is finding a location and then it places that chunk into the buffer. Notice that there was no time involved in handling the request – all those actions took place at time 0.050 seconds. The visual-location requests always finish immediately which reflects the idea that there is a perceptual system operating in parallel that makes these visual features immediately available.

If you step through the model using the Stepper you can use the "Buffer viewer" to see that the chunk **visual-location0-0-1** will be in the **visual-location** buffer after that last event:

```
VISUAL-LOCATION0-0-1
  ISA VISUAL-LOCATION
   SCREEN-X  130
   SCREEN-Y  160
   DISTANCE  15.0
   KIND  TEXT
   COLOR  BLACK
   VALUE  TEXT
   HEIGHT  10
   WIDTH  7
   SIZE  0.19999999
```

There are a lot of slots in a visual-location, but most are not important for this unit, and can be ignored. The first two, **screen-x** and **screen-y**, are the only ones we are concerned with right now. They encode the exact coordinates of the object in the visual scene. 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.

2.5.1.1 The attended request parameter

If we look at the request which was made of the **visual-location** buffer in the **find-unattended-letter** production:

```
 +visual-location>
```

```
      ISA          visual-location
      :attended    nil
```

we see that in addition to specifying "isa visual-location" it includes ":attended nil" in the request. However, looking at the chunk-type for a visual-location we find that it does not have a slot called attended or :attended (calling **chunk-type** with no parameters will print out all currently defined chunk types):

```
VISUAL-LOCATION
   SCREEN-X
   SCREEN-Y
   DISTANCE
   KIND
   COLOR
   VALUE
   HEIGHT
   WIDTH
   SIZE
```

This :attended specification is called a request parameter. It acts like a slot in the request, but does not correspond to a slot in the chunk-type specified. A request parameter is valid for any request to a buffer regardless of the chunk-type specified. Request parameters are used to supply general information to the module about a request which may not be desirable to have in the resulting chunk that is placed into the buffer. A request parameter is specific to the particular buffer and will always start with a ":" which distinguishes it from an actual slot of the chunk-type. We will discuss a couple different request parameters in this unit and later units as we introduce more buffers.

For a visual-location request one can use the :**attended** request parameter to specify whether the vision module returns the location of an object which the model has previously looked at (attended to) or not. If it is specified as **nil**, then the request is for a location which the model has not attended, and if it is specified as **t**, then the request is for a location which has been attended previously. There is also a third option, **new**. This means that not only has the model not attended to the location, but also that the object has recently appeared in the visual scene.

The **attend-letter** production applies when the goal state is find-location, there is a visual-location chunk 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        free

==>
```

```
   +visual>
      ISA         move-attention
      screen-pos  =visual-location
   =goal>
      state       attend
)
```

On the LHS of this production are two conditions 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, a query is made of the **visual** buffer.

2.5.2 Checking a module's state

On the LHS of **attend-letter** a query is made of the **visual** buffer to test that the **state** of the vision module is **free**. All buffers will respond to a query for the module's **state** and the possible values for that query are **busy, free,** or **error** as was shown in unit 1. The test of **state free** is a check to make sure the buffer being queried is available for a new request. If the **state** is **free**, then it is safe to issue a new request, but if it is **busy** then it is usually not safe to do so.

Typically, a module is only able to handle one request to a buffer at a time. This is the case for both the **imaginal** and **visual** buffers which require some time to produce a result. Since all modules operate in parallel it might be possible for the procedural module to select a new production which makes a new request to a module that is still working on a previous request. If a production were to fire at such a point and issue another request to a module which is busy and only able to handle one request at a time, that is referred to as "jamming" the module. When a module is jammed, it will output a warning message in the trace to let you know what has happened. What a module does when jammed varies from module to module. Some modules ignore the new request, whereas others abandon the previous request and start the new one. As a general practice it is best to avoid jamming modules. Thus, when there is the possibility of jamming a module one should be sure to query its state before making a request.

Note that we did not query the state of the **visual-location** buffer in the **find-unattended-letter** production before issuing the **visual-location** request because we know that those requests always complete immediately and thus the **visual-location state** is always **free**. We also did not test the state of the imaginal module before making the request to the **imaginal** buffer in the **encode-letter** production. In that case, a carefully written model would check such a situation, but because this model only makes one such request we omitted the check because we knew that the state would be free at that time. However, that is a risky practice, and it is always a good idea to query the state in every production that makes a request that could potentially jam a module even if you know that it will not happen because of the structure of the other productions. Doing so makes it clear to anyone else who may read the model, and it also protects you from problems if you decide later to apply that model to a different task where the assumption which avoids the jamming no longer holds.

In addition to the state, there are also other queries that one can make of a buffer. Unit 1 presented the general queries that are available to all buffers. Some buffers also provide queries that are specific to the details of the module and those will be described as needed in the tutorial. One can also find all the queries to which a module responds in the reference manual.

2.5.3 Visual buffer

On the RHS of **attend-letter** it makes a request of the **visual** buffer which isa move-attention and it specifies the **screen-pos**[ition] as the chunk from the **visual-location** buffer. A request of the **visual** buffer for a move-attention is a request for the vision module to move its attention to the specified location, create a chunk which encodes the object that is there, and place that chunk into the **visual** buffer. The following portion of the trace reflects this operation:

```
0.100   PROCEDURAL      PRODUCTION-FIRED ATTEND-LETTER
0.100   PROCEDURAL      MODULE-REQUEST VISUAL
0.100   PROCEDURAL      CLEAR-BUFFER VISUAL
0.100   VISION          Move-attention VISUAL-LOCATION0-0-1 NIL
0.185   VISION          Encoding-complete VISUAL-LOCATION0-0-1 NIL
0.185   VISION          SET-BUFFER-CHUNK VISUAL TEXT0
```

Note that the request to move-attention is made at time 0.100 seconds but that the encoding does not complete and result in a chunk being placed into the **visual** buffer until 0.185 seconds. Those 85 ms represent 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 to execute the attention shift and object encoding, it takes 185 ms to create the chunk that encodes the letter on the screen.

As you step through the model you will find this chunk in the **visual** buffer after those actions have occurred:

```
TEXT0-0
  ISA TEXT
   SCREEN-POS  VISUAL-LOCATION0-0-0
   VALUE  "v"
   STATUS  NIL
   COLOR  BLACK
   HEIGHT  10
   WIDTH  7
```

The chunk is of type **text** which is a chunk-type created by the vision module for encoding text from the screen. The **screen-pos** slot holds the location chunk for that object. 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.

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>
      state       respond
   +imaginal>
      isa         array
      letter      =letter
)
```

which makes a request to the **imaginal** buffer to create a new chunk which will hold a representation of the letter as was described in the section on the imaginal module.

2.6 Learning New Chunks

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 **perceptual** buffers, but they will be stored in declarative memory as a permanent chunk encoding what has been perceived once those chunks leave the buffers. That process occurs for all buffers – whenever a chunk is cleared from a buffer it becomes part of the model's declarative memory. Thus this is also happening for the **imaginal** and **goal** buffers' chunks when they are cleared.

2.7 Visual Re-encoding

There is another line in the trace of the model which shows the vision module doing something which will be addressed in this unit:

```
0.970   VISION        Encoding-complete VISUAL-LOCATION0-0-1 NIL
0.970   VISION        No visual-object found
```

At time 0.970 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 at time 0.885 seconds. 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. This re-encoding takes 85 ms just as an explicit request to attend an item does. If the visual-object chunk representing that item is still in the **visual** buffer it will be updated to reflect any changes. If there is no longer a visual item on the display at the location where the model is attending (as is the case here) then the trace will show a line indicating that no object was found and the vision

module will report a **state** of **error** through the **visual** buffer until there is another successful encoding (very much like a memory retrieval failure in the **retrieval** buffer).

2.7.1 Buffer Status

You can see the current query information for the buffers using the "Buffer Status viewer" button in the Control Panel or by calling the **buffer-status** command. That will show the required queries for the buffers along with the current value (either **t** or **nil**) for such a query at this time. Some buffers will also show additional information which can be queried and the documentation of the module in the reference manual will describe those other queries.

2.7.2 Re-encoding Cont.

This automatic re-encoding process of the vision 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. This is one reason it is important to query the visual **state** before all visual requests to avoid jamming the vision module since there may be activity other than that requested explicitly by the productions.

2.7.3 Stop Visually Attending

If you do not want the model to re-encode an item is it possible to make it stop attending to the visual display. This is done by issuing a clear command to the vision module as an action:

```
+visual>
   isa clear
```

This will cause the model to stop attending to any visual items until a new move-attention request is made and thus it will not re-encode items if the visual scene changes.

2.8 The Motor Module

When we speak of motor actions in ACT-R 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 hands can also be used to move a mouse or other device. Information about these features or extending the motor module is available in the reference manual and the documentation on extending ACT-R.

The buffer for interacting with the motor module is called the **manual** buffer. Unlike other buffers however, the **manual** buffer will not have any chunks placed into it by its module. It is

used only to issue commands and to query the state of the motor module. 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. The **manual** buffer query to test the state of the module works the same as the one described for the vision module:

```
?manual>
    state free
```

That query will be true when the module is available.

The motor module actually has a more complex **state** than just **free** or **busy** because 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 in the tutorial, and will only test on the overall state i.e. whether the entire module is **free** or **busy**. The **respond** production from the **demo2** model shows the **manual** buffer in use:

```
(P respond
   =goal>
      ISA         read-letters
      state       respond
   =imaginal>
      isa         array
      letter      =letter
   ?manual>
      state       free
==>
   =goal>
      state       done
   +manual>
      ISA         press-key
      key         =letter
)
```

This production fires when a letter has been encoded, the goal state is respond, and the **manual** buffer indicates that the motor module is available. Then a request is made to press the key corresponding to the letter from the letter slot of the chunk in the **imaginal** buffer and the state slot of the chunk in the **goal** buffer is changed to done. The type of action requested of the hands is specified in the isa slot of the **manual** buffer request. The **press-key** request used here assumes that the model's hands are located over the home row on the keyboard and the fingers will be returned there after the key has been pressed. There are many other requests that can be made of the hands, but for now, key presses are all we need. If you are interested you can find more details in the documentation of the motor module in the reference manual. The motor module actions from the trace that result from this production firing are shown here:

```
0.485   PROCEDURAL     PRODUCTION-FIRED RESPOND
0.485   PROCEDURAL     MODULE-REQUEST MANUAL
0.485   PROCEDURAL     CLEAR-BUFFER MANUAL
0.485   MOTOR          PRESS-KEY v
0.735   MOTOR          PREPARATION-COMPLETE
0.785   MOTOR          INITIATION-COMPLETE
0.885   MOTOR          OUTPUT-KEY #(4 5)
1.035   MOTOR          FINISH-MOVEMENT
```

When the production is fired a request is made to press the key, at time 0.485 seconds. 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 .885 seconds, however the motor module is still busy until time 1.035 seconds. The **press-key** request does not model the typing skills of an expert typist, but it does represent one who is able to touch type individual letters competently which is often a sufficient mechanism for modeling simple tasks.

2.9 Strict Harvesting

Another mechanism of ACT-R 6.0 is displayed in the trace of this model. It is a process referred to as "strict harvesting". It states that if the chunk in a buffer is tested on the LHS of a production (also referred to as harvesting the chunk) and that buffer is not modified on the RHS of the production, then that buffer is automatically cleared. This mechanism is displayed in the events of the **attend-letter**, **encode-letter**, and **respond** productions which harvest, but do not modify the **visual-location**, **visual**, and **imaginal** buffers respectively:

```
0.100   PROCEDURAL     PRODUCTION-FIRED ATTEND-LETTER
…
0.100   PROCEDURAL     CLEAR-BUFFER VISUAL-LOCATION

0.235   PROCEDURAL     PRODUCTION-FIRED ENCODE-LETTER
…
0.235   PROCEDURAL     CLEAR-BUFFER VISUAL
…
0.485   PROCEDURAL     PRODUCTION-FIRED RESPOND
…
0.485   PROCEDURAL     CLEAR-BUFFER IMAGINAL
```

By default, this happens for all buffers except the **goal** buffer, but it is controlled by a parameter (:do-not-harvest) which can be used to configure which (if any) of the buffers are excluded from strict harvesting.

If one wants to keep a chunk in a buffer after a production fires without modifying the chunk then it is valid to specify an empty modification to do so. For example, if one wanted to keep the chunk in the **visual** buffer after **encode-letter** fired we would only need to add an =visual> action to the RHS:

```
(P encode-letter-and-maintain-visual-chunk
   =goal>
      ISA         read-letters
      state       attend
   =visual>
      ISA         text
      value       =letter
==>
   =goal>
      state       respond
  +imaginal>
      isa         array
      letter      =letter

  =visual>
)
```

2.10 More ACT-R Parameters

The model code description document for unit 1 introduced the **sgp** command for setting ACT-R parameters. In the **demo2** model the parameters are set like this:

```
(sgp :seed (123456 0))
(sgp :v t :needs-mouse nil :show-focus t :trace-detail high)
```

All of these parameters are used to control how the system operates and do not affect the model's performance of the task. These settings are used to make working with this model easier, and are things that you may want to use when working with other models.

The first **sgp** command is used to set the :seed parameter. This parameter controls the starting point for the pseudo-random number generator used by ACT-R. Typically you do not need to use this parameter; however by setting it to a fixed value the model will always produce the same behavior (assuming that all the variation is attributable to randomness generated using the ACT-R mechanisms). In this model, that is why the letter "V" is always the one randomly chosen. If you remove this parameter setting from the model you will see different letters chosen when the experiment is run. For the tutorial models, we will often set this parameter in the demonstration model of a unit so that the model you have produces exactly the same trace as presented in the text, but you should feel free to remove that to further investigate the models.

The second **sgp** call sets four parameters that are useful for debugging a model. The :v (verbose) parameter controls whether the trace of the model is printed in the listener. If :v is **t** (which is the default value) then the trace is displayed and if :v is set to **nil** the trace is not printed. It is also possible to direct the trace to an external file, and you should consult the ACT-R 6.0 reference manual for information on how to do that if you would like to do so. Without printing out the trace the model runs significantly faster, and that will be important in later units when we are running the models through the experiments multiple times to collect data. The :needs-mouse

parameter is used to specify whether or not the model needs to control the mouse cursor. In some Lisp implementations, ACT-R can directly control the mouse cursor and will move it around on its own as needed. While this is important for the model to perform some tasks, it can be difficult to work with when it is not needed because you will be fighting with the model for control of the cursor. So letting the system know whether or not that is necessary and turning it off when not needed (as is done here by specifying **nil**) is often a useful setting. The :show-focus parameter 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 it because it could obscure other things you want to see. Finally, the :trace-detail parameter, which was described in the unit 1 experiment description document, is set to high so that all the actions of the modules show in the trace.

2.11 Unit 2 Assignment

Your assignment is to extend the abilities 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 press the key that corresponds to the letter that is different from the other two. The Lisp code to perform the experiment, two initial chunk-types, and an initial goal chunk are contained in the model **unit2-assignment**.

To run the experiment, call the new **do-unit2** function defined in the assignment model file. Like the **do-demo2** function, providing the symbol human to **do-unit2** will cause the task to run you instead of the model. 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-unit2 'human)
CORRECT
```

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

```
> (do-unit2 'human)
NIL
```

Your task is to write a model that always responds correctly when performing the task, and to run the model through the task you just need to call **do-unit2** without including the human symbol. In doing this you should take the model in **demo2** as a guide. It reflects the way to interact with the imaginal, 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 the same as the one used in the **demo2** model and only contains the control state information:

```
(chunk-type read-letters state)
```

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

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

There is an additional chunk-type which has slots for holding the three letters which should be used by the imaginal module:

```
(chunk-type array letter1 letter2 letter3)
```

You do not have to use these chunk types to solve the problem. 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. (Nonetheless, we would like your solution to keep any control state information it uses in the **goal** buffer and separate from the problem representation in the **imaginal** buffer.)

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. Cognitive 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.