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. These additional modules provide a model with visual, motor, auditory, and vocal capabilities as well as the mechanisms for interfacing those modules to the world. The mechanisms which we will use in the tutorial allow the model to interact with the computer i.e. process 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. You should load the "demo2.lisp" file now.

After you load the model you can perform the task yourself if you are running the ACT-R Environment or if 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
0.185   PROCEDURAL    QUERY-BUFFER-ACTION IMAGINAL
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
0.435   IMAGINAL      SET-BUFFER-CHUNK IMAGINAL CHUNK0
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 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
```

There 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 and 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, 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 specifies 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, the chunk that was 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 had been done with ACT-R models in the past, but since ACT-R 6, a more distributed representation of the model's "state" is the preferred means of modeling. Now, we 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 based on the read-letters chunk-type. 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 based on the chunk-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 is 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 a slot named state (or something similar) 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 need to 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 slots and 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>
      value       =letter
   ?imaginal>
      state       free
==>
   =goal>
      state       respond
   +imaginal>
      isa         array
      letter      =letter
)
```

We will explain the details of how the 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 which has a slot named letter that has the value of the variable =letter. 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
0.435   IMAGINAL      SET-BUFFER-CHUNK IMAGINAL CHUNK0
```

When the encode-letter production fires it 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.

Something to notice in the trace is that the chunk was 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 (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 does matter. In such situations one may need to verify that the module is available to create a new chunk and that is done with a query of the buffer in the conditions, as this production does. More details on querying modules will be described later in the unit.

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 it has an interface that one can use to extend it when necessary for more complicated features.

The vision module has two buffers. There is a **visual** buffer that holds a chunk which 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. 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 the value start in the state slot (which is how the chunk is initially created):

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

It makes a request of the **visual-location** buffer and it changes the goal state slot 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.

A **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 a 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 concept that there is a perceptual system operating in parallel within the vision module 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
   VALUE  TEXT
   COLOR  BLACK
   HEIGHT  10
   WIDTH  7
   SCREEN-X  130
   SCREEN-Y  156
   DISTANCE  1080
   KIND  TEXT
   SIZE  0.19999999
```

There are a lot of slots in the chunk placed into the **visual-location** buffer, and when making the request to find a location all of those features can be used to constrain the results. None of them are important for this unit, but we will describe the **screen-x** and **screen-y** slots here. They encode the exact coordinates of the object in the visual scene. The upper-left corner of an experiment 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 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>
     :attended   nil
```

we see that all it consists of is ":attended nil" in the request. This :attended specification is called a request parameter. It acts like a slot in the request, but does not correspond to a slot in any of the chunk-types specified for the model (one can print out all the chunk-types by calling the chunk-type command with no parameters). A request parameter is valid for any request to a buffer regardless of any chunk-type that is specified (or when no chunk-type is specified as is the case here). Request parameters are used to supply general information to the module about a request which may not correspond to any information that would be included in a chunk that is placed into a buffer. A request parameter is specific to a particular buffer and will always start with a ":" which distinguishes it from an actual slot of some 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 should try to find 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**, which means that the model has not attended to the location and that the object has also recently appeared in the visual scene.

2.5.2 The attend-letter production

The **attend-letter** production applies when the goal state is find-location, there is a 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>
   ?visual>
      state       free
==>
   +visual>
      cmd         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 there are no constraints specified for the slots of the chunk in that buffer test. All that is necessary for this production is that there is a chunk in the buffer and the details of its slot values do not matter. Then, a query is made of the **visual** buffer.

2.5.3 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 production which makes a request to a module that is still working on a previous request. If a production were to fire at such a point and issue a request to a module which is busy and that is 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 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 did however test the state of the imaginal module before making the request to the **imaginal** buffer in the **encode-letter** production. That query is not necessary in this model and removing it will not change the way the model performs. That is because that is the only request to the imaginal module in the model and that production will not fire again because of the change to the **goal** buffer chunk's state slot. Thus there is no risk of jamming the imaginal module in this model, but omitting queries which appear to be unnecessary is a risky practice. 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.

2.5.4 Chunk-type for the visual-location buffer

You may have noticed that we did not specify a chunk-type with either the request to the **visual-location** buffer in the **find-unattended-letter** production or its testing in the condition of the **attend-letter** production. That was because we didn't specify any slots in either of those places (recall that :attended is a request parameter and not a slot) thus there is no need to specify a chunk-type for verification that the slots used are correct. If we did need to request or test specific features with the **visual-location** buffer there is a chunk-type named visual-location which one can use that has the slots screen-x, screen-y, distance, kind, color, value, height, width, and size.

2.5.5 Visual buffer

On the RHS of **attend-letter** it makes a request of the **visual** buffer which specifies two slots: cmd and screen-pos:

```
   +visual>
      cmd         move-attention
      screen-pos  =visual-location
```

Unlike the other buffers which we have seen so far the **visual** buffer is capable of performing different actions in response to a request. The cmd slot in a request to the **visual** buffer indicates which specific action is being requested. In this request that is the value move-attention which indicates that the production is asking the vision module to move its attention to some location, create a chunk which encodes the object that is there, and place that chunk into the **visual** buffer. The location to which the module should move its attention is specified in the screen-pos (short for screen-position) slot of the request. In this case that location is the chunk that is in the **visual-location** buffer. The following portion of the trace shows this request and the results:

```
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
```

The request to move-attention is made at time 0.100 seconds and the vision module indicates that at that time as well. Then 0.085 seconds pass before the vision module reports that it has completed encoding the object at that location and places a chunk into the **visual** buffer at time 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
   SCREEN-POS  VISUAL-LOCATION0-0-0
   VALUE  "v"
   COLOR  BLACK
   HEIGHT  10
   WIDTH  7
   TEXT  T
```

Most of the chunks created for the **visual** buffer by the vision module are going to have a common set of slots in them. Those will include the **screen-pos** slot which holds the location chunk which represents where the object is located (which will typically have the same information as the location to which attention was moved) and then **color**, **height**, and **width** slots which hold information about the visual features of the object that was attended. In addition, depending on the details of the object which was attended, other slots may provide more details. When the object is text, the **value** slot will hold a string that contains the text encoded from the screen. As seen above for the text item there is also a slot named **text** which has the value t (the Lisp truth symbol) to indicate that the item is classified as text. Information about the chunk-types available for visual items will be described below.

After a chunk has been placed in the **visual** buffer this model harvests that chunk with the **encode-letter** production:

```
(P encode-letter
   =goal>
      ISA         read-letters
      state       attend
   =visual>
      value       =letter
   ?imaginal>
      state       free
==>
   =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.

2.5.6 Chunk-types for the visual buffer

As with the **visual-location** buffer you may have noticed that we also didn't specify a chunk-type with the request of the **visual** buffer in the **attend-letter** production or the harvesting of the chunk in the **encode-letter** production. Unlike the **visual-location** buffer, there actually are slots specified in both of those cases for the **visual** buffer. Thus, for added safety we should have specified a chunk-type in both of those places to validate the slots being used.

For the chunks placed into the **visual** buffer by the vision module when attending to an item there is a chunk-type called visual-object which provides these slots: screen-pos, value, status, color, height, and width. For the objects which the vision module can process by default the visual-object chunk-type is always an acceptable choice. Therefore a safer specification of the **encode-letter** production would include this:

```
(P encode-letter
   =goal>
      ISA         read-letters
      state       attend
   =visual>
      isa         visual-object
      value       =letter
   ?imaginal>
      state       free
==> ...)
```

If one extends the capabilities of the vision module so that it can process other items in the visual scene then different chunk-types which include additional features may be required. As was noted above, there is also a slot named text in the chunk in the **visual** buffer for this task. We will come back to that in a later unit of the tutorial.

For the request to the **visual** buffer there are a couple of options available for how to include a chunk-type to verify the slots. The first option is to use the chunk-type named vision-command which includes all of the slots for all of the requests which can be made to the **visual** buffer:

```
   +visual>
      isa         vision-command
      cmd         move-attention
      screen-pos  =visual-location
```

Because that contains slots for all of the different requests which the **visual** buffer can handle it is not as safe as a more specific chunk-type which only contains the slots for the specific command being used. For the move-attention command there is another chunk-type called move-attention which only contains the slots that are valid for the move-attention request. Therefore, a more specific request to the **visual** buffer for the vision module to move attention to an object would be:

```
   +visual>
      isa         move-attention
      cmd         move-attention
      screen-pos  =visual-location
```

Specifying move-attention twice in that request looks a little awkward. There is a way to avoid that redundancy and still maintain the safety of the chunk-type declaration, but we will not describe that until later in the tutorial.

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 are two other lines 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 note a failure in the **visual** buffer and indicate 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 standard 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>
   cmd clear
```

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

If one wants the safety of a chunk-type declaration with that, then as indicated above the vision-command chunk-type could be used:

```
+visual>
   isa vision-command
   cmd clear
```

There are also other options for declaring the chunk-type for that which will be described later on in the tutorial.

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>
      cmd         press-key
      key         =letter
)
```

This production fires when a letter has been encoded in the **imaginal** buffer, the goal state is slot has the value respond, and the **manual** buffer indicates that the motor module is free. 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 specific action requested of the hands is specified in a slot named cmd in a request to the **manual** buffer. The **press-key** action used here assumes that the model's hands are located over the home row on the keyboard (which they are by default). From that position a press-key request will move the appropriate finger to touch type the character specified in the key slot of the request and then return that finger to the home row position. There are many other actions that can be performed with the model's hands, but for now, key presses are all we need. 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 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 at time 0.485 seconds a request is made to press the key, the buffer is automatically cleared (even though the motor module does not put chunks into its buffer the procedural module still performs the clear action), and the motor module indicates that it has received a request to press the "v" 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 0.885 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 without looking at about 40 words per minute, which is often a sufficient mechanism for modeling simple tasks.

2.8.1 Motor module chunk-types

Like the **visual-location** and **visual** buffer requests the production which makes the request to the manual buffer did not specify a chunk-type. The **manual** buffer request is very similar to the request to the **visual** buffer. It has a slot named cmd which contains the action to perform and then an additional slot to specify details for performing that action. The options for declaring a chunk-type in the request are also very similar to those for the **visual** buffer.

One option is to use the chunk-type named motor-command which includes all of the slots for all of the requests which can be made to the **manual** buffer:

```
+manual>
   isa         motor-command
   cmd         press-key
   key         =letter
```

Another is to use a more specific chunk-type named press-key that only has the valid slots for the press-key action (cmd and key):

```
+manual>
   isa         press-key
   cmd         press-key
   key         =letter
```

Again, that repetition is awkward and we will come back to that later in the tutorial.

2.9 Strict Harvesting

Another mechanism of ACT-R 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** and **temporal** buffers, 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-keep-chunk-in-visual-buffer
   =goal>
      ISA         read-letters
      state       attend
   =visual>
      value       =letter
   ?imaginal>
      state       free
==>
   =goal>
      state       respond
   +imaginal>
      isa         array
      letter      =letter
   =visual>
)
```

Strict harvesting also applies to the buffer failure query. A production which makes a query for buffer failure will also trigger the strict harvesting mechanism for that buffer and clear it as an action unless that buffer is modified in the production. Clearing the buffer in this situation will result in the failure notice being removed from the buffer.

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 software 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 randomly chosen letter is always "V". If you remove this parameter setting from the model, save it, and then reload, you will see different letters chosen when the experiment is run. For the tutorial models, we will often set the :seed parameter in the demonstration model of a unit so that the model always 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. 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 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. Thus 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. 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 for this task.

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. To run the model through the task you just need to call **do-unit2** without including symbol human. 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 you may use for specifying the goal chunk, and the starting model already creates one and places it into the **goal** buffer. This chunk-type is the same as the one used in the **demo2** model and only contains a slot named state:

```
(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 specified which has slots for holding the three letters which can be used for the chunk in the **imaginal** buffer:

```
(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.