Interfacing ACT-R with Tasks

Dan Bothell

Table of Contents

Introduction

This document will describe ACT-R mechanisms which can be used to connect ACT-R to tasks that are not built using the AGI tools (ACT-R GUI Interface)[1] along with some of the issues to consider when building tasks. The main assumption with this document is that one is creating the task or has access to the implementation of the task and can modify it. Essentially, this is more about interfacing the task to ACT-R than the other way around. That said, if one has a task that cannot be modified but does provide its own external interface, then many of these concepts would still apply to the process of creating a "bridging" interface to connect the task and ACT-R. However, in that case, you will likely be much more constrained by what is provided in the task's interface.

This document is not going to discuss the low-level details of the communication between the task and ACT-R. The assumption is that the task (or bridging interface) will connect to the built-in ACT-R remote interface[2], and this will describe the activity in terms of using that interface – adding, monitoring, and evaluating commands. It is also not going to describe most of the ACT-R commands that are used because their details can be found in the reference manual.

There are many possible ways to handle the interaction and running of a model (or models) connected to a task, and what works best for one task may be completely inappropriate for a different one. Because of that, this document is not going to describe a single approach for handling the interaction, and will provide information about possible options one can use in different situations. Similarly, this document cannot possibly cover every programming language, graphics library, OS interface, etc that one might want to use for a task and how to implement the mechanisms in that environment. It will describe the concepts generally and at a fairly high level. In some places it may touch on some of the possible implementation details involved, but is not going to provide any specific examples of implementing them. There are however examples of some of the basic concepts in the ACT-R tutorial and in the examples/model-task-interfacing directory (although most of those do rely upon the AGI tools).

---

[1] The tutorial units cover many aspects of building tasks with the AGI, and there is also an AGI manual describing the AGI in the docs directory of the ACT-R distribution. Many of the same concepts described in the tutorial for building tasks with the AGI also apply to building tasks in general.

[2] The remote interface is described in the "remote" manual in the docs directory of the ACT-R distribution, and examples of implementing that interface in C, Java, Lisp, MATLAB, Node.js, Python, R, and Tcl/Tk can be found in the examples/connections directory of the ACT-R software.

Overview

There are four significant issues to consider when interfacing a model to a task: time management, running the model, model inputs, and model outputs. The first two are often intertwined, but it can be helpful to consider the timing situation first because that can constrain which running mechanisms are reasonable options to consider. Each of those issues will be described in a section of this manual. Some of those issues were discussed generally in the ACT-R tutorial code documents, and this documentation will assume that one has already worked through the tutorial and has a solid understanding of how ACT-R works and how to build simple tasks for it.

Timing

Determining how time will be handled for the task and model is usually the first thing that should be considered because it can play a large part in how the interactions will take place. There are two basic issues to consider with respect to timing: how do the model and task interact with respect to time and what controls the passage of time. The interaction approaches will be described in this section, and the next section will cover approaches for running the model and controlling the passage of time based on the particular interface needed.

There are three basic timing interaction approaches: untimed, asynchronous, and synchronous, with a few subcategories to synchronous interaction. The purpose of the modeling effort usually dictates how timing interaction needs to be handled. However, when creating a connection to an existing task, its provided interface (or lack thereof) may dictate a particular timing approach. Typically, a single timing approach is used for a task, but occasionally different stages of a task may require different approaches.

Untimed

If the timing of the model actions is not important for the task and the task does not include a timing component then timing issues are irrelevant. That situation often happens in tasks where the model is making predictions or classifications of information based on prior data or experience which is not time sensitive – all that matters is the result which the model generates not the detailed timing of how it got there. In this case, the model can run using the ACT-R clock with no need for any other timing considerations.

Asynchronous

The asynchronous situation typically occurs when the task itself is not tracking the time, but the timing in the model is important. Continuous prediction or classification tasks where the model is learning the information as it goes, and the model's processing time of that information matters, would be one type of situations where this occurs. With respect to timing, this is similar to the previous case since the model is free to run in its own time frame, but there may be some additional control needed to make sure the model's timing is properly maintained.

Synchronous

In synchronous mode both sides are dependent upon timing information and it is necessary to keep them running together on the same timeline. There are three types of synchronous timing that typically occur when interfacing to tasks which we will call: completely, step-wise, and real time.

Completely

In a completely synchronous system both the model and task are using the same clock to advance all of their actions and the clock only advances when both sides have completed all of their actions at the current time. These types of tasks are often driven by a discrete event simulation system to perfectly track the time course of actions, and when the sources of randomness in the model and task are controllable (for example by setting a seed for a random number generator) allows for perfect recreation of actions when run again.

Step-wise

A step-wise synchronous system runs the model and task in intervals during which they are not kept in sync with the clock, but do agree on the time at the beginning/end of each interval. A common situation for this is a task that updates in discrete time steps which provide a constraint on the time that the model has to operate, but the timing of the model's actions within a step are not relevant to the task, for example, a game which is updating its state at fixed intervals (like 60 updates per second) and only depends on whether an action occurred since the last update. In many ways this is similar to a completely synchronous system, but without requiring that every time step be agreed upon provides for some potentially easier implementation approaches.

Real Time

When both the model and task are dependent upon the passage of the real time clock or a task based clock which does not wait for an acknowledgment, that is loosely synchronous in that they are advancing with the same clock, but they are not strictly synchronizing their actions. There are many situations where this level of synchronization is required, and usually it's a real time constraint, for example, interacting with a task that was built with no external interface which runs in real time, operating in tasks which also include humans, or embedding the model in a robot for which human-level performance is desired. Although this situation is generally easy to implement in terms of running the model, the situations which require it are often more difficult in terms of providing the inputs to the model and performing the model's actions. In addition to that, there are some potential issues with respect to how the ACT-R system runs in real time mode which will be described in the next section.

Running

This section will describe ways to run a model with a task based on the timing interaction needed. Before that however, it will start with an overview of how ACT-R runs, a summary of the commands available for running ACT-R, and some general issues with respect to running ACT-R and implementing tasks.

Running ACT-R

The ACT-R software is built around a discrete event simulation system which was custom built to provide the necessary operations for implementing the ACT-R cognitive architecture. That system is referred to as the meta-process, and although it was built specifically for running ACT-R, it can be used to implement other tasks as well. To use the meta-process, one schedules events to occur at particular times (where an event is basically just a command to call and the parameters to pass it). The events are ordered by their times, and when ACT-R is run the meta-process evaluates the actions associated with the events one at a time in order. More details on the use of the meta-process and how to schedule the events can be found in the reference manual.

The meta-process can run the events in two modes: simulated time and real-time. In simulated time mode the events on the queue provide the clock. The events are evaluated with no delays in between, and each event advances the simulation time immediately to that event's time. In that case, it will be necessary for all of the actions or updates from the task to be scheduled as events or to be triggered during the evaluation of some other event so that the timing remains consistent. In real time mode the meta-process is driven by a clock which determines when it is appropriate to evaluate the next event. Before evaluating an event the clock is checked and if the time from the clock is equal to or greater than the time of the event then the event is evaluated, and if it is not the appropriate time then the meta-process will wait and check again later. The default real time mode clock is one built into Lisp which tracks the passage of real time, and the default delay is a simple loop which just yields the processor before checking again. Both of those can be changed to other mechanisms which can allow for better synchronization between the meta-process and a task (see the configuring real time operation section of the reference manual for more details).

When running ACT-R using a real time mode clock that is not explicitly synchronized with a clock in the task there are timing issues to be aware of because the meta-process is built to be no faster than real time since it waits to evaluate the events. Although it is generally able to run the system significantly faster than real time, it could be inconsistent in operation over any particular interval for

reasons internal to ACT-R or because of other demands on the computer. If an event is delayed because of other computer processes or takes longer to evaluate in real time than the simulated time it represents, then there will be a delay in the event processing. After the delay, events may be run faster than the real time intervals between them until the system catches up to the real time clock. Therefore, there is no guarantee of determinism or repeatability in a task when running in real time mode with an unsynchronized clock.

Running commands

There are a few available commands for running the meta-process. They differ in how the meta-process determines when to stop running. In all cases it will stop running immediately if an explicit break event is encountered. If there are no events left to evaluate in simulated time mode then all of the commands will also stop running immediately, and most will also stop when there are no events left in real-time mode as well. Here are the commands along with their stopping condition:

run

Run is given a time interval in seconds and will run the model up to that amount of time from the current time stopping early if there are no events left to process.

run-full-time

Run-full-time is given a time interval in seconds and will run the model for exactly that length of time. In simulated time mode it will stop immediately when there are no events left to process and advance the simulation clock to represent that the whole interval has passed, but in real time mode it will continue to run until the real time clock being used indicates that the interval has passed regardless of whether there are events to evaluate or not. Since events can be scheduled in parallel with the meta-process running, in real time mode it is possible for it to evaluate events which are created while it is waiting for the time interval to complete if the time on those events have a time which falls within that interval.

run-until-time

Run-until-time works the same as run-full-time except that it is given an explicit time in seconds instead of a duration.

run-until-condition

Run-until-condition is given a command to evaluate before every event, and it will stop running when that command returns a true value or there are no events to process. The command will be passed one parameter which is the time of the next event in milliseconds.

run-until-action

Run-until-action is given the name of an action and it will run until an event which evaluates that action occurs or there are no events to process.

Running performance

If you are concerned with the time that it takes to run ACT-R and the task then there are a couple things to consider. In general, the fewer calls that are made to run the system the faster it will perform because there is some overhead to starting and stopping a run as well as the communication costs when the running call is made remotely. Thus, running the system continuously through the whole task instead of incrementally will usually be faster. However, the run-until-condition command (which can provide an easy approach to running through the full task since it takes a user specified command to determine the ending condition) is very costly to use since it requires calling that condition for every event which occurs, and if that command is a remote one that cost can be significant. Therefore there is often a trade off between run-time performance and implementation complexity to be considered when creating the interface.

Running approaches

The following sections will describe ways to use the running mechanisms in creating interfaces for the different timing interactions presented above along with notes on potential issues and additional considerations.

Untimed

The untimed situation usually lends itself to a fairly straight forward running interface since there is no need to coordinate the timing. In many cases the model is going to be reset before each run to generate the result, and even when it is not, there's usually no real benefit to running it continuously in these situations since the model typically runs long enough for the overhead of starting and stopping to be negligible. Thus, the typical approach is to build the model so that it effectively stops itself when it is run i.e. it performs the needed action(s) and then has nothing else to do until there is a change in the state. Then, all that is necessary is to reset the model or setup its initial conditions as needed, and then just call the run command with a time that is sufficiently long enough for the model to finish what it needs to do, or use run-until-action if there is a specific action the model is going to perform that indicates completion. When that run returns, the task can process the result of the model as needed and then repeat. The one thing to be careful with when having the model stopping itself is to be sure that it can appropriately handle possible issues, like retrieval failures, so that it does not

stop prematurely or end up in an endless loop because in this case the model is essentially part of the task implementation as well.

Asynchronous

The asynchronous situation can often be implemented the same as the untimed one, but if there are constraints on how much time the model is allowed to run before providing a response or receiving additional data then just allowing the model to run until it is done may not be acceptable. One option would be to use run-full-time or run-until-time instead to provide those timing constraints. Alternatively, since the model is likely not going to be reset before each run, it might be useful to schedule events that provide the additional information or perform whatever action causes the time restrictions and also that collect the model's responses or output if necessary. Then, the model can be run continuously through the entire task instead of incrementally through each section.

Real Time Synchronous

In most real time synchronous cases the model will be running continuously through the task and a simple call to run in real time mode (using either the default real time clock or a custom real time clock which gets its time from the task) with a very large time will be sufficient. However, if the task is not continuous and the model will need to be stopped and started then other approaches may be required. If the intervals are of known lengths then run-until-time and/or run-full-time could be used in real time mode. If the stopping conditions are not known in advance then one option would be to use run-until-condition with a custom command that can determine when it is time to stop. However, because the condition is called for every event the meta-process evaluates it can be costly, and that may impede the model's ability to actually keep up with running in real time. Instead, it may be more useful for the task to schedule an action to stop the run when needed. That can be done by scheduling a break event, or by adding a new command and using run-until-action to stop when an event with that action occurs. The break event would be the most efficient, but the test for a specific action is not very costly time wise and might be useful if there's something that the task code needs to do when the model stops, like recording a result.

Completely Synchronous

The important consideration for the completely synchronous case is whether ACT-R's clock is being used or something outside of ACT-R controls the clock. If ACT-R's clock is not being used, then the details of the task's timing interface are going to determine how the model can be integrated with the task, but in my experience, tasks' timing interfaces are usually of a step-wise synchronous nature (the

task does not care about times between its own events) and a truly synchronous task which would require all of the model actions or time changes to be acknowledged are rare.

ACT-R controls the clock

If the task is being built for ACT-R, then using the discrete event system in ACT-R to build the task makes for an easy way to run it with a model since the task and model would be running off of the same event queue and clock. To do that one would simply need to add commands to perform the operations of the task and schedule them at the appropriate times to do the task. Then, depending upon the structure of the task, an appropriate ACT-R running command could be used to run both together. That might also involve monitoring for model actions as triggers to then schedule additional task actions as well.

If the task has its own discrete event system and an interface which accepts an external clock, then the ACT-R time could be sent to the task in what would basically be a step-wise synchronous manner (since ACT-R does not care about the time between its actions unless it is running in real time mode), and again one of the ACT-R run commands could be used to run the model and provide the task's clock. The recommended way to synchronously provide ACT-R's time is with an event hook. An event hook is a command that is called at every event and passed the details of that event, which includes its time. To create an event hook the add-pre-event-hook or add-post-event-hook command is used, and the difference is whether the command is called before or after the event occurs. Because it is called for every event is will be called very frequently which may be costly, and it will likely also provide repeat times since multiple events can occur at the same ACT-R time (of course one could have the event hook only relay the time when it changes if that is what is needed).

Clock external to ACT-R

If the clock is external to ACT-R and the system is completely synchronous, then how to run ACT-R is going to depend strongly on the external system's interface and requirements (does it provide a clock pulse, does it provide event scheduling, does it require every action to be acknowledged or only time updates, etc). Some very general approaches for running ACT-R will be listed here, but only briefly since this is an unlikely and very idiosyncratic situation. One approach would be to call run-until-time at every clock pulse provided by the other system to advance ACT-R. Another would be to run ACT-R continuously and use a pre-event hook to handle the synchronization. The pre-event hook would schedule an action in the external system with the current ACT-R event's time and then waits for that action to be triggered by the external system before allowing the event to occur in

ACT-R, effectively mirroring the ACT-R events on the external event stream. The final approach would be to run ACT-R in real-time mode and specify a custom clock for ACT-R using the mp-real-time-management command which would get the current time from the external system and likely also need to specify the slack function to acknowledge the time changes from ACT-R to the other system.

Step-wise Synchronous

This situation occurs often with external tasks, and usually has one of two forms of timing information from the external system: providing a duration until the next synchronization or the time at the current synchronization.

Duration based

There are two common ways that a duration based interface provides the timing information: either it provides a fixed time period once at the start of the task or it repeatedly provides a time until the next synchronization. In either case, the same mechanisms can be used to run the model, the only distinction is in which side initiates the synchronization. If the model was provided a fixed period, then it will send the sync notice and wait for a response. If the task is providing durations, then the model will wait for a duration message and then send an acknowledgment when it is done.

The simple approach is to run the model incrementally using run-full-time with the provided duration at each step, but as described above that is not always an efficient way to run the model. One alternative with a continuous run would be to schedule an event to perform the synchronization at the appropriate time. If there's a fixed period then the schedule-periodic-event command could be used to have that event automatically scheduled as it runs, but if the duration is provided incrementally, then the event would need to wait for the duration value to schedule the next one explicitly. Since there will always be events scheduled to happen for the model, the run command could be used with a large time to run it. Another alternative for a continuous run would be to create a custom clock function that is driven by the durations, install that with mp-real-time-management, and then run the model in real-time mode. That is likely to be more complicated than just scheduling events, but if the interval mode of the clock is used it should not be too difficult to implement.

Current time based

When the task provides a current time the options are similar to those for when it provides a duration. The simple (but possibly inefficient) case is using run-until-time repeatedly with the time that is

given. An option for running continuously is to schedule an event for the time given where that event sends the acknowledgment and waits for the new time to schedule the next event. Since there will always be an event for the end time on in the meta-process queue the run command with a sufficiently long time can be used to run the mode. The other option for running continuously is to use the provided time as the basis for a custom clock. In this case, since the time itself is provided, the absolute clock mode can be used with mp-real-time-management to provide the clock, and a slack function could be created to send the acknowledgment when the model has no more events at the current time and get the next time.

Inputs to the Model

Most tasks are going to need to provide some data to the model. There are many ways to handle that, and this section will describe some options for doing so using the vision and audio modules in ACT-R as well as some general approaches. After describing how to provide input there will be some additional information about implementation considerations.

Visual Input

Providing visual input to a model is done using the add-visicon-features command. That command allows the modeler to specify the contents of the visual-location and visual-object chunks for one or more new features. The vision module will then add those features into the model's visicon. Once a feature has been added it can be changed using the modify-visicon-features command. It can be removed using delete-visicon-features, or all features can be removed using delete-all-visicon-features. The only requirement for specifying visual features is that the visual-location chunk must contain a position, which by default are the screen-x and screen-y slots but the names of the position slots can be changed when creating the feature. Details on those commands are found in the reference manual, and the examples/vision-module directory of the distribution contains several examples of using them.

Auditory Input

To provide a model with auditory input there is one general command and a few commands for that simplify creating specific types of sounds. The general command is new-other-sound which allows one to specify all of the auditory parameters relevant to the aural module for the sound as well as including custom features. The specific commands are new-digit-sound, new-tone-sound, and new-word-sound, which create sounds for a number, simple tone, and word respectively. Unlike visual features, once the sound is created it cannot be modified or removed, and will occur at the appropriate time in the model based on the specification provided and the audio module's parameters.

Other Input

In addition to making perceptual information available to the model, there are three other available mechanisms: adding chunks to its declarative memory, using buffers, and directly calling commands from within productions. Each of those will be described in a section below.

Declarative Memory

In most models one provides a set of initial chunks for the model's declarative memory using the add-dm command. That command, and the similar add-dm-fct command, can be used at anytime to add additional chunks to the model's memory, and the declarative parameters for those chunks can be changed as needed using the sdp command. Alternatively, if instead of adding new chunks, one wants to strengthen existing knowledge without having to adjust the declarative parameters, then there are commands for adding a reference to existing chunks in the same way that it updates the references automatically when buffers are cleared. The merge-dm and merge-dm-fct commands take descriptions of chunks just like add-dm and add-dm-fct. If a description matches a chunk in declarative memory then that existing chunk will be strengthened, but if the description does not match a chunk in declarative memory a new chunk will be added for it. The merge-dm-chunks and merge-dm-chunks-fct commands take the names of chunks instead of chunk descriptions, and then check whether there are chunks which match those given in declarative memory. If a matching chunk is found then it is strengthened, otherwise the indicated chunk is added to declarative memory. An important consideration when using declarative memory to provide input to a model is that a model's declarative memory may not be modified other than through addition of new information – chunks in declarative memory cannot be modified or removed.

Buffers

Buffers can be used to provide new chunks to a model, and the contents of a chunk in a buffer may be modified to update the information. The set-buffer-chunk command can be used to place a copy of a chunk into any buffer, and the mod-buffer-chunk command can be used to modify a chunk in a buffer. Typically, only the goal and imaginal buffers are manipulated directly because most of the other buffers are set and adjusted by their modules in ways that could conflict with direct manipulation (the automatic stuffing of new chunks into the perceptual buffers for example).

One can also create new buffers by adding a module to the system so that there are no potential conflicts. Another advantage of adding a module and a new buffer is that one can then also create custom queries that can be used with that buffer to test for information without having to create a chunk for the buffer.

Direct Calls

The !eval! and !bind! operators in productions can be used on the LHS to test or acquire information, and they can call any command which has been added in addition to evaluating Lisp expressions.

When used on the LHS of a production they must return a true value for the production to match, but just because a command has been called from the LHS of a production does not mean the production which made the call will be the one that matches and fires because other constraints may still fail to match. Similarly, failure of other constraints may prevent the command from being called since it is unnecessary. Thus, the command being called should not make any assumptions about the activity of the model based just on having been called.

Input Considerations

Timing

With all of these input mechanisms, an important issue to consider is that of timing. To ensure that the information is provided to the model at the appropriate time, the recommendation is to only change or add information when the model is not running or through an event evaluated by ACT-R while it is running. For buffer manipulations, the recommendation is to only use the commands shown above in code that is called during a scheduled event, and at any other time the actions should be scheduled to occur using the corresponding commands schedule-set-buffer-chunk and schedule-mod-buffer-chunk. The vision and aural modules will schedule actions to process the percepts given, but those scheduled events are based on the current time when the calls happen thus the initial calling time still matters. As for commands called directly from productions, if the same command can be called from multiple productions care should be taken to ensure that the results are stable during the entire conflict-resolution process to avoid discrepancies in the matching, and again, a safe way to do so is to only change state as a result of a scheduled action.

Task Integration

“Where/when do I call <some input function>?” is a frequent type of question that arises, and unfortunately, as with most of the task implementation issues, there is no simple answer. In some cases the answer is easy because of the task constraints e.g. if the task provides a single state output mechanism or there are specific perceptual output streams (image processing for example), then you would have to add the calls into the code that receives and processes that data. In the cases where one is implementing the task or modifying the source code for a task however, it is basically a software design issue that is not dependent upon the ACT-R interface. As long as it is being called at the appropriate time for the task it does not matter if it is done as an explicit call in the task code, if it is done in an abstraction layer that has been built to separate model output from human output, if it is

tied into the GUI library so it happens automatically when an interface is built, or anywhere else that it may be placed.

Output from the Model

Most tasks are going to require some form of output or data from the model. This section will cover options which are essentially the reciprocals of the input mechanisms: the motor and speech modules, reading information from declarative memory and buffers, and making calls directly to task commands.

Motor Output

The ACT-R motor module controls the model's hands, and can produce many types of actions. Typically however one does not directly interact with the motor module's low level actions. Instead, one installs a device to convert the low level actions into more meaningful events which generate signals that can be monitored, and there are two devices included with ACT-R which will be discussed here: a keyboard and a cursor (which can be controlled by a mouse or joystick). It is possible to create new devices, but that requires an understanding of the low level motor actions which are not going to be discussed in this document. However, there is also a way to add new actions to the motor module which can call custom commands directly. Those commands can work with or without a device, and that will be described in the section on direct calls.

Motor Devices

When using the AGI, the experiment window device automatically installs the keyboard and mouse devices for the motor module, but if you are not using the AGI you will need to install them explicitly. That is done with the install-device command. The interface name for the motor module is "motor". The keyboard device is named "keyboard" and no details are necessary. The cursor device is named "cursor" and the details must be provided as one of "mouse", "joystick1", or "joystick2" to indicate which type of cursor is being used. Each of the available cursors is treated as a separate cursor with its own position, and they have different scaling parameters for the Fitts's law calculation of movement timing. For each of the cursors, it is assumed that there is a button located under each finger when the model's hand is on the cursor device.

Keyboard Event

When the model performs an action that results in pressing a key on the keyboard device it will generate an output-key signal. That signal can be monitored to respond to model keypreses. The

signal is passed two parameters. The first is the name of the model, and the second is a string which indicates the key that was pressed.

Cursor Events

There are three signals that can be generated when the model interacts with a cursor. The first is that whenever a cursor is moved (which includes the initial placement when it is installed) a move-cursor signal is generated. If that is monitored, the monitoring command will be passed three parameters: the name of the model that performed the action, the name of the cursor which was moved, and the new position of the cursor (which will be a list of three numbers representing the x, y, and z coordinates respectively). When the model performs an action which results in a finger being pressed on the cursor device a signal is generated. If it is a "mouse" cursor then it will be a click-mouse signal and any monitoring command will be passed three values: the name of the model, the list of the cursor's position at the time the click occurs, and the name of the finger which performed the action. For any other cursor, a finger press will generate a click-cursor signal and any monitoring commands will be passed four parameters: the name of the model, the name of the cursor, the position of the cursor at the time the click occurs, and the name of the finger which performed the action.

Speech Output

The ACT-R speech module provides the model with two actions: speaking and subvocalizing. As with the motor module, one typically does not directly interact with the speech module's low level actions. Instead, a device is installed to interpret those actions and one monitors for the signal generated by that device.

Speech Device

There is only one device provided for interacting with the speech module, named "microphone". The microphone device detects the model's speak actions, but does not respond to subvocalize actions. It provides one signal at the beginning of a model's speech output.

Microphone Event

When the model performs a speak action the microphone device will generate an output-speech signal when the model's output begins. That signal can be monitored to respond to a model's vocal

output. The signal is passed two parameters. The first is the name of the model, and the second is a string which contains the speech output.

Reading Chunk Information

Instead of responding to model actions, another common form of output from a model that a task may use is the chunks from declarative memory or in the buffers. This is often done when the model does not need to respond with human-like actions and response times either because they are not part of the task or that performance is unnecessary for the modeling objectives.

Declarative Memory

The dm command can be used to get a list of the names of all the chunks in the model's declarative memory. It will also print out those chunks if the command trace is enabled, but that can be turned off if it is not needed. If one only wants some of the chunks, then the sdm command can be used to search the model's declarative memory. That command is passed a specification of the chunks desired using the same chunk specification mechanisms as are used for buffer tests on the LHS of a production (called a chunk-spec in the reference manual), and that chunk-spec can include variables and modifiers. It will return a list of the names of all of the chunks which match that specification in the model's declarative memory. Like the dm command, it will also print them to the command trace if it is enabled. The declarative memory parameters of the chunks can be accessed using the sdp command. The declarative module will also record the details of the requests that it processes if the "retrieval-history" stream is recorded. The data returned from that stream contains all of the retrieval requests that were made which includes all of the chunks that matched a request and the details of how their activations were computed.

Buffers

To get a list of all the chunks in buffers the buffer-chunk command can be used without providing any buffer names, and it can also be used to get the names of chunks in specific buffers if passed the buffers' names. It will also print the chunks to the command stream. Another command, buffer-read, can also be used to get the name of a chunk in a buffer and it does not print anything.

Chunk Contents

The previous two sections describe how to get the names of chunks, but typically it is a chunk's content that is important. To get the value of a chunk's slot the chunk-slot-value command is used

which takes the name of a chunk and the name of a slot, and it returns the value in that slot of that chunk. It is also possible to get a list of the names of slots which contain values in a chunk using the chunk-filled-slots-list command.

Calling Task Commands

There are several ways that one can call a custom command from within a running model, and this section will provide information on many of those.

Hook Functions

In the synchronous timing section it talked about using an event-hook to add a call to a custom command whenever the meta-process executed an event. The procedural and declarative modules also provide hooks to call custom commands during their operation, and some of those hooks can be used to modify how the module operates. Here we will just list some of the commonly used hooks, and the details of the hooks can be found in the reference manual. The declarative module provides hooks to call commands when new chunks are added to memory, when chunks are merged with existing chunks in memory, when a retrieval request is made, after the chunks matching a request have been determined, when the retrieved chunk is selected, and several to modify the individual components of the activation calculation itself. The procedural module provides hooks to call commands when the set of matching productions has been determined, when a production fires, and when procedural partial matching is enabled there is a hook to adjust how the mismatch penalties are computed.

Production Actions

The !eval! and !bind! operators in productions can be used on the RHS to call any command which has been added or to evaluate Lisp expressions. When used on the RHS of a production the command or expression will be called when the production fires (specifically during the production-fired event of the procedural module). If that call is going to change the contents of buffers, then it must schedule those changes to occur to avoid any problems with other actions in the production which may be expecting the buffers to be as they were when the production matched the buffers' contents.

Through Modules

Another option for making direct calls from the model is by creating a new module and then making requests to that module's buffer(s). One reason for choosing to use buffer requests instead of !eval! and !bind! calls is how they interact with the production compilation mechanism. The ! actions (which must be marked as safe for compilation purposes) will be added as-is during the composition process, but with requests there are mechanisms in the composition process for combining multiple requests into one and also for eliminating a request. Production requests also automatically generate an event in the meta-process which can be useful for inspecting and debugging a model's actions.

Two of the provided modules also have the ability to extend their possible actions without the need to create a new module to add calls to new commands. That is typically much easier than creating a new module, and can also be useful if the effect of the new command should be attributed to one of those modules for purposes of computing the fMRI predictions from the model.

New Motor Module Actions

The motor module command extend-manual-requests can be used to add new functionality to the motor module. However, that command is currently not available for use remotely i.e. it must be written in Lisp. The reference manual has details on that command, and there is an example in the examples/extend-manual-requests directory which shows how the motor module's "style" mechanism can be used to create a new action that has the same processing steps as the built-in actions.

New Imaginal Actions

The imaginal module has a second buffer called imaginal-action which can be used to attribute actions to the imaginal module. Requests to that buffer are very similar to a !eval! in that they must specify a command to call and the module then calls that command, but the difference is that with this action the imaginal module is marked as busy for some time after the call (how long depends on how the request is specified and it can be a fixed time of arbitrarily determined by the command that is called). Details can be found in the reference manual and there is a simple example found in the building sticks task model in unit 8 of the tutorial.

Output Considerations

Timing

As with everything else, timing of the output actions can be important. The signals from the speech and motor module are generated at the appropriate ACT-R time in the model run which is useful if

the task requires collecting the response times. When calling custom commands through other mechanisms, it may be necessary to have those commands schedule events to perform their output if it has a timing component. If the model is interacting with a real time synchronous task, there is going to be some cost to making the call from ACT-R to the task, and that may require additional work to compensate or adjust for the lag in the action relative to the operation of the model.

Task Integration

Making calls to task created commands, and sending output to tasks with specified input mechanisms is typically a straight forward process. However, when monitoring a model's output from the motor and speech modules there are often multiple possibilities and task considerations which are not constrained by the ACT-R interface and are open software design decisions. For a simple task, a common approach is to just have the monitoring command record the data and update the task directly as needed (as is done in all of the tasks for the ACT-R tutorial). If the task is already implemented with a real GUI that has an interface for providing simulated input, then it may be useful to build a general interface for connecting the ACT-R commands to that interface, particularly if that same GUI is used for multiple tasks. Other times, it may be necessary or useful to create real actions based on the model actions by making system level calls to generate keypresses, mouse movements, and button clicks which are detectable by other programs, and similarly, a text to speech library may be useful to create real sounds for the model's speech.