Unit 4 Code Description

There are only a couple of new commands used in the models for this unit and one of them, set-all-base-levels, was discussed in the unit text. So there is not really much of anything new to discuss about the functions used to run the models, but there is a different approach to running the models used for the zbrodoff experiment compared to the paired associate task and all of the previous units' tasks. We will start by describing the differences between those approaches, and then look at the code which implements them with the focus more on the overall structure of the experiment implementations and model interaction than the details of every function call.

So far you have seen what can be described as an iterative or "trial at a time" approach to writing the experiments for models. The experiments run by executing some setup code for a trial, running the model to completion on that trial, recording a result and then repeating that process for the next trial until all the trials are done. That style is a commonly used approach, but it has some drawbacks which you may have encountered. For instance, when running an experiment it is not possible to stop it using the Stop button in the Stepper tool of the ACT-R Environment because the Stop button only stops the current "run" of ACT-R, but it cannot affect the code which is making those calls to run things. Instead you have to terminate the execution of the task somehow, which can be difficult if it wasn't written with a way to do so.[1] For models with few trials or that get reset on each trial that may not be a significant issue, but for large experiments or models that need to learn from trial to trial that can make things difficult to work with, particularly if there is a problem with the model on a later trial that forces one to abandon a very long run by terminating the experiment code which likely doesn't have any way to continue where the model left off for debugging or investigating the problem.

An alternative way to write the experiments is with a more event-driven approach. The system which runs the ACT-R models is a general discrete-event simulation system which can be used to run other things as well, like an experiment for a model (or a person if precise timing information isn't required). Calling one of the ACT-R running commands causes all of the events which have been created to be executed in either a simulated time or real time sequence. Up to now those events have been mostly generated by the model, e.g. production firing, memory retrieval, and key presses, or by ACT-R commands like goal-focus. However, as was seen in the sperling task, arbitrary events can also be scheduled to execute at particular simulated times. One can also use the interface events generated by the model (like output-key) to do things other than just record the response. By scheduling events to occur at appropriate times and putting some of the control into the functions that handle the model's actions the experiment can run "with" the model instead of "around" it. Such an experiment only needs to call the run function one time to complete the whole experiment instead of once (or more) per trial.

Having the model running in an event-driven experiment with a single call to run typically allows for more interactive control of the task as a whole. The Stepper tool in the

---

[1] Hitting control-C or some other interrupt key in a Lisp running ACT-R may stop things, but if it's the ACT-R system code that is interrupted instead of the task code then it may cause problems for running ACT-R and require exiting and restarting.

Environment will also pause on user scheduled events and the Stop button will stop the whole experiment if it is being driven by the events that are run. That allows one to see exactly what is happening at specific points in the experiment without having to abort the experiment function. Additionally, to continue after stopping all one needs to do is usually call run again to have the model and the experiment continue from where they left off since the events are still scheduled to occur at the appropriate times. It can also make writing the model itself easier because one doesn't have to make sure that the model "knows" when to stop for the task code to update and can just focus on having it respond to the events that occur as they occur instead of as a sequence of separate interactions.

The two models for this unit use those two different approaches. The paired associate task is written using the iterative approach, and the zbrodoff task is written as an event-driven experiment. The comments in the files describe how they work, and the paired associate task is similar to the tasks from prior units. Because the zbrodoff task is written in the event-driven style, it looks a little different than the tasks from previous units, and some of the details of the approach as used in that task will be covered later in this document.

New Commands

Run-full-time/run_full_time – this function takes one required parameter which is the time to run a model in seconds and an optional parameter to indicate whether to run the model in step with real time. The model will run until the requested amount of time passes whether or not there is something for the model to do i.e. it guarantees that the model will be advanced by the requested amount of time. If the optional parameter is provided and is a true value, then the model is advanced in step with real time instead of being allowed to run as fast as possible in its own simulated time, and if a number is provided as the optional parameter then that sets the scale to use for advancing model time relative to real time.

print-warning and **print_warning** can be used for outputting information in the ACT-R warning trace. The Python function takes a single parameter which is a string and prints that as an ACT-R warning message. The Lisp version is a little more general and works similar to the Lisp format command. It takes a format string and then any number of parameters after that to be used in the format string.

AGI command defaults

Two of the commands which we have seen in previous units are used slightly differently in the zbrodoff experiment. Previously when clear-exp-window was used we passed it the window to clear, but here we did not. If there is only one window opened by the AGI then most of the commands will default to working with that window and it does not need to be provided. Thus, this experiment assumes that there is only one open window and doesn't pass one to the clear-exp-window command. Similarly, when using add-text-to-exp-window (and other similar functions for buttons and lines which will be used later in the tutorial) the first parameter can be specified as nil (Lisp) or None (Python) to indicate that the default window should be used instead of specifying one. If there are multiple

windows open when a call is made that indicates using the default window it will result in a warning and nothing will happen.

The :ncnar Parameter

As was mentioned in the main text there is a new parameter being set in the models for this unit - :ncnar (normalize chunk names after run). This parameter toggles whether or not the system cleans up the references to merged chunks' names. If the parameter is set to **t**, which is the default, then the system will ensure that every slot of a chunk in the model which has a chunk as the value references the "true name" of the chunk in the slot i.e. the name of the original chunk in DM with which any copies have been merged. That operation can make debugging easier for the modeler because all the slot values will be consistent with the chunks shown to be in DM. However, if a model generates a lot of chunks and/or it makes many calls to one of the ACT-R commands to "run" the model it can take time to maintain that consistency. Thus it can be beneficial to turn this parameter off by setting it to **nil** when model debugging is complete and one just wants to collect the results or when the real time needed to run a model is important. For the models in the tutorial, leaving it enabled will typically not result in much of a run time increase (the paired model is the worst performer in this respect running around 8% slower with it enabled and the zbrodoff model is about 5% slower when normalizing the chunk names), but for tasks with more chunks in DM and/or more calls to run ACT-R one may find the savings from turning it off to be more significant.

Zbrodoff task

In addition to being written in a different style, there is also additional information needed for recording the data for the task since the block and addend need to be recorded along with the response and time. To keep track of that extra information this task creates a structure/class (Lisp/Python) to hold the information. Then, to make the running easier it creates an instance for each of the trials to present in advance with the relevant presentation information, and to make it easier to stop and resume the task it records that information in a global variable so that it is not lost if the task code or model stops unexpectedly.

The **zbrodoff** task is written using the event-driven approach, and there are basically two significant differences between it and the other tasks you've seen in the tutorial to implement that. The first difference between how this experiment runs relative to the others you've seen is that it only needs to call run once to run all of the trials of the task. That call appears in the collect-responses/collect_responses function and runs the model for up to 10 seconds per trial to be presented (which should be sufficient time to perform all of the trials). The advantage of only calling run once, along with recording the trials to present and the collected data in global variables, means that if the task is interrupted then it can be resumed simply by calling the collect-responses/collect_responses function again and the data will still be available when it is done (just calling run will not work since the collect-responses/collect_responses function removes the output-key monitors so they need to be restored). The other difference is that instead of presenting the information to the model from a loop in the code that's also calling run (as previous experiments have

done) it does the update when the model responds – the output-key event is the trigger to perform the update not a model that stops running when a trial is complete. In this task that just requires the function that is monitoring output-key to handle the presentation for the next trial in addition to recording the current response and time. Essentially, the code is still doing the same operations, but the difference is in where the "looping" happens – in an explicit loop in the code or a loop driven by the actions of the participant. While it may seem a little more complicated to implement, the savings from writing the experiment in this style are often worth the effort because it simplifies creating the model (since it doesn't have to "know" when to stop running) and getting it to do the task properly (since you can pickup where it left off in the task when there's a problem instead of having to start over).

Monitoring function notes

As discussed with the unit 2 code, functions that are called as monitors are evaluated in separate threads and may require additional protection on changes to items which are also accessed outside of that function. What wasn't mentioned at that time is that when ACT-R is running to generate those actions there is some protection because the actions performed in the model are not evaluated in parallel – the events are evaluated one at a time. Therefore, when a monitor for output-key is called because the model pressed a key, you do not need to worry about threading issues between the monitoring function and other functions which are called as events by ACT-R or the code which called run, but that is still a potential problem when a person is performing the task since the person could press the key in parallel with any of the code which is running the task while the monitor is active.

However, there is another issue to consider, and that is whether any functions that are used in the monitoring function themselves have threading issues. In particular, when running a model you need to be careful about the ACT-R commands that are used since even though ACT-R will only evaluate one event at a time ACT-R itself is still "running". Most of the ACT-R commands presented in the tutorial are safe to use while ACT-R is running, in particular all of the AGI commands for creating and manipulating windows are safe, but commands which run ACT-R cannot be used while it is already running and the commands for resetting and reloading a model cannot be used while it is still running. If you are using other ACT-R commands which are not described in the tutorial you will want to verify with the reference manual that they are safe for use while ACT-R is running, and if you're using the Lisp version it is strongly recommended that you not call any undocumented ACT-R functions since you won't know if they are safe to use.