

# ACT-R Environment Manual

*Working Draft*

Dan Bothell

# Table of Contents

Table of Contents.....	2
Preface.....	3
Introduction.....	4
Running the Environment.....	6
Environment Overview.....	9
Current Model.....	11
Model.....	13
Control.....	15
Inspecting.....	30
Tracing.....	43
History Tools.....	70
BOLD tools.....	84
Miscellaneous.....	100
Window Positions and Sizes.....	105
Extending or Changing the Environment.....	106
Available Environment Extras.....	108
Standalone Environment Tools.....	113
Advanced Issues.....	115

## **Preface**

This document is a work in progress to describe the operation of the ACT-R Environment. The content is accurate, but may not cover all the components of the Environment. It may also make reference to sections or other documents which are not yet available. The hope is that although it is not yet complete, this working version will be of some use to ACT-R modelers.

## Introduction

The ACT-R Environment is a GUI written in Tcl/Tk which can be connected to ACT-R to help users with running, inspecting, and debugging models. It also provides a way to display a GUI with which a model is interacting for Lisps which don't have such capabilities, and for the standalone versions of ACT-R (pre-built applications which run without needing to have a Lisp in which to run ACT-R) it also provides some basic text editing capabilities for model files.

This document assumes the user has a basic understanding of ACT-R. [If you do not, then you should probably start with the ACT-R tutorial.] It will focus primarily on running the Environment in conjunction with ACT-R running in a separate Lisp application. All of the same tools are available when running the standalone version, and the documentation on the additional Environment tools included with the standalone version of ACT-R is included in the [Standalone Environment Tools](#) section.

To generate the information displayed by the Environment it uses the same ACT-R commands that are available to the user when using ACT-R from a command line. Thus, for the most part, it does not provide anything new for working with ACT-R. However, because it displays the command outputs in separate windows and updates those windows as the model runs it can be a lot easier to use when debugging, and for some things, like the buffer traces and BOLD response predictions, it displays the data graphically instead of just textually.

## Requirements

There are a few requirements necessary to run the ACT-R Environment:

- You need to have the ACT-R software.
- If you are using Windows or Mac OS X then you can use the pre-built Environment application which is included with the distributions.
- If you are not using Window or Mac OS X or if you prefer not to use the pre-built Environment application, then you will need a wish interpreter to run the Environment from the Tcl/Tk source code which is included in all the ACT-R software distributions. It should run in any Tcl/Tk version 8.1 or newer, but 8.3.4 or newer is recommended.
- The Lisp in which you run ACT-R must have the ability to open and communicate via TCP/IP sockets, it must have multiprocessing capabilities, and there must be an appropriate ACT-R interface for those Lisp capabilities. The ACT-R 7 distribution includes interfaces for Allegro Common Lisp, LispWorks, Clozure Common Lisp, CMUCL, and SBCL. If your Lisp has the necessary capabilities but is not one of those, it is possible to extend the ACT-R interface to include it (see the Adding Lisp Support section under advanced topics).
- You will need to have TCP functionality on your machine, but it's not necessary to have an active internet connection as long as you are going to run the Environment and ACT-R on the same machine.

## Running the Environment

This section assumes that you are loading ACT-R into a supported Lisp application. If you are using a standalone version of ACT-R then you should consult the documentation which came with it as to how to run that. The Environment runs as a separate application from the Lisp in which ACT-R is running. It communicates with ACT-R via a TCP/IP socket connection and can be run on the same machine or a different machine than the Lisp running ACT-R. It is also possible to have more than one Environment connected to the same ACT-R session. In this section the assumption will be that there is one Environment connection occurring on the same machine which is running the Lisp with ACT-R. For details on other situations (remote connections and multiple concurrent Environments) see the advanced sections.

Here are the standard steps to follow to run ACT-R with the Environment on all systems (there is a shorter alternative for some Lisp and OS combinations covered later):

1. The first thing to do is load ACT-R into your Lisp application (see the ACT-R reference manual for details).
2. Start the Environment application.
  - a. If you are using one of the pre-built applications (“Start Environment.exe” on Windows or “Start Environment OSX” on Macs) run it like you would any other application. Note: if you are using Mac OS 10.8 and get an error dialog indicating that the file is "damaged and can't be opened" the issue is probably due to permissions. Open your System Preferences and under "Security & Privacy" set the "Allow applications downloaded from:" to Anywhere, and then try running it again. If it successfully runs, then you can change your preferences back to a safer setting.
  - b. If you are using Linux/Unix or do not want to use the prebuilt applications then you can run the Environment from the source files. To do so you must have Tcl/Tk installed. Then, all you need to do is either execute the

starter.tcl script found in the environment/GUI directory or you can use wish (the Tcl/Tk interpreter) to run it i.e. "wish starter.tcl".

3. Once the Environment is running you should have seen a “Powered by ONR” splash screen briefly and then have a window titled “Control Panel” open which says “Waiting for ACT-R” at the top.
4. To connect ACT-R to the Environment you need to call the start-environment function in Lisp.
5. That should result in another splash screen opening briefly to show the ACT-R version information and then the “Waiting for ACT-R” message should be removed from the “Control Panel” window and several buttons should appear there instead.

Once the buttons appear the Environment is ready to use. When you are done using the Environment you should close the connection from the Lisp by calling the stop-environment function. That should put the “Control Panel” back into the waiting state. At that point you can close the Environment application if you want, or you can leave it running to connect to again when you need it. You should only close the Environment application (the “Control Panel” window) when it is in the waiting state.

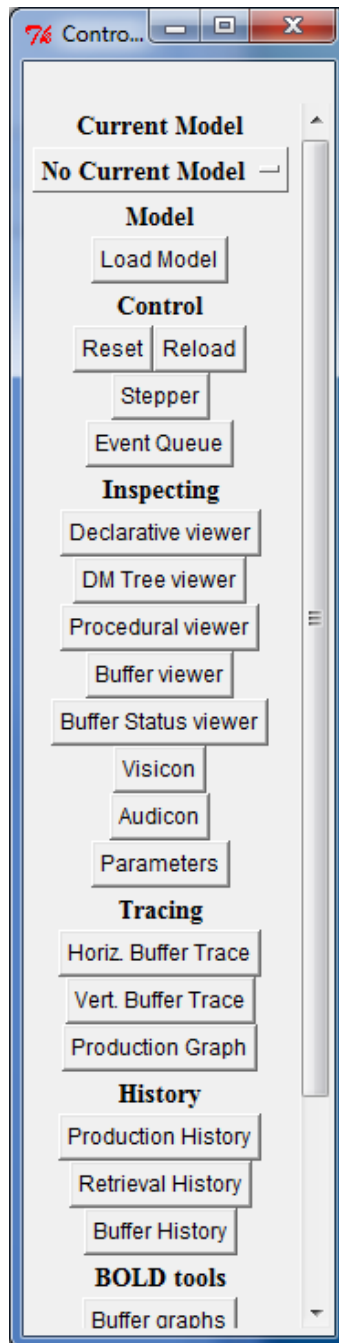
If you are using LispWorks or Allegro Common Lisp under either Mac OS X or Windows or are using Clozure Common Lisp under Mac OS X, Windows, or Linux, then you may be able to replace steps 2-4 listed above with a single step. After loading ACT-R you can call the function run-environment instead of the start-environment function. That should automatically run the appropriate Environment application and then initiate the connection between ACT-R and the Environment, but run-environment may not work on all machines for a variety of reasons. If the Environment application does not start, then you should use the standard instructions described above. If the Environment application is slow to start and there is an “Unable to connect” message displayed in the Lisp you can ignore those and wait for it to try again. If that happens regularly and you want to avoid the “unable to connect” warnings, then you can increase the delay before ACT-R attempts to

connect to the Environment. The delay can be provided as an optional parameter to run-environment indicating how many seconds to wait before connecting. The default delay is 5 seconds. A longer delay may be necessary in some cases, and on some machines a shorter delay will work just fine.



## Environment Overview

Once it is connected the Control Panel should look similar to the image below. The appearance may vary based on which operating system you are using, but all of the same components should be available.



The Control Panel consists primarily of buttons which open the tools that it provides. Those buttons are grouped into sections based on their functions. Each section of the Control Panel and its buttons will be described below. Note that there is a scroll bar on the right of the control panel and some of the buttons may not be visible without scrolling the window or making it larger.

One thing to note about the Environment is that it was originally designed to work with a single ACT-R model at a time. It has recently been extended to allow all of the tools to work when there are multiple models currently defined within the default meta-process (it still does not work with multiple meta-processes). Details on using the Environment with multiple models can be found in the [Current Model](#) section below. That change to allow support for multiple models is still a work in progress, so if you encounter any problems with the Environment, either with a single model or multiple models, please contact Dan ([db30@andrew.cmu.edu](mailto:db30@andrew.cmu.edu)) with details.

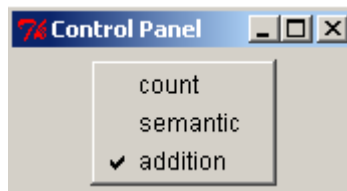
## Current Model

The Current Model section has only one item which is a button that serves two purposes. The text shown on the button displays the name of a currently defined model or the text “No Current Model” if there is no model currently defined. By default, the Environment assumes that there will only be one model defined at a time. In the single model mode the inspecting tools will always work with the currently defined model, if there is one. Thus, if a different model is loaded to replace the current one the open inspector windows will then begin working with that new current model.

## Multiple Models

It is possible to use the environment with multiple simultaneously defined models. To enable that, the “Allow the environment to work with multiple models” option needs to be set (see the [Options](#) section). When there are multiple models simultaneously defined the button will still show the name of a single model. With multiple model support enabled, when an environment tool is opened it will always be associated with the model that was current when it was initialized and that model name will be shown in the title of the tool.

To change which model is current, press the button which shows the current model. That will bring up a menu with all of the currently available models in it. That would look like this if there were three models named count, addition, and semantic defined:



The one with the checkmark next to the name is the one currently being used, and clicking on one of the other names will switch that model to the current one in the Environment.

Because the inspector tools are associated with specific models when there are multiple models defined those tools will become unusable if a model is no longer available and

trying to use one may result in warnings or problems within the Lisp running ACT-R. There is an options setting which will cause the inspector tools for models which are no longer available to be closed automatically, but that is not always desirable for a couple of reasons and by default the option is disabled. Probably the strongest reason for not enabling the switch is that when clear-all gets called it deletes all the current models which means loading a model file which contains a clear-all will result in closing all the inspector windows. The Reload button in the environment is sensitive to that and if that button is used to reload a model it will not close the inspector windows if the option is enabled, but any other reloading of the model file will e.g. calling the reload function from Lisp.

## Model

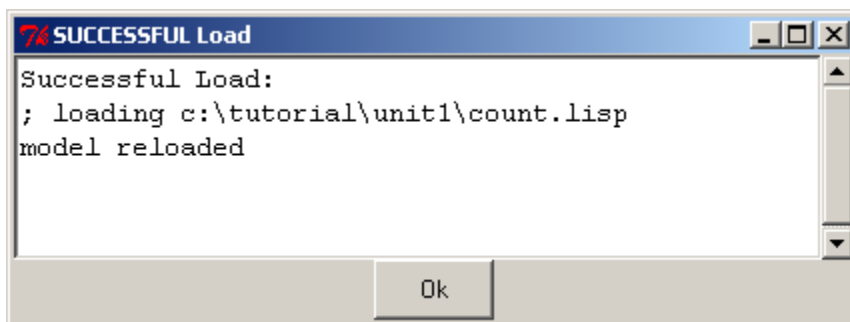
The model section contains controls for working with model files. When using the Environment with a Lisp running ACT-R it will have only one button which is described in the next section. The standalone version of the environment has some additional buttons which provide access to a simple text editor. Details on those buttons can be found in the [Standalone Environment Tools](#) section later in the manual.

### Load Model

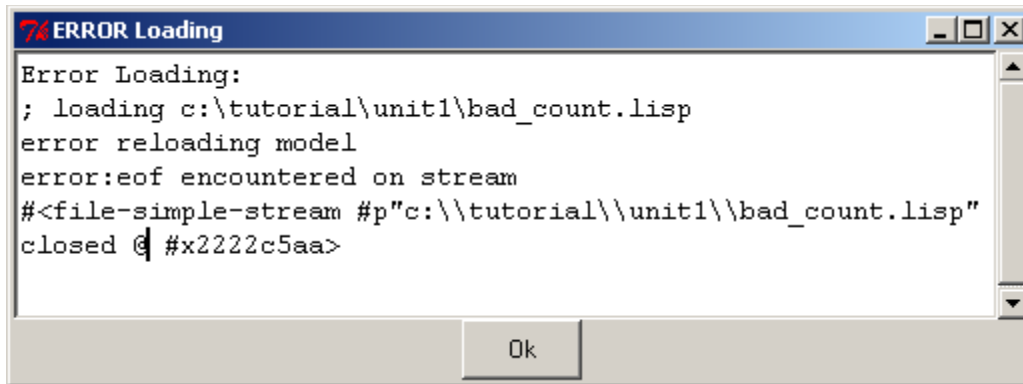
The “Load Model” button can be used to load a model file into Lisp. This button will open a file selection dialog and the file which is chosen will be loaded into the Lisp running ACT-R.

If the compile definitions option of the Environment (described under options in the miscellaneous section) is enabled, then that file will be compiled and loaded.

After loading the file a dialog window will be opened to show any output, warnings, or errors which occurred during the load. If the load completed successfully then it will say “Successful Load” at the top of the dialog like this:



If there is a problem it will indicate that by saying “Error Loading” at the top like this:



In either case, the “Ok” button on the resulting dialog should be pressed to close the dialog before doing anything further with the Environment.

This button will only work if the Environment is running on the same machine as the Lisp running ACT-R. Also, if the Lisp you are using has a menu or other easy to use mechanism for loading and compiling files then you should use that instead of this button because that is likely to provide much better handling of errors or other unusual circumstances.

## Control

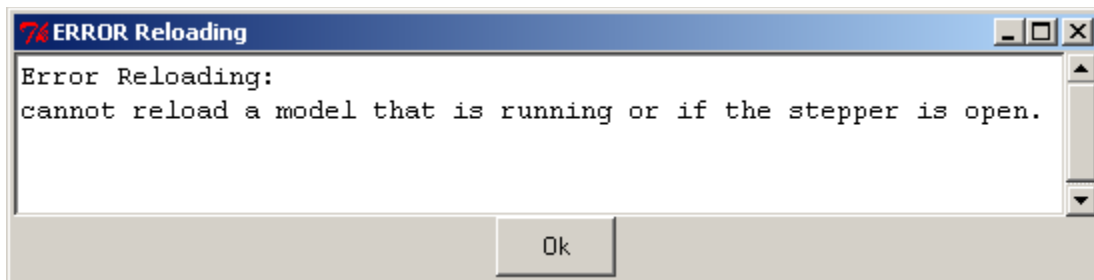
The control section contains buttons for stepping through the trace of running models, for viewing the events that are scheduled, and for restoring the models to their initial conditions.

### Reset

The “Reset” button is used to return models to their initial state. Pressing the “Reset” button is equivalent to calling the ACT-R **reset** command.

### Reload

The “Reload” button is used to load the last model file which was loaded into the Lisp again (a model file is defined as a file which calls the ACT-R clear-all command at the top-level in the file). Pressing the “Reload” button is equivalent to calling the ACT-R **reload** command. The “Reload” button will not function if ACT-R is currently running or if the stepper tool is open and it will open a dialog to indicate that if it happens:



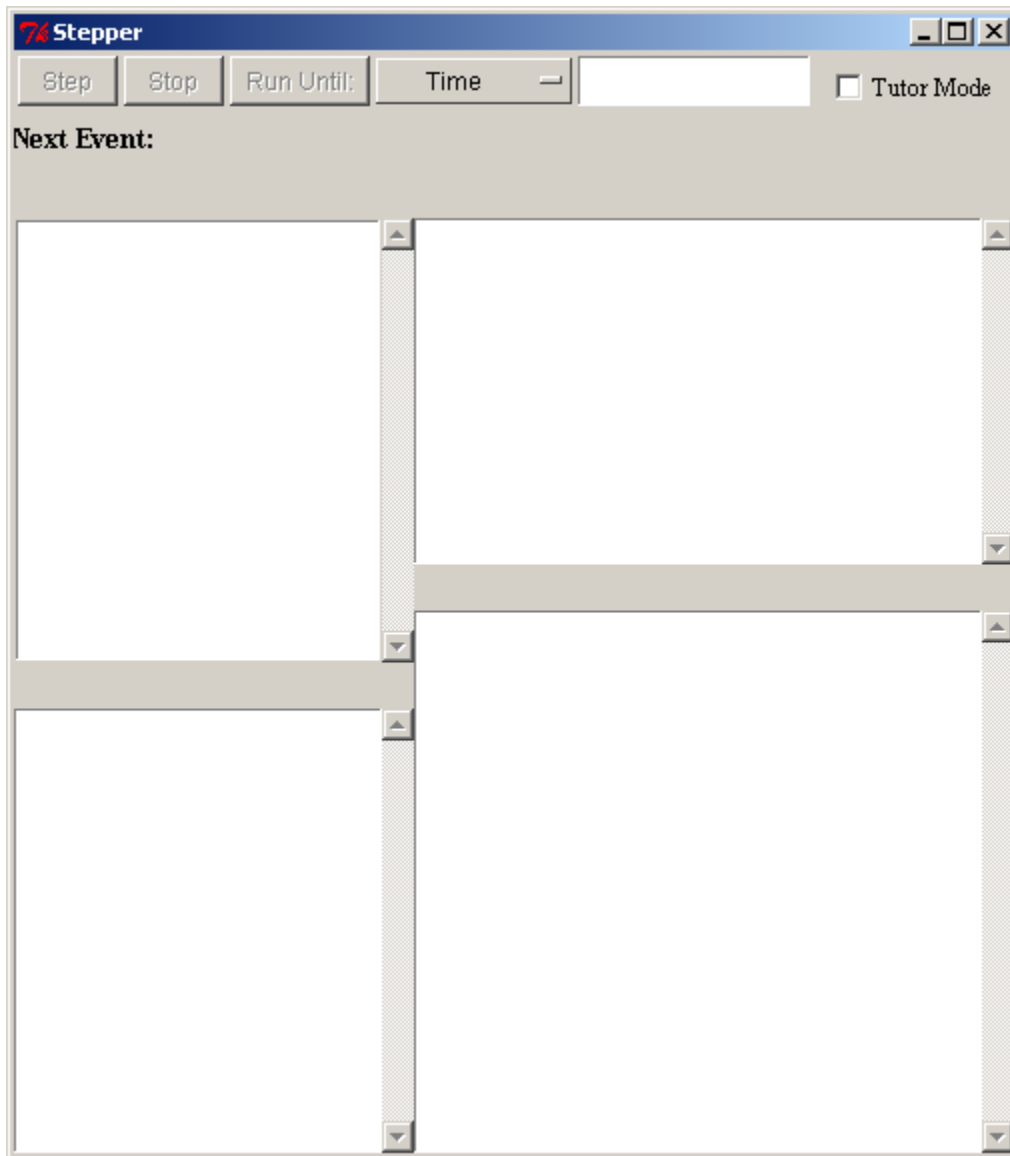
That dialog should be closed by pressing the “Ok” button before continuing with any of the other tools.

### Stepper

The “Stepper” button is perhaps the most useful tool in the Environment. When it is pressed it will open the stepper if it is not already open. If it is open, then pressing this button will bring it to the front – there can be only one stepper open in the Environment

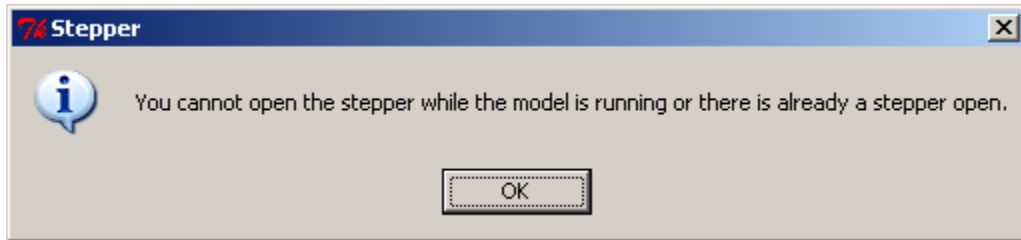
even if there are multiple models defined. The stepper is used to “step” ACT-R through its execution one event at a time.

The stepper looks like this when you first open it:



To use the stepper you should have it open before you start the model running. If you try to open it while the model is currently running it will display a dialog like this indicating that it is unavailable:





Thus, the proper way to use it is to open the stepper and then call the appropriate function to run the model from Lisp. If you close the stepper while the model is running the model will continue to run to its natural completion from that point.

When the stepper is open it will pause ACT-R before every event that will be printed in the trace. The event which is about to occur will be displayed in the stepper after the “Next Event:” heading, and for some events, additional information will be displayed in the windows below that. ACT-R is suspended at that point and the modeler may use any of the Environment’s inspection tools at that time to investigate the currently defined models. The model will execute the displayed event once the user either presses one of the buttons along the top of the stepper or closes the stepper. The details of what the buttons do and the additional information available for some events will be described below.

Note that when the stepper is initially opened the three buttons which will advance the system are disabled. They only become enabled and useable while ACT-R is running.

## Step

The “Step” button allows the system to continue operation. It will execute the “Next Event” which is displayed and continue to run to the next event which will be handled by the stepper, if there is one. This is the button that is most often used with the stepper – it “steps” the system through its operation one event at a time.

## **Stop**

The “Stop” button will stop the current run after it executes the event displayed. There are two important things to note about the “Stop” button. First is to emphasize that the “Next Event” shown will be executed by the model when the button is pressed – there’s no clean way to prevent an event from occurring once it has been stepped to. The other is that it only stops the current run. If the system is being run from Lisp code which contains multiple calls to one of the ACT-R running functions then the system may continue to be run by the next call from that code after the “Stop” button is pressed i.e. the stepper’s “Stop” button does not affect the Lisp code which may be providing an experiment or other interface for the model or models.

## **Run Until**

The “Run Until:” button works in conjunction with the two interface items to its right which are a selection menu and a text entry box. Pressing the “Run Until:” button will execute the “Next Event” and allow the model to run without being paused by the stepper until the condition specified by the combination in the selection menu and text entry box occurs.

There are three options for the selection menu: time, production, and module. That choice determines what should be entered into the text entry box. If an invalid value is entered into the text entry box when “Run Until:” is pressed then a warning will be printed in the trace indicating the issue and the system will be paused at the next available event as if the “Step” button had been pressed. Here are the details for specifying each of the options for “Run Until”.

### ***Time***

When “Time” is selected the text entry box should have a number entered into it which represents the ACT-R time when the stepper should next pause the system. If that time has already passed then the model will pause on the next event as usual. If there is no event at that specific time, then the system will be paused at the first event after that time.

## ***Production***

When “Production” is selected the text entry box should have the name of a production entered into it. The system will then be allowed to run until the next time that named production is either selected or fired. If there are multiple models defined then the system will step until the next time any of those models selects or fires a production of that name.

## ***Module***

When “Module” is selected the text entry box should have the name of a module entered into it. The system will then be allowed to run until the next event which is generated by that named module. As with the production option, if there are multiple models defined then it will step to the first event generated by the specified module regardless of which model generated that event.

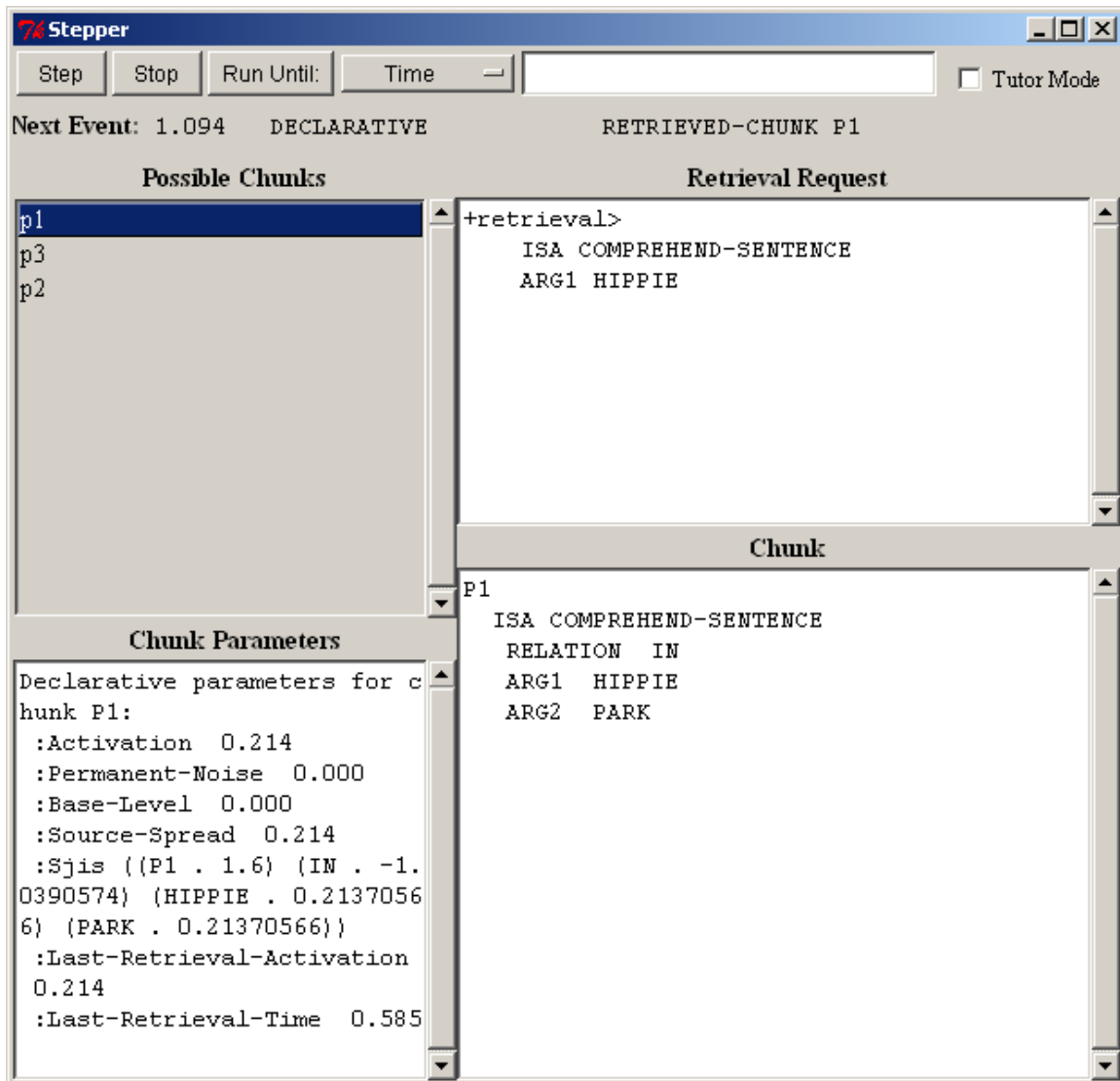
There are two things to be careful of with the module option. The first is that the module’s actual name is required. Some modules have the same name as their buffer, like goal and imaginal, but others do not, for example the module which controls the manual buffer is named :motor. That brings up the other issue. Some modules use a keyword for a name, but the prefixed colon doesn’t actually show up in the trace when printing the module’s name – the :motor module’s events just shows “motor” in the trace. To see the list of all the modules’ true names you can use the ACT-R **all-module-names** command.

## **Additional Information**

The bottom portion of the stepper will currently show detailed information for three specific events within a model: the declarative module’s retrieved-chunk, the procedural module’s production-selected, and the procedural module’s production-fired. For all other events the lower portion of the window will be blank.

### ***Retrieved-chunk***

When a retrieved-chunk event occurs the stepper will fill in the lower boxes like this (taken from a run of the fan model from unit 5 of the tutorial):



The upper-right pane, labeled “Retrieval Request”, shows the request which was made to the declarative module for which this chunk was retrieved.

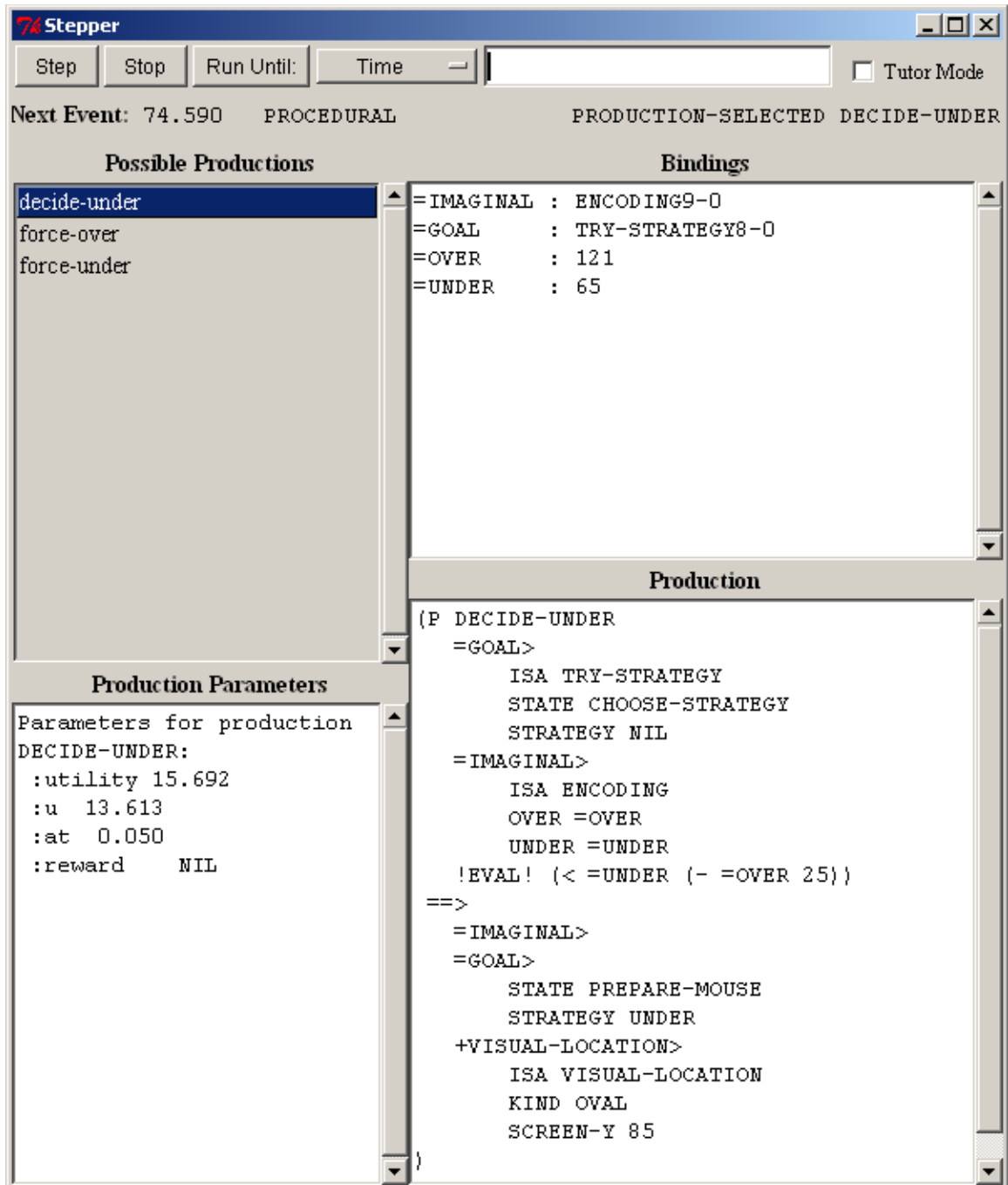
The upper-left pane, labeled “Possible Chunks”, shows the list of chunks which matched the request. They are ordered based on their activations with the highest activation, and thus the chunk which is retrieved, at the top. Selecting a chunk in this list will cause the lower two panes to be filled with the information appropriate for that chunk.

The lower-right pane, labeled “Chunk”, displays the standard ACT-R printout of the selected chunk. The lower-left pane, labeled “Chunk Parameters”, will be empty unless

the subsymbolic computations are enabled for the model. If they are enabled, then that pane will display the declarative memory parameters for the selected chunk as reported by the **sdp** command.

### ***Production-selection***

When a production-selection event occurs and the “Tutor Mode” box is not checked (see [below](#) for details when it is) the stepper will look like this (taken from a run of the building sticks task in unit 7):



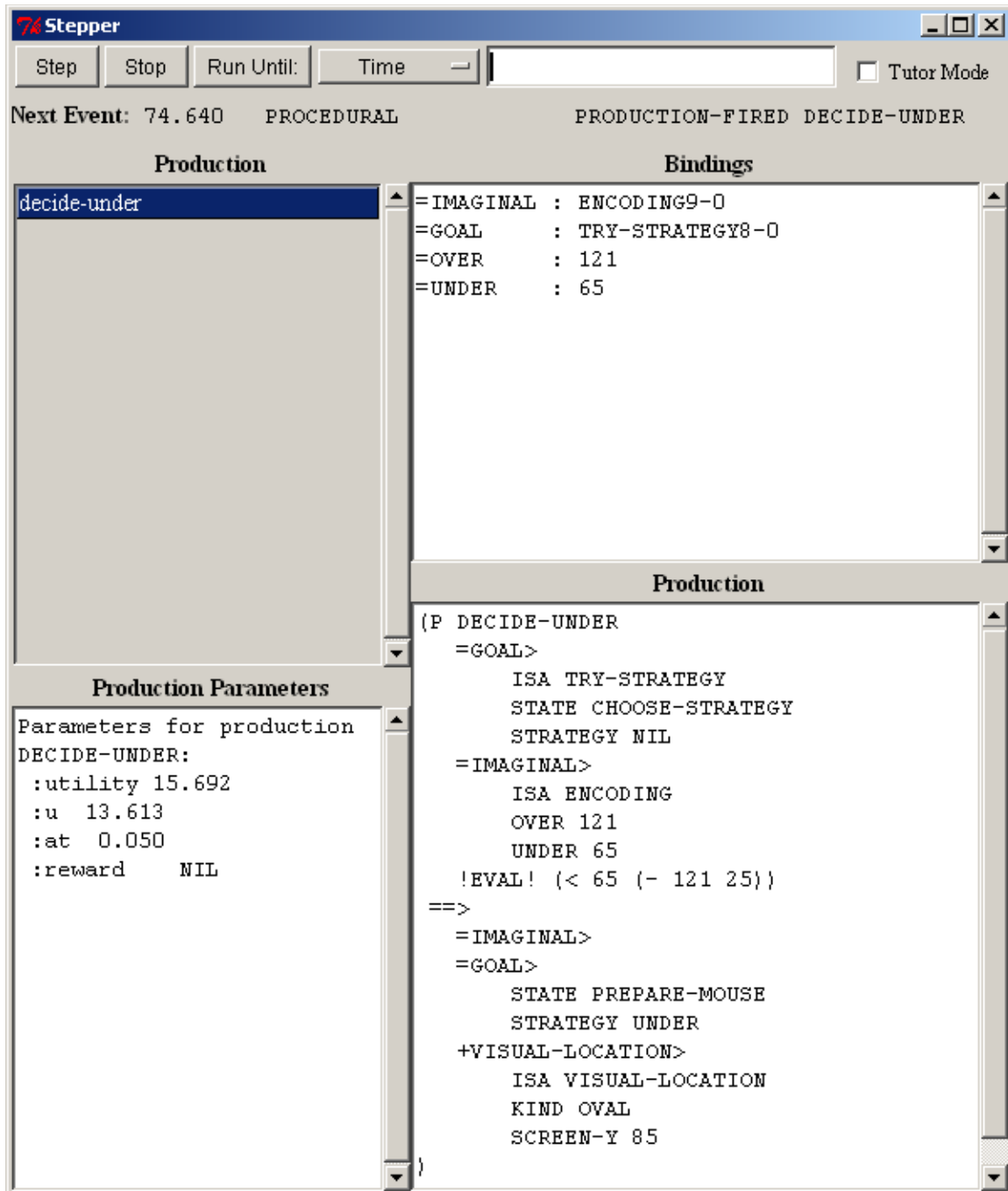
The upper-left pane, labeled “Possible Productions”, shows the list of productions which matched the current state during the last conflict-resolution event. They are ordered based on their utilities with the highest utility, and thus the production which was selected, at the top. Selecting a production in this list will cause the other panes to be filled with the information appropriate for that production.

The lower-right pane, labeled “Production”, displays the text of the production. The lower-left pane, labeled “Production Parameters”, will be empty unless the subsymbolic computations are enabled for the model. If they are enabled, then that pane will display the procedural parameters for the selected chunk as reported by the **spp** command.

The upper-right pane, labeled “Bindings”, shows all of the variables used in the production and the value that they have while matching the current state.

### ***Production-fired***

When a production-fired event occurs the stepper will look like this (which is the production firing that follows the production selection shown above):



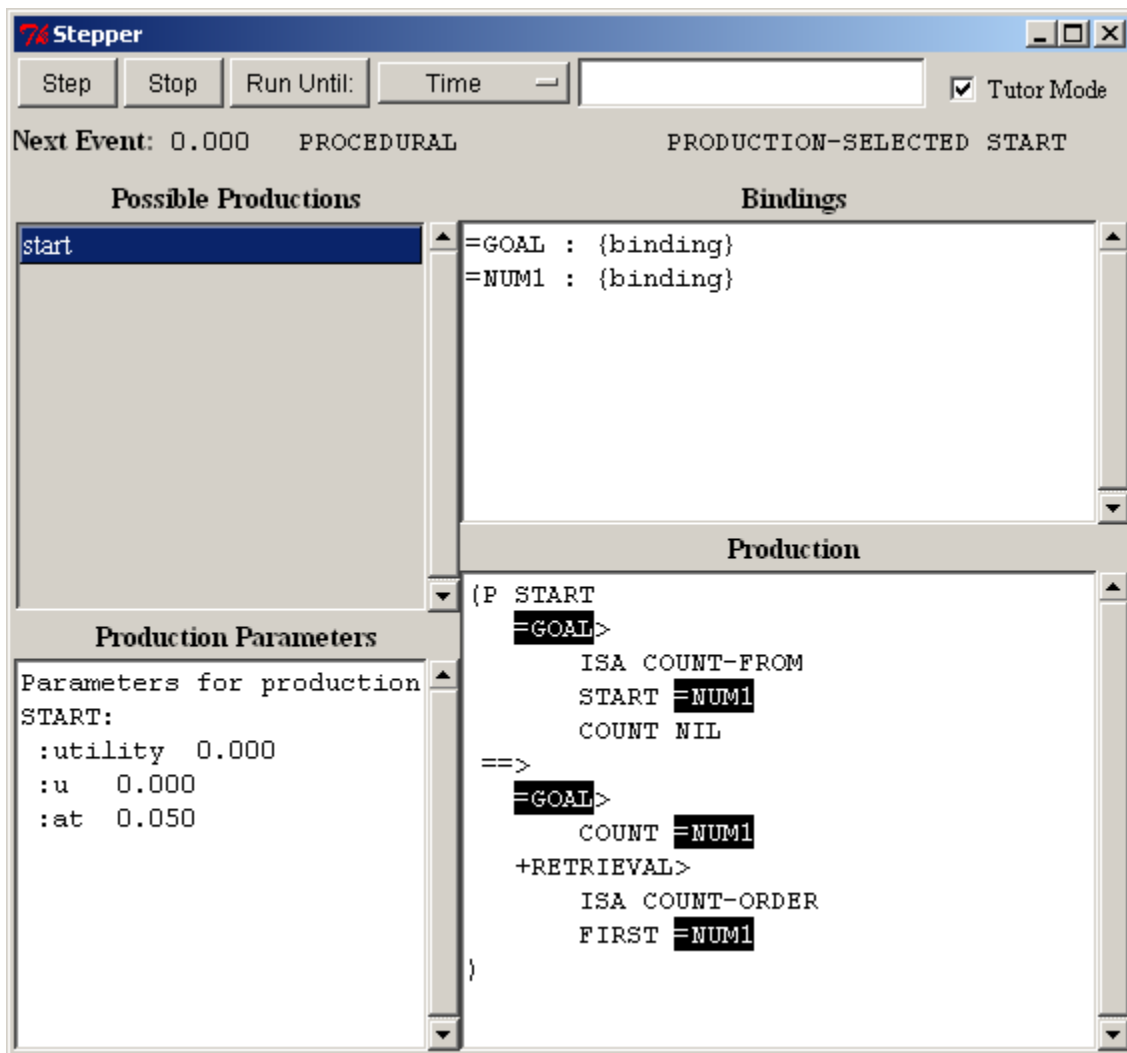
The displays are similar to those for the production-selected event. Now however, only the production which fired is listed in the upper-left pane. The “Bindings” and “Production Parameters” panes display the same information for that production which they did for the production-selected event. The information in the lower-right pane differs in that now it shows the instantiation of the production instead of the production text. The production’s



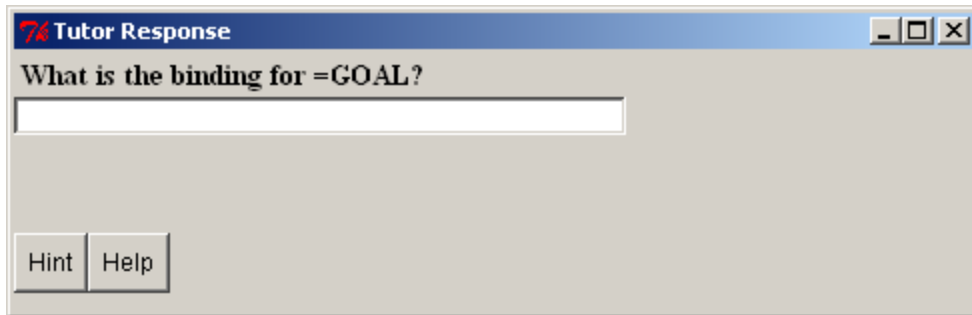
instantiation displays the production text with the variables replaced with their corresponding bindings.

## Tutor Mode

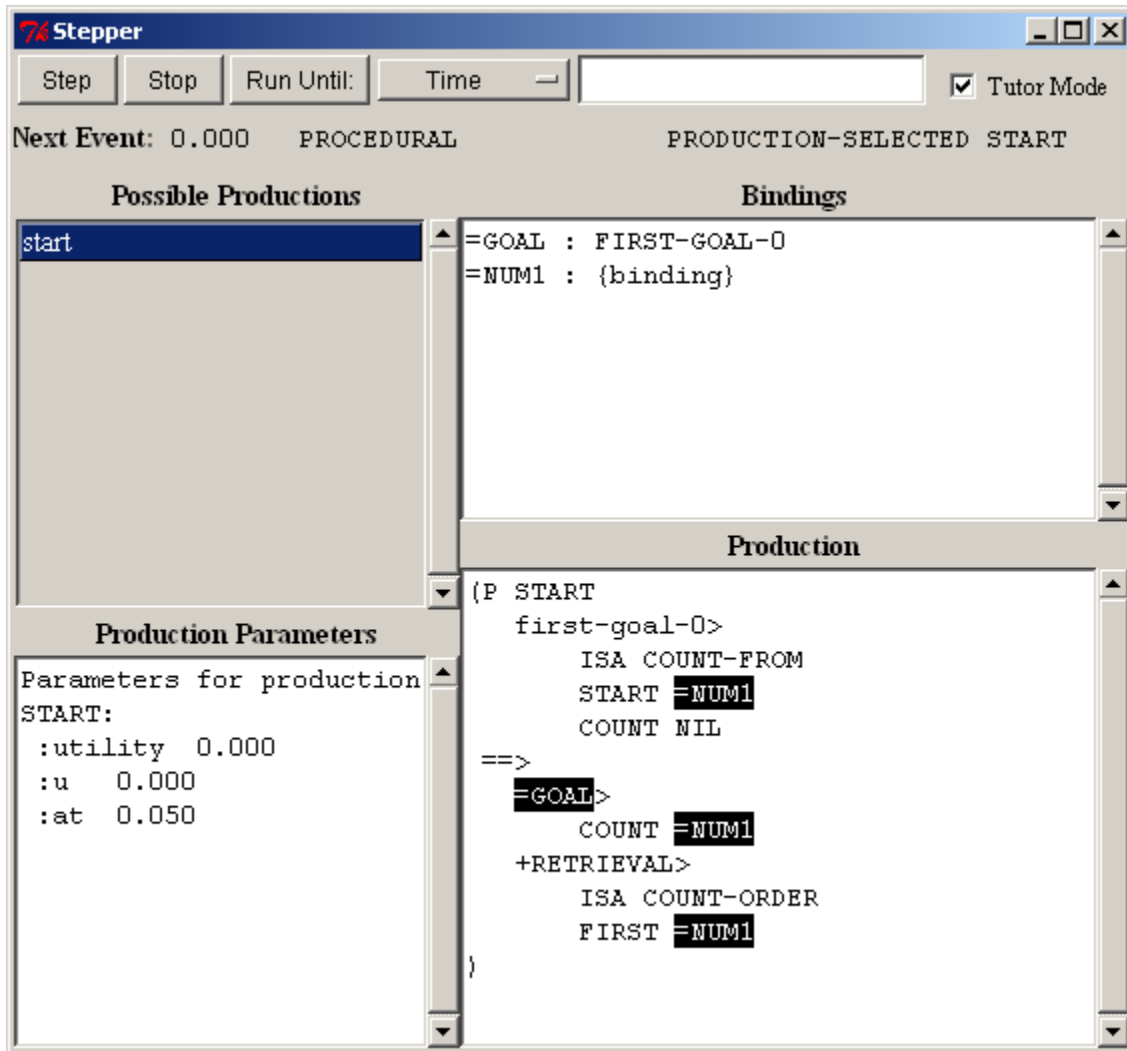
The “Tutor Mode” check box is for use with the models in unit 1 of the ACT-R tutorial. When the box is checked the stepper requires additional interaction from the user to continue past a production-selected event. Instead of displaying the production and its bindings for such an event the production is displayed with all of its variables highlighted and the bindings unset like this (from the count model in unit 1 of the tutorial):



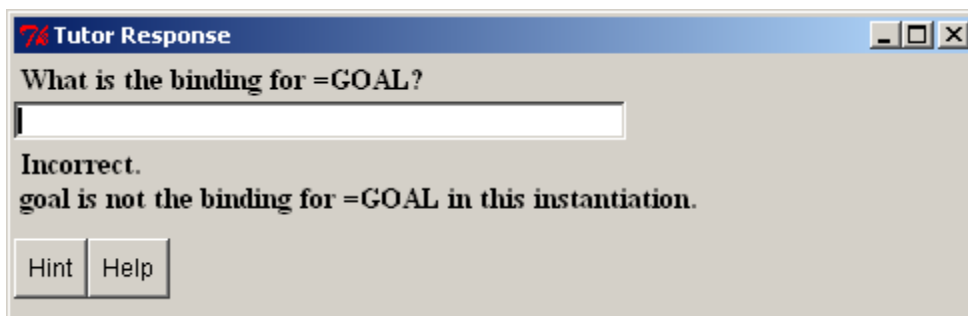
The user is then required to click on each of the highlighted variables and enter the appropriate binding. When one of the variables in the “Production” pane is clicked a new dialog opens in which the binding should be entered:



If the correct value is given then the “Tutor Response” dialog will close and it will replace the variable in the display and the value will be shown under the bindings:



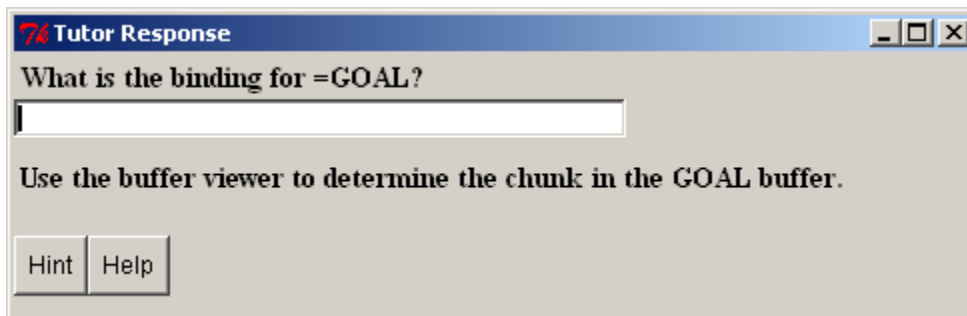
If an incorrect answer is given then it will indicate that the entry is incorrect and wait for another value to be entered:



The two buttons at the bottom of the “Tutor Response” dialog will provide additional information to help the user get the correct answer.

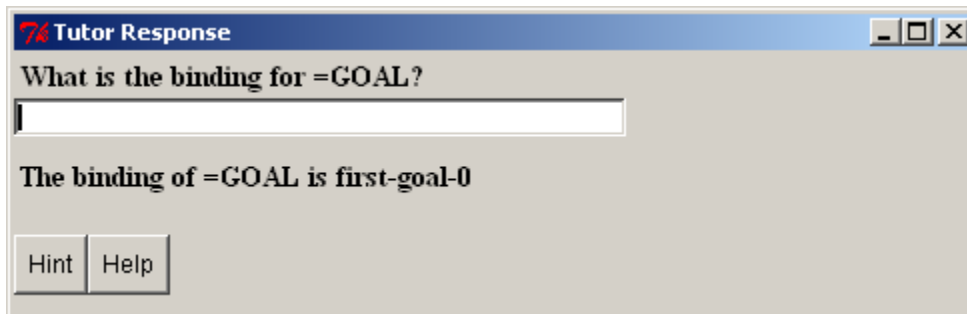
### ***Hint***

Hitting the “Hint” button will display a suggestion in the “Tutor Response” dialog indicating which other tool in the Environment may be used to help find the correct answer:



### ***Help***

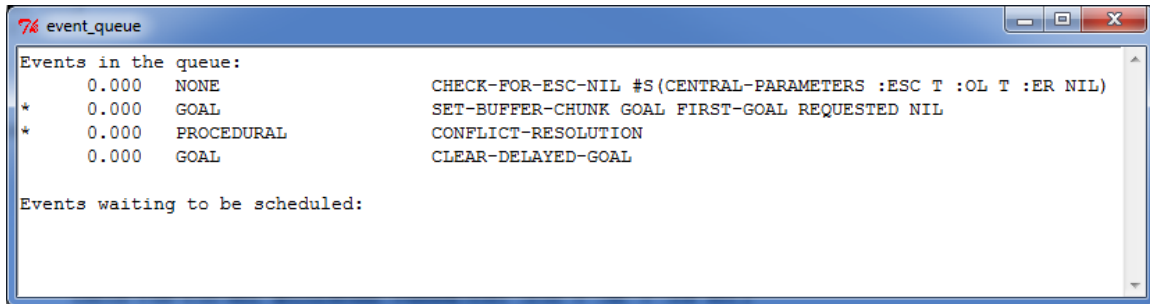
Hitting the “Help” button will print the correct answer in the “Tutor Response” dialog:



## **Event Queue**

The “Event Queue” button is used to see all of the events in the meta-process. When it is pressed it will open the event queue window if it is not already open. If it is open, then pressing this button will bring it to the front – there can be only one event queue window open in the Environment even if there are multiple models defined because there is only

one event queue for all the models. It shows the output from the mp-show-queue and mp-show-waiting commands, which displays all of the events that are currently scheduled and those which are waiting for some other event to trigger their scheduling.



Only the events marked with a “\*” at the start of the line will be displayed in the model’s trace with the current parameter settings. Only those marked events will be displayed in the stepper if it is being used.

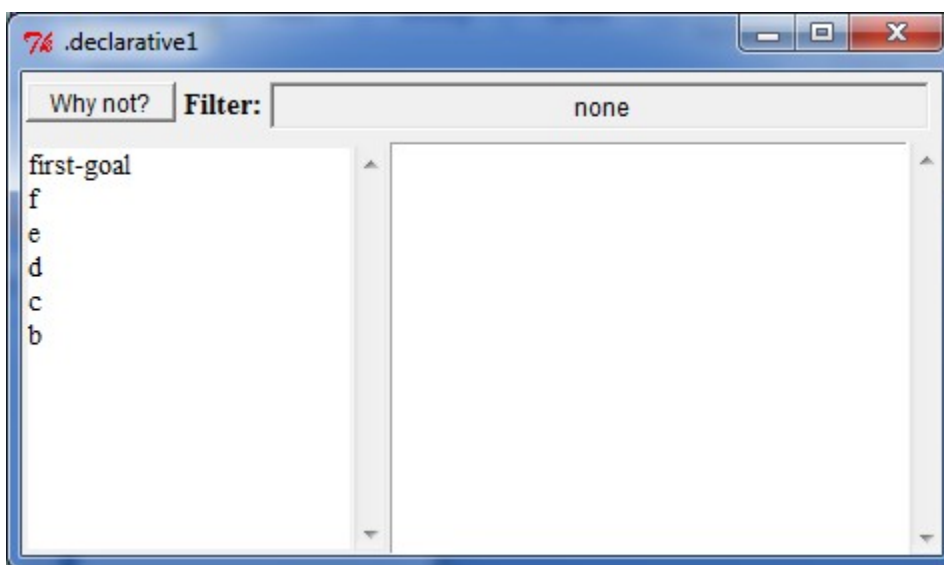
## Inspecting

The inspecting section of the Control Panel contains buttons for viewing detailed information about particular components of the model(s). The inspection buttons can be useful in conjunction with the stepper to view the current state of things before and after a particular event occurs.

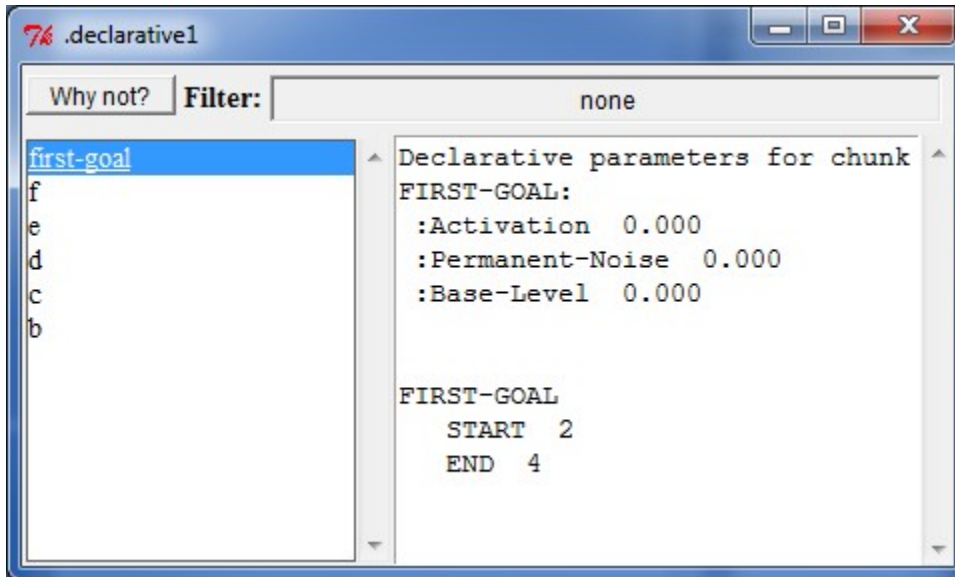
The contents of the windows opened by the inspection tools are automatically updated as the system runs. However, if the system is running at “full speed” i.e. not in real time and without the stepper open, then the windows may not be able to refresh fast enough and the contents could lag behind the running system. Even in real time mode, if there are a lot of inspection windows open they may start to fall behind the current model state.

## Declarative Viewer

The declarative viewer allows the user to inspect the chunks in a model’s declarative memory. Pressing the “Declarative viewer” button opens a new declarative window for inspecting the declarative memory of the currently selected model and any number of such windows may be open at the same time. This is what a declarative viewer will look like (from the count model in unit 1 of the tutorial):



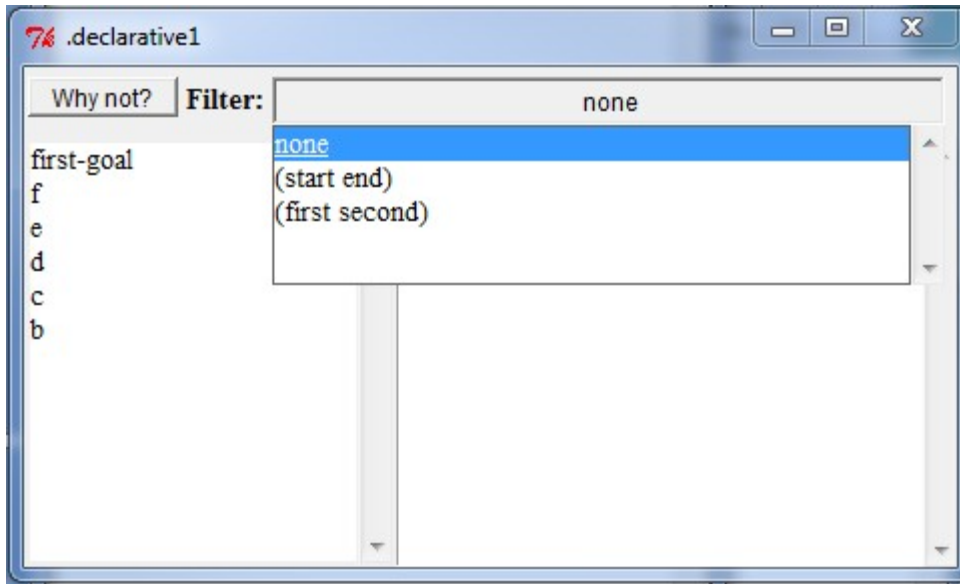
The list on the left shows all the chunks in declarative memory by default (see [filter](#) below for how to change that). Selecting one of those chunks will then cause the details of that chunk to be displayed in the window on the right:



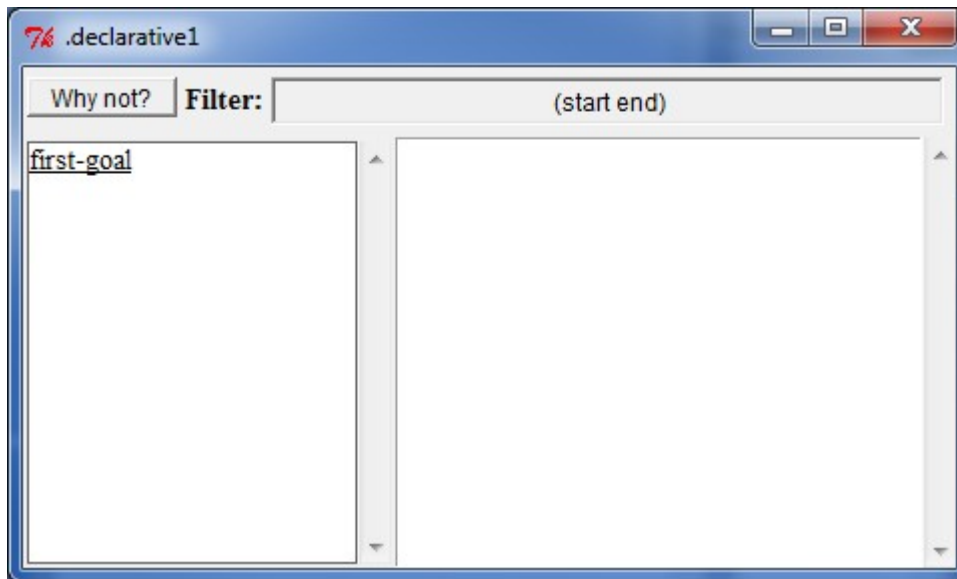
The chunk will be printed in the window, and if the subsymbolic computations are enabled then the window will also show the chunk's parameters as reported by **sdp** at the top.

## Filter

At the top of the window is a filter which allows one to restrict the display to only chunks which have a particular set of slots. The default of "none" means that all chunks in declarative memory will be displayed. To change the filter, click on the box containing the current filter value. That will open a window which contains lists of all the different sets of slots which exist among the chunks in declarative memory. If you pick a set of slots from that list that will be displayed in the filter and only chunks which contain that set of slots will be displayed. Here is the selection window:



Here is the result of selecting the (start end) set of slots:



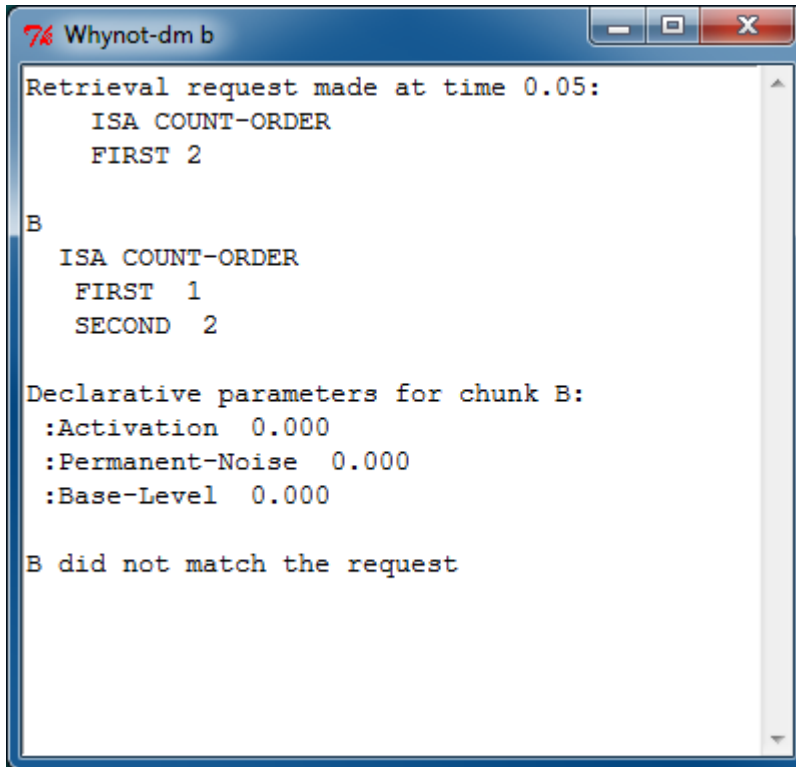
## Why not?

The “Why not?” button at the top of the declarative window can be used to get information about whether a chunk was retrieved or not during the last retrieval request the model made. Pressing the “Why not?” button will open another window and display the results of calling the ACT-R **whynot-dm** command for the currently selected chunk in

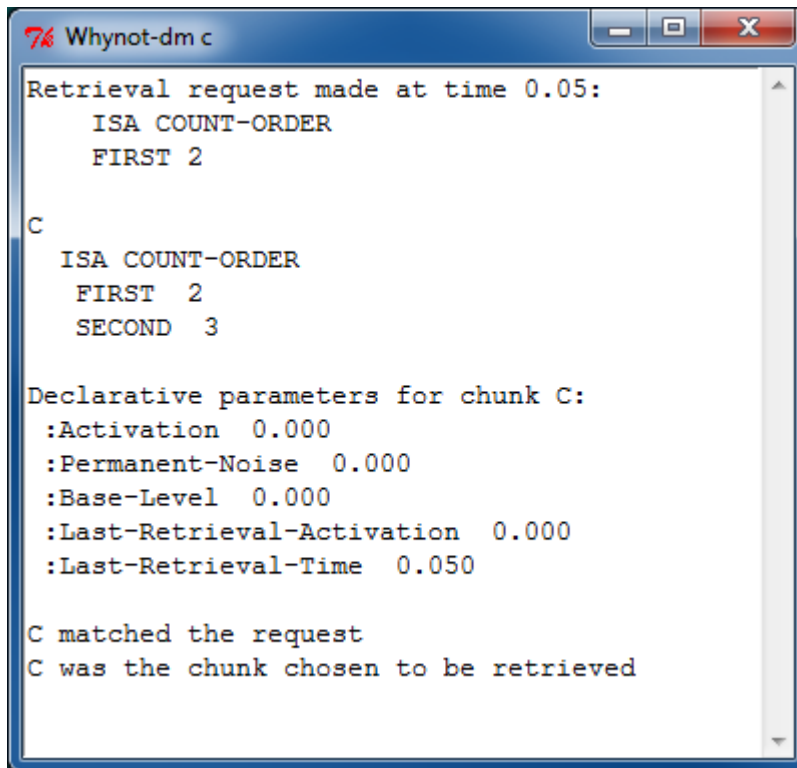


the declarative viewer.

The Whynot window will display the last retrieval request the declarative memory module received and then display the details of the selected chunk and indicate whether or not it matched that request. Here is a Whynot display for the chunk b .1 seconds into the run (after the model makes its first retrieval request):

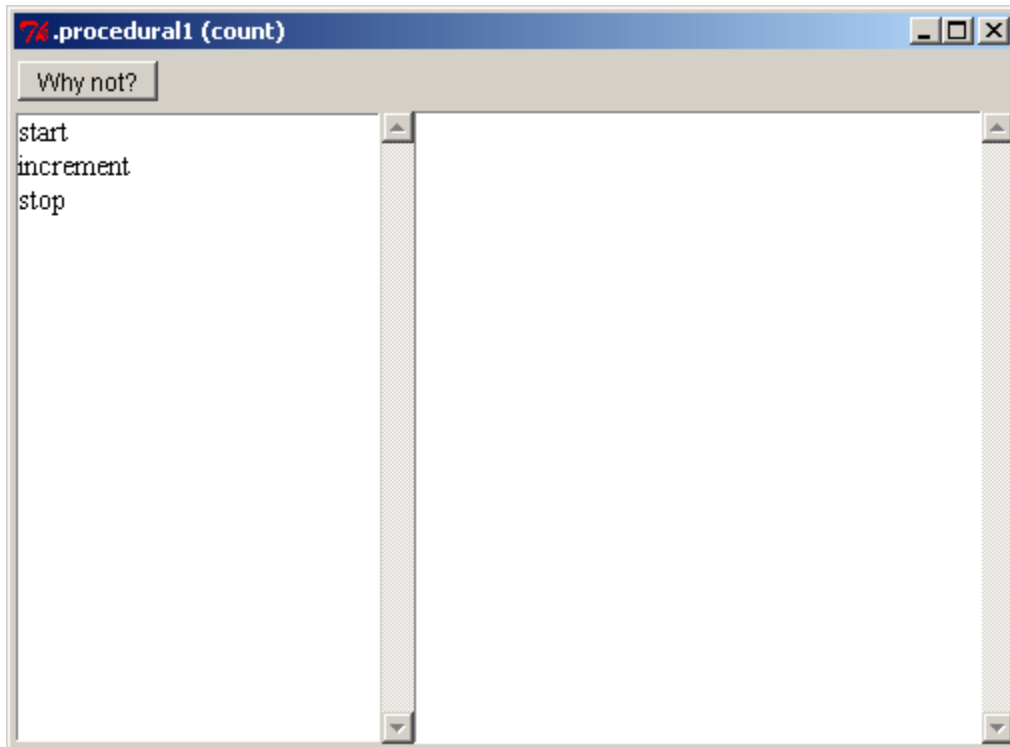


Here is a Whynot window showing results for the chunk c:

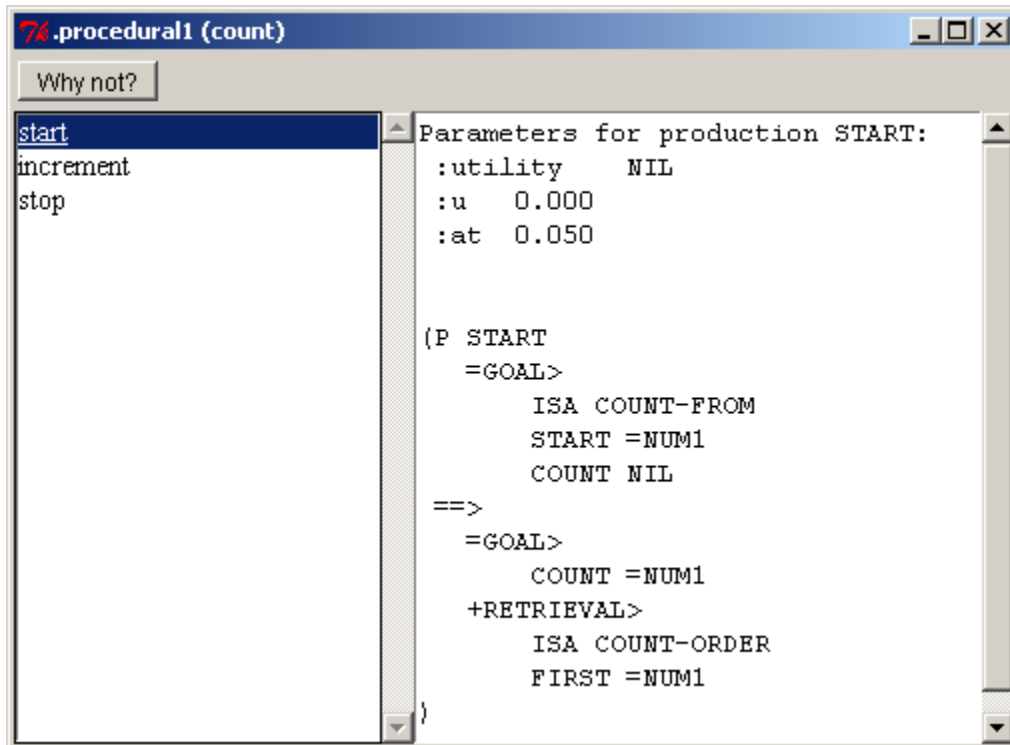


## Procedural Viewer

The procedural viewer allows the user to inspect the productions in the model's procedural memory. Pressing the "Procedural viewer" button opens a new procedural window for inspecting the productions of the currently selected model and any number of such windows may be open at the same time. This is what a procedural viewer will look like (from the count model in unit 1 of the tutorial):



The list on the left contains all of the productions in the model. Selecting one of those productions will cause the details of that production to be displayed in the window on the right:



The production's text will be printed in the window, and if the subsymbolic computations are enabled then the production's parameters from **spp** are displayed at the top of the window.

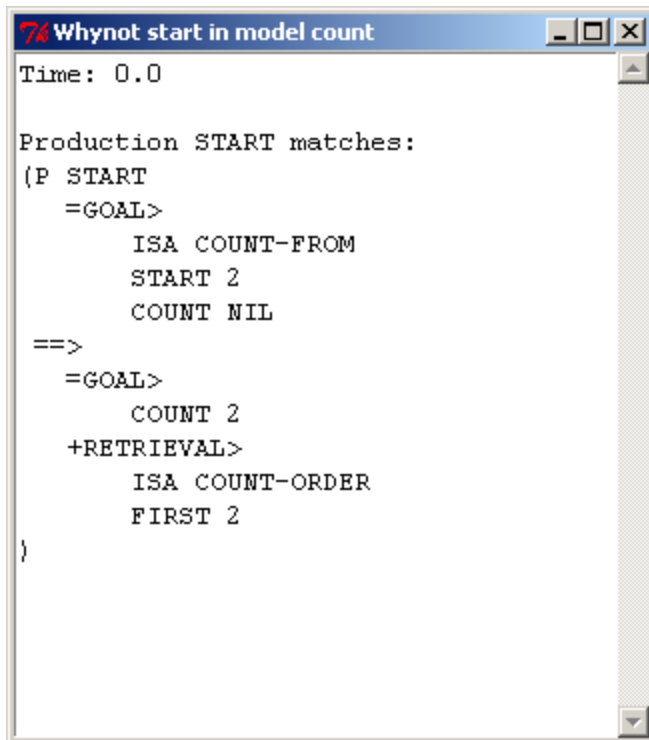
## Why not?

The "Why not?" button at the top of the procedural window is an important debugging aid. Pressing the "Why not?" button will open another window and display the results of calling the ACT-R **whynot** command for the currently selected production in the procedural viewer along with some other relevant information.

The Whynot window will display the time at which the whynot was generated and whether or not the LHS of that production currently matches. If it does match, that will be followed by the instantiation of the production. If it does not match, then it will print the text of the production and indicate the first condition which did not successfully match.

Here is a Whynot display for the start production in the count model at the beginning of

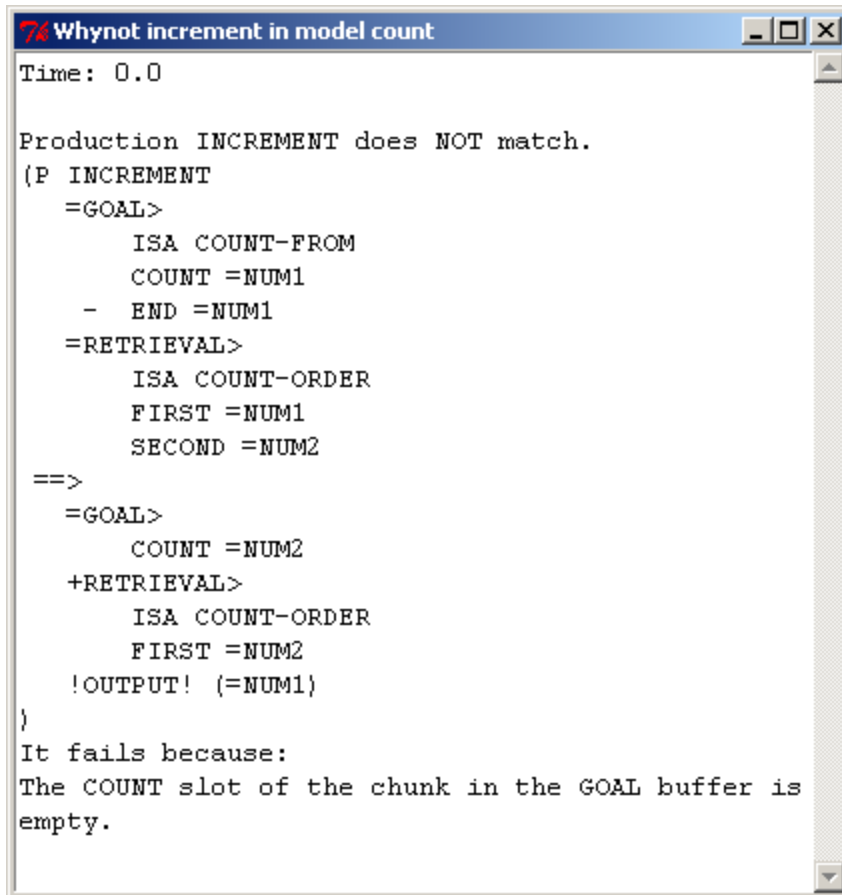
the run when it matches:



```
Time: 0.0

Production START matches:
(P START
  =GOAL>
    ISA COUNT-FROM
    START 2
    COUNT NIL
  ==>
  =GOAL>
    COUNT 2
  +RETRIEVAL>
    ISA COUNT-ORDER
    FIRST 2
)
```

Here is a Whynot window showing the increment production at that same time which does not match:

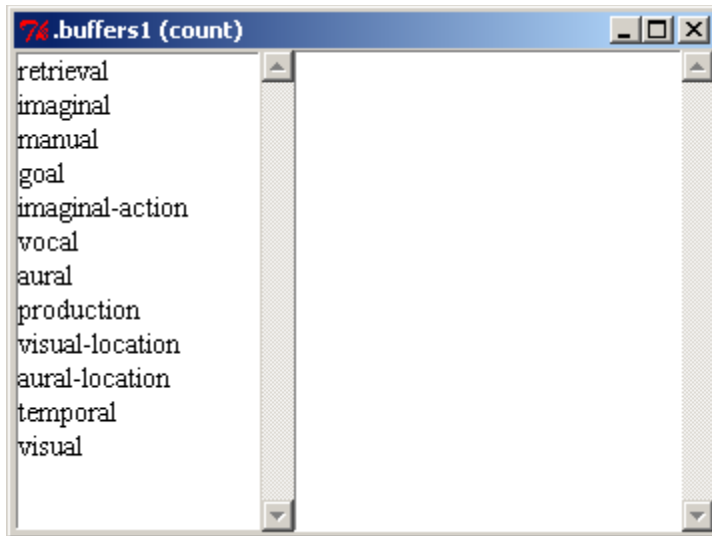


```
Time: 0.0

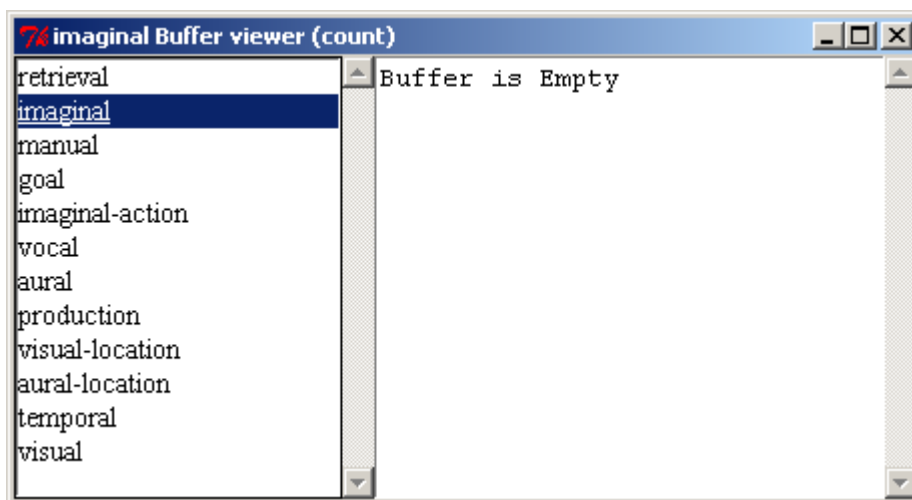
Production INCREMENT does NOT match.
(P INCREMENT
  =GOAL>
    ISA COUNT-FROM
    COUNT =NUM1
  - END =NUM1
  =RETRIEVAL>
    ISA COUNT-ORDER
    FIRST =NUM1
    SECOND =NUM2
==>
  =GOAL>
    COUNT =NUM2
  +RETRIEVAL>
    ISA COUNT-ORDER
    FIRST =NUM2
  !OUTPUT! (=NUM1)
)
It fails because:
The COUNT slot of the chunk in the GOAL buffer is
empty.
```

## Buffer Viewer

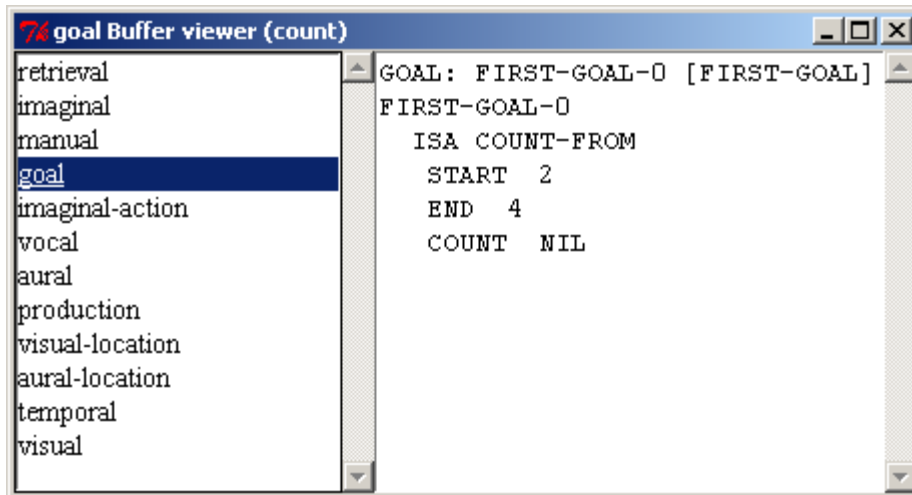
The buffer viewer allows the user to inspect the chunks in the currently selected model's buffers. Pressing the "Buffer viewer" button opens a new buffer window for inspecting the buffer chunks and any number of such windows may be open at the same time. This is what a buffer viewer will look like:



The list on the left shows the names of all the buffers in the model. Selecting a buffer from that list will cause the title of the window to change to show the buffer being displayed and to show the contents of that buffer in the window on the right. If the buffer is empty then it will print that:

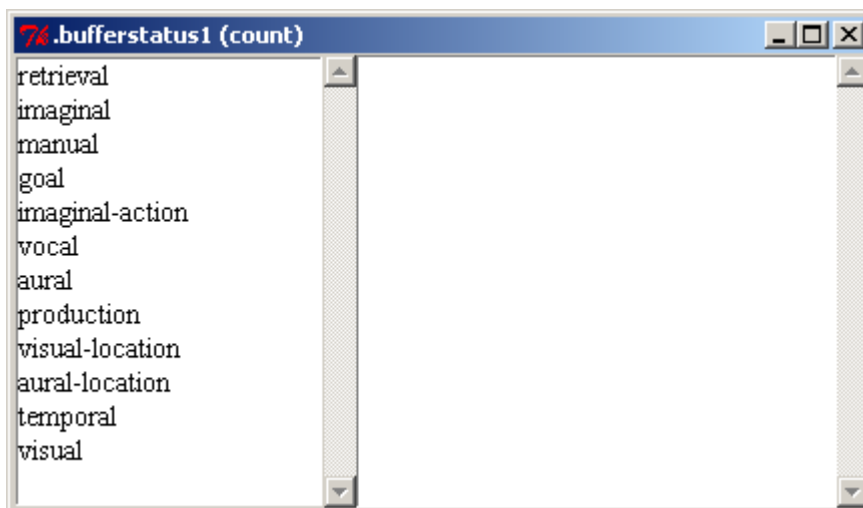


If there is a chunk in the buffer, then that chunk will be displayed using the **buffer-chunk** command (this is from the count model in unit 1 of the tutorial):



## Buffer Status viewer

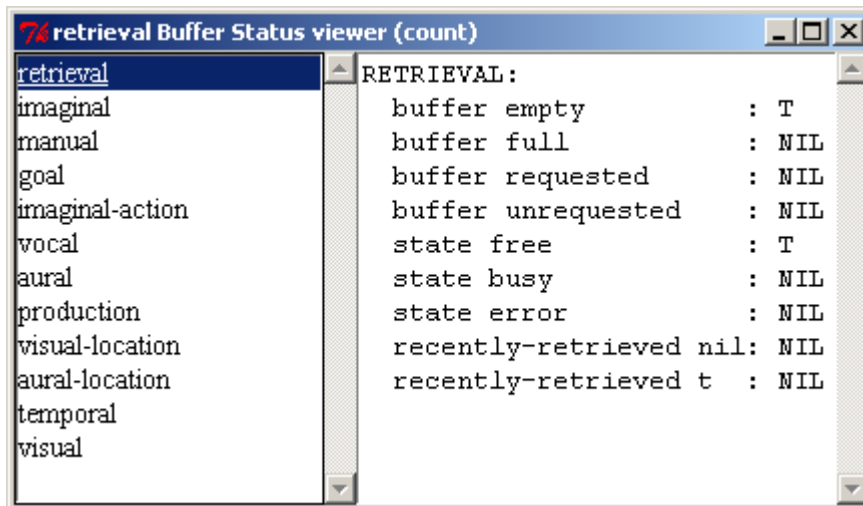
The buffer status viewer allows the user to inspect the results of the queries which can be made through the currently selected model's buffers. Pressing the "Buffer Status viewer" button opens a new buffer status window for inspecting the buffer queries and any number of such windows may be open at the same time. This is what a buffer status viewer will look like:



The list on the left of the window shows the names of all of the buffers in the model. Selecting a buffer from that list will cause the title of the window to change to show the



buffer status being displayed and then show the results of the **buffer-status** command for that buffer on the right:



The **buffer-status** command shows the queries available for the buffer along with whether or not that query is currently true (t) or false (nil).

## Visicon

Pressing the “Visicon” button will open a window showing the information currently available to the currently selected model’s vision module. Only one such window will exist in the environment for each available model. If a visicon window is already open for the current model, then pressing the button will bring that window to the front. The visicon window displays the information returned by the **print-visicon** command and here is an example using the sperling model from unit3 of the tutorial:

Visicon (sperling)						
Loc	Att	Kind	Value	Color	ID	
{ 80 111}	NEW	TEXT	"v"	BLACK	VISUAL-LOCATION0	
{ 80 161}	NEW	TEXT	"c"	BLACK	VISUAL-LOCATION1	
{ 80 211}	NEW	TEXT	"w"	BLACK	VISUAL-LOCATION2	
{130 111}	NEW	TEXT	"n"	BLACK	VISUAL-LOCATION3	
{130 161}	NEW	TEXT	"r"	BLACK	VISUAL-LOCATION4	
{130 211}	NEW	TEXT	"j"	BLACK	VISUAL-LOCATION5	
{180 111}	NEW	TEXT	"t"	BLACK	VISUAL-LOCATION6	
{180 161}	NEW	TEXT	"y"	BLACK	VISUAL-LOCATION7	
{180 211}	NEW	TEXT	"g"	BLACK	VISUAL-LOCATION8	
{230 111}	NEW	TEXT	"z"	BLACK	VISUAL-LOCATION9	
{230 161}	NEW	TEXT	"k"	BLACK	VISUAL-LOCATION10	
{230 211}	NEW	TEXT	"f"	BLACK	VISUAL-LOCATION11	

## Audicon

Pressing the “Audicon” button will open a window showing the information currently available to the currently selected model’s audio module. Only one such window will exist in the environment for each available model. If an audicon window is already open for the current model, then pressing the button will bring that window to the front. The audicon window displays the information returned by the **print-audicon** command and here is an example using the sperling model from unit3 of the tutorial:

Audicon (sperling)								
Sound event	Att	Detectable	Kind	Content	location	onset	offset	Sound ID
AUDIO-EVENT0	NIL	NIL	TONE	1000	EXTERNAL	0.150	0.650	TONE0

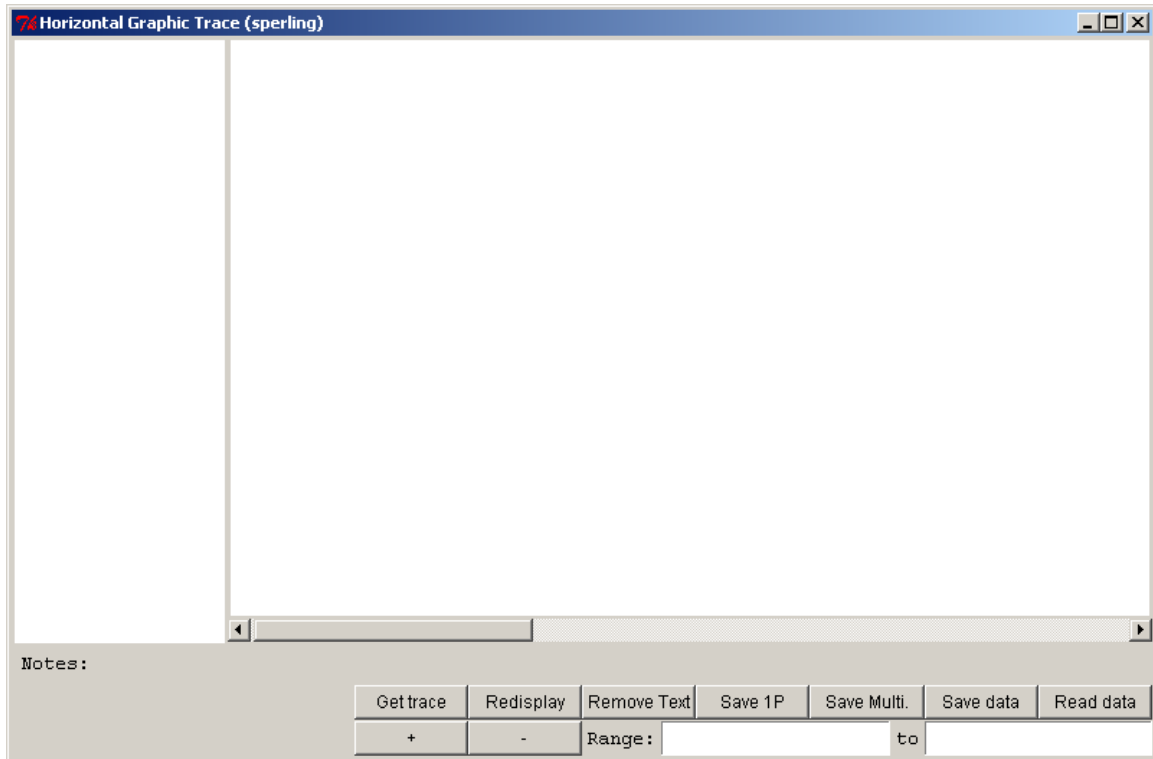
## Tracing

The tracing tools provide a graphic representation of a model's operation. The first two tools in this section work similarly and will be described in the [Graphic Trace](#) section below. The other tool provides a different view focused on the sequence of productions which fired and is described in the [Production Graph](#) section.

### Graphic Trace

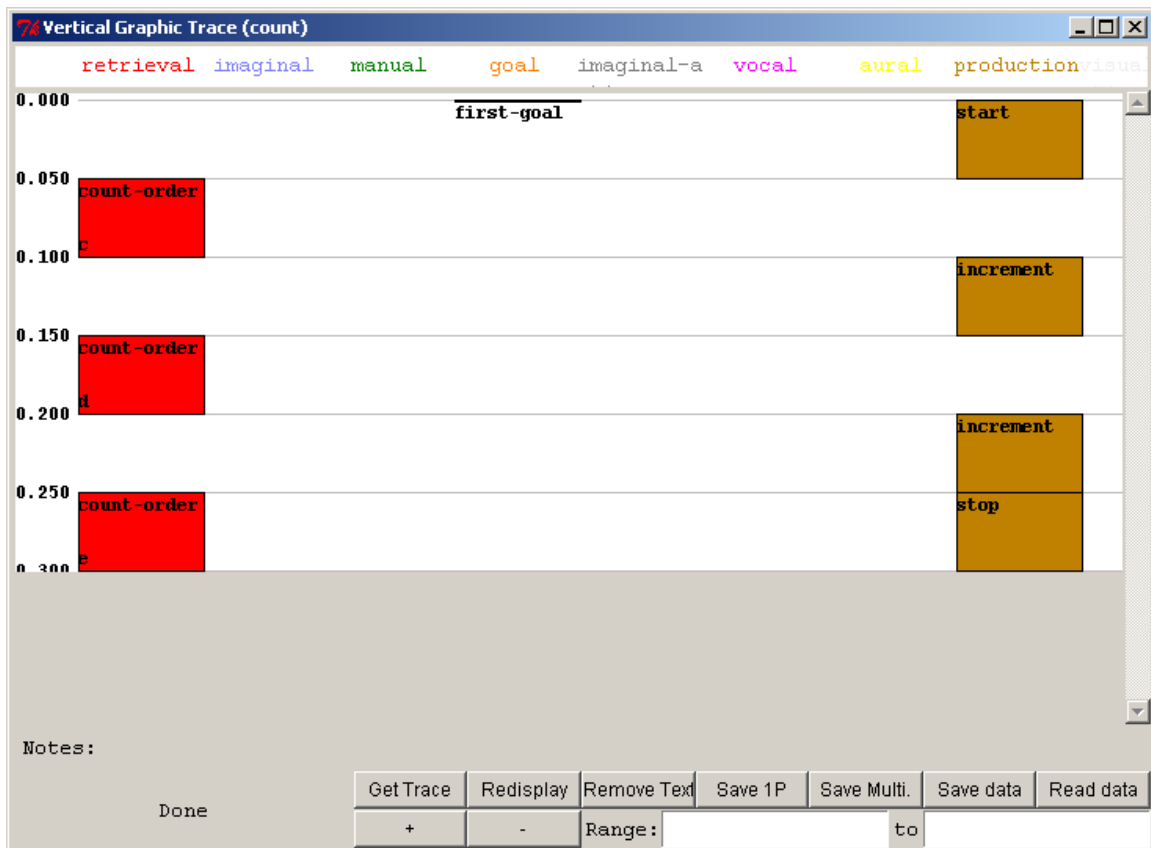
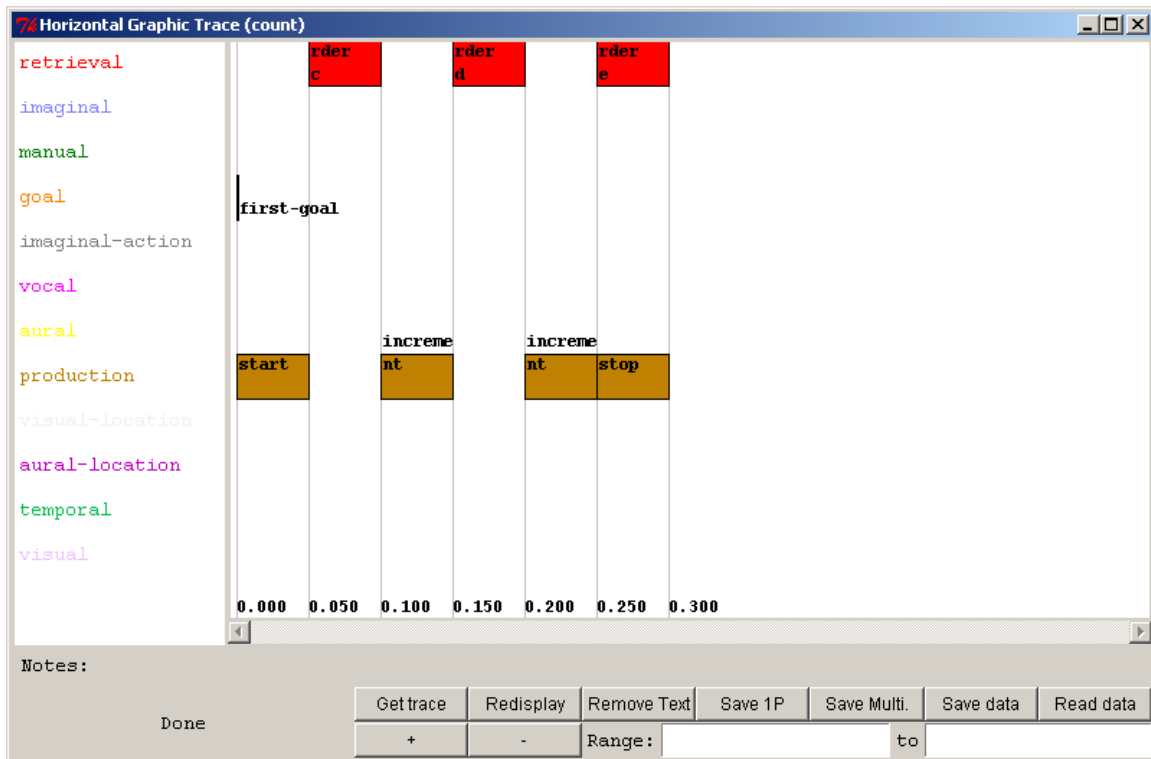
The graphic trace is similar to the buffer trace which can be displayed when a model runs instead of the default event based trace. Other than the orientation of the display, both tracing tools work the same and will be described together.

Pressing the “Horiz. Buffer Trace” button will open a new “Horizontal Graphic Trace” window for the currently selected model and pressing the “Vert. Buffer Trace” button will open a new “Vertical Graphic Trace” window for the currently selected model. Any number of either type of window may be open at any time. When opened, the window will look like this:



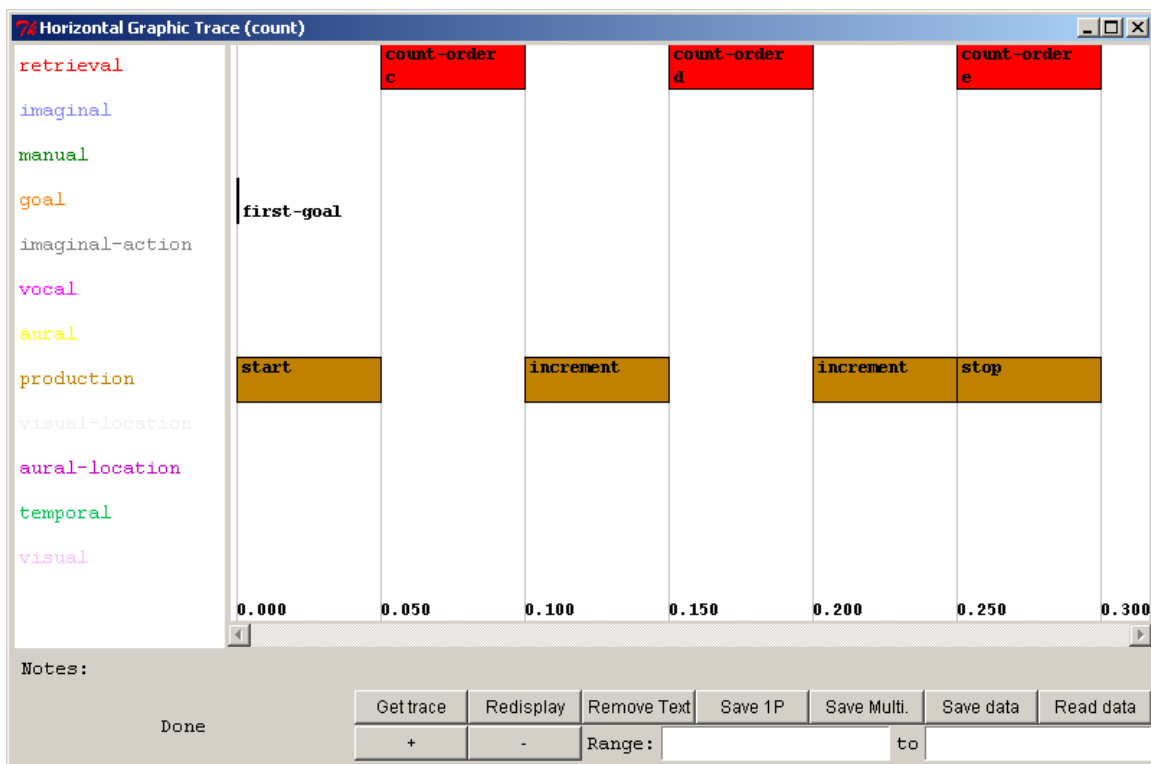
There will be no information displayed in the window until the model is run and the information is requested as described below.

To use the tracing tools you must either set the parameter :save-buffer-trace to t in the model or open a trace tool window prior to running the model. After the model has been run, the graphic trace can be displayed by pressing the “Get trace” button in the tracing window. Here are the horizontal and vertical traces from running the count model from unit 1 of the tutorial:



For a longer trace it may take some time for the display to be fully drawn. While the display is being drawn the word “Busy” will be shown in the lower left corner of the window, and it will display the word “Done” in the lower left corner (as seen above) once it finishes drawing the trace. None of the other controls should be used until the drawing is complete.

The “+” and “-” buttons at the bottom of the window can be used to zoom in or out on the trace. Pressing the “+” button for that display will zoom in and result in this display which better shows the information in the boxes:



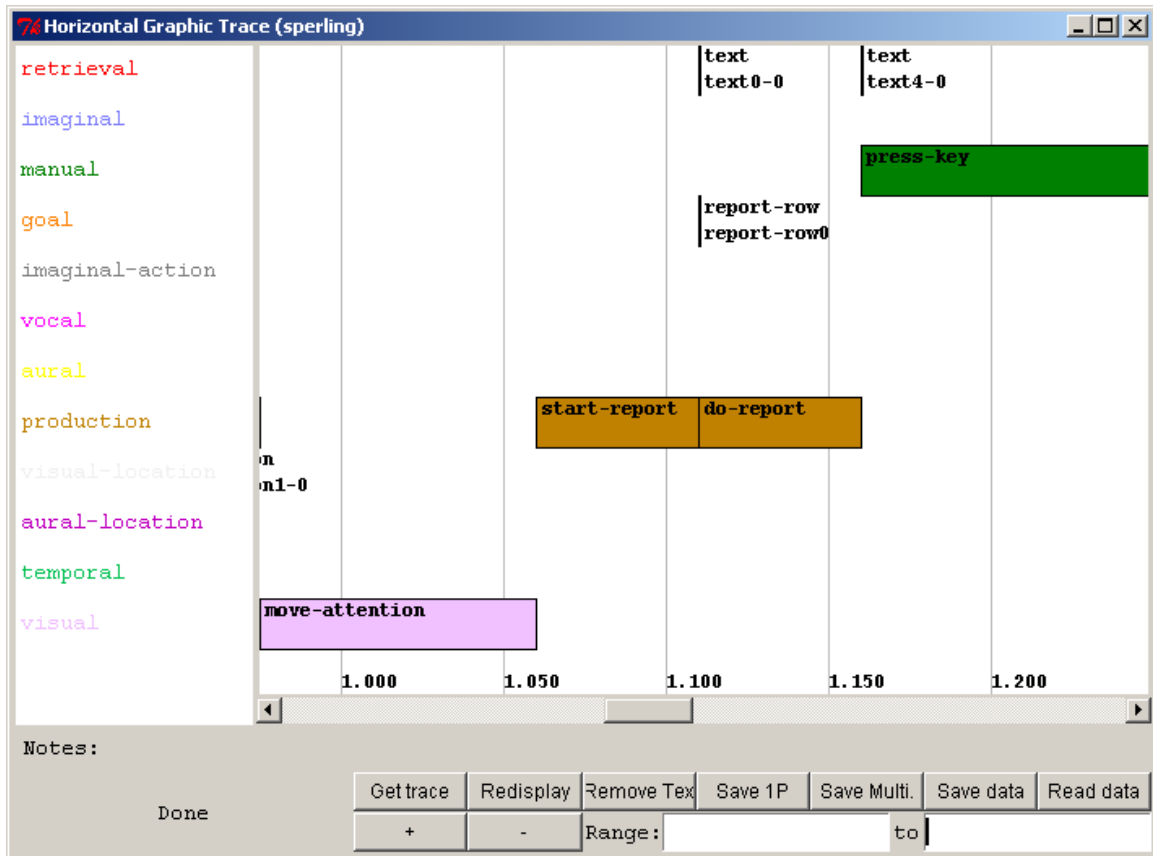
Each row of the horizontal trace or column of the vertical trace corresponds to one of the buffers in the model. The time runs along the bottom of the horizontal trace and along the left edge of the vertical trace. Boxes in a row or column indicate a time period during which that particular buffer reports that its module was busy, and typically represents the module’s processing of a request. The text in the box shows two things. Generally, the text at the top of the box represents the chunk-type of a request that was made to the buffer, if there was one, and the text at the bottom shows the name of the chunk which

was placed into the box as a result of that request.

There are a couple of exceptions however in the text displayed. The first is for the buffer named “production” in the trace. That does not represent a “real” buffer in the model and instead represents when the procedural module fires productions. The text in those boxes is for the name of the production which was selected and fired during that time. Another exception occurs in the vertical trace when an action takes no time i.e. the box is only a line. In that case only the lower line (the chunk name) is displayed. [Note that for the event at time 0 in the goal buffer on the horizontal trace shown above there also isn’t a top line of text. In that case however, it is because the chunk was not created by a module request since it was set directly with the goal-focus command.]

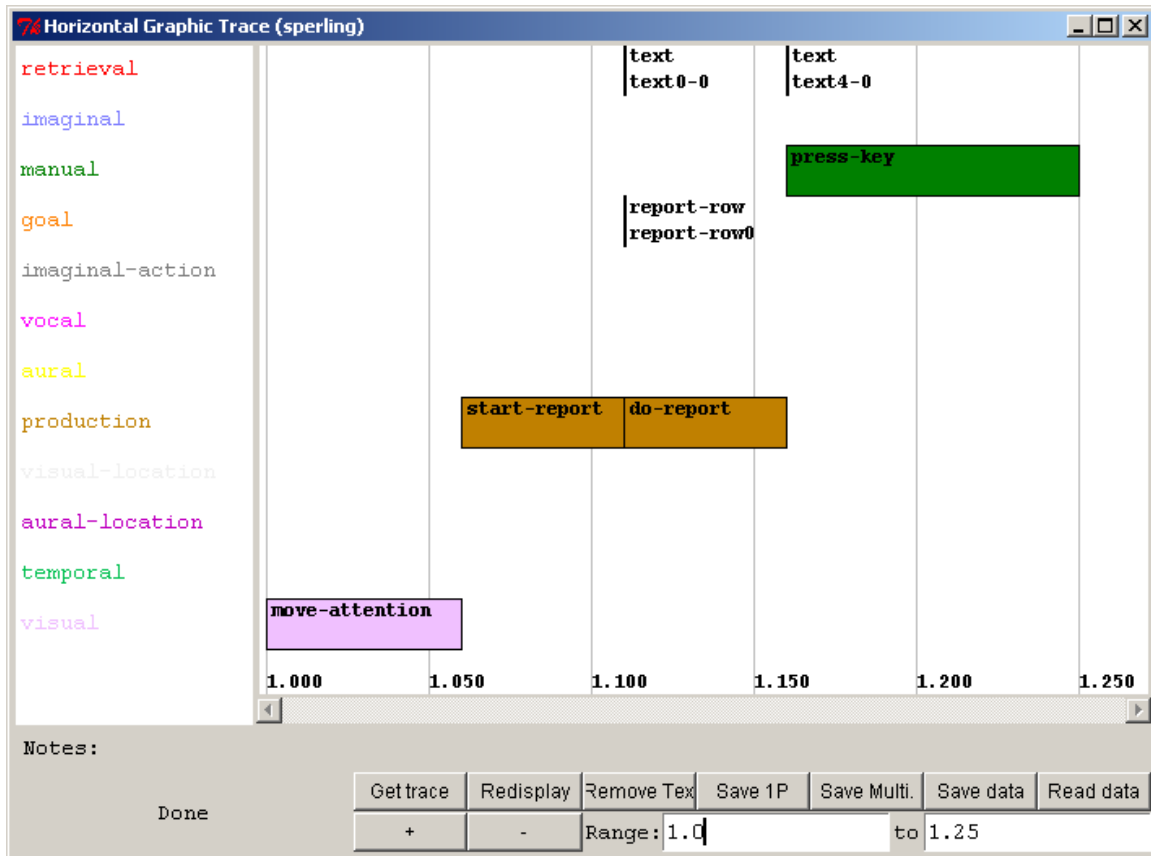
If the trace is larger than the window the scroll bar along the edge of the trace can be used to scroll the display, or the “Range:” boxes at the bottom of the window can be used to restrict the trace to a particular segment of the run. To restrict the display to a particular range of the trace times must be entered into both of the boxes. The times are measured in seconds and the first box must have a time less than the time entered into the second box. Then the “Redisplay” button must be pressed to have the trace redrawn.

Here is part of the sperling model from unit 3 scrolled to the time of the first retrieval:

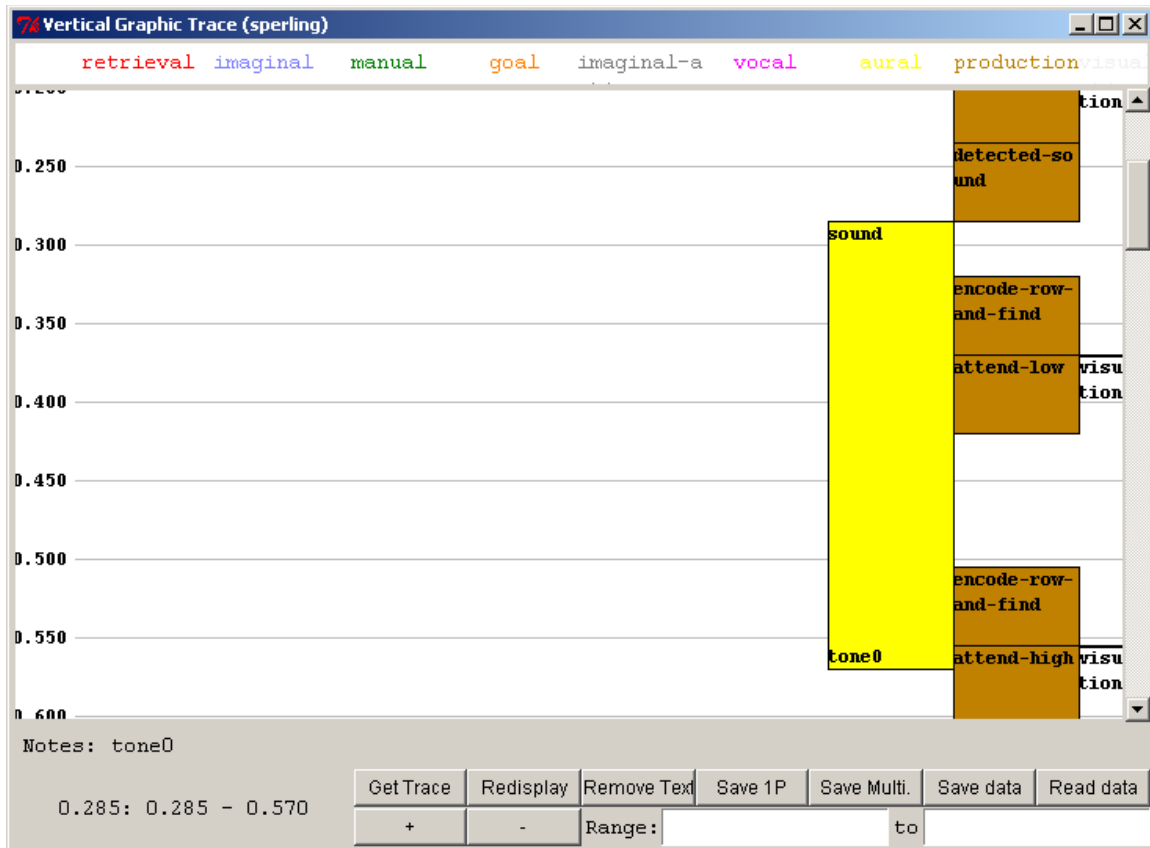


Below is that same trace restricted to the range of 1.0 seconds to 1.25 seconds:

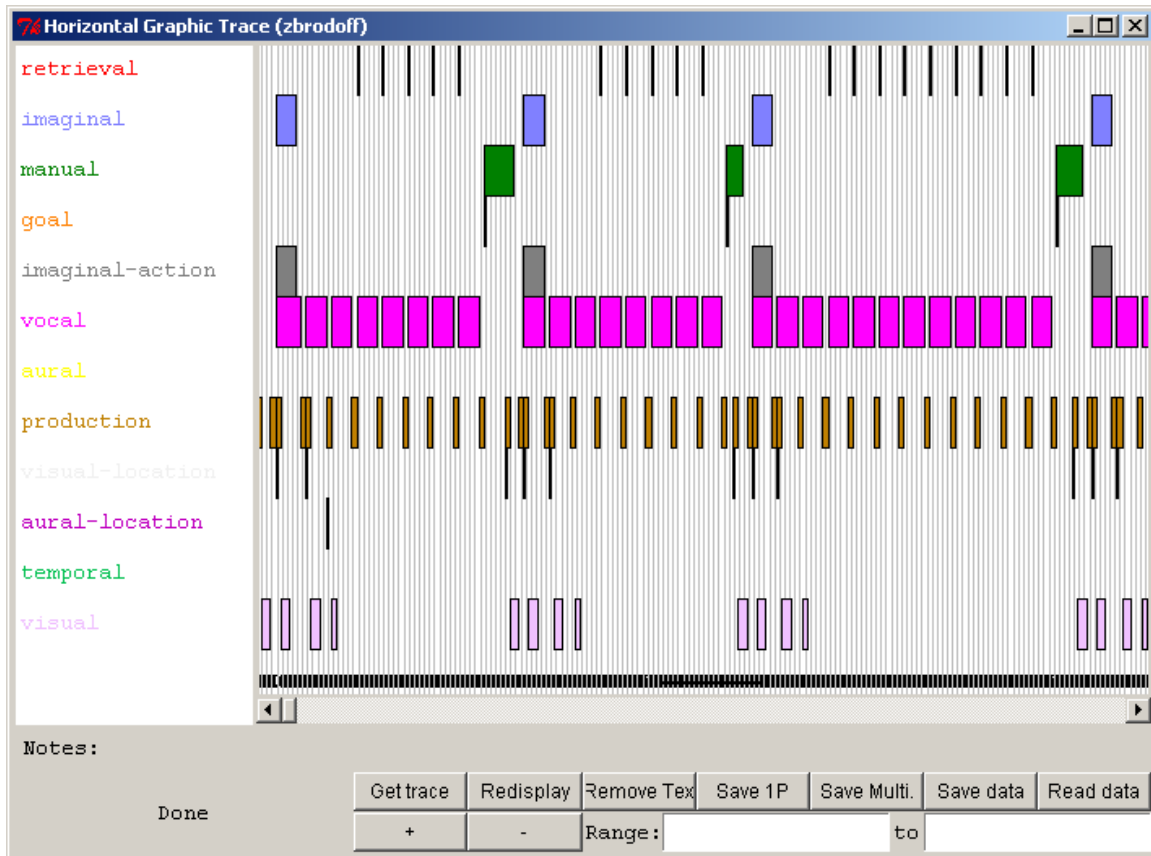




Placing the mouse cursor over a box in the trace will cause some additional details to be shown in the bottom of the display. In the lower left corner it will show the length of the box in seconds followed by the start and stop times. Also, along the line which starts with “Notes:” it will display some additional information. By default, that will be the name of the chunk (the lower line of the box) if there is one available and the request/production (the top line of the box) if not, but it is also possible to add some code to the model to override the default notes with something else. How to do that is described [below](#). Here is a section of the trace from the sperling task with the mouse positioned over the box in the aural buffer column:



Sometimes, it's not important to see the text details in the boxes of the trace, for example when zooming out on a large trace to get a more general view for what is happening in the model. In cases like that hitting the "Remove Text" button will clear the text and show only the boxes. Here is a trace of a run of the zbrodoff model from unit 4 of the tutorial zoomed out to see a couple of trials of the task with the text removed:



Hitting the “Redisplay” button will redraw the window and restore the text if desired.

## Saving Graphic Traces

The remaining four buttons on the graphic trace window are for use in saving or restoring the data from a trace.

### ***Save 1P***

The “Save 1P” button can be used to save an image of the graphic trace as an Encapsulated PostScript file. Pressing that button will bring up a file creation dialog in which you must provide the name for the file in which to save the data. The image will be saved as a single page graphic which contains the whole trace, or the currently displayed range if one is specified. Encapsulated PostScript files can be imported in many

applications which are used for word processing and generating presentations.

### ***Save Multi.***

The “Save Multi.” button can be used to save an image of the graphic trace as a PostScript file. Pressing that button will bring up a file creation dialog in which you must provide the name for the file in which to save the data. The image will be saved as a multiple page document, and for the horizontal trace the page is generated in landscape mode.

### ***Save data and Read data***

The “Save data” button can be used to save the internal data needed to generate the graphic trace to a file. Pressing that button will bring up a file creation dialog in which you must provide the name for the file in which to save the data. That data can then be loaded back in later using the “Read data” button to recreate the trace. Pressing the “Read data” button in a graphic trace window will open a file selection dialog. A graphic trace data file which was saved using the “Save data” button should be selected. The trace data in that file will be used to draw the trace in the current graphic trace window. Note that the data saved is orientation specific i.e. if it was saved from a vertical trace it can only be loaded and drawn correctly in another vertical trace window.

### **Inspecting Items**

For the Retrieval and Production entries of the graphic traces one can click on the boxes and open the appropriate inspecting tool. Clicking on a box in the retrieval row for which there is a chunk retrieved will open a “Declarative viewer” window and display the information for the chunk which is indicated in the retrieval box. Similarly, clicking on a box in the production row will open a “Procedural viewer” window and display the information for the production which is specified in the box. Note that if the model for which the trace was generated is no longer available then the inspectors will not be opened.

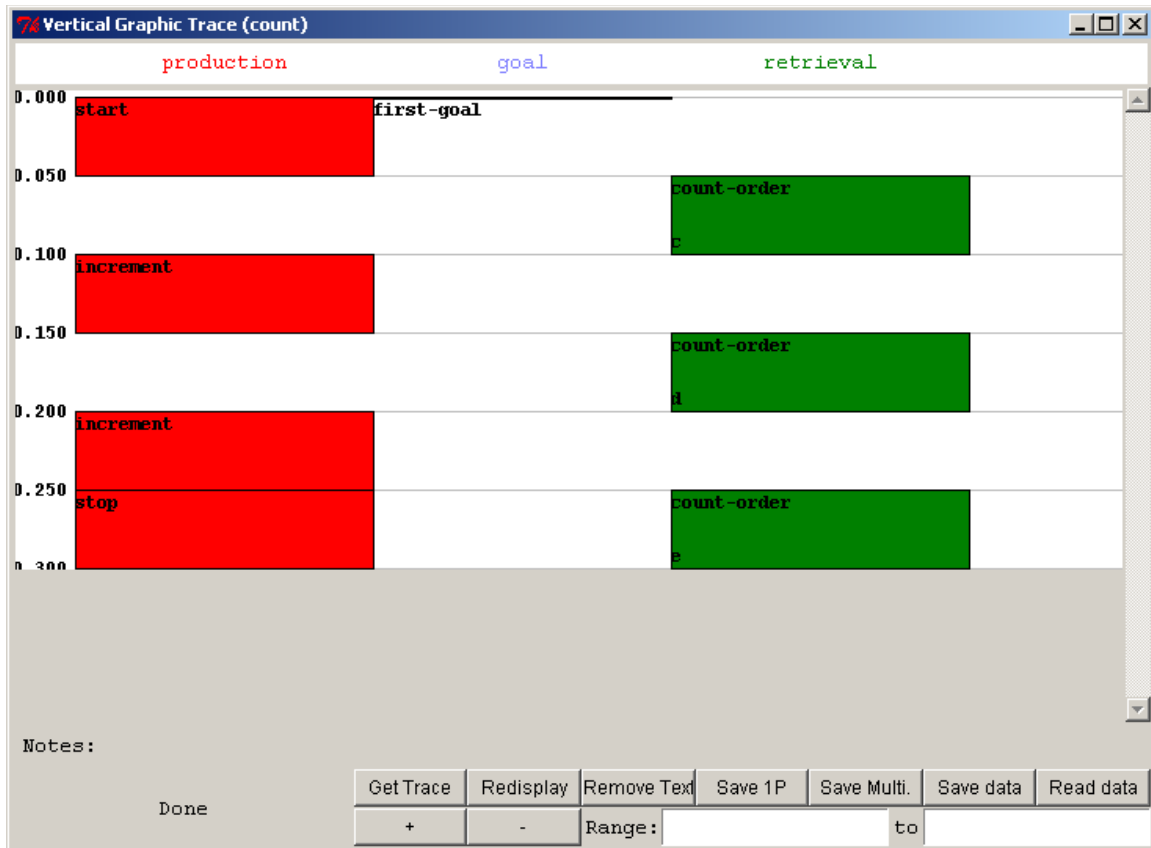
## Display Options

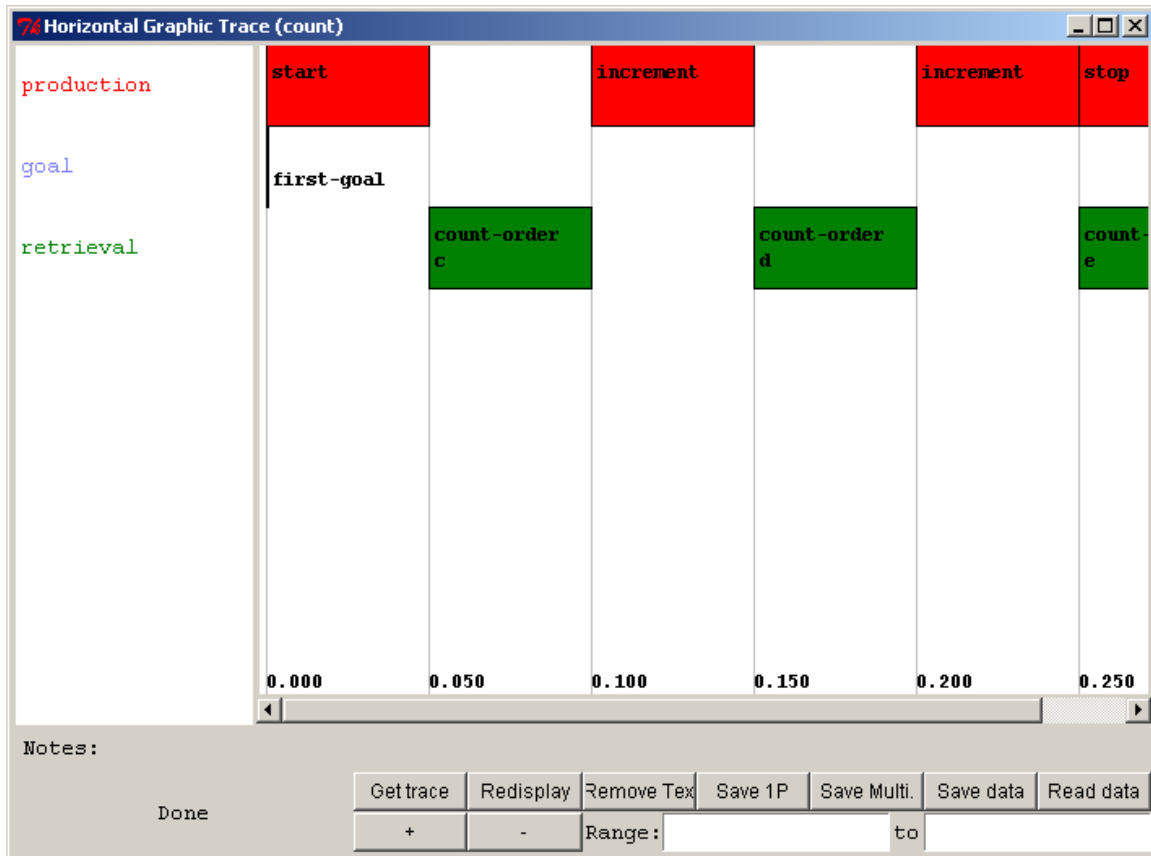
There are commands and parameter settings in ACT-R which can be used to control what gets displayed in the graphic trace and configure how things are displayed. Those options are described in the following sections.

### *Buffers*

By default, all of the buffers in the model are displayed and the order in which they are displayed is not specified and may not always be the same. However, it is possible to restrict the set of buffers which are used and to specify the order in which they are displayed. The **:traced-buffers** parameter in the model is used to specify which buffers should appear in the trace and the order in which to draw them. The parameter should be set to a list of buffer names and only those buffers will be drawn in the order specified (left to right for the vertical trace or top to bottom in the horizontal trace). Here are the traces from the count model of unit 1 with the **:traced-buffers** parameter set like this:

```
(sgp :traced-buffers (production goal retrieval))
```

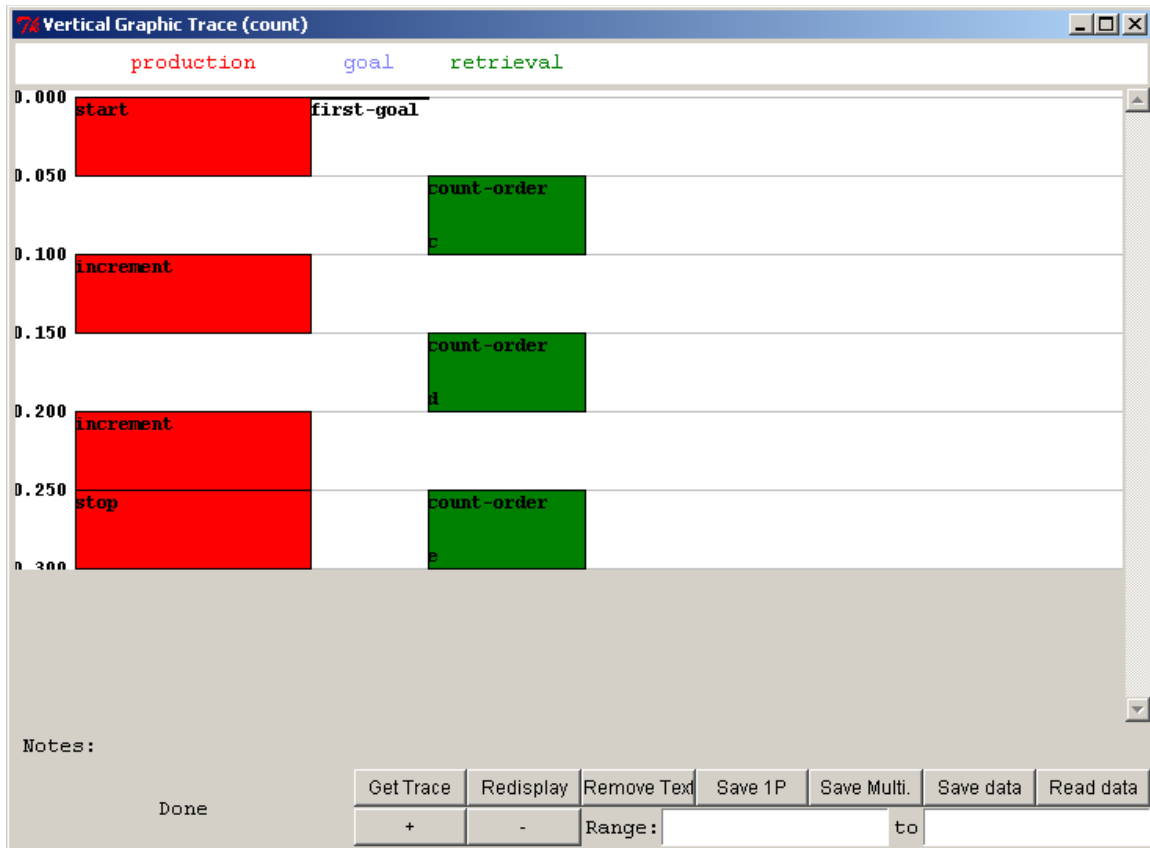




### ***Buffer widths***

Note that for the vertical trace when there are fewer buffers displayed the boxes are drawn wider. The width of the columns in the vertical trace can be specified using the **:graphic-column-widths** parameter in the model. There is no corresponding setting for the horizontal trace. That parameter can be set to a list of numbers where each number represents the width in pixels of the corresponding column in the trace. If there are fewer numbers specified than there are buffers (columns), then the remaining ones are drawn in the default width. There is no restriction in the setting of how wide the trace can be drawn, but because the vertical trace window does not have a horizontal scroll bar you may not be able to see the entire trace if you make it wider than your monitor can display. Here is a vertical trace of the count model with the following settings in the model:

```
(sgp :traced-buffers (production goal retrieval))
(sgp :graphic-column-widths (150 75 100))
```



### ***Buffer Colors***

By default, the colors chosen for each column (or row) are taken from a predefined list of colors. However, it is possible to specify a particular color for each column or row. Like the **:graphic-column-widths** parameter described above, the **:buffer-trace-colors** parameter can be set to a list of color designators and those colors will be used for the columns.

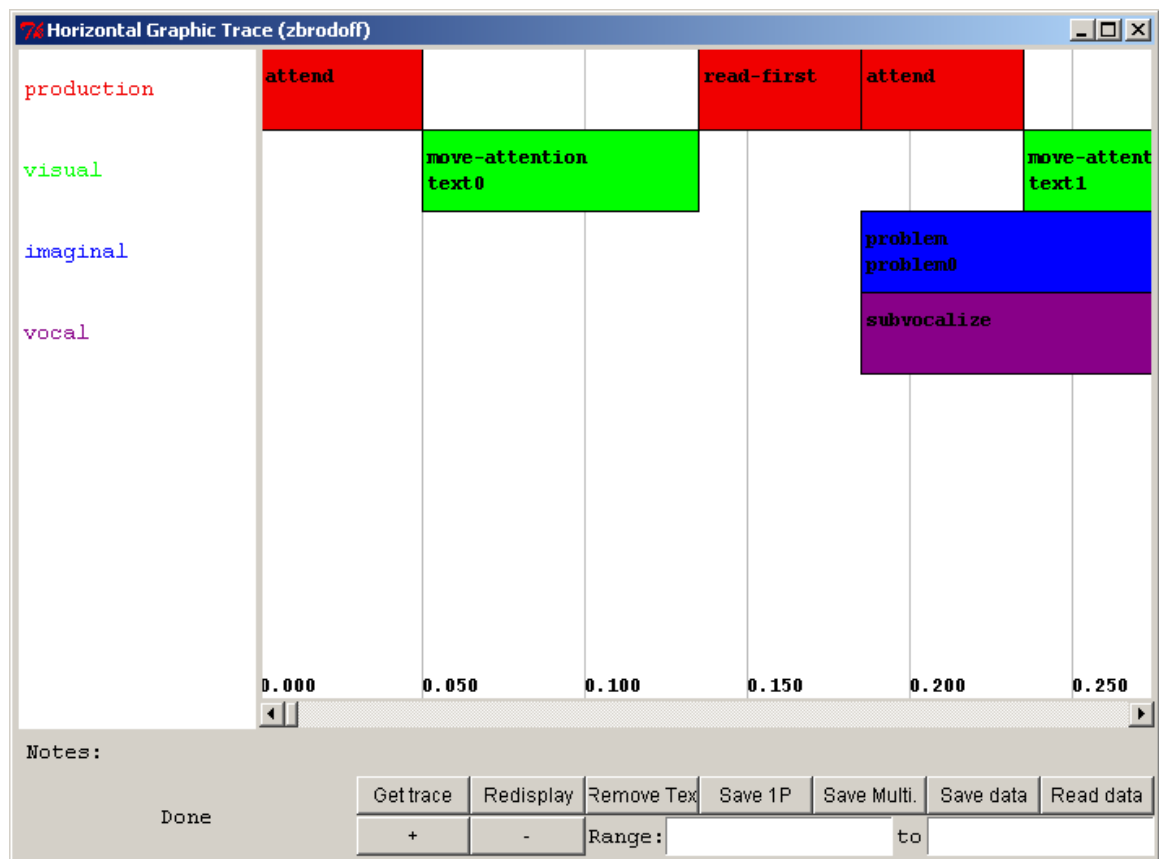
A color designator is a Lisp string which is either 4, 7, or 10 characters long. It must start with the character # and the remaining items represent three hexadecimal values for the red, green, and blue value of the color. Each of the color components can be specified using either 1, 2, or 3 hex digits (depending on the color depth desired) and they all must have the same number of digits. Thus, fully saturated red could be specified as “#F00”, “#FF0000”, or “#FFF000000”. Any combination of red, green and blue values may be



given in a particular color designator, but the actual color displayed will depend on the monitor and video capabilities of the machine.

Here is a horizontal trace from the beginning of a run of the zbrodoff model from the tutorial with these settings in the model:

```
(sgp :traced-buffers (production visual imaginal vocal))
(sgp :buffer-trace-colors ("#F00" "#00FF00" "#000000fff" "#880088"))
```



### ***Production Colors***

In addition to being able to control the color of each buffer column or row, for the production boxes it is possible to specify a particular color for each production. If no color is given to a production then it is drawn in the color specified for the production

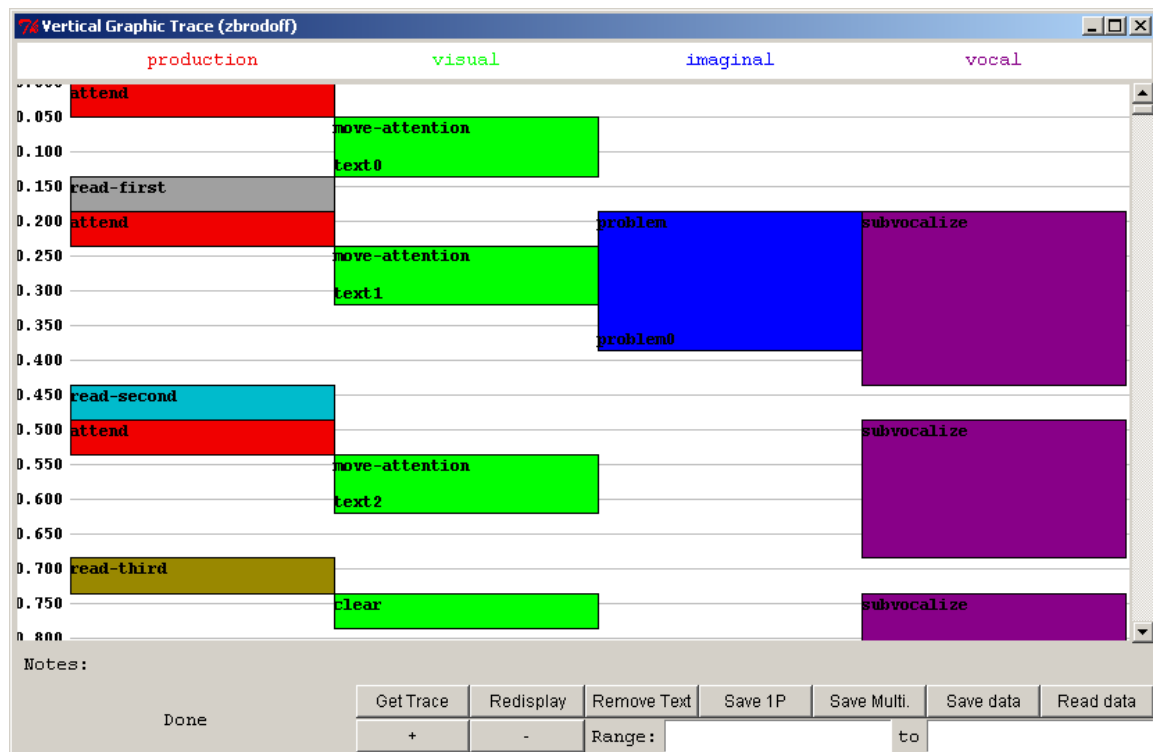
column or row i.e. that is the default color for a production. To specify a color for a production you need to setf the production-color value of the production's name to a color designator (as described above for the buffer colors). For example, to set the color for a production named read-first to blue you would add this to the model after the definition of the read-first production:

```
(setf (production-color 'read-first) "#00f")
```

Below is a section of the zbrodoff model trace with these settings in the model:

```
(sgp :traced-buffers (production visual imaginal vocal))
(sgp :buffer-trace-colors ("#F00" "#00FF00" "#000000fff" "#880088"))

(setf (production-color 'read-first) "#aaa")
(setf (production-color 'read-second) "#00bbcc")
(setf (production-color 'read-third) "#990888000")
```



### *Adding custom “Notes”*

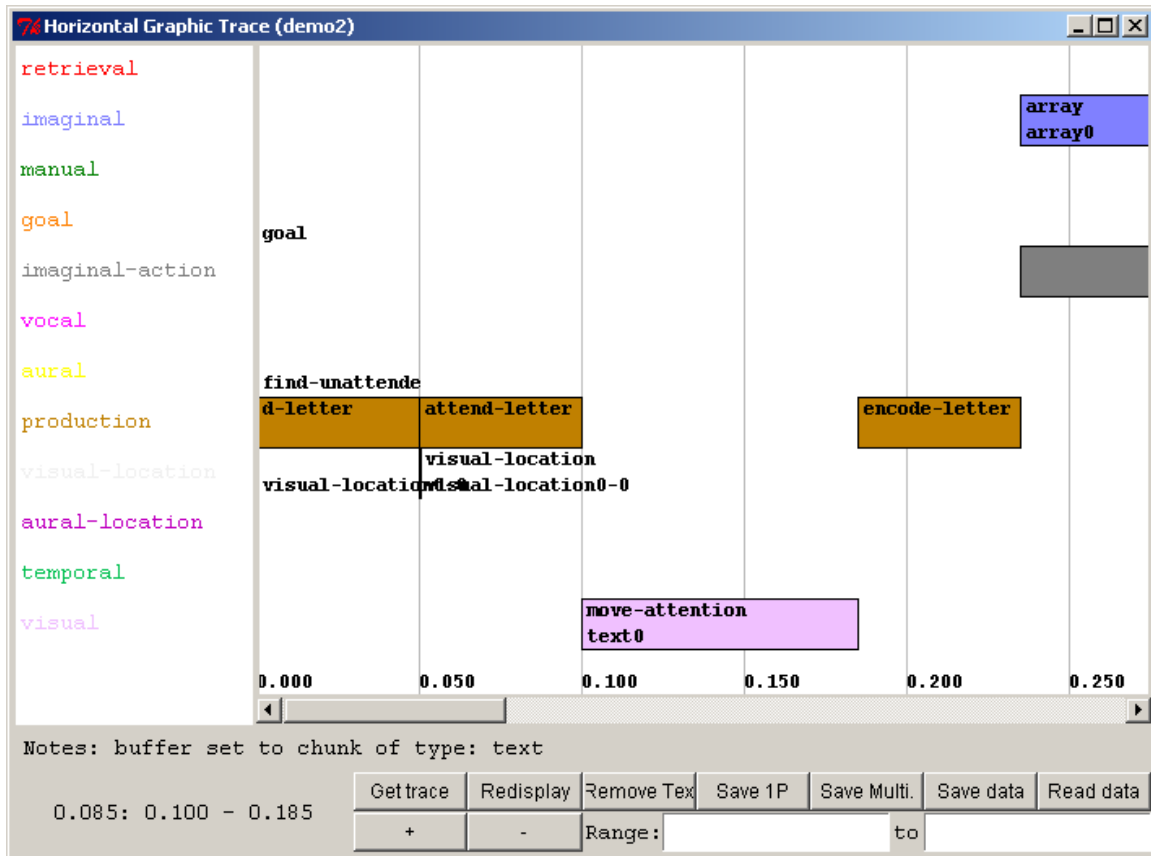
The ACT-R **add-buffer-trace-notes** command can be used to place additional notes into the buffer trace. It takes two parameters which are the name of a buffer and something to store in the trace (it can be anything). It adds that note to the buffer trace at the time which it is called. When the mouse is placed over a box in the graphic trace tools, if there are any custom notes which have been added for that buffer during the time the box covers, then the last such custom note is printed on the “Notes:” line using the `~a` argument in a format call.

Adding custom notes is something which would most likely occur in the module’s request processing code to make that additional information available, but it could be called from the user code or perhaps from one of the event hook functions as well. Here is some code which creates an event hook in the model which looks for any **set-buffer-chunk** actions and then adds a note for that buffer indicating the chunk-type which was set:

```
(defun note-buffers-chunk-type (x)
  (when (eq (evt-action x) 'set-buffer-chunk)
    (add-buffer-trace-notes
      (first (evt-params x))
      (format nil "buffer set to chunk of type: ~A"
        (chunk-chunk-type-fct (second (evt-params x)))))))

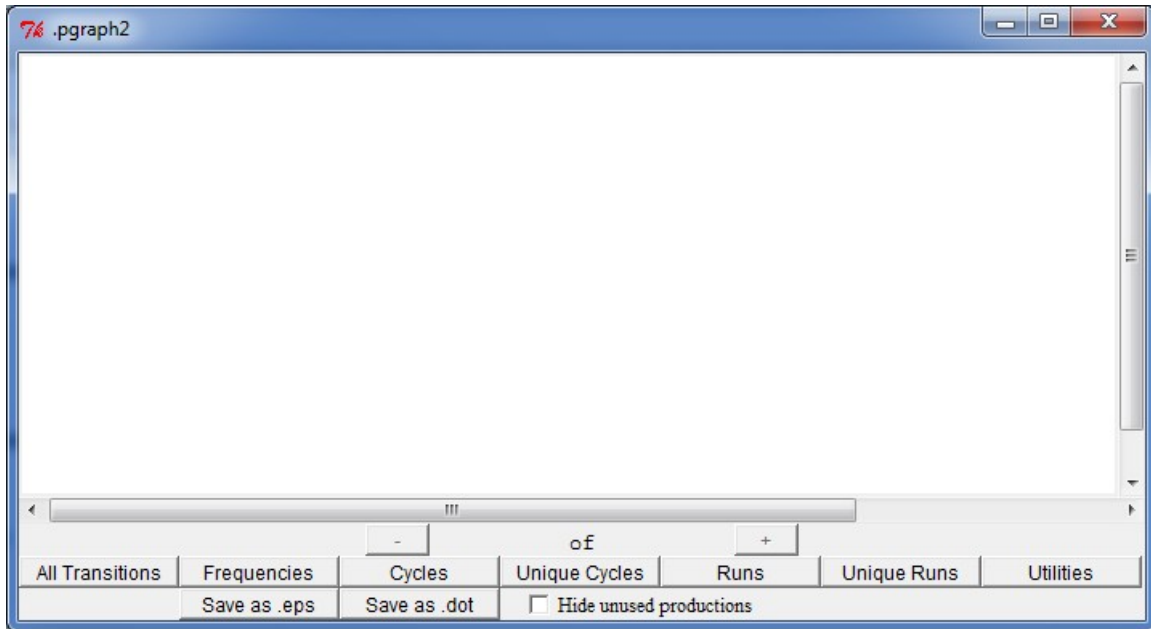
(add-pre-event-hook 'note-buffers-chunk-type)
```

Here is a trace of the `demo2` model from unit 2 of the tutorial with that hook function added and the mouse cursor placed over the first box in the visual buffer’s row:



## Production Graph

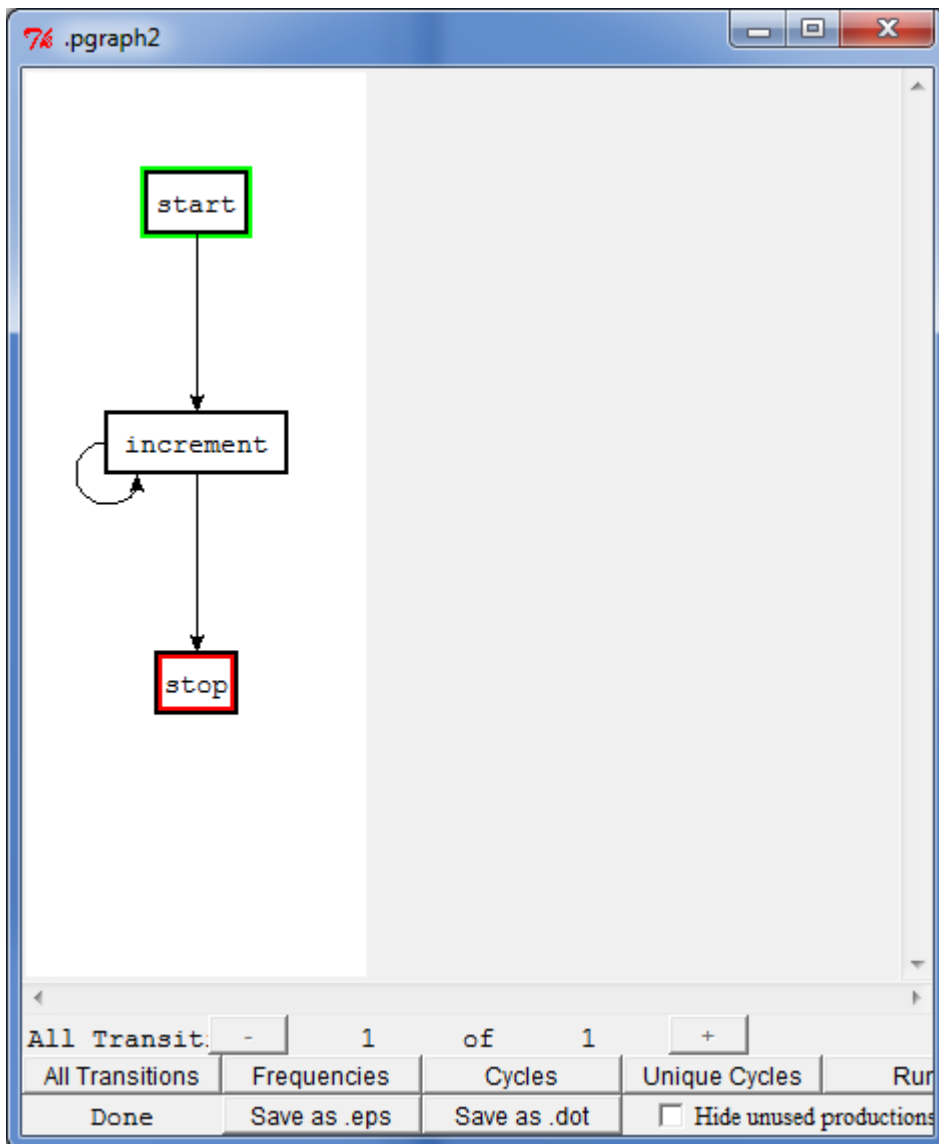
Pressing this button opens a new “Production Graph” window for the current model and any number of those windows may be open. The tool provides a graph of the production transitions which occurred in the model run. Here is a display of the window without any data shown (how it will always appear upon opening):



To use the tool either a “Production Selection History” (described [later](#) in the manual) or “Production Graph” window needs to be open before running the model or the **:save-p-history** parameter needs to be set to **t** in the model to make sure that the data is recorded. After running the model you need to press one of the seven data display buttons: “All Transitions”, “Frequencies”, “Cycles”, “Unique Cycles”, “Runs”, “Unique Runs”, or “Utilities” to have the data displayed. All of the buttons work similarly, but each provides a slightly different view of the data. The similar operation will be described first, and then the specific details of each will be discussed.

After pressing one of the data display buttons the window will show the word “Busy” in the lower-left corner and all of the controls will be disabled. When it completes it will show the data, print the word “Done” in the corner, and the buttons will be available

again. Here is the display after running the count model from unit 1 of the tutorial and pressing the “All Transitions” button:



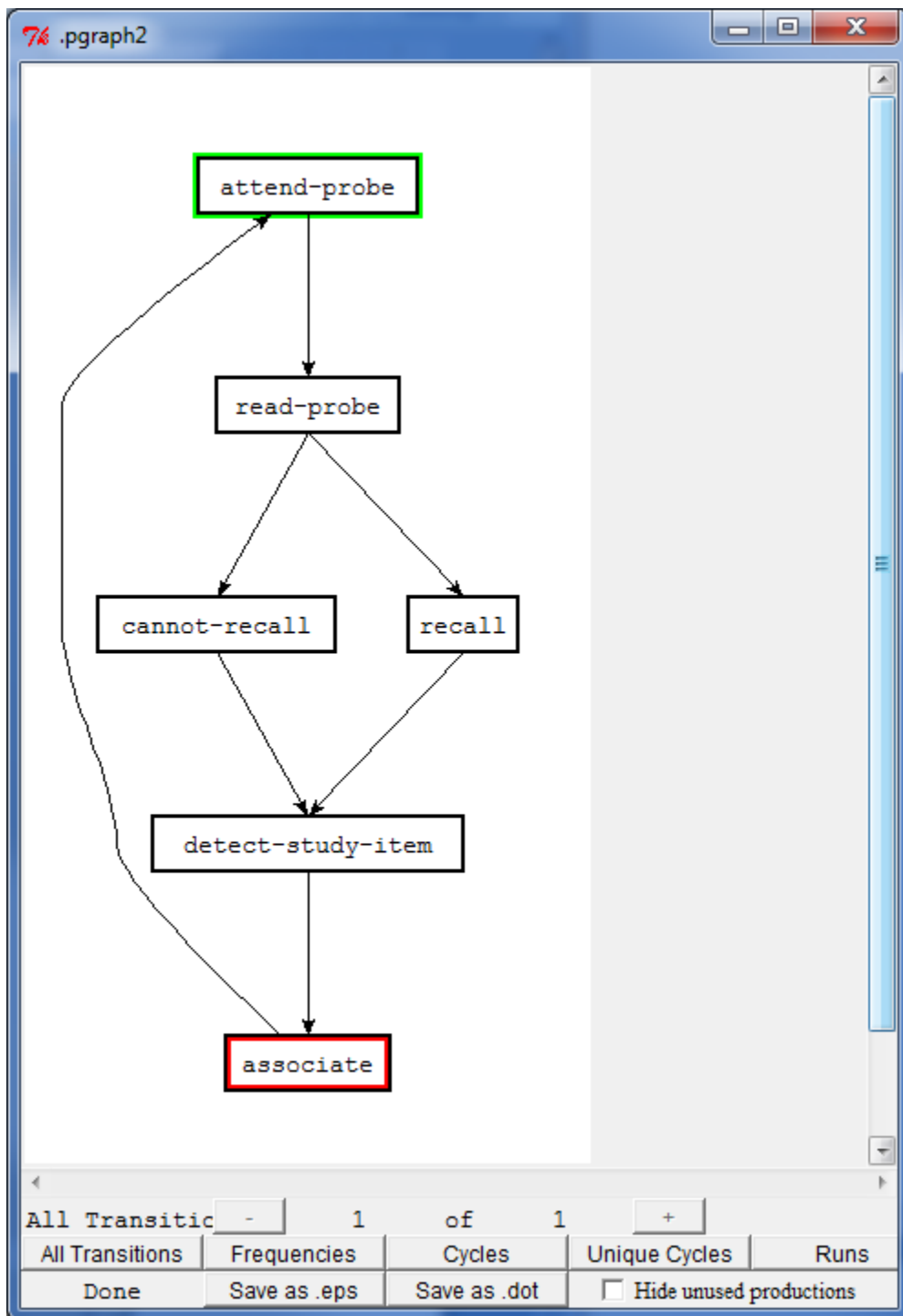
The display will be a state chart diagram for the productions in the model indicating the order in which they were selected and fired. Each production in the model will be drawn in a box, and like the “Production History” tool, if you click on the name of a production then it will open a new Procedural Viewer window with that production. If the border of the box is black then that production was selected and fired at some time during the run of the model for which the data was recorded. If the box border is gray then that production

was not selected and fired. [If the “Hide unused productions” box is checked before pressing a data display button then the gray boxes will not be displayed.] The box with the green highlight indicates the first production which was selected during the data period being displayed and the box with the red highlight was the last one selected. The arrows indicate the sequencing of the productions. An arrow from a production A to a production B means that production B was in the conflict set after production A fired. If the arrow is a solid black line then production B was selected and fired after production A, but if the arrow is dashed and gray then production B was not selected and fired even though it did match the current state.

Below the display it will show which type of graph is being displayed along with how many different graphs are available (which can differ based on the button pressed to display the data) and which one of those is currently being shown. When there is more than one graph available the “+” and “-” buttons allow you to change which of those available graphs is shown.

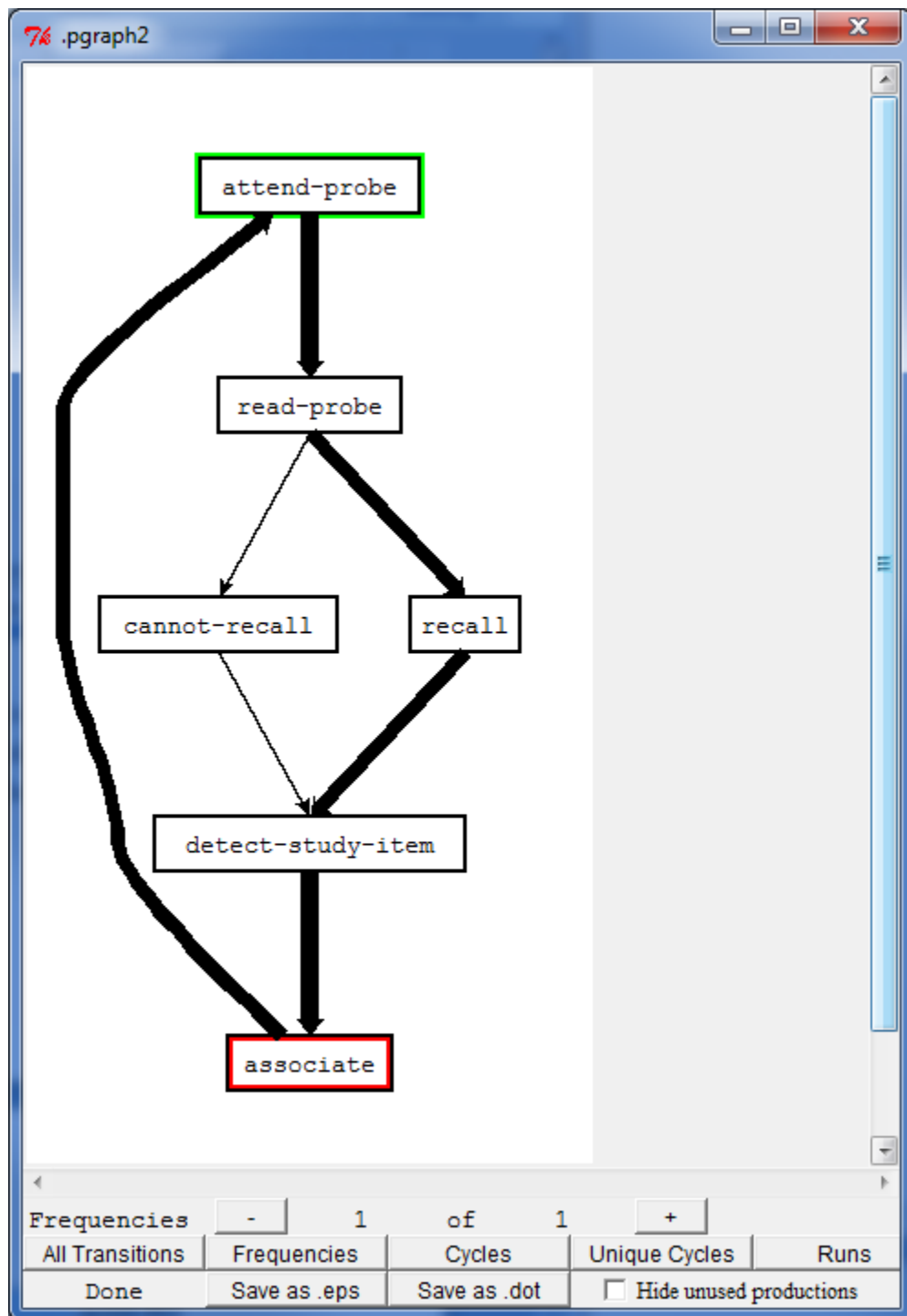
The differences between the displays for each button will be described next and examples for most will be shown for running the paired model from unit 4 of the tutorial using the command (paired-task 1 4).

The “All Transitions” button will always show only one graph. It will contain all of the transitions which occurred in the production data which was recorded. That data may involve multiple cycles in the graph (a loop which passes through a production multiple times) and there is nothing in the display to separate different cycles. Here is what that shows for the paired task:



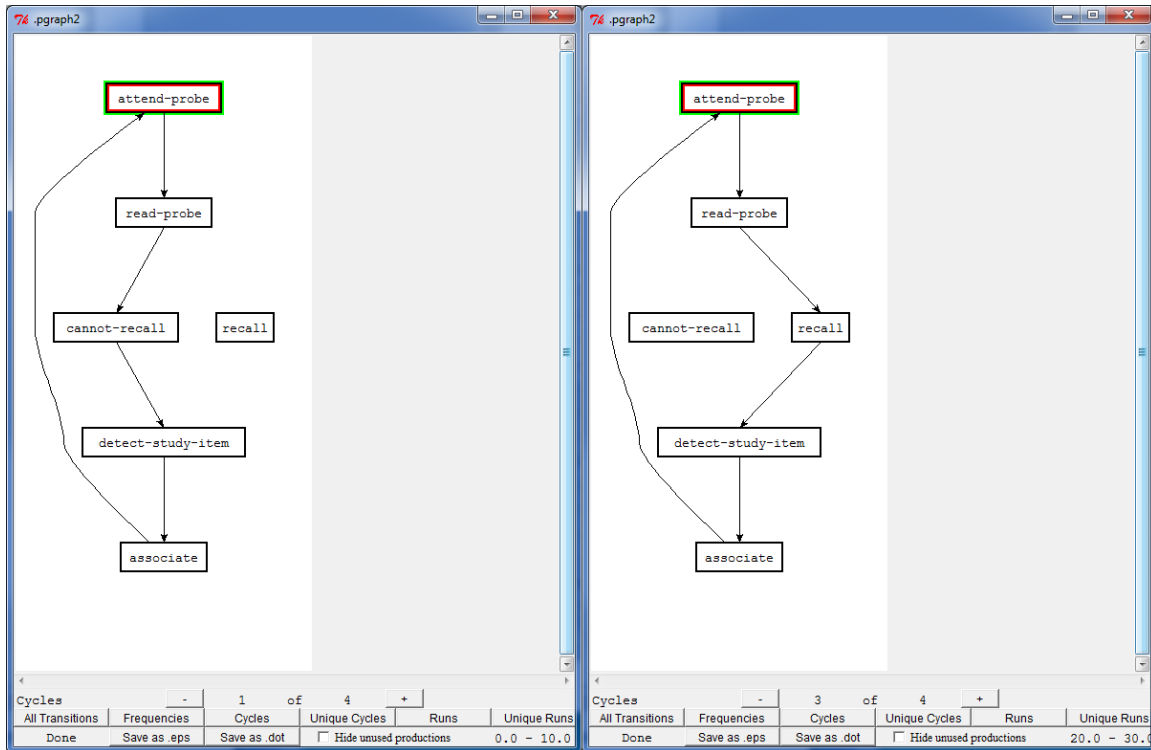
Pressing the “Frequencies” button will display the same graph as is shown for “All Transitions” except that the thickness of the links will indicate their relative frequencies with thicker lines being more frequent. Here is the result of that from the example run:





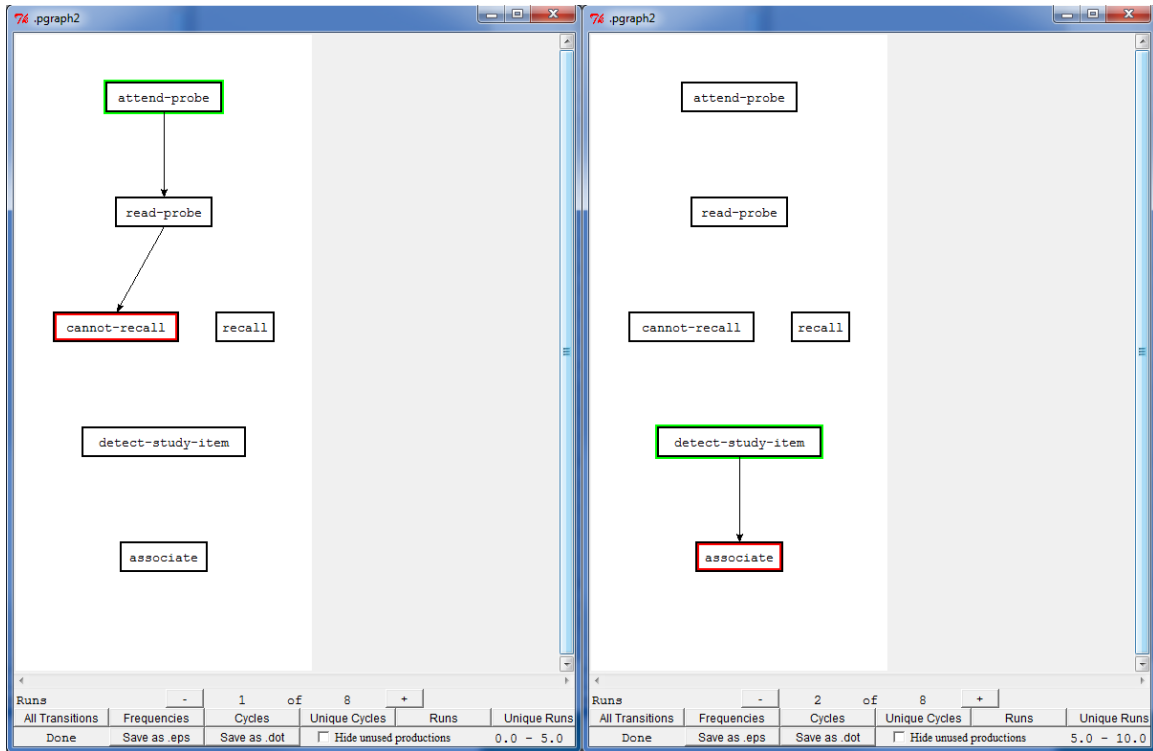
Pressing the “Cycles” button will break the data up into one display for each cycle which

occurs (and possibly an incomplete cycle at the end if it does not form a loop). For each of the cycles displayed it will also show the model's time at the start and end of that cycle at the bottom of the window. Here are two of the cycles from the example run:



The “Unique Cycles” button works similar to the “Cycles” button, except that it only provides one copy of each different cycle which occurs in the data and does not provide the timing information. In the example run there are only 3 unique cycles among the 4 cycles of the data.

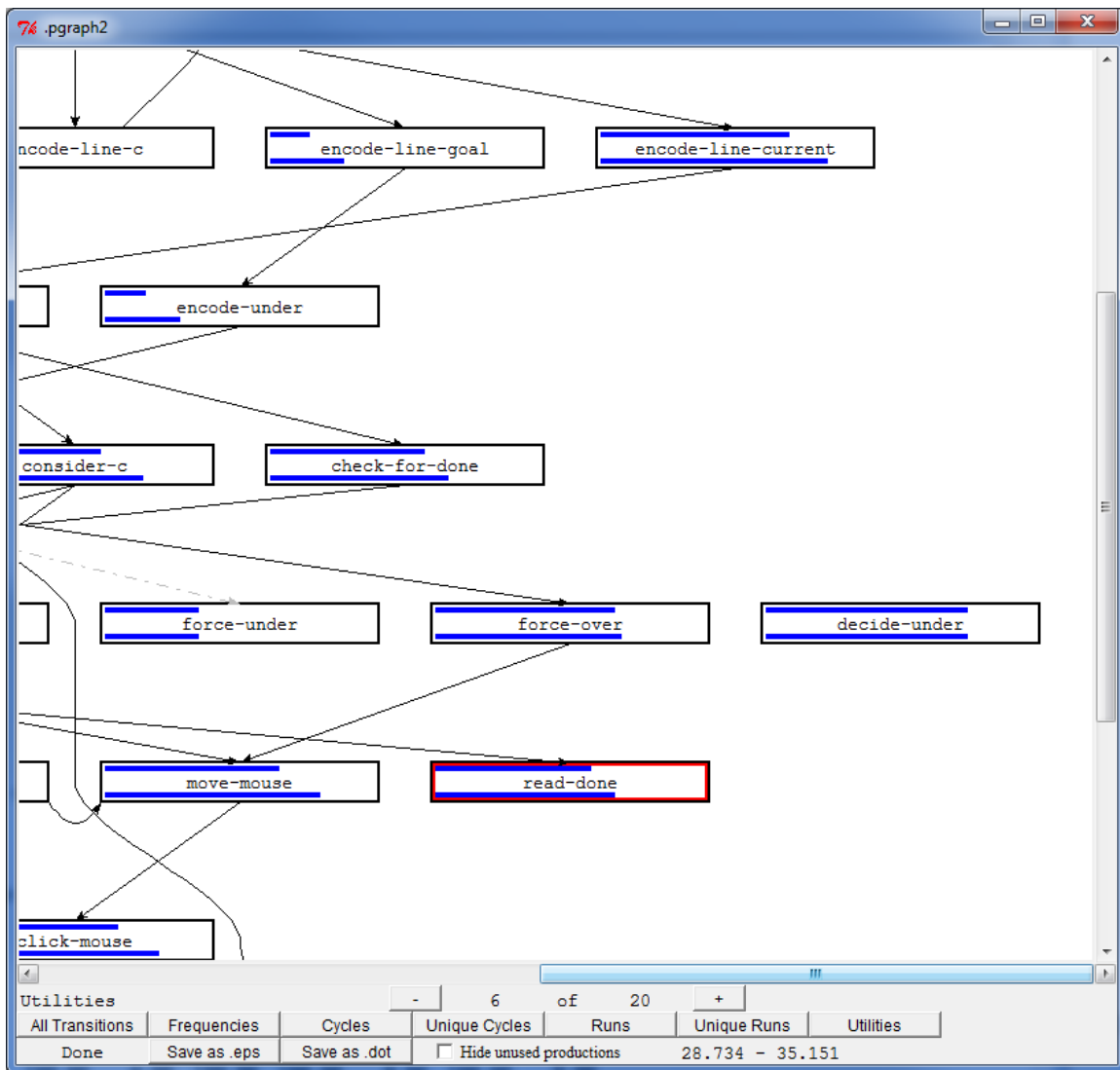
The “Runs” button will break the data up into graphs based on when one of the ACT-R running commands is called, and will provide a separate graph for each run during which at least one production selection occurred. It will show the start and stop times for that run at the bottom of the display. Here are two of the “Runs” graphs from the example:



The “Unique Runs” button works similar to the “Runs” button, except that it only provides one copy of each different run which occurs in the data and does not provide the timing information. In the example run there are only 3 unique runs among the 8 runs which occurred.

The “Utilities” button works similar to the “Runs” button except that the data is broken into graphs based on when the model receives rewards. Each reward marks the end of a graph, and there may be one additional graph at the end which does not represent a reward being presented if the model has fired additional productions after receiving the last reward. The display for the “Utilities” graph is slightly different than the others. First, all of the productions are represented in boxes of the same width instead of being sized based on the production names. In addition to that each production box may have a blue bar displayed along both the top and bottom of the box. Those bars represent the relative utility of that production (the true utility not counting any noise which may have occurred during the run). The bar along the top represents the utility prior to the reward being provided and the bar along the bottom represents the utility after the reward has been propagated. The utilities are scaled across all productions and all graphs so that the

maximum utility which any production has is represented by a bar which fills the box from left to right. Here is an example showing part of the graph after running the building sticks learning model from unit 6 of the tutorial. On this trial we can see that the force-over production was selected among the strategy selection productions and that its utility went up when read-done provided a reward.



The “Save as .eps” button at the bottom of the display can be used to save an image of the current production graph as an Encapsulated PostScript file.

The “Save as .dot” button at the bottom of the display can be used to save a text description of the currently displayed graph in DOT format for use with Graphviz or other purposes. There is one minor issue currently with the DOT files and that is that the frequency graphs do not currently include any information about the link thickness. Thus, the DOT file for the frequency graph will look exactly like the DOT file for the all transitions graph.

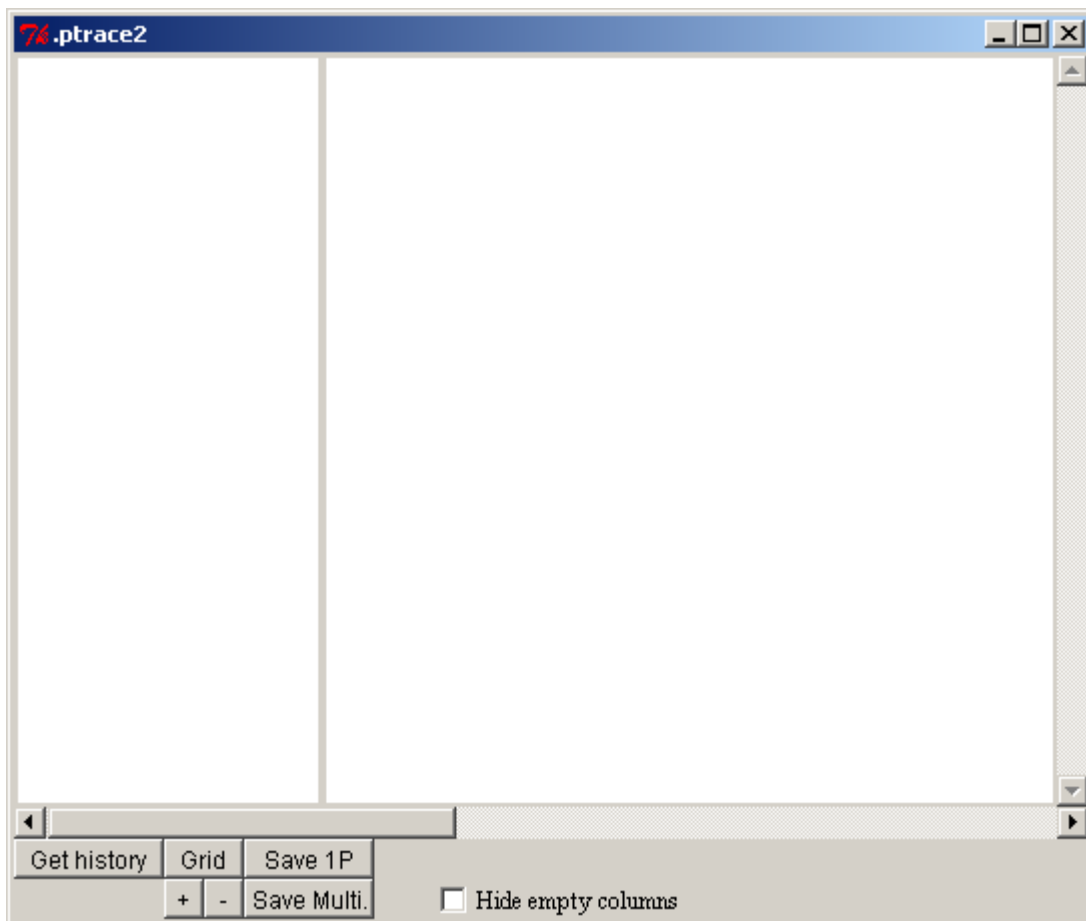
There are two parameters which can be set in the model to adjust the spacing of the productions in the display. The **:p-history-graph-x** parameter specifies the horizontal pixel spacing between the production boxes in a row and defaults to 40. That also determines the maximum thickness of a link for the “Frequencies” display which will be  $\frac{1}{4}$  of the horizontal spacing. The **:p-history-graph-y** parameter specifies the vertical spacing between rows of productions and defaults to 90.

## History Tools

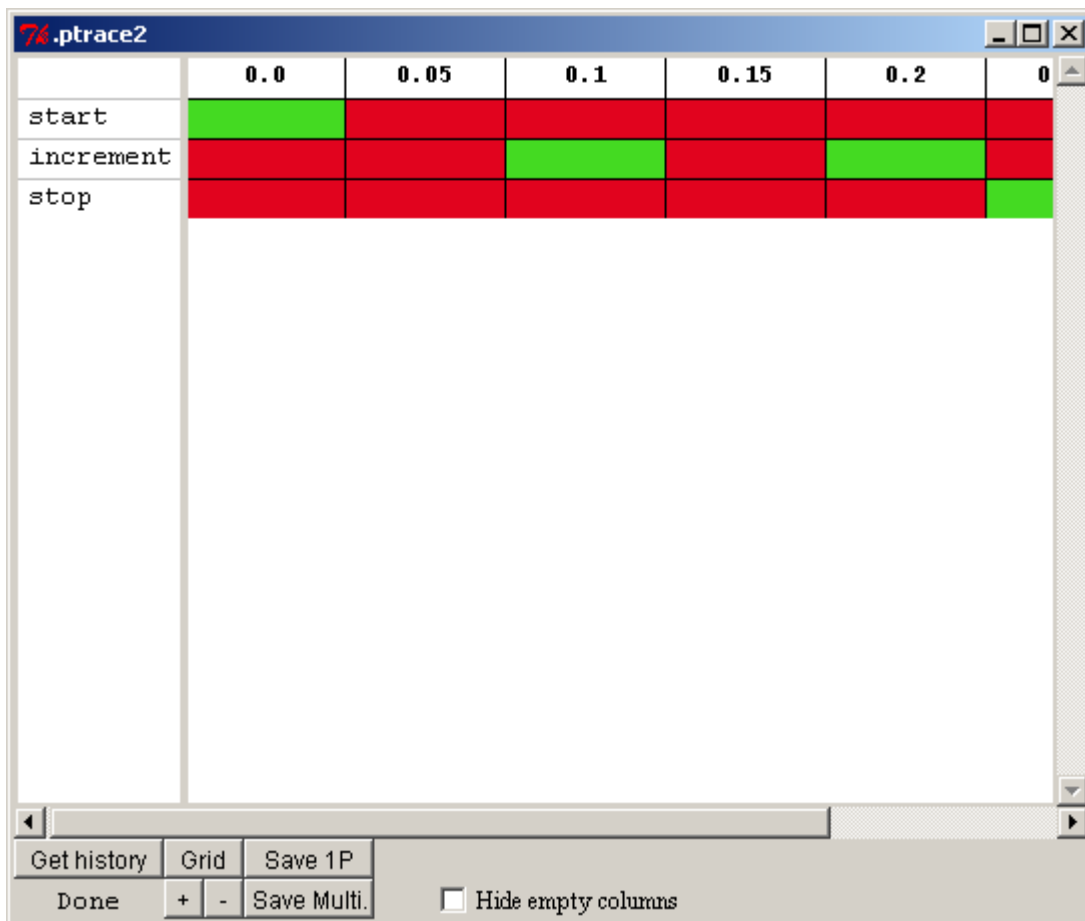
The history tools can be used to record the history of production matching, declarative retrievals, and buffer changes which occur while a model runs and then display that information after the run. Each of those tools operates differently and they will be described individually below.

### Production History

Pressing this button opens a new “Production Selection History” window for the current model and any number of those windows may be open. The tool works similar to the horizontal and vertical buffer tracing tools described above. Here is a display of the window without any data shown (how it will always appear upon opening):



To use the tool either the “Production Selection History” window needs to be open before running the model or the **:save-p-history** parameter needs to be set to **t** in the model to make sure that the data is recorded. After running the model you need to press the “Get history” button in the lower-left corner of the “Production Selection History” window to get the history data displayed. While that is being generated the window will show the word “Busy” in the lower-left corner and the other controls will be disabled. When it completes it will show the word “Done” in the corner and the buttons will be available again. Here is the display after running the count model from unit 1 of the tutorial:



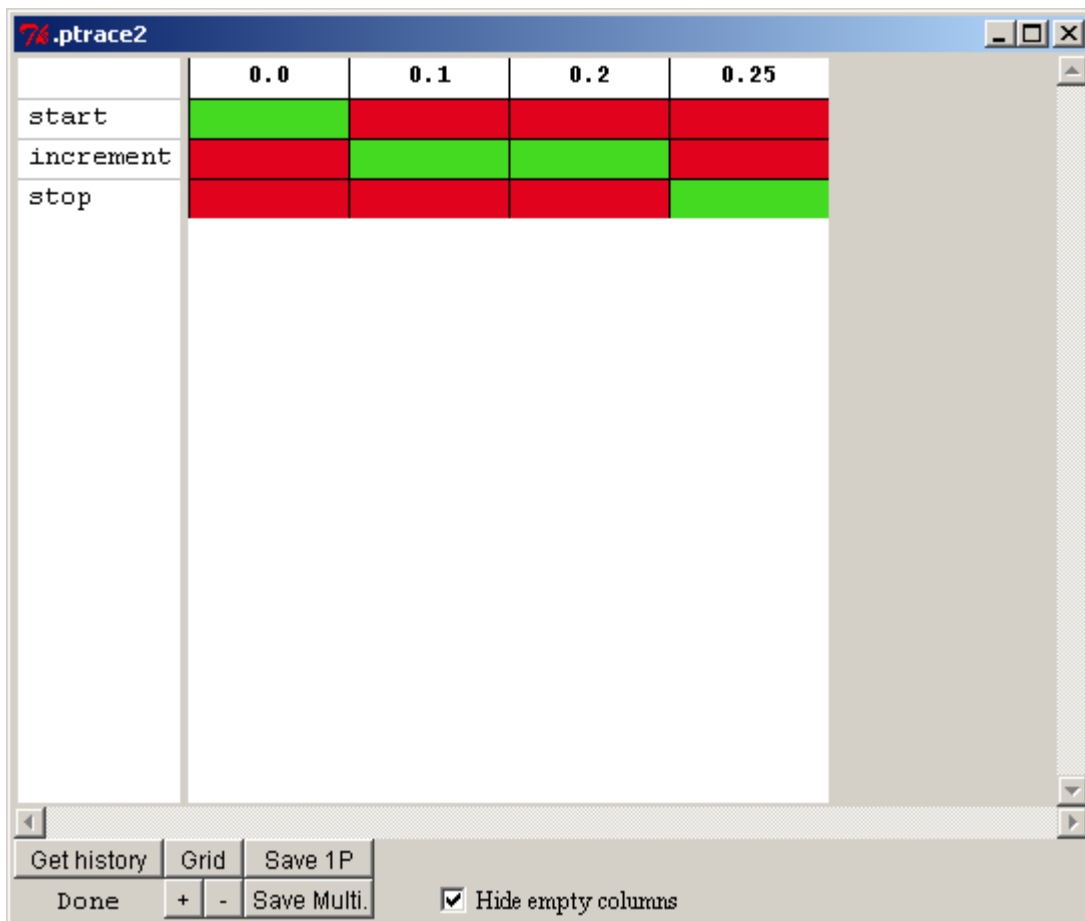
The screenshot shows a window titled ".ptrace2" with a table of data. The table has 7 columns labeled 0.0, 0.05, 0.1, 0.15, 0.2, and 0. The rows are labeled start, increment, and stop. The cells are colored green or red. Below the table is a scroll bar and a control panel with buttons: Get history, Grid, Save 1P, Done, +, -, Save Multi., and a checkbox for Hide empty columns.

	0.0	0.05	0.1	0.15	0.2	0
start	Green	Red	Red	Red	Red	Red
increment	Red	Red	Green	Red	Green	Red
stop	Red	Red	Red	Red	Red	Green

Control panel: Get history, Grid, Save 1P, Done, +, -, Save Multi., ☐ Hide empty columns

The left column displays all the names of the productions in the model, one per row, and if you click on the name of a production then it will open a new Procedural Viewer window with that production selected for viewing the production’s text and parameters (assuming that the model is still currently available in the Environment). To the right of that there is

a column for each time that there was a conflict-resolution event in the model with the time of that event listed at the top (times increasing to the right). By default each conflict-resolution event will have a column. However, there is a parameter called **:draw-blank-columns**, which defaults to **t**, but can be set to **nil** if you do not want to display columns for conflict-resolution events that did not result in the selection and firing of a production. Alternatively, you can check the “Hide empty columns” box at the bottom of the window which will cause it to redraw the graph with the empty columns removed. Here is that same display after checking the “Hide empty columns” box:



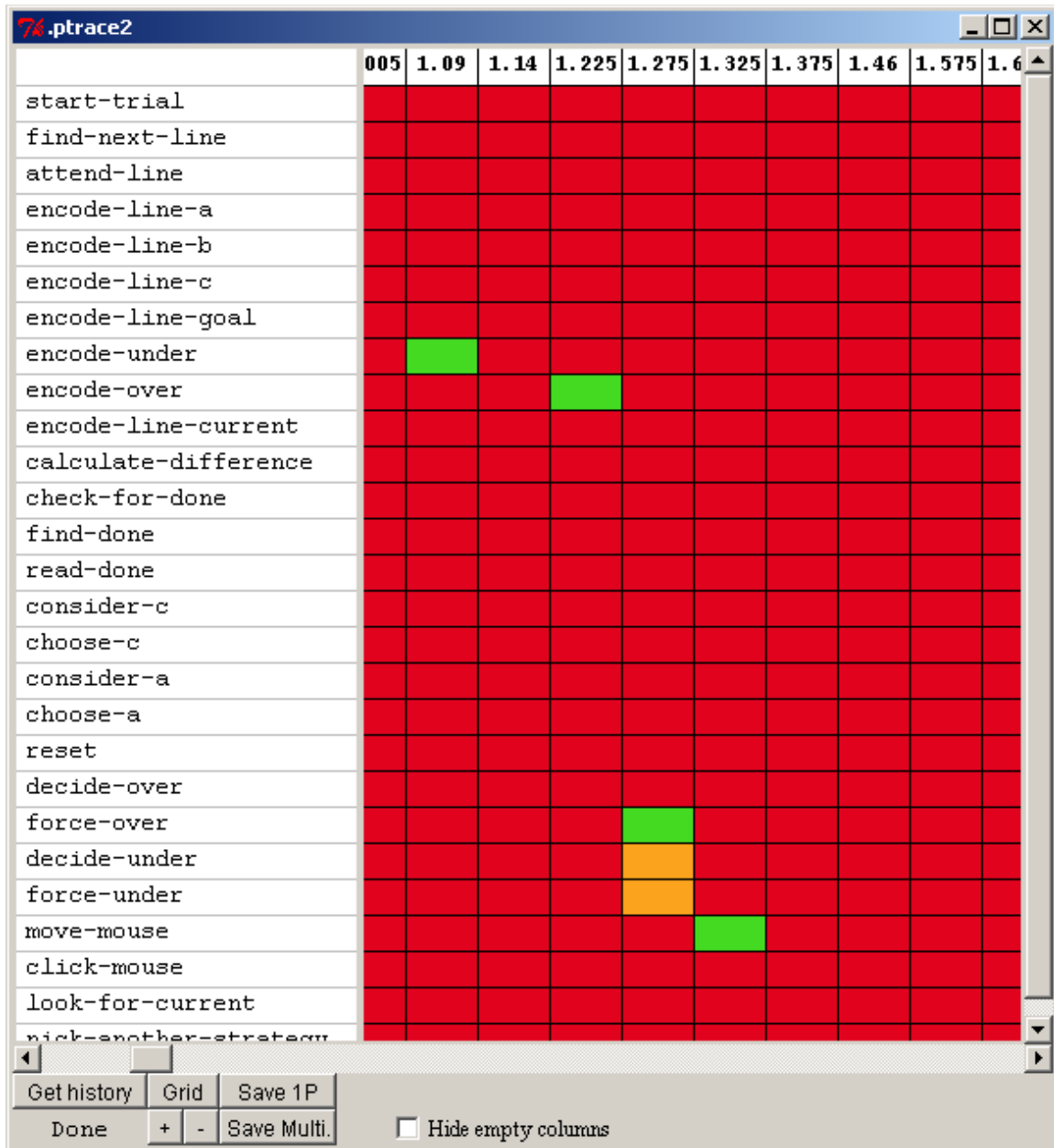
The color of the cell for a production’s row in a column indicates whether or not that production matched during that conflict-resolution event and whether or not it was the production which was selected. If the cell is green, then the production matched and was selected. If the cell is orange then the production matched but it was not selected, and if



the cell is red then the production did not match. The colors used can be changed by setting the **:p-history-colors** parameter. It takes a list of three color string values (as described for the tracing tools) and uses those colors for the selected, matched, and mismatched items respectively. If a production was generated later in a run, typically through production compilation, then for the columns of the times before it existed it will not have any of those colors and will show up as the background white.

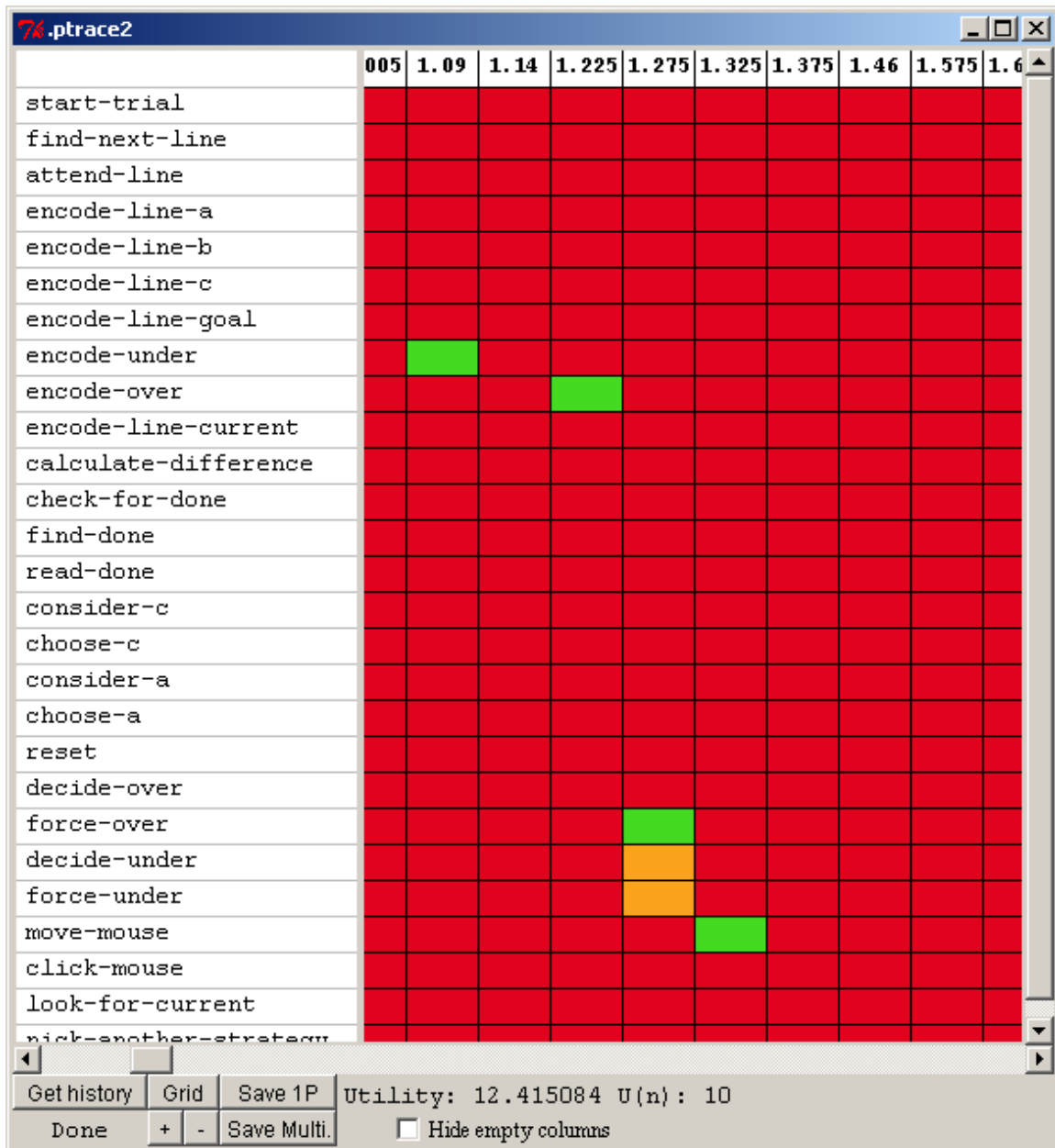
The scroll bar along the bottom of the display allows you to scroll through the history. The “+” and “-” buttons allow you to zoom in or out on the display, and the “Grid” button cycles through three options for whether or not the black grid lines are drawn for the columns and rows: both drawn, only the row lines, none of the lines.

Below is a run from the bst-nolearn model from unit 6 of the tutorial zoomed out and scrolled to see some productions which competed, in this case the force-over, decide-under, and force-under productions:

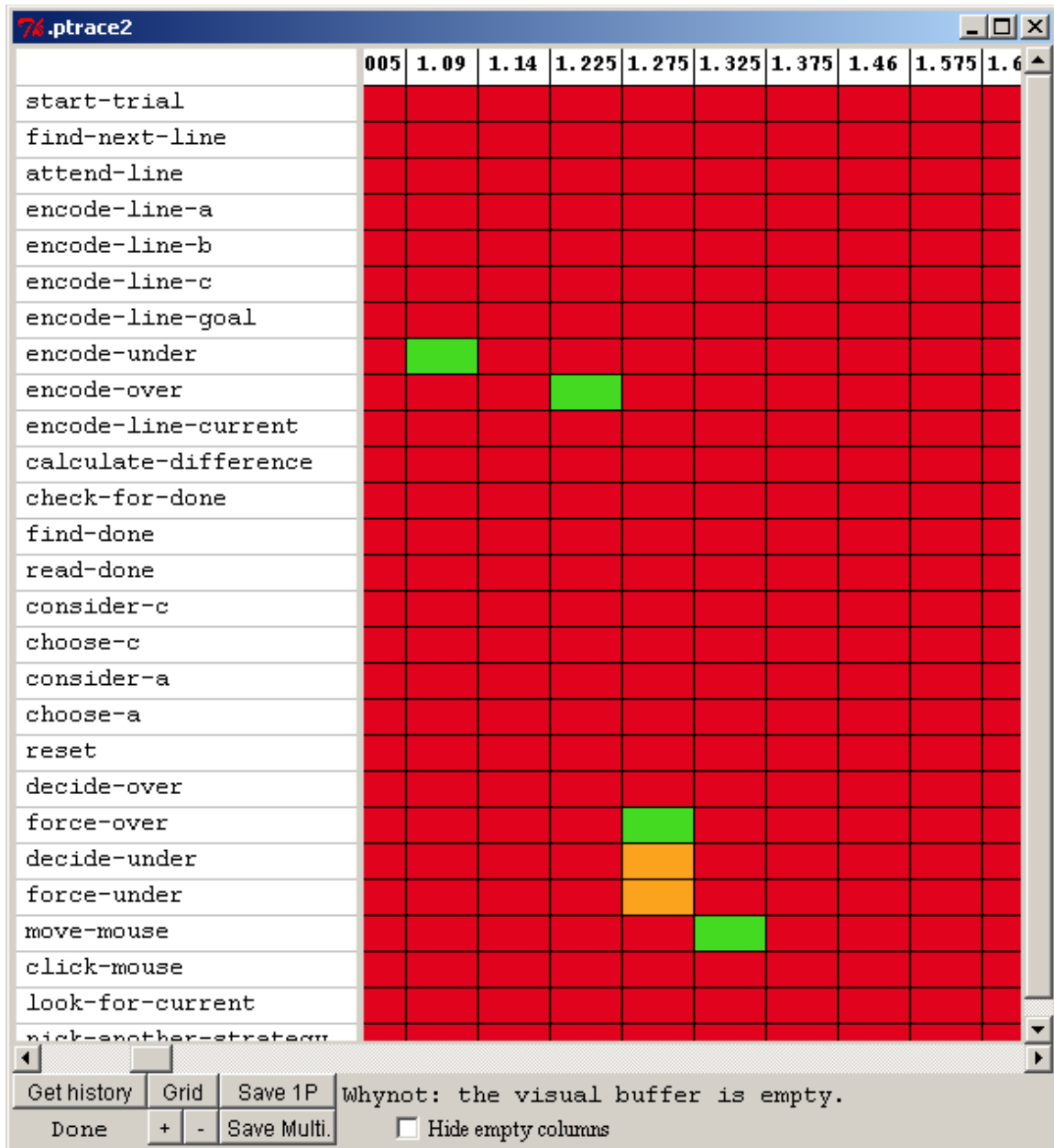


Placing the mouse cursor over the cells in the display will result in additional information being displayed along the bottom of the window. If the cell is orange or green, because the production did match during that conflict-resolution event, then both the noisy utility value of that production at that time (which is what determined which one was chosen) and the true  $U(n)$  value of the production at that time will be displayed. Here is that

display with the mouse over the force-over production's cell at time 1.275:



If the mouse is placed over one of the red cells, a production which did not match at that time, then the reason returned from the ACT-R **whynot** command at that time is displayed to indicate why that production did not match. Here is the display with the mouse over the encode-over cell at time 1.14:

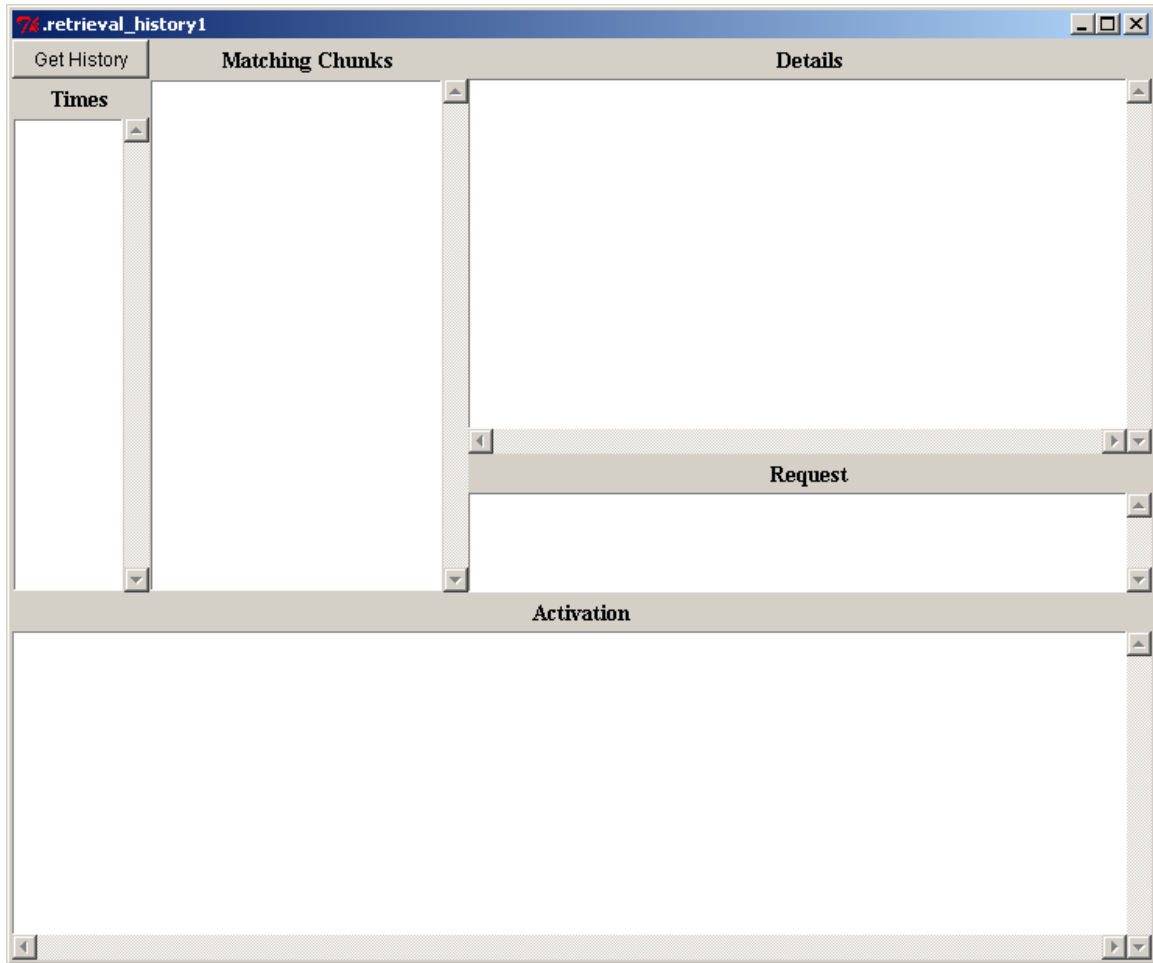


One potential use for this tool is in investigating the productions which are generated through production compilation. Often many new productions are generated and it can be difficult to determine which ones are becoming generally useful or which ones are never being used. This tool would provide some graphic feedback to help locate the learned productions which are never matching and those which are matching and being selected often.

The “Save 1P” and “Save Multi.” Buttons at the bottom of the display can be used to save an image of the entire production matching grid in the same way the buttons with those names can be used save the graphic trace displays. The “Save 1P” button saves an image of the graphic trace as an Encapsulated PostScript file as a single page graphic. The “Save Multi.” button saves an image of the graphic trace as a PostScript file saved as a multiple page document generated in landscape mode.

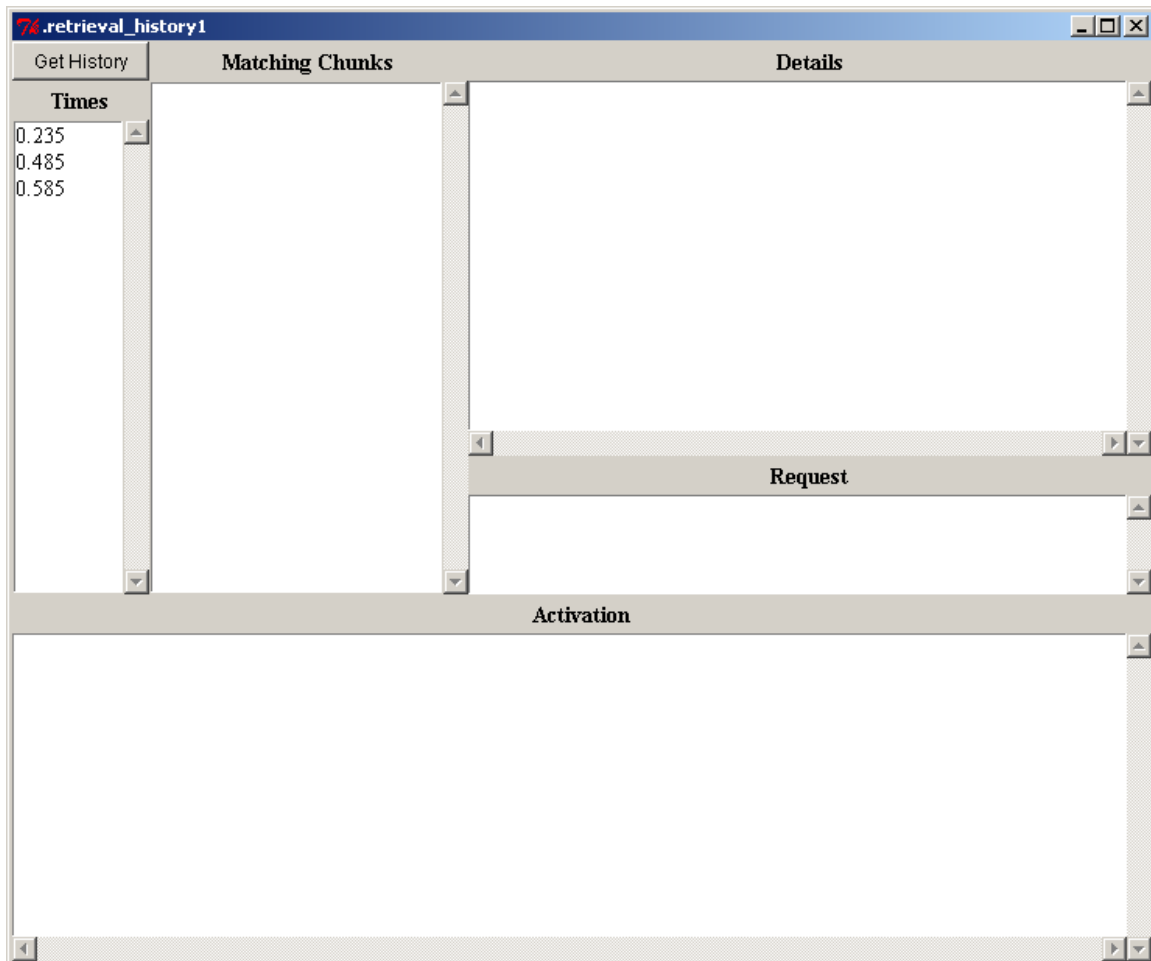
## Retrieval History

Pressing this button opens a new “Retrieval History” window for the current model and any number of those windows may be open. Here is a display of the window without any data shown (how it will always appear upon opening):



To use the tool either a “Retrieval History” window needs to be open before running the model or the **:save-dm-history** parameter and the **:sact** parameter need to be set to **t** in the model to make sure that the data is recorded. After running the model you need to press the “Get History” button in the upper-left corner of the “Retrieval History” window to get the history data displayed. Here is what the window shows after pressing “Get

History” following a run of the fan model from unit 5 of the tutorial for the sentence “the hippie is in the park”:



The left column displays all the times at which a retrieval request was made. Selecting one of those times will cause the “Matching Chunks” section of the window to list all of the chunks that were in declarative memory and matched the request at that time. The item at the top of the list is the one which was retrieved, or will be the keyword :retrieval-failure if no chunk was retrieved. The rest of the chunks in the list are in no particular order. The “Request” section of the window will display the request which was made at that time.

Selecting one of the chunks from the “Matching Chunks” list will result in the “Details” section being filled with a printing of the chunk along with the parameter values for that chunk at the time of the retrieval request. Note that the tool assumes the normal

operation of the system in which chunks in declarative memory cannot be changed. Thus, while the parameters were recorded at the time of the request the printing of the chunk itself is based on the chunk at the current time in the model. The “Activation” section of the display will show the detailed activation trace of how that chunk’s activation was computed at that time. Here is the tool after selecting the 0.585 second time and the first chunk on the resulting list, p1:

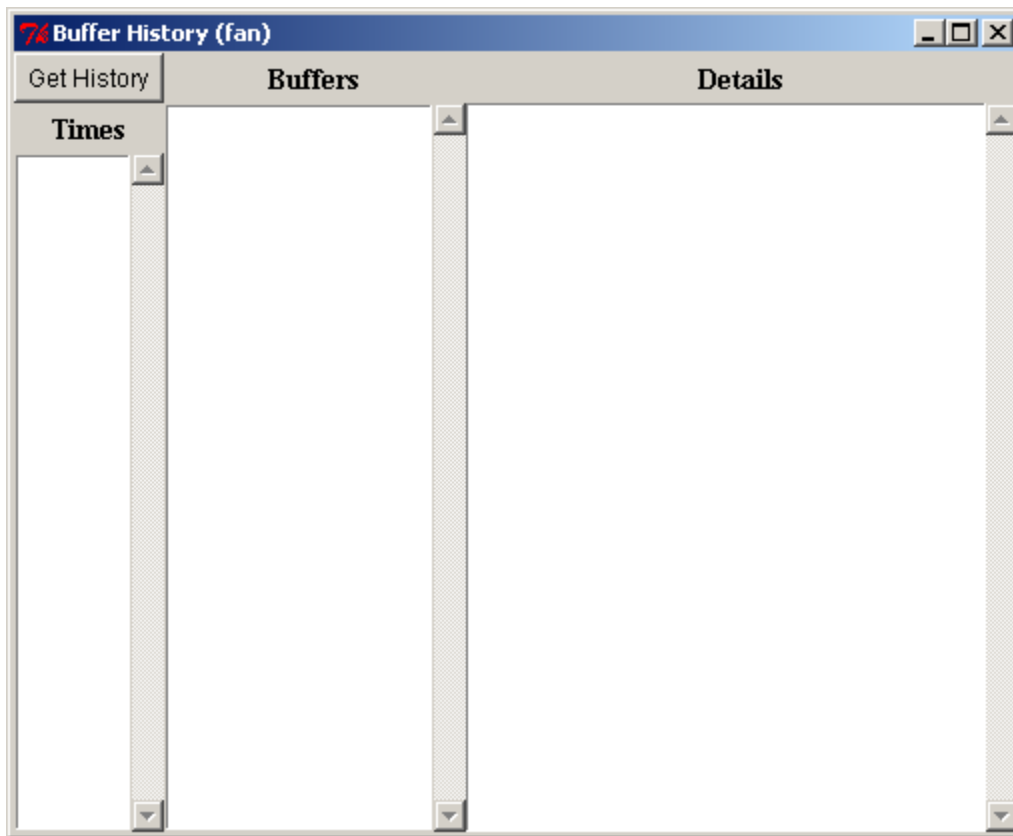
The screenshot shows a software interface titled "74.retrieval\_history1". It has three main panes: "Get History", "Matching Chunks", and "Details".

- Get History:** A list of times: 0.235, 0.485, and 0.585. The time 0.585 is selected and highlighted in blue.
- Matching Chunks:** A list of chunk identifiers: p1, p3, and p2. The chunk p1 is selected and highlighted in blue.
- Details:** This pane displays information for the selected chunk P1.
  - ISA COMPREHEND-SENTENCE
  - RELATION IN
  - ARG1 HIPPIE
  - ARG2 PARK
  - Declarative parameters for chunk P1:
    - :Activation 0.214
    - :Permanent-Noise 0.000
    - :Base-Level 0.000
    - :Source-Spread 0.214
    - :Sjis ((P1 . 1.6) (IN . -1.0390574) (HIPPIE . 0.213'
    - :Last-Retrieval-Activation 0.214
  - Request:**
    - ISA COMPREHEND-SENTENCE
    - ARG1 HIPPIE
  - Activation:**
    - Computing activation for chunk P1
    - Computing base-level
    - Starting with blc: 0.0
    - Total base-level: 0.0
    - Computing activation spreading from buffers
    - Spreading 1.0 from buffer IMAGINAL chunk COMPREHEND-SENTENCE0-0
    - sources of activation are: (HIPPIE PARK)
    - Spreading activation 0.10685283 from source HIPPIE level 0.5 times Sji 0.21370566
    - Spreading activation 0.10685283 from source PARK level 0.5 times Sji 0.21370566
    - Total spreading activation: 0.21370566
    - Adding transient noise 0.0



## Buffer History

Pressing this button opens a new “Buffer History” window for the current model and any number of those windows may be open. Here is a display of the window without any data shown (how it will always appear upon opening):



To use the tool either a “Buffer History” window needs to be open before running the model or the **:save-buffer-history** parameter needs to be set to **t** in the model to make sure that the data is recorded. After running the model you need to press the “Get History” button in the upper-left corner of the “Buffer History” window to get the history data displayed. Here is what the window shows after pressing “Get History” following a run of the fan model from unit 4 of the tutorial for the sentence “the hippie is in the park”:

Buffer History (fan)		
Get History	Buffers	Details
Times	retrieval	
0.0	imaginal	
0.05	manual	
0.1	goal	
0.185	imaginal-action	
0.235	vocal	
0.25	aural	
0.25	production	
0.3	visual-location	
0.35	aural-location	
0.435	temporal	
0.485	visual	
0.535		
0.585		
1.094		
1.144		
1.444		

The left column displays all the times at which a change occurred in some buffer, where a change is any of: the buffer clearing, a chunk being placed into the buffer, the chunk in the buffer being modified, or a change in one of the buffer queries of “state free”, “state busy” or “state error”. The middle column shows the names of all the buffers in the model. Selecting a time and one of the buffers will result in the “Details” section being filled with the results from calling **buffer-chunk** and **buffer-status** for that buffer at the time specified. Here is the tool after selecting the 0.25 seconds time and the imaginal buffer:

Buffer History (fan)		
Get History	Buffers	Details
Times	retrieval	IMAGINAL: COMPREHEND-SENTENCE0-0
0.0	imaginal	COMPREHEND-SENTENCE0-0
0.05	manual	ISA COMPREHEND-SENTENCE
0.1	goal	RELATION NIL
0.185	imaginal-action	ARG1 NIL
0.235	vocal	ARG2 NIL
0.25	aural	
0.3	production	IMAGINAL:
0.35	visual-location	buffer empty : NIL
0.435	aural-location	buffer full : T
0.485	temporal	buffer requested : T
0.535	visual	buffer unrequested : NIL
0.585		state free : T
1.094		state busy : NIL
1.144		state error : NIL
1.444		

One thing to note is that the information shown for the buffer is how it was reported at the “end” of the time selected. There may have been multiple changes occurring in the buffer during that time step (multiple concurrent events), but only the final state is recorded.

## BOLD tools

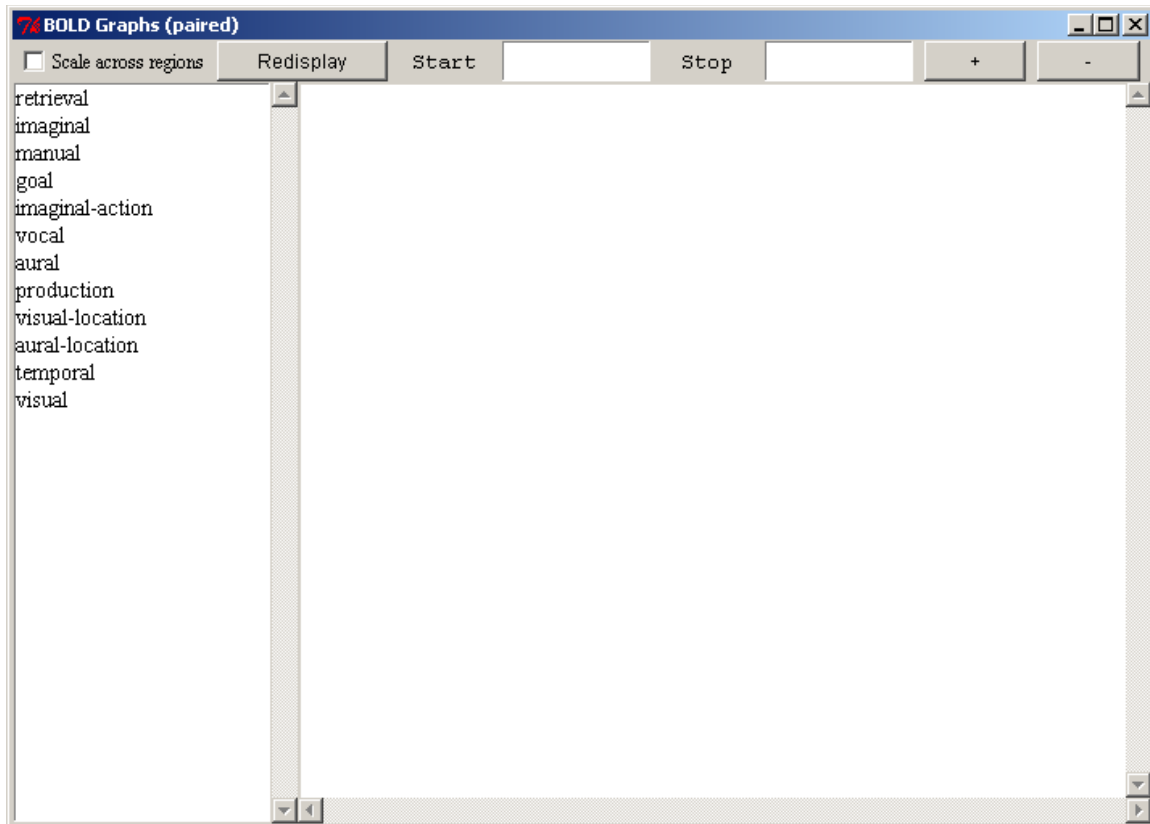
The BOLD tools section provides graphic representations of the BOLD (Blood Oxygen Level Dependent) response prediction data which a model generates. A full description of how that is computed is beyond the scope of this document, but a very brief description will be given here before describing the tools.

For each buffer in ACT-R the pattern of use, as shown in the graphic traces, can be recorded. That recorded pattern of use over a run can then be considered as a metabolic demand on the brain which can be combined with a hemodynamic response function to create a prediction of a BOLD response. Past research has lead to associating each of the buffers in ACT-R with a particular region of the brain. Thus, the patterns of use of the buffers lead to predictions for a BOLD response seen across various areas of the brain.

Similar to the Tracing tools, the BOLD tools require that you set the parameter **:save-buffer-trace** to **t** in the model to record the data required to produce these displays. As with the Tracing tools, some of the BOLD tools can be opened in advance of running the model and will set that parameter automatically, but others will not and should only be opened after the model is done running. The descriptions below will indicate when a particular tool should be used relative to running the model.

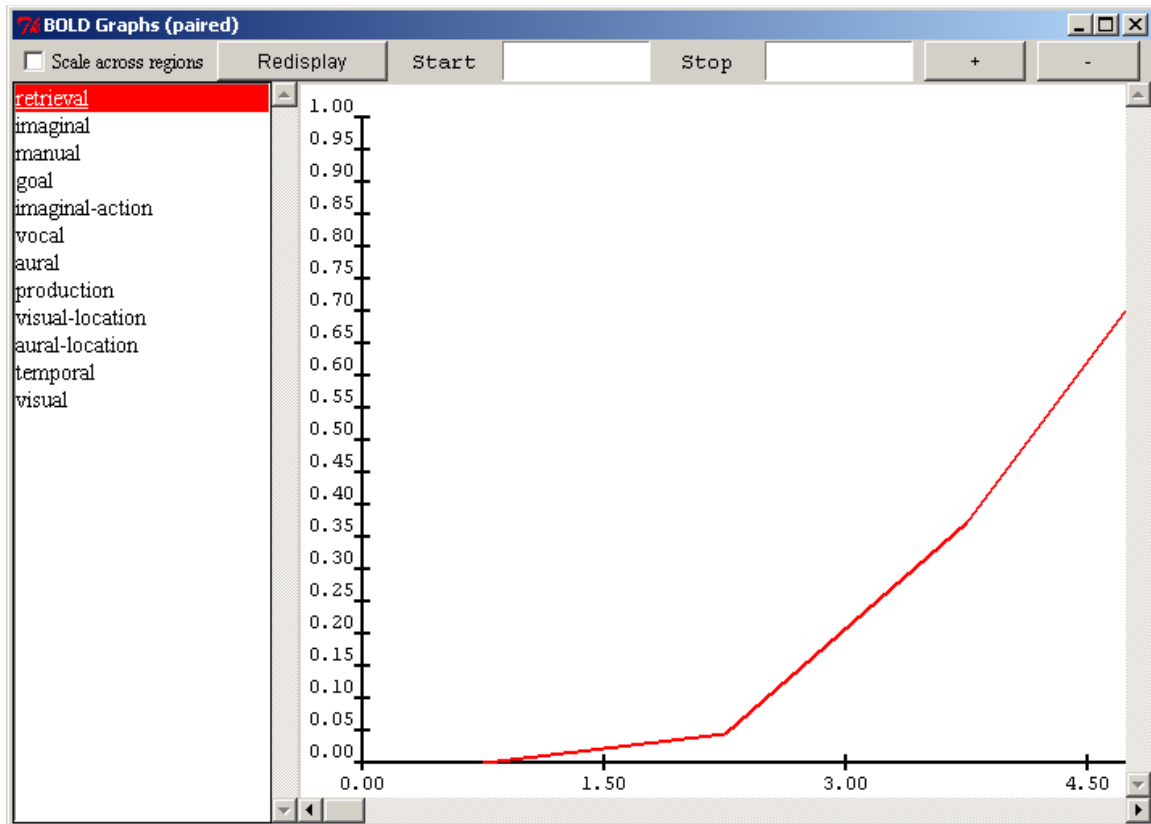
### Buffer graphs

The “Buffer graphs” button will open up a new “BOLD Graphs” window for the current model which looks like this:

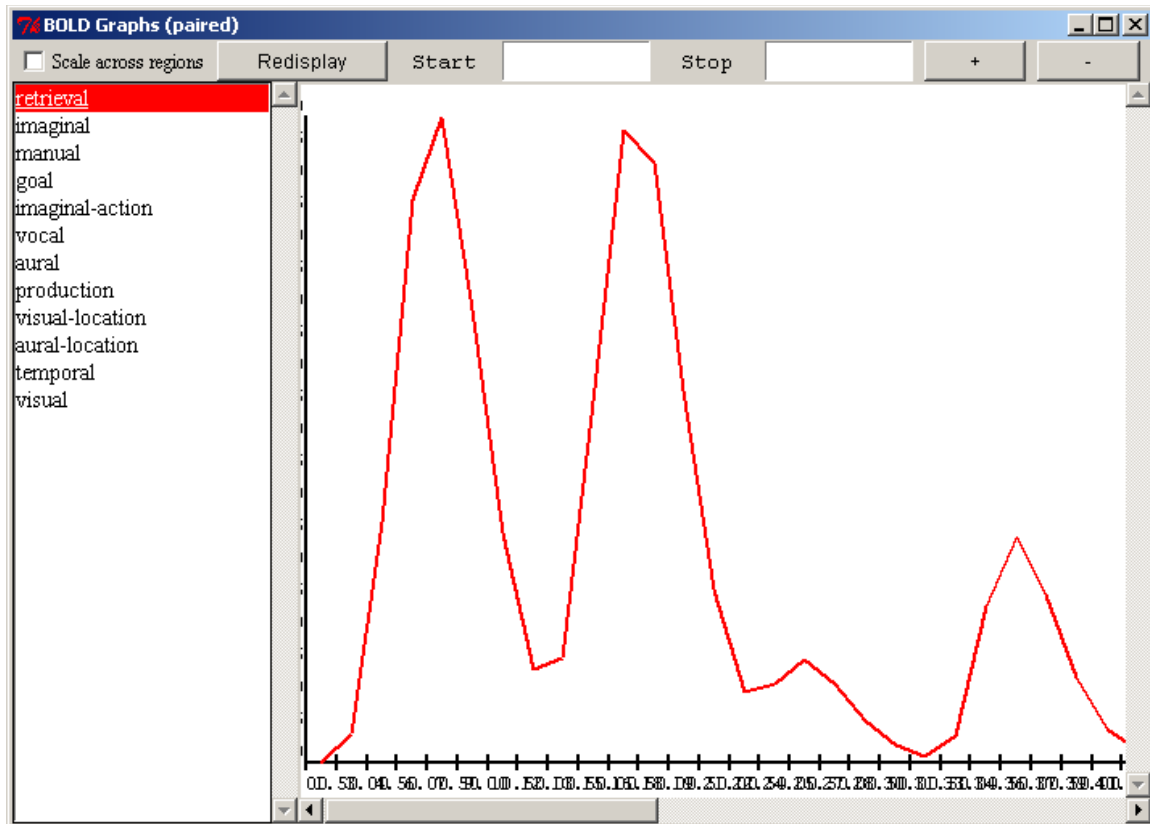


Any number of such windows may be open at a time. Opening this window before the model runs will automatically set the **:save-buffer-trace** parameter to **t**.

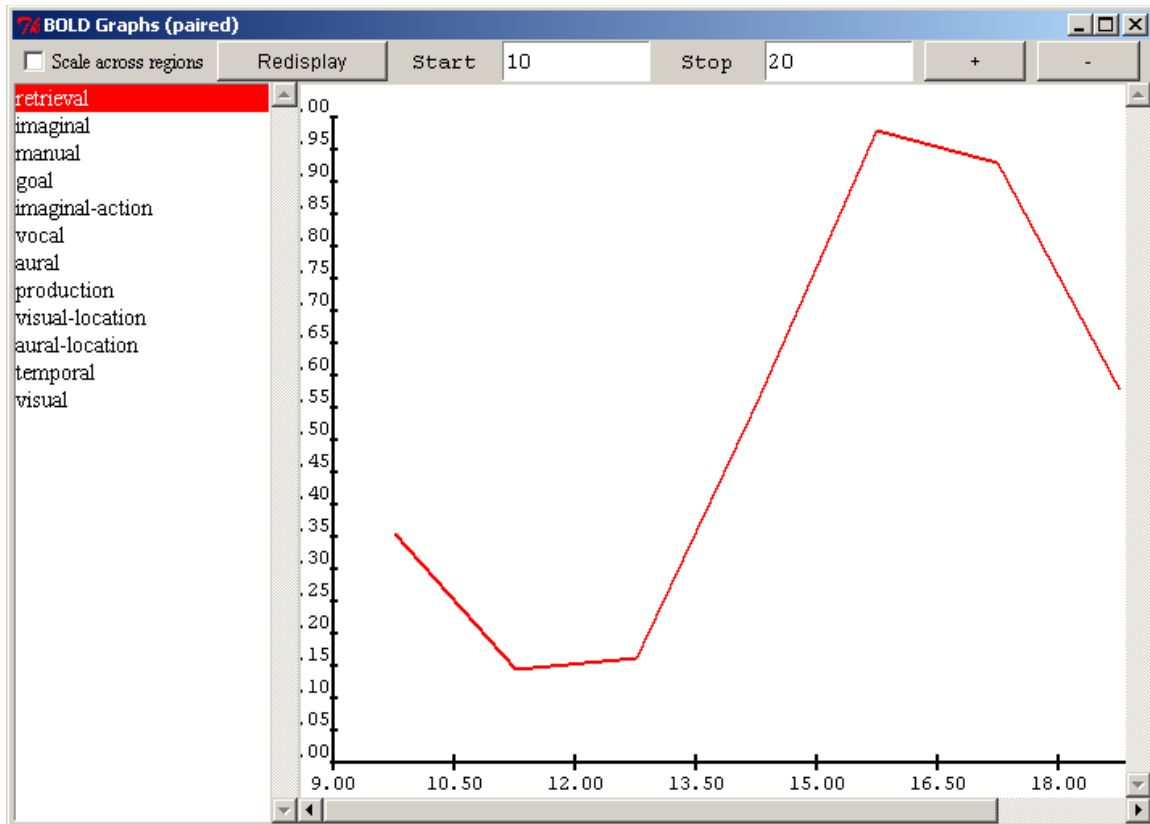
The column on the left side lists all the buffers in the model. Selecting a buffer from the list will result in a graph being drawn in the pane on the right of the window showing the data returned by the ACT-R **predict-bold-response** command for that buffer scaled into the range 0.0-1.0. Here is the graph for the retrieval buffer after running the paired associate learning model from unit 4 of the tutorial for two items five trials each:



The “+” and “-” buttons at the top can be used to zoom in or out on the graph and here is that same graph after zooming out to show more of the data:



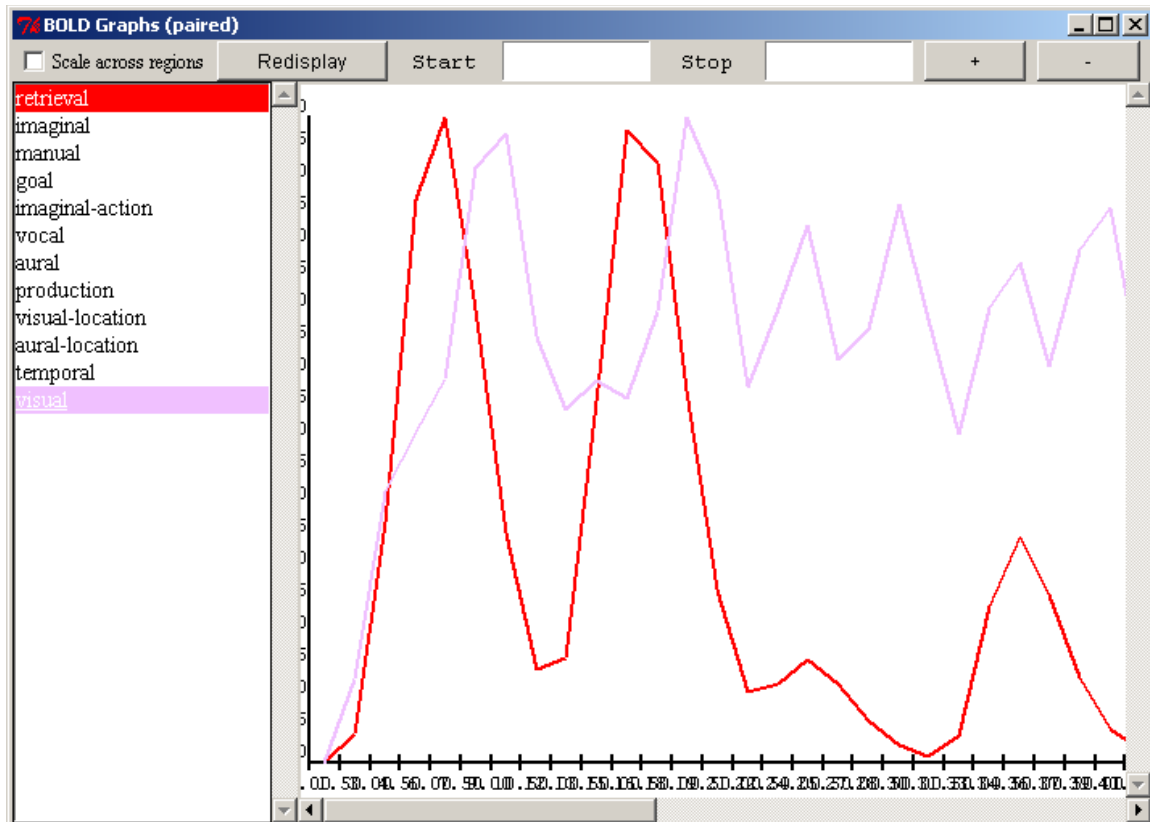
The Start and Stop boxes can be used to restrict the display to a particular segment of the run. Each box can have a time in seconds entered in it. If the Start box is empty then the data is started at time 0s, and if the Stop box is empty then the end time is the current model time. After adjusting the Start and Stop values you must hit the “Redisplay” button to have the graph redrawn. Here is that same trace restricted to the time between 10 and 20 seconds in the run at a different zoom level:



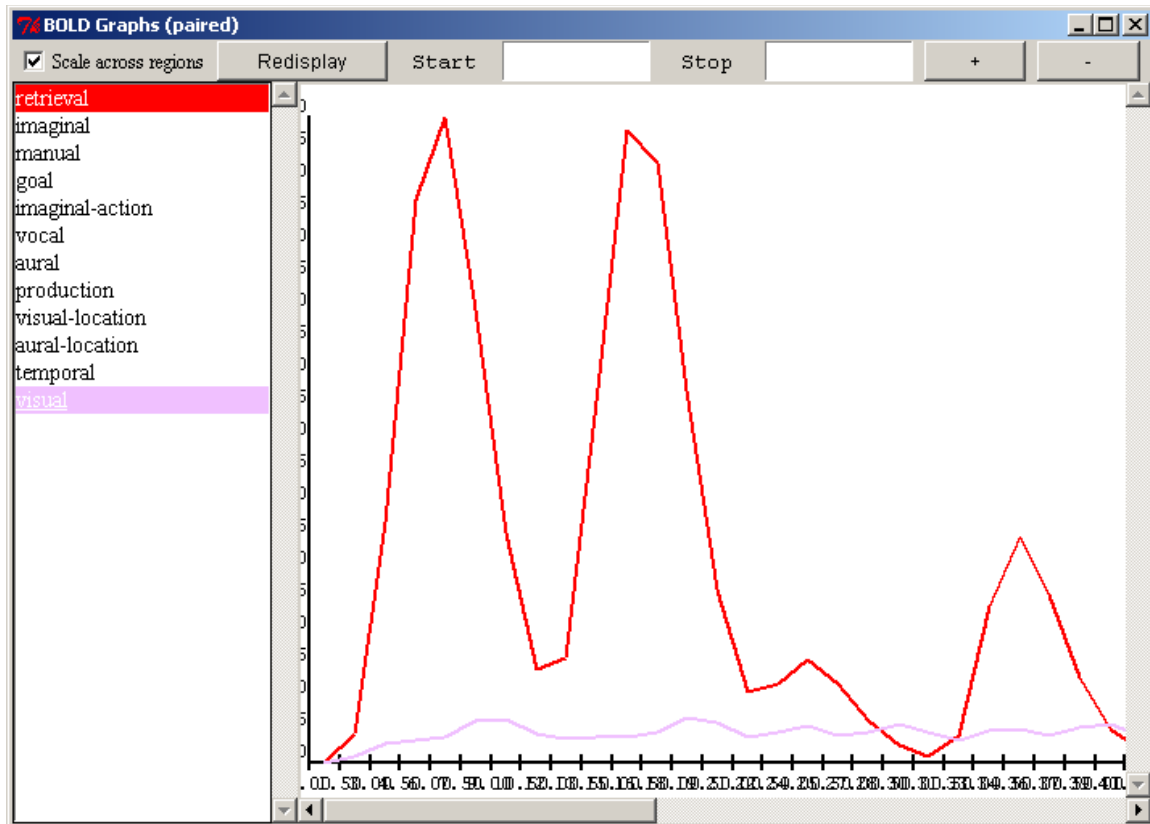
Note that the data may not always fit into exactly the times specified. That is because the data is generated based on an interval specified in the model with the **:bold-inc parameter** which defaults to 1.5 seconds, and adjusting the Start and Stop times does not change the interval used. It always starts incrementing from time 0 and plots the data based on the middle of each the interval.

It is also possible to select more than one buffer in the column on the left. Each selected buffer will be drawn in the current display. The selection color of the buffer corresponds to the color of that buffer's data in the graph. Here is the data from the retrieval and visual buffers both shown in the same display for that task:



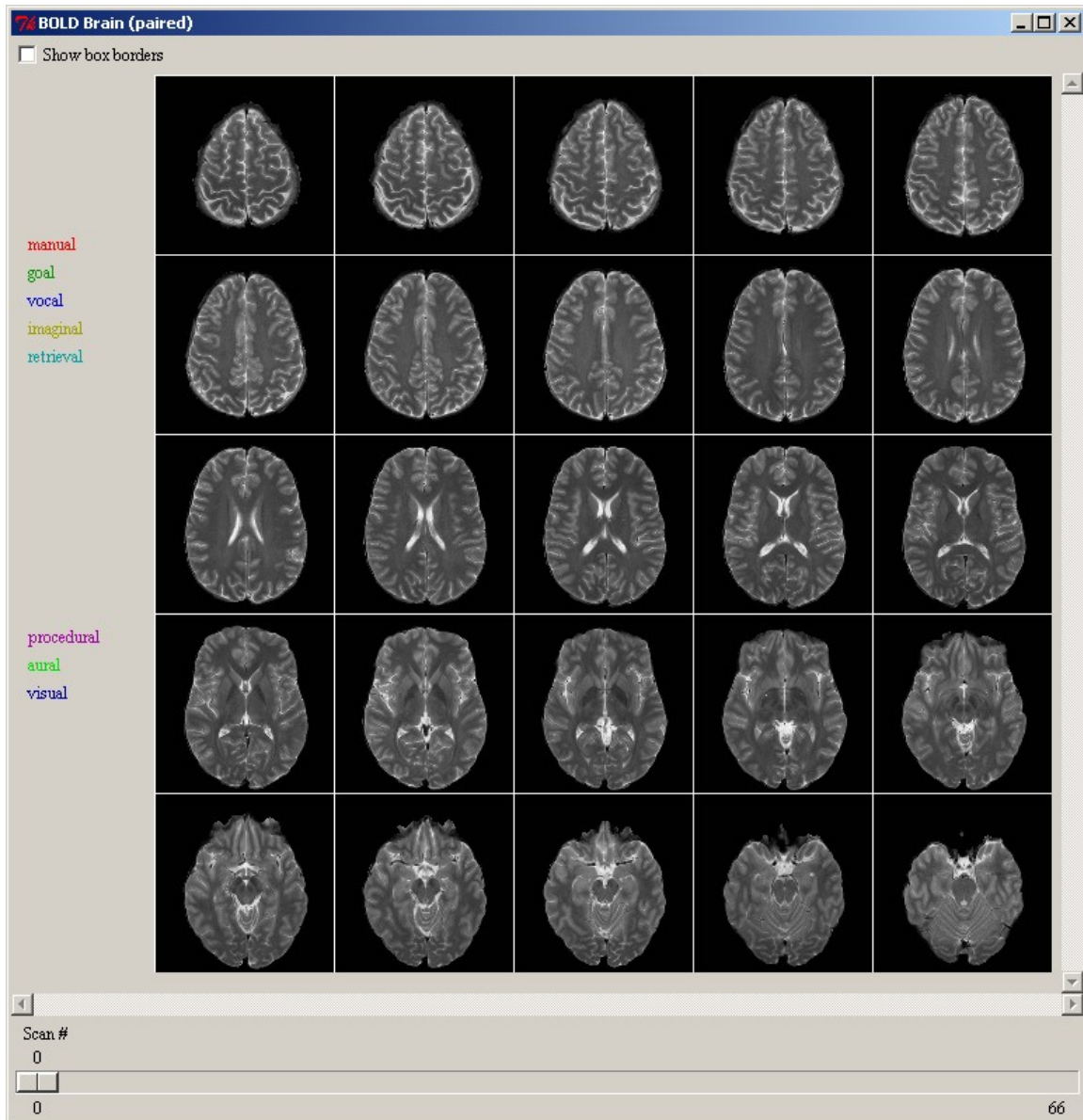


If the “Scale across regions” box is unchecked then for each buffer the BOLD data is scaled to the range of 0.0-1.0 for display based on the maximum value for that buffer. If the “Scale across regions” box is checked, then the data for all buffers is scaled to the 0.0-1.0 range based on the maximum value among all the buffers. This allows one to see the effects more clearly for a given buffer or to compare the results from different regions when desired. Here is the same data with the “Scale across regions” box checked:

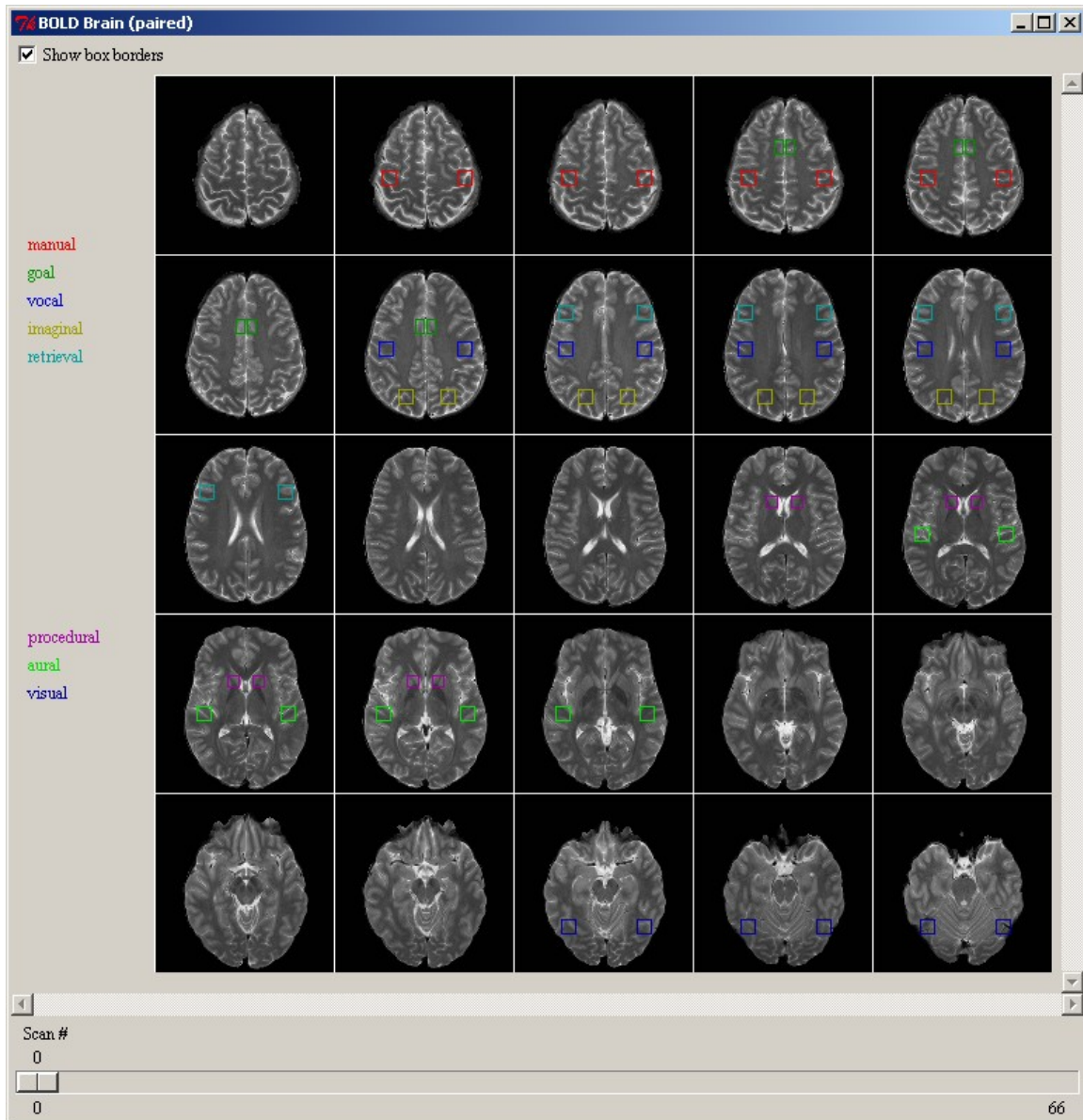


## 2D brain

The “2D brain” button will open a “BOLD Brain” window for the current model if there is not already one open or bring it to the front if it is already open because there can be only one open “BOLD Brain” window per model. The “BOLD Brain” window shows image slices of a reference brain and displays the BOLD data values for the buffers as color coded boxes in the images in the areas with which the corresponding buffer has been associated. This tool should only be opened after the model runs because it does not refresh its data after it has been opened. Because of that, it does not set the **:save-buffer-trace** parameter. Here is what the window looks like:

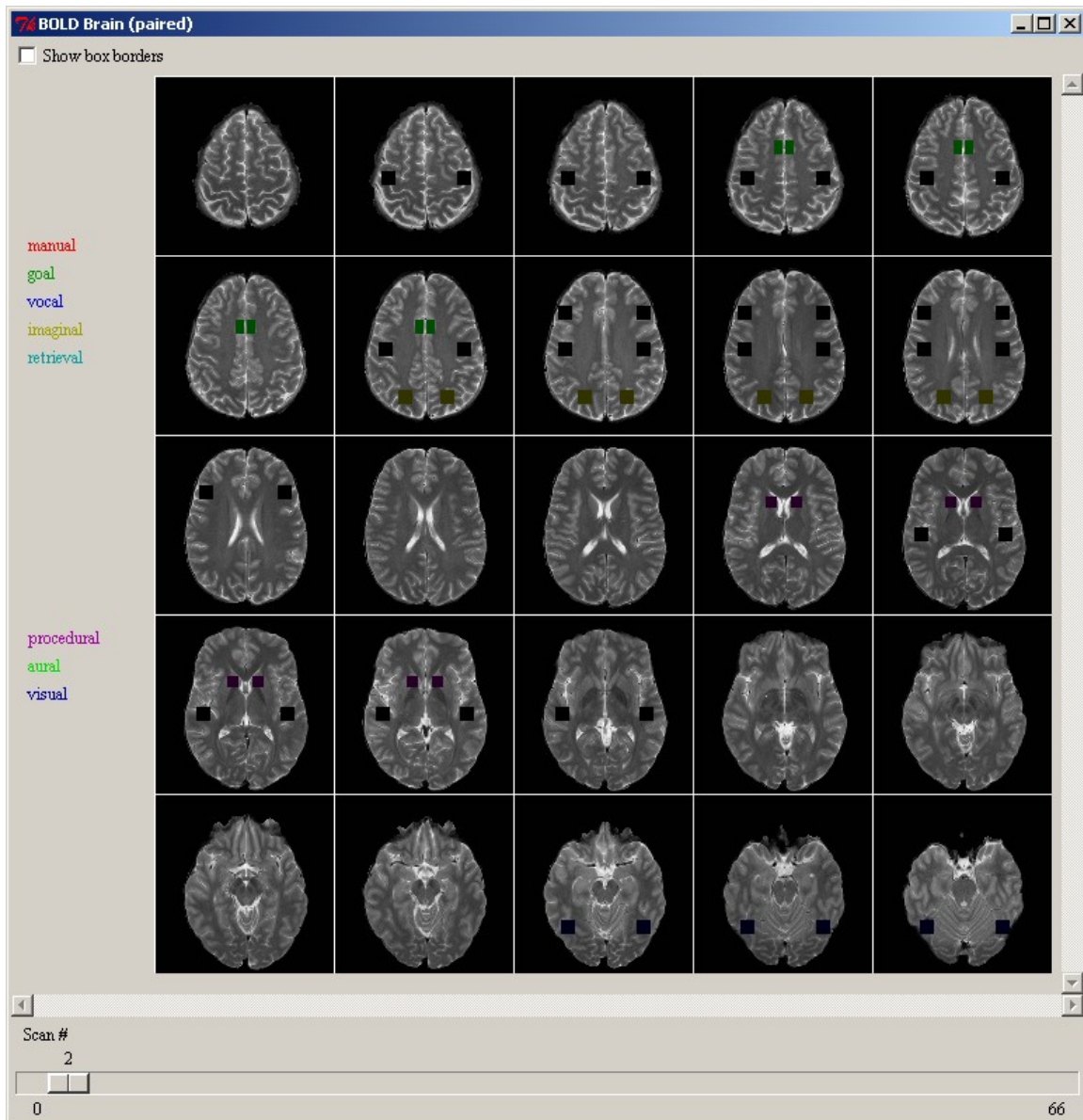


The buffers for which a brain association is defined are displayed on the left in the color which will be used to draw them in the images and in the order in which they are drawn in the images i.e. the manual buffer is drawn in the top slices and the visual buffer in the lower ones. If the “Show box borders” button is selected then a colored box will be displayed for each buffer whether or not there is any activity. This can be used to see where the regions are in the absence of any activity and looks like this without any other data displayed:

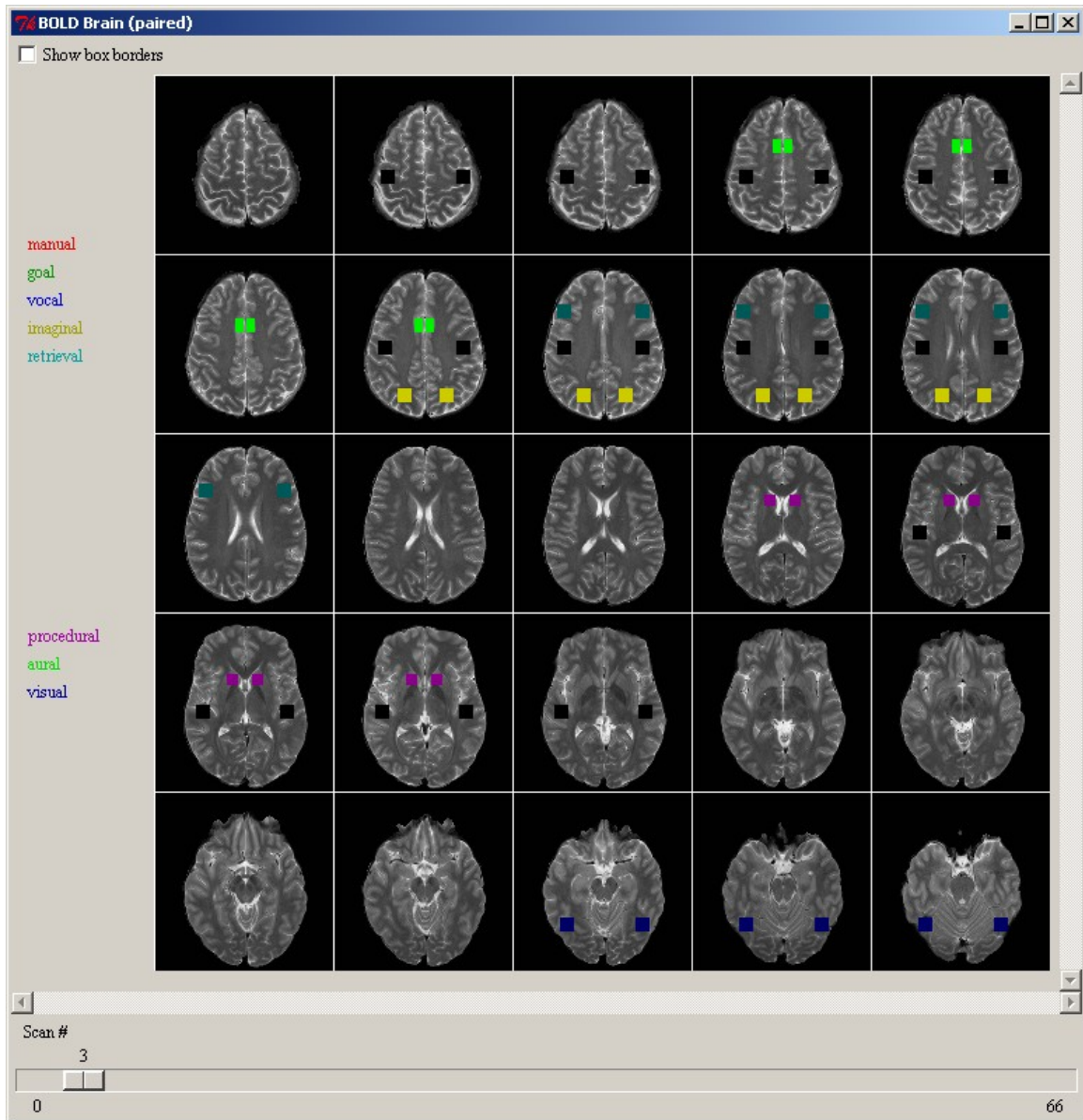


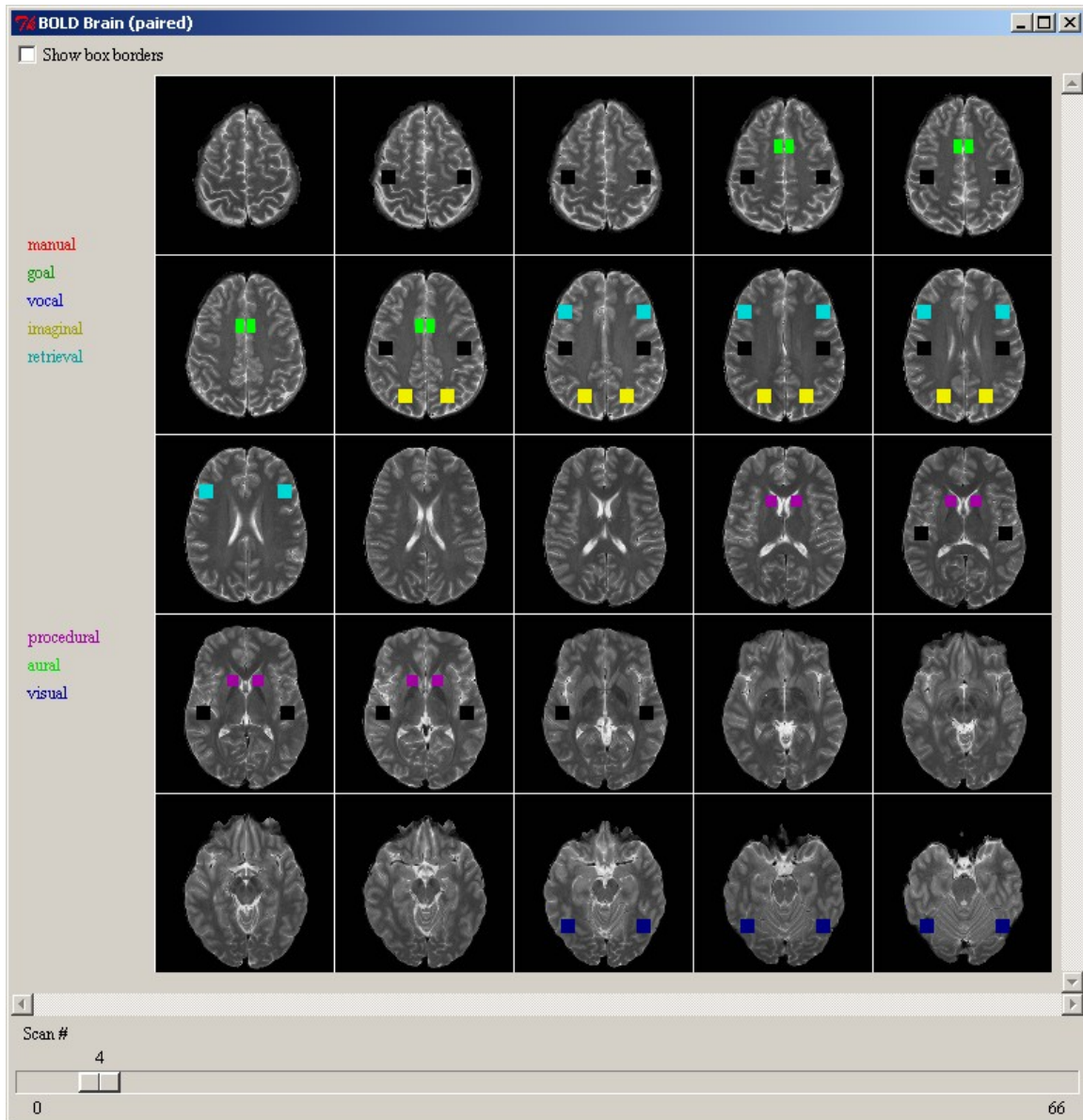
The slider along the bottom allows one to select the specific scan from the run for which the data should be displayed. The scans occur based on the value of the **:bold-inc** parameter, with a scan occurring every **:bold-inc** seconds. On each scan the brightness of the corresponding boxes indicates the BOLD activity in that buffer. Each buffer has its BOLD data scaled from 0.0-1.0 individually and that is used as a brightness value in displaying the color. Thus, if there is no activity, a value of 0, then the box will be black and if there is a lot of activity, a value near 1.0, then the box will be brightly colored. Here is an image from the paired associate model as run for the graphing data at scans 2-4

showing activity in several buffers increasing at the start of the task:





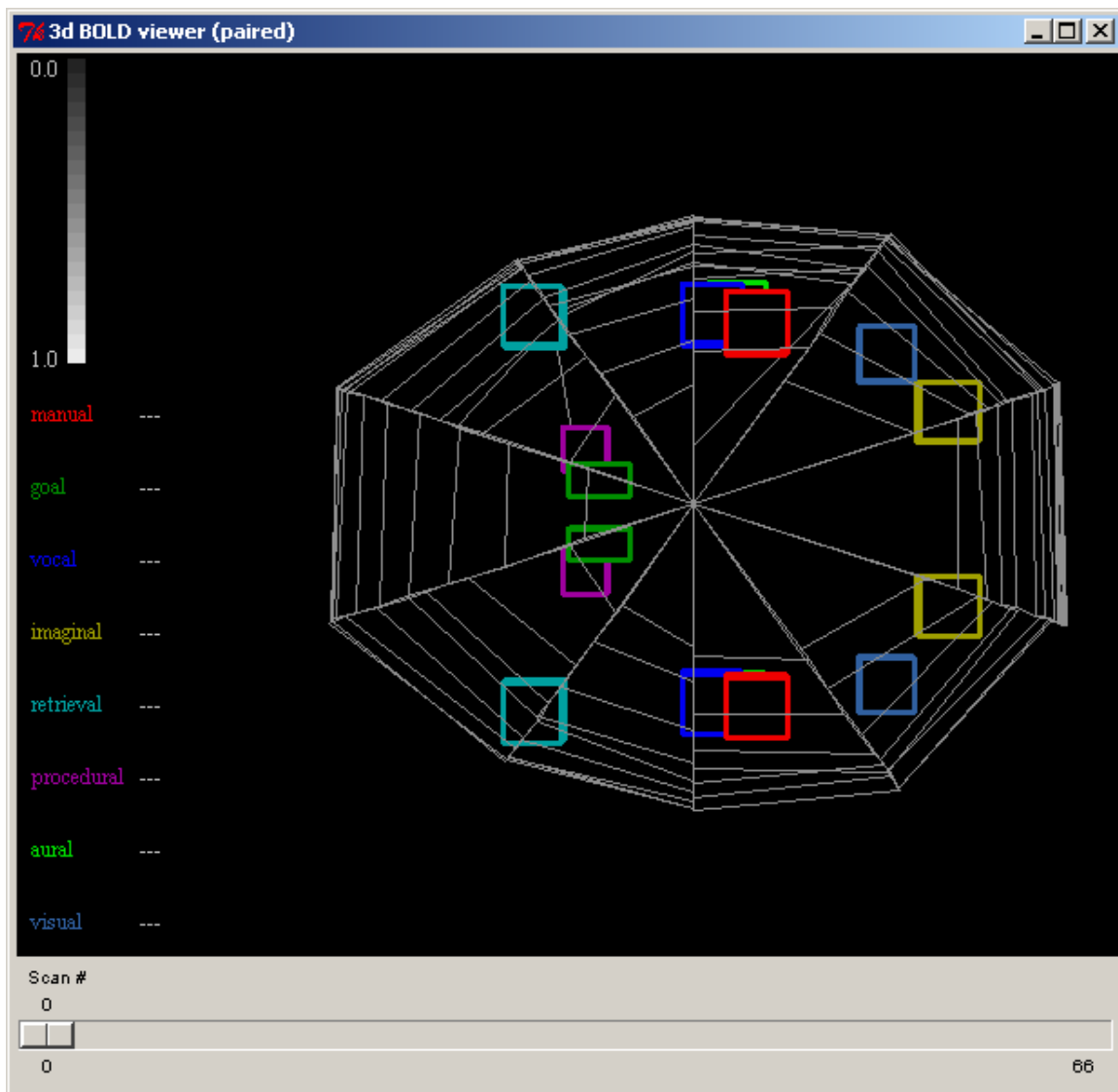




### 3D brain

The “3D brain” button will open the “3d BOLD viewer” window for the current model if it is not already open or bring it to the front if it is already open because there can be only one open per model. The “3d BOLD viewer” window shows the same information as the 2d viewer described above, except instead of the using images from a reference brain the boxes are drawn in a crude three-dimensional wireframe brain model. Like the 2D viewer,

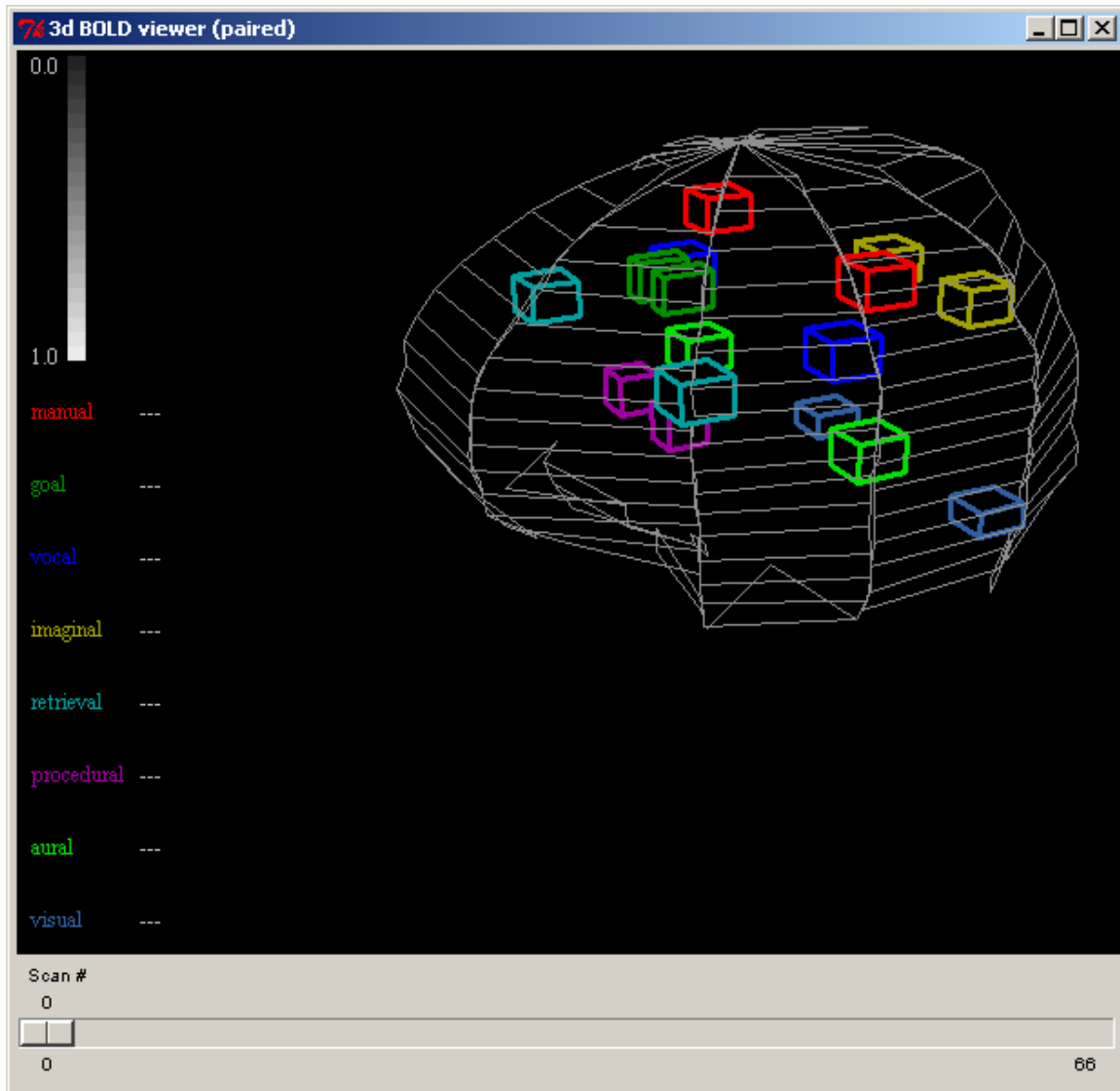
the data for the display is not updated after opening the window and thus it should not be opened until after the model has been run. Here is what the window looks like by default:



The buffers for which a brain association is defined are displayed on the left in the color which is used to draw the outline of the region's box in the image. The default view is top-down with the front of the brain to the left, but the brain can be rotated and moved by clicking on it and moving the mouse. If the left mouse button is clicked and held moving the mouse will rotate the brain around its center point. If the right mouse button is clicked and held moving the mouse up and down will zoom in and out on the image, and if the

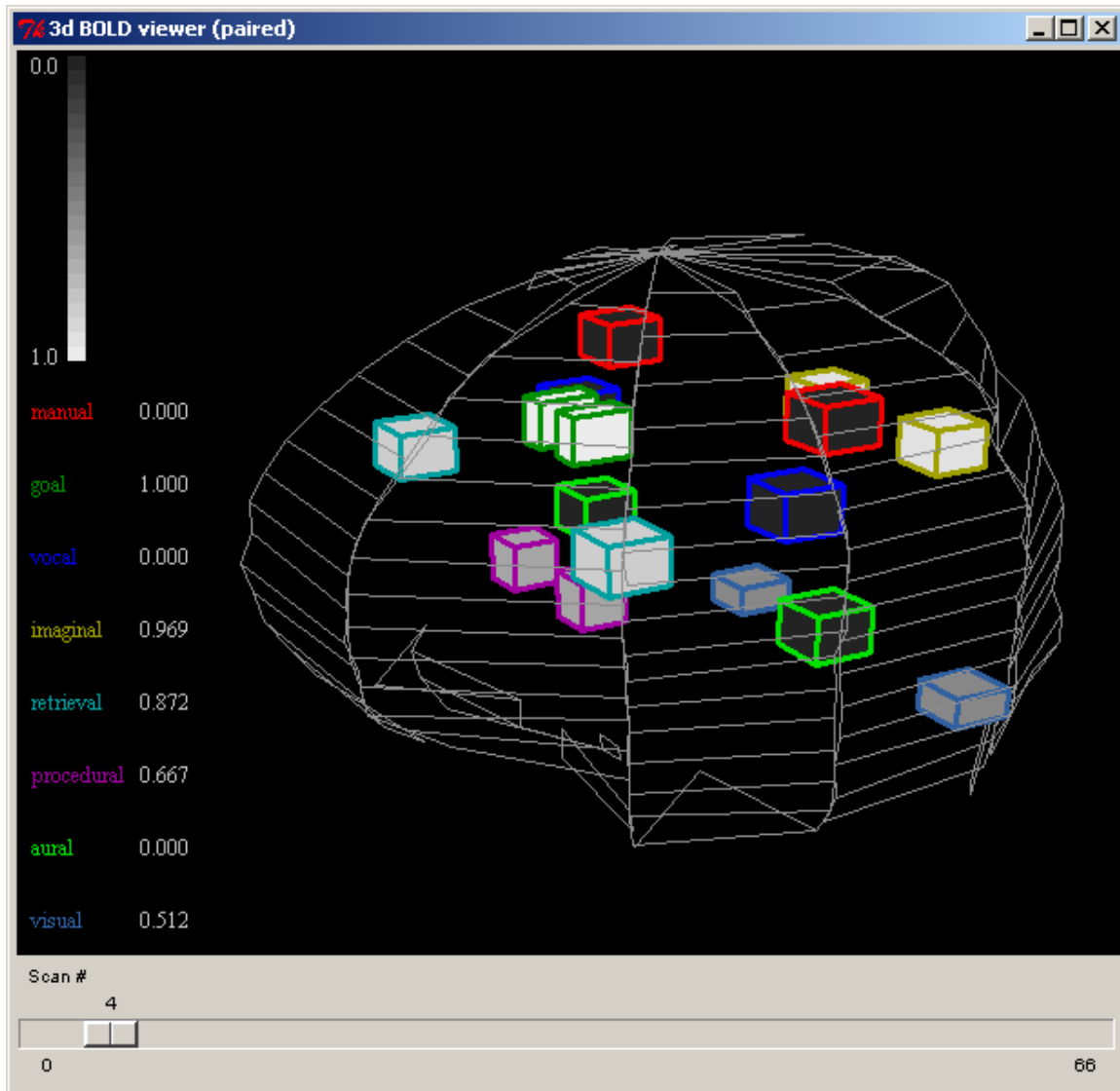


middle mouse button is clicked and held moving the mouse will move the brain around in the window without rotating it. Here is a view of the image after it has been moved and rotated:



The slider along the bottom allows one to select the specific scan from the run for which the data should be displayed in the same way that it does for the 2D viewer. On each scan the boxes for each buffer will be filled with a gray-scale color which indicates the BOLD activity in that buffer with the reference colors indicated by the gradient shown in the upper left of the window. The box outlines will always be drawn with the brightly colored

edges. Each buffer has its BOLD data scaled from 0.0-1.0 individually and that is used as a brightness value in displaying the color and that number is also shown on the left of the window after the buffer name. Here is an image from the paired associate model as run for the graphing data on scan 4 showing activity in several buffers:



## Run-time 3D brain

The "Run-time 3D brain" button will open the "3d BOLD run-time viewer" window for

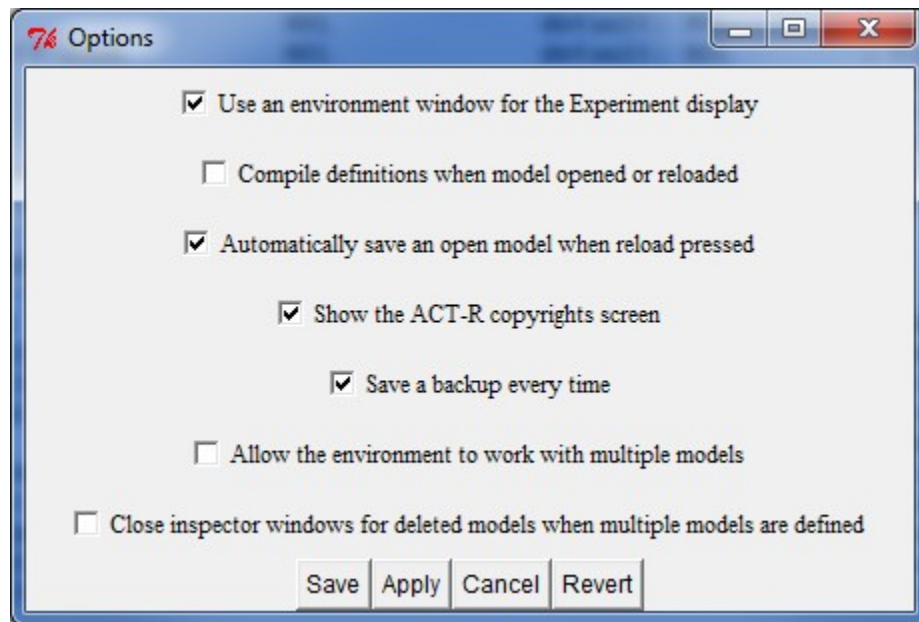
the current model if it is not already open or bring it to the front if it is already open because there can be only one open per model. The “3d BOLD run-time viewer” window shows the same information and works the same way as the 3D brain viewer described above, except that it does not have a slider bar for picking scans and instead is updated as the model runs. It should be opened before the model is started. It automatically sets the **:save-buffer-trace** parameter and will update as the model runs. This should only be used if the model is running in real time or if the stepper is being used because otherwise the display will not be able to keep up with the model data and all of the Environment’s tools will become non-responsive as a result.

## Miscellaneous

The Miscellaneous section contains the controls which are not involved with the actually modeling and thus do not belong to one of the other sections. The only control in the current Environment is the Options button which allows the user to specify some settings for how the Environment should operate.

### Options

Pressing the “Options” button will bring up the “Options” window if it is not already open or bring it to the front if it is open because there can only be one such window open at a time. Here is what the “Options” window looks like:



It has several options which can be enabled or disabled by checking or unchecking them. When the window is opened the current setting of the options will be shown in the selections. Some of the options are only meaningful when using the editor with the standalone version of the Environment, but they will all be described below for completeness. First, the buttons at the bottom of the window will be described, and then each of the options themselves.

## **Save**

Pressing the “Save” button will apply the current selections to the Environment as well as save them to a file so that the next time the environment is started those settings will be used instead of the defaults. It will also close the “Options” window after saving the settings.

## **Apply**

Pressing the “Apply” button will apply the current selections to the Environment to change how it operates. It does not save those setting for future use nor does it close the window.

## **Cancel**

Pressing the “Cancel” button will close the window without applying or saving any of the changes which have been made to the options since the last save or apply occurred.

## **Revert**

Pressing the “Revert” button will return all of the options to the values which they had when last applied or saved, undoing any changes that have been made. It does not close the window, nor does it save or apply the values.

## **Use an Environment window for the Experiment display**

This option is only meaningful if ACT-R is running in a Lisp which has GUI tools available and which also has the appropriate support in the ACT-R graphics interface. At this time those Lisps would be ACL w/IDE, LispWorks, MCL, and CCL. If ACT-R is running in any other Lisp, then the selection of this box is ignored.

The default for this option is selected (enabled) which means that when the Environment is connected to ACT-R any experiment window opened with the ACT-R **open-exp-window** command which is visible will be displayed in a window opened by the Environment in

Tcl/Tk instead of with the Lisp's native GUI tools. If the option is disabled then the Lisp's native GUI tools will be used to display visible windows opened with the **open-exp-window** command.

### **Compile definitions when model opened or reloaded**

This option controls how models are loaded or reloaded through the Environment. If the box is unchecked (the default) then files which are loaded or opened with the “Load Model” or “Open Model” buttons (note that “Open model” is only available by default with the standalone version) are loaded directly with the Lisp command `load`, and when the “Reload” button is pressed the ACT-R **reload** command is called with no parameters.

If the option is enabled then when a file is loaded or opened the file is first compiled with the Lisp command `compile-file` and then that compiled file is loaded with the `load` command. When the “Reload” button is pressed with this option enabled the value of `t` is provided as the optional parameter to the **reload** command which will cause it to also compile the file before loading. One thing to note about having the parameter enabled is that if you directly load a compiled file then the “Reload” button will not work for that file because it will not be able to compile the already compiled file and it will print this warning instead:

```
#|Warning: To use the compile option the pathname must have type lisp. |#
```

### **Automatically save an open model when reload pressed**

This option only affects the operation of the Environment when using the “Open model” button available with the standalone version (or enabled by the user as described in the [Extending or Changing the Environment](#) section). If this option is enabled, which is the default, then any changes made to the currently open model file will be saved before the **reload** command is called in response to pressing the “Reload” button. If this option is not checked then the changes will not be saved by the Environment prior to the **reload** command being called and thus the previously saved version will be loaded.

## Show the ACT-R copyrights screen

This option controls whether or not the window showing the ACT-R copyright information is displayed every time the Environment is started. If it is enabled then the window will be displayed and if it is disabled then it will not be displayed.

## Save a backup every time

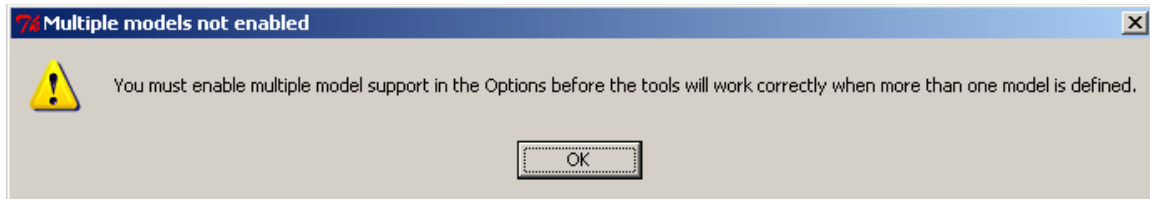
This option only affects the operation of the Environment when using the “Open Model:” button. If this option is enabled, then whenever the model file is saved (including the automatic saving upon **reload** if enabled) a backup copy of the existing file is made first. The backup copy will have the same name as the original file with an increasing number added to the end of the file’s extension. It will be written to the same directory as the original file. Thus, if the count.lisp file were opened and then reloaded a file named count.lisp-0 would be created in the same directory as the original count.lisp file and would be a copy of the file count.lisp before any current changes are saved into it. If it were reloaded again then a file named count.lisp-1 would be created, and so on.

These backup files are intended only as protection against a crash of the system or other error which may cause the loss of the file being worked on. ***The original file will always be the most recently saved version of the model and you should not open or load the backup files directly unless absolutely needed.*** After the Environment has been successfully closed you should feel free to delete any and all of the backup files. If the system does crash or for some other reason you would like to use one of the backup files you should first rename it to something meaningful if you plan to open it in the Environment because otherwise it will also have backups made of it which would then look something like count.lisp-1-0 and that can become confusing very quickly.

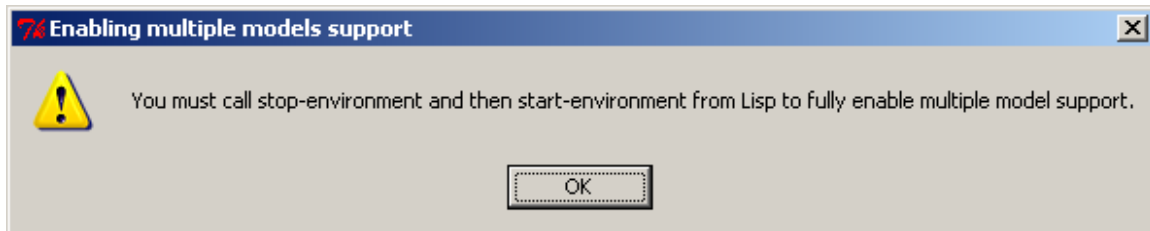
## Allow the environment to work with multiple models

This option controls whether the Environment assumes there will only ever be one model defined at a time, or if it will provide separate tools for multiple simultaneously defined models. The default setting is off. In that case if more than one model is defined this

warning will be displayed:



If the setting is enabled, then it will show this dialog when that change is made:



To change the Environment safely from single model mode to multiple model mode requires stopping and starting the connection to Lisp. Thus, when making that change the “Save” button should always be used to make the change because the “Apply” button’s settings do not persist across a stopping and starting of the Environment.

### **Close inspector windows for deleted models when multiple models are defined**

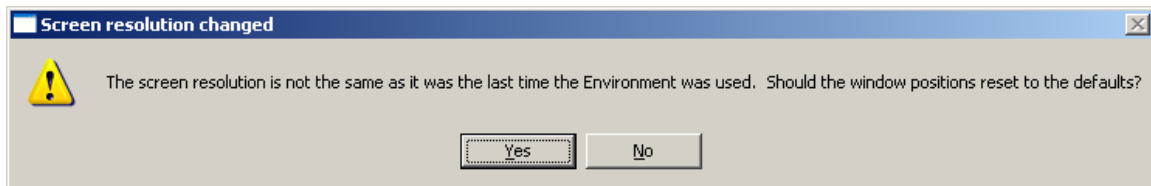
This option only matters if the “Allow the environment to work with multiple models” option is enabled. If that is enabled then this controls what happens to the inspector windows for models which are no longer defined. If the option is set then when a model is deleted the inspector windows for that model will automatically be closed in the Environment. If it is not set then the windows for deleted models will remain open, but will no longer function correctly and trying to use them may result in warnings in the Lisp running ACT-R.



## Window Positions and Sizes

When the Environment is closed it will save all of the settings for the window sizes and positions that were used while it was running. The next time that the Environment is run it will then use those same sizes and positions for all of the windows.

This is generally desirable, but can lead to problems if one uses the same machine with different displays or monitors which have different resolutions which could result in windows or tools no longer being within the bounds of the current display. To accommodate that there is a test performed when the Environment first starts to check whether or not the same screen space is available by testing the current height and width as well as the maximum window size as reported by the system. If those differ, then it will display this dialog before starting with the option of restoring everything to the default size and location instead of using the saved values:



If the “Yes” option is chosen then the default window positions will be used instead of those in the saved configuration.

If for some reason the Environment doesn’t detect that your screen has changed or you are having some other problems whereby the tools are no longer available because they are opening in windows outside of the screen then you can manually remove the file with the saved settings in it and force things into their default positions the next time the Environment is started. The settings are saved in the Environment’s GUI/init directory in the file named 10-userguisettings.tcl. If you delete that file then the next time the Environment starts all of the windows will revert to their default sizes and positions.

## Extending or Changing the Environment

It is possible to add new tools or capabilities to the Environment. Like the ACT-R loader, when the Environment starts it will automatically load source files located in some of its directories. Thus, putting new files into those directories will cause that additional code to be loaded and become a part of the Environment. Changes to the operations can also be made on the Lisp side of the Environment (which may affect how things are displayed or otherwise generated for display on the Tcl/Tk side) by placing files into the ACT-R directories that get loaded automatically.

When the Environment starts it first loads all of the .tcl files in the GUI/init directory. Then, it opens the Control Panel window and loads all of the .tcl files located in the GUI/dialogs directory. The files are loaded in ascending order based on their file names.

It is also possible to remove tools that you don't need, if you don't want them included in the Environment, by deleting those files from the GUI/dialogs directory. Each of the tools in the Environment is generally implemented in a separate file located in the dialogs directory and the names of the files indicate the tool that they implement, for example "38-visicon.tcl" implements the "Visicon" button. Deleting that file will remove the "Visicon" button from the Environment. The numbers on the fronts of the names are used to ensure that they are loaded and created in a specific order.

Most of the tools are independent and can be deleted without affecting the others, but there are a couple of exceptions. First, the "00-copyrights.tcl" file should not be deleted because it also controls the initial connection between ACT-R and the Environment. To eliminate the copyright window you should instead disable the "Options" setting which controls it. Also, the standalone version of the Environment's buttons for closing and saving model files depend on operations defined in the open model tool. Thus, if the open button is removed the save and close buttons should also be removed.

There are some additional controls or modifications for the Environment included with the ACT-R extras which can be used and are described in the sections below. There are also some disabled buttons (like the Open button mentioned above) included in the

GUI/dialogs directory. They were disabled by changing the extension on the file name from .tcl to .tcx. Returning that to .tcl will enable that button.

## Available Environment Extras

In the extras directory of ACT-R 6 there are two additional tools which can be added to the Environment. Each of those will be described below along with how to add it to the Environment.

### Sorted Items

There is a file called “sort-environment-lists.lisp” in the extras/environment-sorter directory. To use this file it should be moved to the other-files directory before loading the main ACT-R load file. With this file loaded the list boxes of items in the inspector windows will have their values sorted alphabetically instead of the default ordering returned by ACT-R. This will affect the listings in the Declarative, Procedural, Buffer, and Buffer Status tools.

One minor note about the sorted items modification is that it is currently incompatible with the retrieval and buffer history tools. If the sort-environment-lists.lisp file is used it will result in errors occurring if either of those tools is attempted to be used.

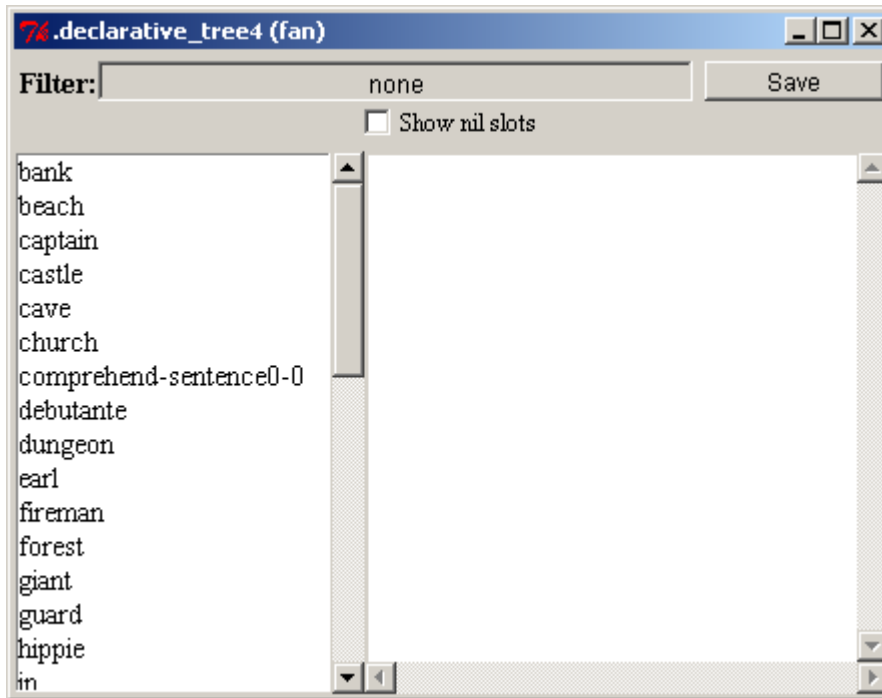
### DM Tree Viewer

In the extras/chunk-tree-viewer directory are several files which describe and implement a declarative viewer with an alternative display of chunks. The representation was developed by Andrea Heiberg, Jack Harris, and Jerry Ball working at the Air Force Research Laboratory and is described in detail in the HeibergHarrisBall.pdf file found in that directory. The tool for the Environment is a slight modification to their representation and is described in detail below.

To use the tool the chunk-tree.lisp file must be placed into the ACT-R other-files directory and the 35a-declarative-tree.tcl file should be placed into the Environment’s GUI/dialogs directory. Doing so will add a new button called “DM Tree viewer” to the Inspection section of the Control Panel.

Pressing that button will open a new declarative\_tree window for the current model and

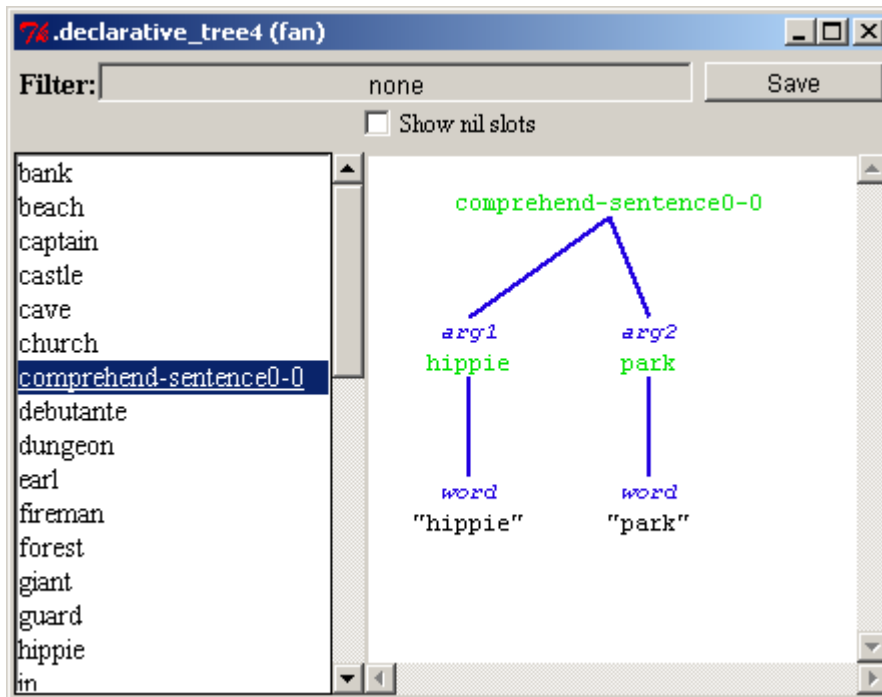
any number of those windows may be open at a time. Here is what the window will look like after running the fan model from the tutorial unit 5 (note that the sorted items addition described above is also loaded here):



Like the normal declarative viewer the list on the left shows the names of all the chunks in the model's declarative memory and the filter at the top can be used to restrict that list to chunks of a particular chunk-type. The difference in the display of the selection along with the additional controls available for this dialog will be described below.

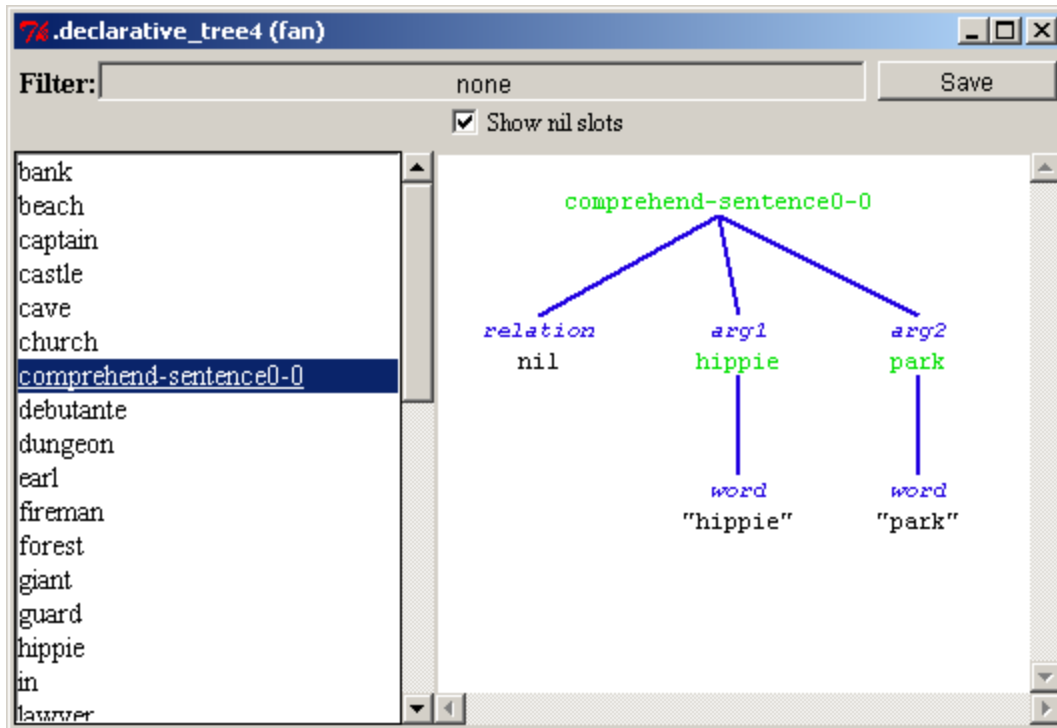
Selecting one of the chunks from the list will result in an image being generated to show the contents of that chunk. The chunk name will be displayed at the top of the window in light green and then for each slot a blue line will be drawn below that name with the name of the slot written in blue italic text at the end of the line. Below the slot name the contents of the slot will be displayed. If the content is not a chunk then it will be displayed in black with nothing below it. If the content is a chunk, then it will be displayed in one of three ways. If it is a chunk which is not in the model's declarative memory and which doesn't occur further up in the current branch of the tree then it will be displayed in dark green text with its slot contents displayed recursively below it. If it is a chunk which does

occur in the model's declarative memory and which doesn't occur further up in the current branch of the tree then it will be displayed in light green text and its slots will be recursively displayed below it. If it is a chunk which has already been displayed in the current branch of the tree (a circular reference) then it will be displayed in red text and its slots will not be displayed. Here is the display for the comprehend-sentence0-0 chunk after the model has done a trial for "the hippie is in the park":



It shows the contents of the arg1 and arg2 slots as the chunks hippie and park and each of those chunks has a word slot which contains the string representation of that word.

The "Show nil slots" checkbox at the top of the window controls whether or not slots with a value of **nil** (empty slots) are drawn in the display. If it is unchecked (the default) then they are not. If it is checked then all of the slots will be displayed. Here is that same chunk shown with the box checked now showing the unused relation slot in that chunk:



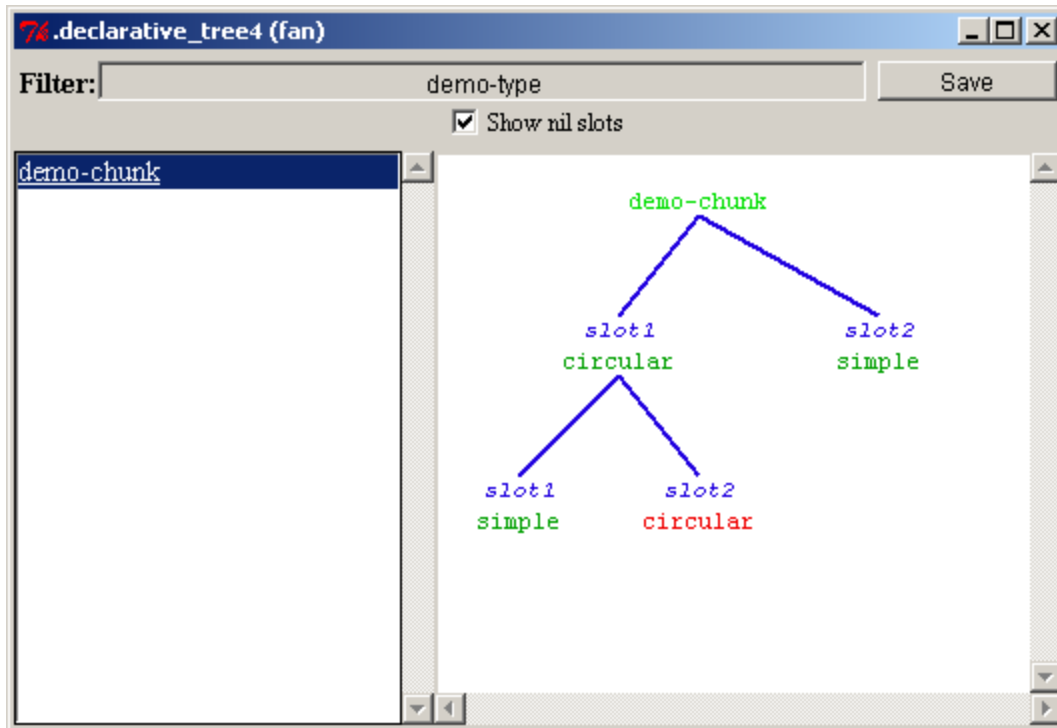
For demonstration purposes these commands have been executed to define some additional chunks for the model and to add a new chunk to the model's declarative memory:

```
(chunk-type demo-type slot1 slot2)

(define-chunks (simple isa chunk)
  (circular isa demo-type slot1 simple slot2 circular))

(add-dm (demo-chunk isa demo-type slot1 circular slot2 simple))
```

Thus, the demo-chunk is in the model's declarative memory while the simple and circular chunks are not, and the chunk named circular has a slot with a circular reference to itself. Here is the display showing the chunk named demo-chunk being drawn:



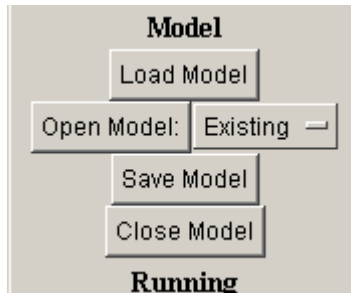
Any of the light green chunks in the display (the chunks which are in the model's declarative memory) can be clicked on to open the Environment's default declarative viewer with that chunk selected to see the parameters and text representation of that chunk.

Pressing the "Save" button at the top right of the declarative tree viewer window will allow you to save an image of the chunk as an Encapsulated PostScript file. A file creation dialog will be opened and the image will be saved into the file which you provide.



## Standalone Environment Tools

The standalone version of the Environment includes some additional buttons in the model section of the Control Panel:



These buttons provide access to a very basic text editor which can be used to edit or create models.

### Open Model:

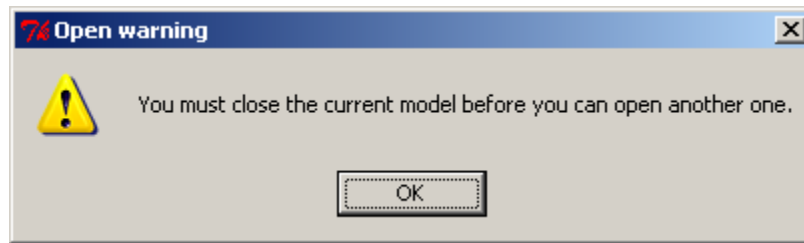
The “Open Model:” button will either create a new text file or open an existing file depending on the setting of the option menu to its right. If the option menu says “Existing” then a dialog will open for you to select a file to open and edit. If the option menu says “New” then a dialog will open asking you for the name and location to save the new file. To change the option menu you can press it to bring up the options which will look like this:



The one with the checkmark is the currently selected option and clicking the other will make it the current option.

In addition to opening the file for editing, it will also be loaded as if it had been chosen using the “Load Model” button. There can only be one model opened for editing in the

Environment at any time. If you try to open a second one it will display a warning dialog indicating that you must close the currently open model first:



You can close an open model using the “Close Model” button on the Control Panel or by just closing the window in which it is being edited.

## Save Model

The “Save Model” button will save the contents of the currently opened model file. If there is no model currently opened for editing, then this button has no effect other than to display a dialog indicating that there is not an open model:



If the “Save a backup every time” option is enabled, then before the file is saved a backup copy of the previous file will be made as described in the [Save a backup every time](#) section.

## Close Model

The “Close Model” button will save the contents of the currently opened model file and close the window in which it is being edited.

## **Advanced Issues**

The sections below contain information about advanced capabilities of the Environment. These mechanisms will likely not be of much use to most users, but are available if needed.

## Running the Environment on a Separate Device

It is possible to run the Environment on a device other than the machine which is running the Lisp with ACT-R. To do that one would perform the same steps described in the [Running the Environment](#) section using two separate devices, one to run the Lisp and one to run the Environment, with one exception. In step 4, instead of just calling `start-environment` to make the connection one of two alternatives must be done.

4'. Set the system parameter `:default-environment-host` to the IP address of the device running the Environment or the full hostname of that device before calling `run-environment`.

4''. Use the `connect-to-environment` command instead of `start-environment` and pass the IP address of the device running the Environment or the full hostname of that device as the host parameter e.g. `(connect-to-environment :host "192.168.123.254")` or `(connect-to-environment :host "foo.bar.edu")`.

You can still call `stop-environment` to put the Environment back to the waiting state. Unlike when the Environment is running on the same machine as the Lisp application, it is possible to quit the Environment application without first stopping the connection from the Lisp side. If that happens then ACT-R will print a message indicating that the Environment has been closed and stop the connection automatically. When the Environment connection is stopped in that way there might also be some notices in the Lisp indicating a problem with the connection being closed or reaching an end of file signal. Such warnings/errors can be safely ignored. That happens because of issues with how the processes handling the Environment in Lisp are shut down and is a known issue to be fixed.

## Running More than One Environment

In addition to running the Environment on a device other than the one running the Lisp as described above, it is also possible to connect more than one Environment to the same Lisp running ACT-R. Without changing the Environment files, it is only possible to run one instance of the Environment on a device, but any number of devices each running an Environment may be connected to the same Lisp running ACT-R.

To connect the first Environment, use either the regular method described in the [Running the Environment](#) section if it's on the same machine or the method described in the [Running the Environment on a Separate Device](#) if it's on a separate device. Then, to connect the second and all subsequent Environments you must use connect-to-environment specifying the host address and also the keyword parameter clean must be specified as nil i.e. (connect-to-environment :host "192.168.123.254" :clean nil). If you don't specify the :clean nil parameter all previously connected Environments will be deactivated. To put all of the connected Environments back into the waiting state you now must call close-all-connections instead of stop-environment.