

**The COGENT Tutorial**

**22<sup>nd</sup> Annual Conference of  
the Cognitive Science Society**

**August 12<sup>th</sup>, 2000**

**Philadelphia, USA**

**Contents**

1. COGENT: An Overview
2. The Tutorial Task: The Modal Model of Memory
3. The COGENT 'Modal Model' Model
4. Building the Short-Term Store
5. Adding the Long-Term Store
6. Decay, Time and Rehearsal

Peter Yule & Richard Cooper

Copyright ©1999, 2000 The COGENT Group  
<http://cogent.psyc.bbk.ac.uk/>

# 1 COGENT: An Overview

COGENT (Cooper & Fox, 1998) is a powerful computational modelling system that provides a flexible environment within which information processing models of cognitive processes may be developed and explored. The system provides a range of functions that allows scientists to explore their ideas and theories without commitment to a particular architecture. COGENT has been designed to simplify rigorous development and testing of models, and to aid data analysis and reporting. Among the functions provided by COGENT are:

- A visual programming environment;
- Research programme management tools (providing a time-oriented project display and editor, version control on models, and support for documentation at the model and project levels);
- A range of standard functional components (including memory buffers, rule-based processes, simple connectionist networks, input sources, output sinks), with mechanisms for controlling inter-component communication;
- An expressive rule-based modelling language and implementation system;
- Automated data visualisation tools (via tables, graphs, and animated diagrams); and
- A powerful model testing environment (with functions for performing monte carlo-style simulations and an “experiment-based” scripting language).

To date, COGENT has been used to develop models of problem solving, memory, reasoning, decision making, categorisation, mental rotation, routine action control and executive process control.

## 1.1 The Visual Programming Environment

COGENT simplifies the process of model development by providing a visual programming environment in which models may be created, edited, and tested. The programming environment allows users to develop cognitive models using a box and arrow notation that builds upon the concepts of functional modularity (from cognitive psychology) and object-oriented design (from computer science). Models are specified by sketching their functional components using COGENT’s graphical model editor. Figure 1 shows a production system model for a multicolumn subtraction task (Young & O’Shea, 1981). The model has five principal functional components: *Working Memory* (a short-term store in which temporary information is stored and manipulated), *Production Memory* (a long-term store containing the rules and procedures for multicolumn subtraction), *Production System Interpreter* (a complex process which selects rules and executes actions), and *Paper* (a representation of a piece of paper on which the task is presented and on which temporary working and the answer is written).

The graphical model editor provides a number of standard types of component, including various kinds of buffer, rule-based processes, simple feed-forward networks, compound boxes (that contain other boxes) and input/output devices. Different shaped boxes are used to represent these different types of component within a COGENT diagram, and arrows are used to indicate different types of communication between the components.

A number of different types of information may be associated with a model. This information may be viewed or edited by selecting the appropriate tab on the main portion of the model editor window (figure 1). The Diagram view is shown in the figure. Other views (Description, Properties, Message Matrix, Messages) provide access to: a text window into which notes or comments on the model may be entered; the set of configurable properties that control aspects of the model’s execution; a two-dimensional map that shows, during model execution, inter-component communication; and a text-based view of messages generated or received by the top-most box.

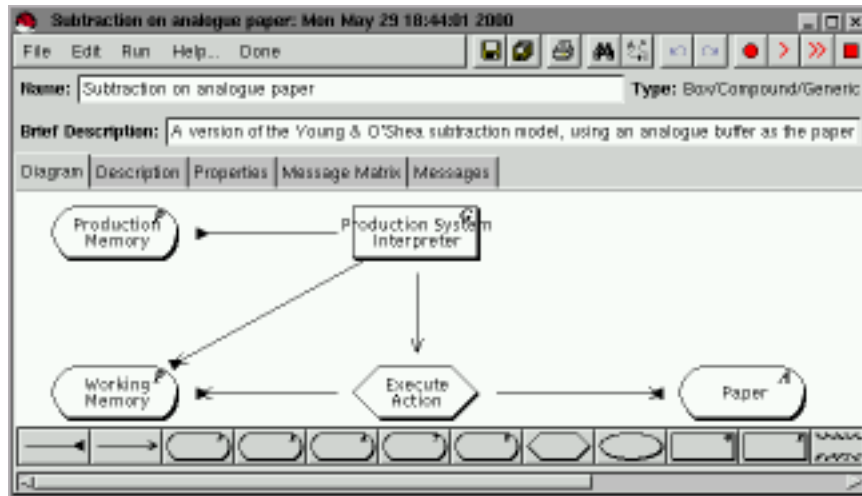


Figure 1: A box/arrow diagram of a production system whose actions include reading from and writing to paper

## 1.2 Research Programme Management

Cognitive models are often developed in stages over extended periods of time. COGENT supports this development through special tools for managing sets of models within a research programme. The research programme manager, shown in figure 2, provides a graphical display of the models contained within a research programme (showing ancestral links between models), facilities for version control on models (e.g., copying, archiving), documentation support, and a front-end to the graphical model editor described above.

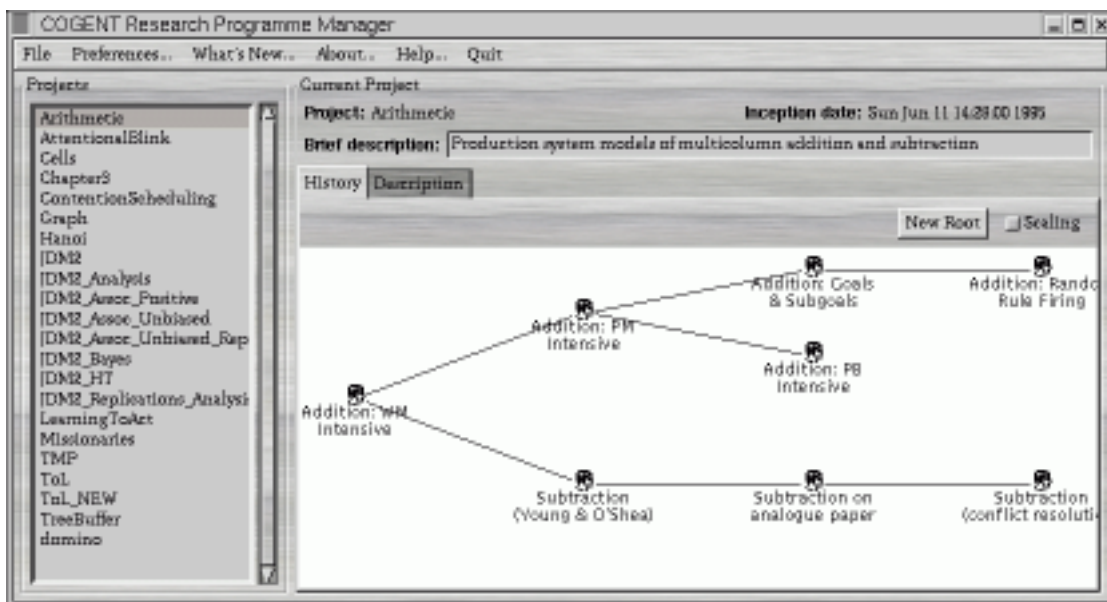


Figure 2: The project history view

The left hand column of the research programme manager (figure 2) shows all research programmes re-

gistered with COGENT. When a research programme is selected its history is displayed in frame on the right in the form of a tree. Each node in the tree corresponds to a separate model. Double-clicking on a node opens COGENT's model editor on the corresponding model. In general, models to the right are derived from models to the left, and links in the tree show relations between successive versions of the same model. As can be seen from the figure, several versions of a model may be explored in parallel.

### 1.3 Standard Component Types

COGENT provides a library of standard configurable components. Models are constructed by assembling these components (in the form of a box/arrow diagram) and then configuring them as necessary. The component library includes:

**Memory buffers** Buffers are general information storage devices that may be used for both short term and long term storage. The detailed behaviour of any instance of a buffer is determined by its properties, which specify (among other things): capacity limitations, decay parameters and access restrictions. The use of properties to specify buffer behaviour (and in fact, the behaviour of all COGENT objects) leads to components that are both flexible (i.e., can perform a variety of functions) and well-specified (i.e., the various property values fully define the computational behaviour of the component). In addition, different subtypes of buffer may be used to store information in different formats (e.g., propositional, tabular, and analogue).

**Rule-based processes** Rule-based processes manipulate information according to symbolically specified rules. A powerful rule language and rule interpreter allows rule-based processes to perform complex manipulations and transformations of information, in a way that may be contingent upon the contents of other COGENT objects.

**Connectionist networks** COGENT is intended primarily for high-level symbolic modelling. Nevertheless, COGENT's generalised processing engine allows direct interface with some simple connectionist object types (two-layer feed-forward networks and competitive networks), making it suitable for a variety of hybrid modelling applications. As in the case of buffers, precise network behaviour is determined by a number of parameters (governing, for example, learning rate, initialisation, etc.).

**I/O sources and sinks** Specialised data source components allow data to be feed into other components in a controlled manner. Data sinks similarly allow the collection of data from other components during model execution. Three types of data sink are provided (text-based sinks, tabular sinks, and graphical sinks) to allow flexible storage and presentation of model output.

**Inter-module communication links** Inter-module communication is achieved within COGENT by arrows of various types which may be drawn between the appropriate components within a COGENT box/arrow model. Two basic arrow types exist: read arrows and write arrows. The presence of such arrows from one component to another enables the relevant communication function between those components.

### 1.4 The Rule-Based Modelling Language

COGENT's rule-based modelling language allows complex processes to be specified in terms of production-like rules. Each rule consists of a set of conditions and a set of actions. Conditions include logical operations whose outcome may be true or false, such as matching some information stored in a buffer or testing the

equality of data elements. Actions allow messages of various forms to be sent to other boxes. A simple rule may have the following form:

```

IF      minuend(X) is in Working Memory
        subtrahend(X) is in Working Memory
THEN  add equal(minuend, subtrahend) to Working Memory
        send difference(0) to Write Answer

```

This rule fires when *Working Memory* contains two elements of the form *minuend(X)* and *subtrahend(X)*. On firing, the rule adds a further element to *Working Memory* (*equal(minuend, subtrahend)*) and sends a message of the form *difference(0)* to another box, *Write Answer*. COGENT's rule language is powerful but complex. Special tools are provided to simplify the process of constructing rules and to allow monitoring of the processing of rules during model execution.

Rules are contained within processes (which themselves are represented diagrammatically as hexagons: see figure 1), and may be supplemented with user-defined conditions. Figure 3 shows the Rules and Condition Definitions view of one such process (*Match Productions* from within the *Production System Interpreter* box of figure 1). The condition definition language is based on Prolog, an AI programming language. It is highly expressive and provides COGENT with substantial flexibility.

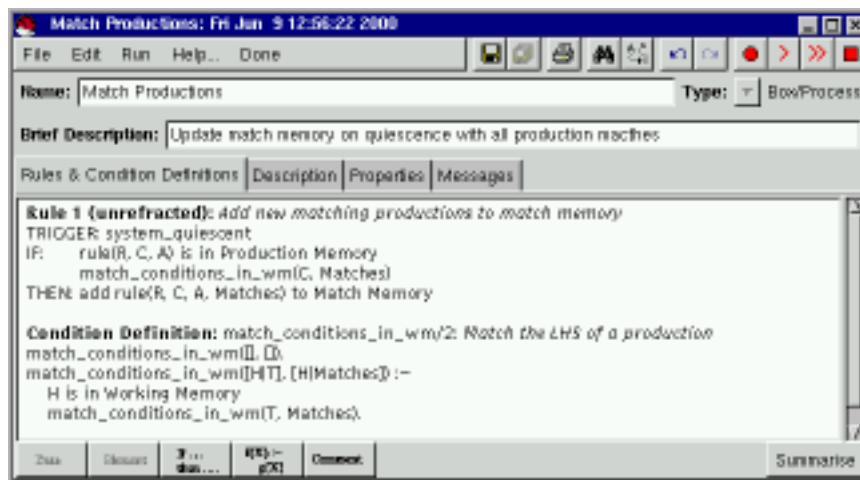


Figure 3: Some rules from the production matching process within a COGENT production system interpreter

## 1.5 Automated Data Visualisation Tools

COGENT provides a number of visualisation tools to assist in the monitoring and evaluation of a model. These tools, which take the form of additional types of box, allow data to be displayed in standard (tabular or graphical) form. More sophisticated visualisations may also be crafted through use of a generalised graphical display box.

**Tables** Tables allow data to be displayed in a standard two-dimensional format (see figure 4). Messages sent to a table specify values for the various cells. Tables are updated dynamically during the execution of a model, with details of table layout (e.g., row width, column height, row and column labels, etc.) for any particular table being governed by the properties of that table.

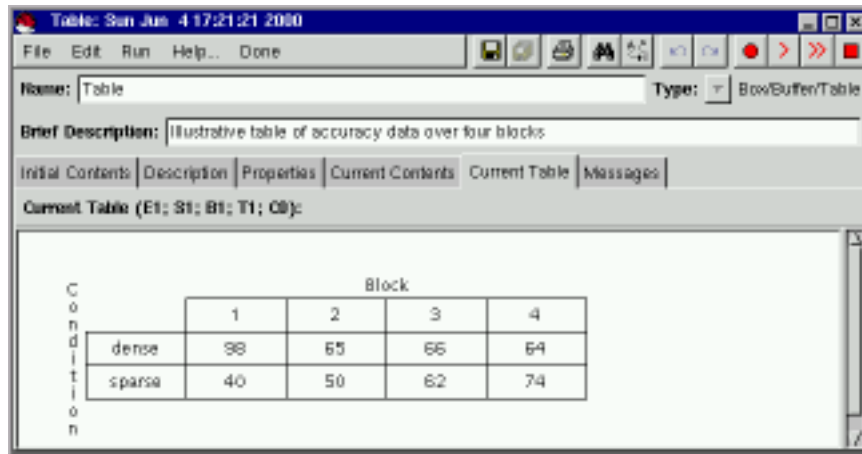


Figure 4: A table buffer, displaying data accumulated over four blocks of a task

Tables come in two flavours. Output tables are write-only: data sent to such tables are displayed but cannot be queried by other components. Buffer tables are read/write: other components connected to the buffer with read access may query the value in any cell. This querying is governed by standard buffer access properties (extended to allow spatial access from left to right, right to left, top to bottom, or bottom to top).

**Graphs** Graphical display devices are also available to display data in several standard graphical formats (line graphs, scatter plots and bar charts). A single graph may be used to display multiple data sets in different colours (see figure 5). Messages sent to a graph specify data points or style information relating to a particular data set. As with tables, graphs are updated dynamically during model execution and presentational details are controlled through properties associated with the graph.

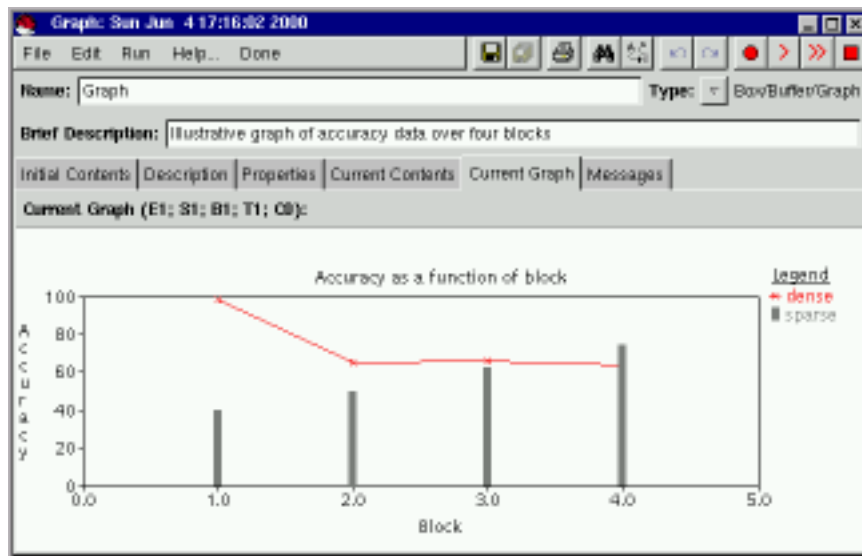


Figure 5: A graphical buffer, showing a line graph and a bar chart

**Generalised Graphical Output** A specialised type of buffer (an analogue buffer) may be used to display data in pictorial form. Such displays are dynamically updated whenever the contents of the buffer change.

Figure 6 shows an analogue buffer being used to display a pictorial representation of the Tower of Hanoi problem.

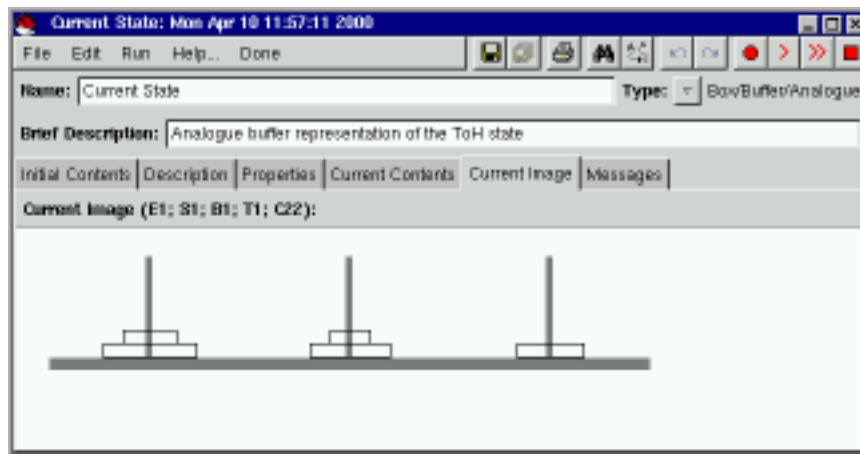


Figure 6: An analogue buffer, showing an intermediate state of the Tower of Hanoi problem

## 1.6 The Model Testing Environment

All COGENT models share an underlying processing system that supports four levels of execution: trial, block, subject and experiment. These levels correspond directly to their analogues in experimental psychology. The simplest way of using COGENT is to run a single trial (i.e., present a single stimulus and gather a single response), but it is also possible to specify extended experimental designs, in which, for example, 20 virtual subjects are run in each of four experimental conditions, with each virtual subject performing five blocks of 30 trials. Such designs are constructed through a special purpose experiment script editor.

The model testing environment provides a range of facilities for monitoring and debugging models, as well as a sophisticated execution environment for running computational experiments. Monitoring is provided through the Messages view available on each component's window. This view, which is dynamically updated during model execution, shows all messages generated by or received by a component. Thus, figure 7 shows the messages relating to the *Execute Action* process of the model shown in figure 1 after 26 processing cycles. Each line shows: the cycle on which the message was received or generated, the message source (e.g., rule 10 of *Execute Action*), the message destination (e.g., *Paper*), and the message content.

Other facilities allow the traffic between components within a compound box to be monitored (through the Message Matrix view of a compound box), and the execution of specific elements within rules to be traced.

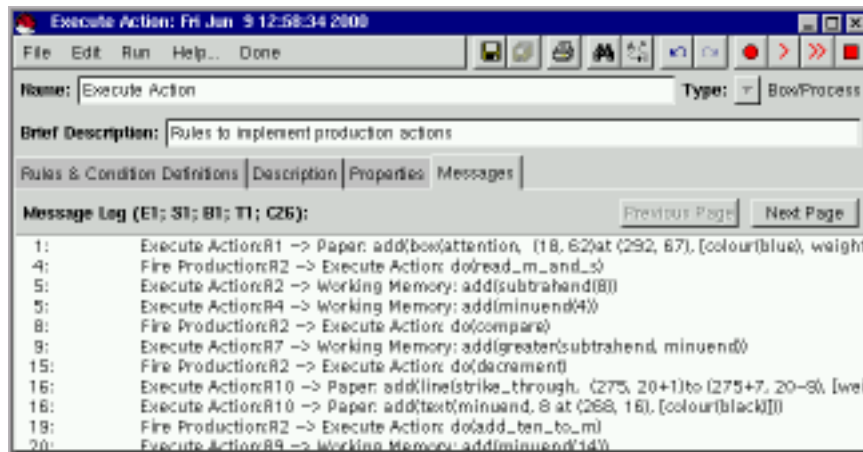
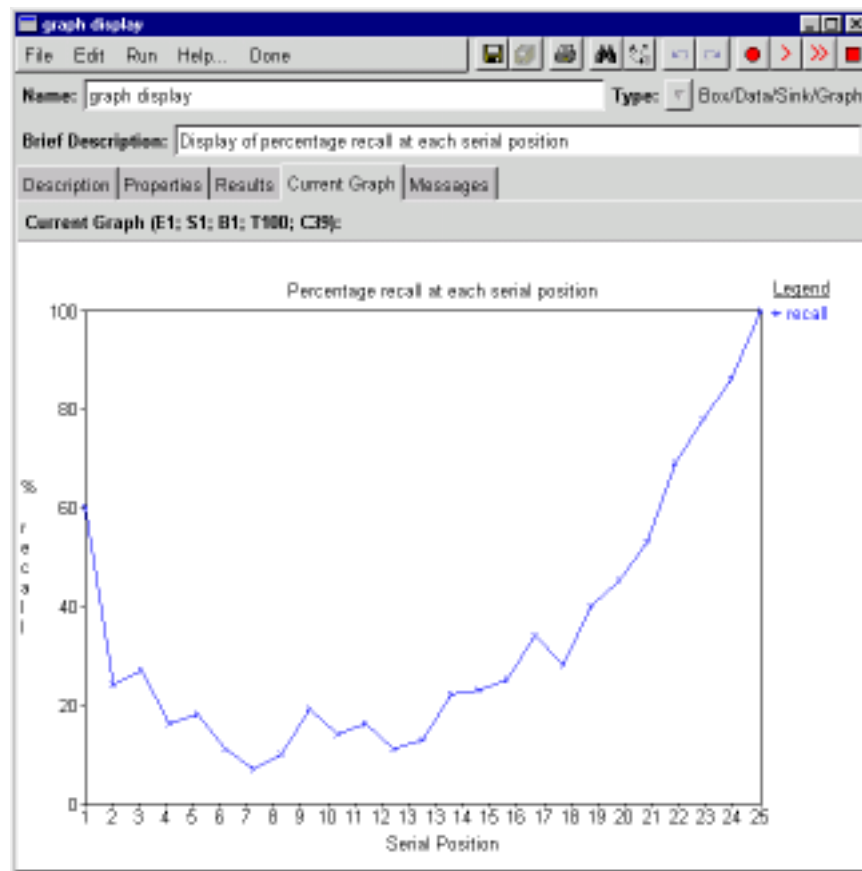


Figure 7: The Messages view of *Execute Process*

## 2 The Tutorial Task: The Modal Model of Memory

This tutorial develops an implementation of some aspects of Atkinson & Shiffrin's (1968, 1971) so-called "Modal Model" of human memory (called "modal", apparently because most psychologists subscribed to it at one point). The Modal Model is concerned to explain the Recency and Primacy effects in free-recall serial position curves (Glanzer & Cunitz, 1966). In a free-recall paradigm, participants are presented with a list of words, one at a time, and instructed to memorise the words as well as possible, in order to recall as many as possible in a subsequent free-recall phase.



Because the words are presented serially, it is possible to plot the average accuracy of recall at each point in the series. The typical finding is that the curve is roughly U-shaped: words at the beginning of the list are recalled relatively well, as are words at the end of the list, but those in the middle are poorly recalled. The peak at the beginning is known as the *Primacy Effect* and the peak at the end is known as the *Recency Effect*. To anticipate slightly (ultimately you will reproduce this), a Serial Position curve produced by a COGENT model is shown above.

Now we can explain the Recency Effect (the rising portion at the right of the diagram) if we postulate two distinct memory stores: a limited-capacity but relatively reliable Short-Term Store (STS), and an unlimited capacity but relatively unreliable Long-Term Store (LTS). By this account, the items in the Recency portion of the free-recall curve are recalled well, because they are still held in the STS, whereas those earlier in the curve are recalled relatively poorly, since they are only held in the decay-prone LTS. If subjects can recall

from either store, we expect a peak at the end of the curve, spanning as many items as the postulated capacity limitation of STS will allow (e.g.  $\approx 7$  items (Miller, 1956)).

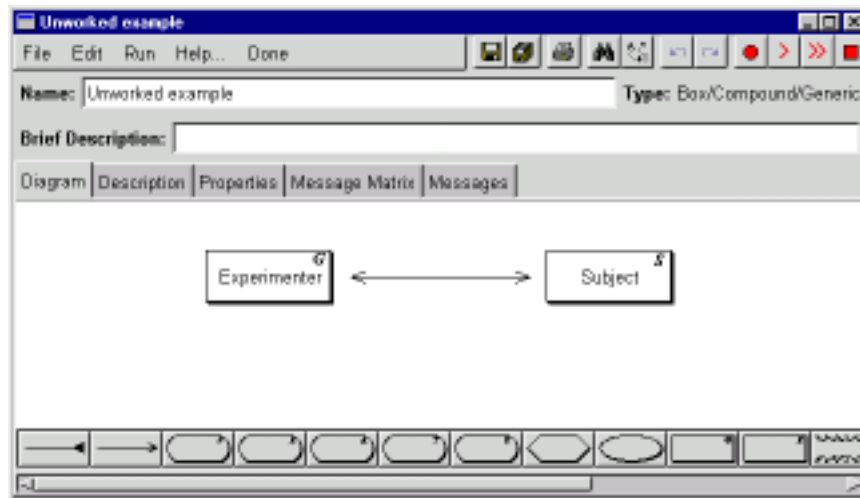
The Primacy Effect (at the far left of the diagram) requires a different explanation; in the Modal Model, it is assumed to arise from the limited-capacity process of *Rehearsal* which is used to transfer information from STS to LTS. The explanation depends on the fact that at the beginning of the list, there are relatively few items to rehearse, so these items receive disproportionately more rehearsal than items later in the list, and so if more rehearsal implies better recall, they should be remembered better.

Students should note that this theory is obsolete, and nobody believes the human memory system is so simple nowadays. Still, it is a good example to illustrate how COGENT can be used to implement concepts and models from the psychological literature.

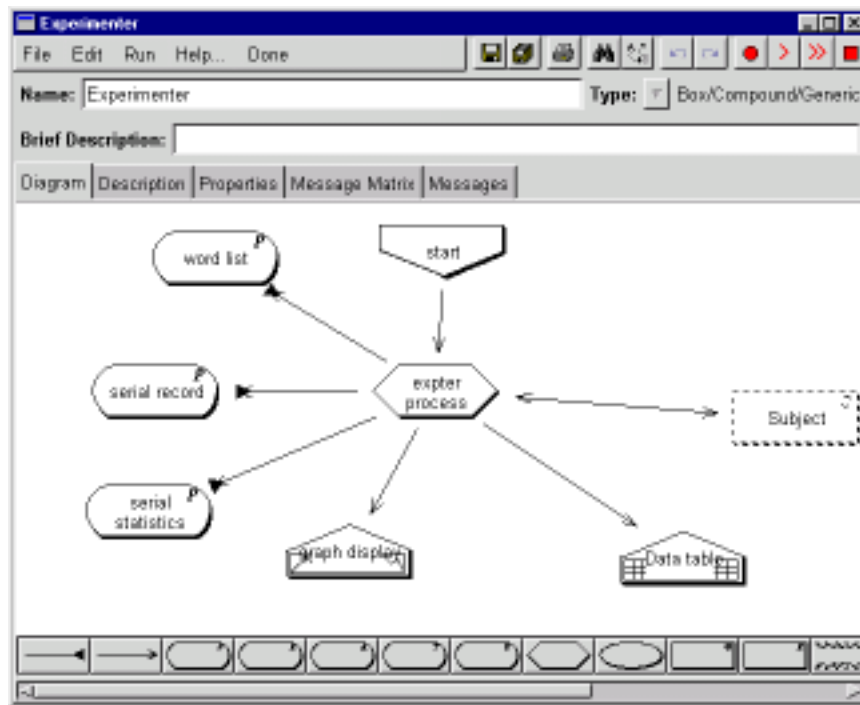
The remainder of this tutorial consists of developing and testing a COGENT implementation of the Modal Model. The box/arrow language of COGENT is very powerful, and it is possible to use the language to specify both cognitive models and computational environments in which those models may be evaluated. That is, COGENT can be used to simulate stimulus presentation and data collection. This tutorial adopts this approach. However, because such aspects may obscure the presentation of the Modal Model, we encapsulate them in functionally distinct, pre-specified, compound boxes.

### 3 The COGENT ‘Modal Model’ Model

Because this is a short tutorial, and since we are concentrating on producing a simple cognitive model rather than a complete experimental program, we start the tutorial with a model in which we already have a developed experimental environment, which presents the stimuli, collects responses and graphs results. This “stub” model should be installed on your machine. Select Tutorial2000 from COGENT’s Research Programme Manager window to and double-click on Unword example to open it. You should see a screen like this:



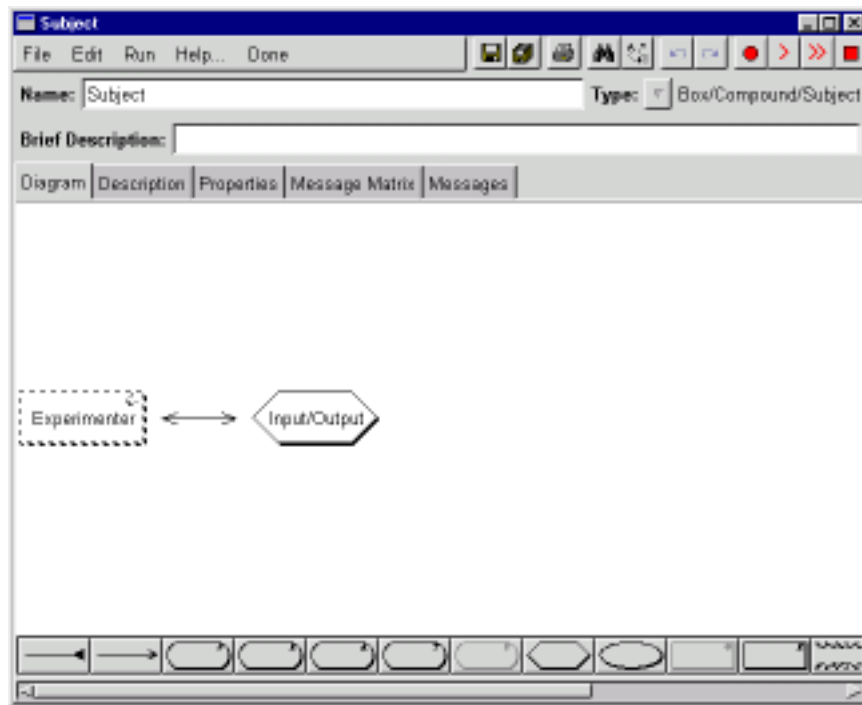
This is the main box-arrow diagram of the model. It comprises two compound boxes, linked by arrows. The box labelled “Experimenter” presents stimuli to, and collects responses from, the box labelled “Subject”. The Experimenter also maintains a graphical representation of the serial position curve (as shown already above). You can find the graph display by double-clicking on the “Experimenter” box, which will open a new window which looks like this:



The graph display itself is opened by double-clicking the box at the bottom of the diagram labelled “graph display”. When you open it, you need to click on the Current Graph tab to view the graph.

Of course there is no graph yet, since there is no data. You should by now be able to see that each box in a box-arrow diagram represents an object, and when you open it, you have a variety of different views available — the exact range of views depends on the object type. Objects that contain other objects, which they display as a box-arrow diagram, are known as *Compounds* and are represented with a rectangle. Rounded boxes are *Buffers*, of which there are a range of types.

Finally in this section, we can introduce the Subject box and its relation to the Experimenter. Return to the main box-arrow diagram, and double-click on the Compound labelled “Subject”. This as its name suggests, is going to contain our model of the human subject in the free-recall task. But since this model isn’t built yet, it contains only a stub, looking something like this:







The hexagonal box labelled “Input/Output” is a *Process*, and it will handle all interaction with the Experimenter. Double-click on it to open it, then switch to the Messages tab, which will initially display a blank screen.

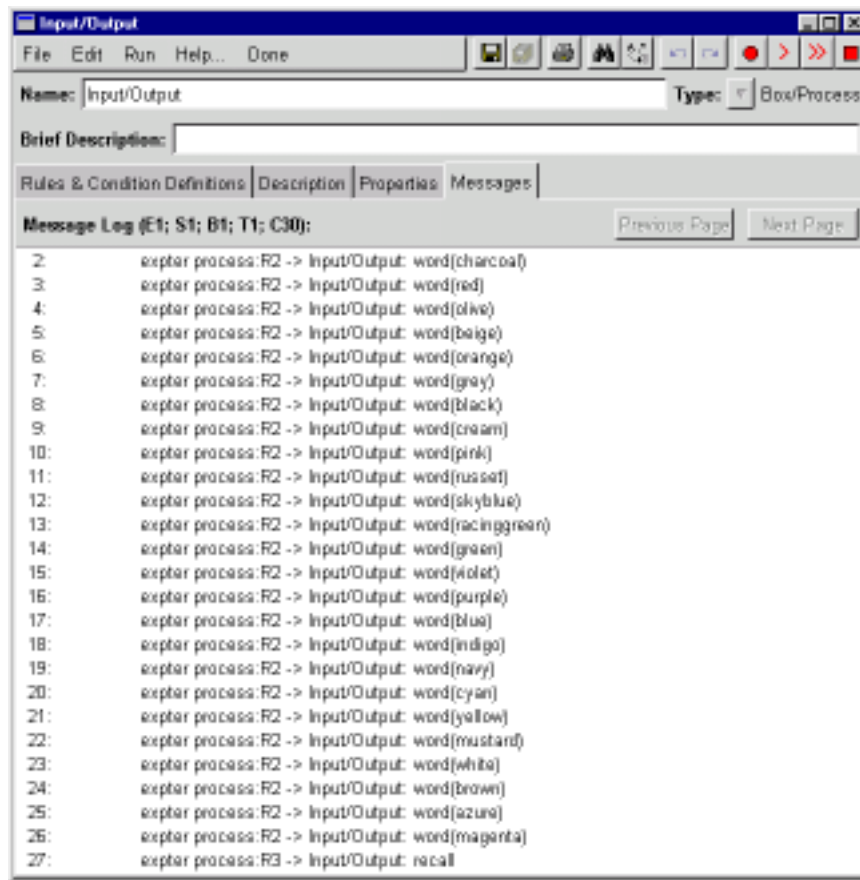
Now notice the row of buttons marked in red on the toolbar at the top of the window, like this:



This is the run toolbar, and it is displayed in the window of every open box. It provides access to the commands used to run a model. You can get more information about the buttons’ functions by placing the mouse pointer over each one, without clicking; the functions are as follows:

Button	Action Performed
	Initialise
	Single Step
	Run
	Stop execution

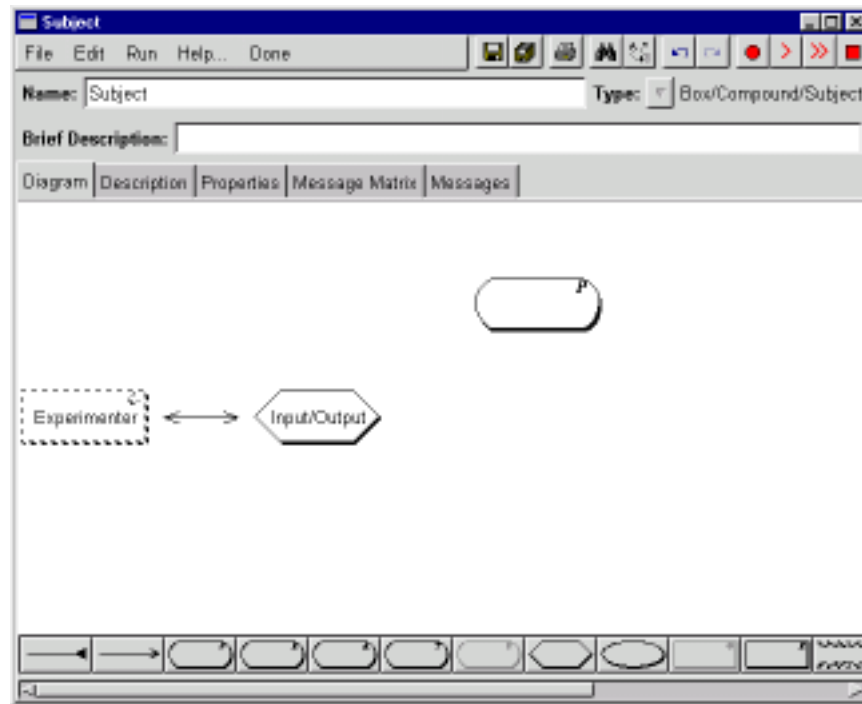
The first thing to do is click on the “Initialise model” button, to initialise the model. Then click the “Run” button; you should see signs of activity, and messages should accumulate in the open window. These messages are being sent by the Experimenter — in this model the Experimenter sends a message of the form `word(Word)` to the Subject on each successive cycle. When the Experimenter has sent 25 words to the Subject, it sends a final `recall` message, like this:



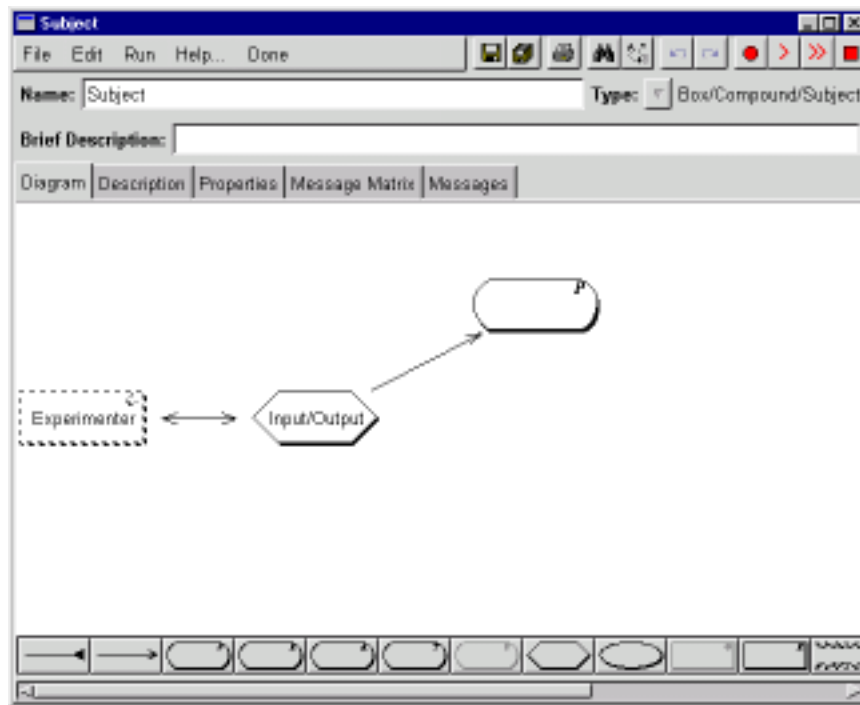
The model will ultimately need to be able to respond appropriately to these messages. The next section describes how to do this.

## 4 Building the Short Term Store

We begin by adding a short-term store to our stub model. (Recall that the short-term store was held to be responsible for the Recency effects in free recall.) From the main box-arrow diagram, double click on the “Subject” box to open it. The model needs to contain two boxes representing memory stores, the STS and LTS. These can be implemented as Propositional Buffers. To add the STS box to the diagram, first note the drawing toolbar at the bottom of the window; it contains a set of buttons representing different box types, and two arrow types. Click once on the “Buffer/Propositional” button (the one showing the letter *P* within a round-cornered rectangle), then click on the diagram canvas. A Buffer box, annotated with a *P* in the top right corner, will appear on the diagram:



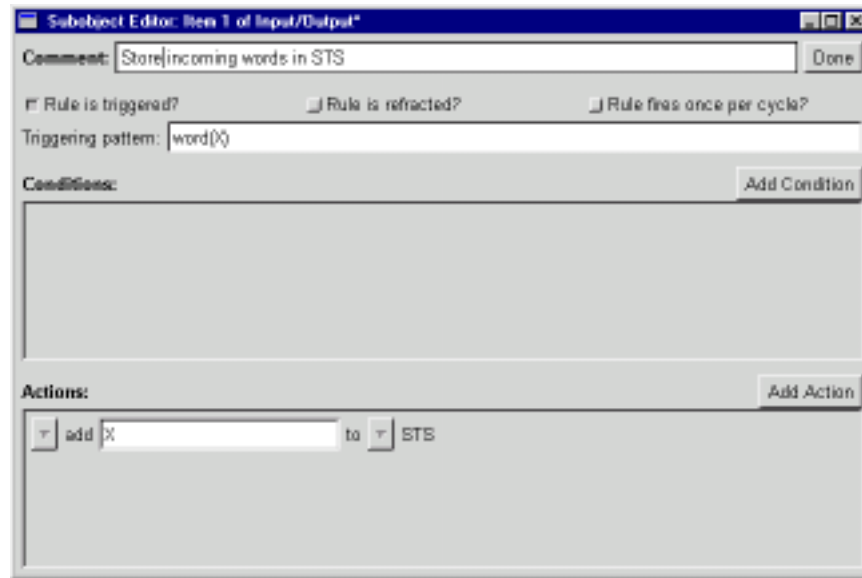
Next, we need to link the “Input/Output” process to the new buffer, to allow the “Input/Output” process to store the incoming words in STS. For this we need a *Send* arrow. Select the appropriate arrow type (by clicking on the rightmost of the two arrow buttons on the drawing toolbar). Now click on the “Input/Output” box, and, without releasing, move the pointer to the new box — an arrow will follow the mouse pointer — and release the button. The diagram should now look like this:



Before continuing, you should double-click the new box to open it, and give it a name, by typing “STS” into the field labelled Name:.

We are now ready to add a rule to the “Input/Output” process that will transfer incoming words to STS. Double-click on the “Input/Output” process box to open it, and view the Rules & Condition Definitions pane. You will see a toolbar at the bottom of the window; to create a rule, click on the button marked “If... then...”, and then click somewhere in the main, white region of the window. A new rule will appear; double-click it and a new window will open. This is the *Rule editor*, and we are going to use it to create a rule which will write each incoming word from the Experimenter into the STS buffer.

First, check the Rule is triggered checkbox. This indicates that this rule responds to trigger messages such as those sent by the Experimenter. The Triggering pattern field, previously greyed out, will become active. You should type the triggering pattern `word(X)` in it. (Make sure you capitalise the X inside the brackets, but not the word outside.) Next, pull down the Add Action menu, and select the add item on the menu. A new entry under the *Actions* part of the window will appear. This entry has two boxes. In the left box, type the letter X (again capitalised — it refers to the same variable as the one in the trigger pattern). The right box offers a selection of names of other boxes (the ones to which it is possible to add items). From these, select STS. Finally, in the box at the top labelled Comment, type a comment to explain what the rule does — such as Store incoming words in STS. The rule editor window should now look something like this:



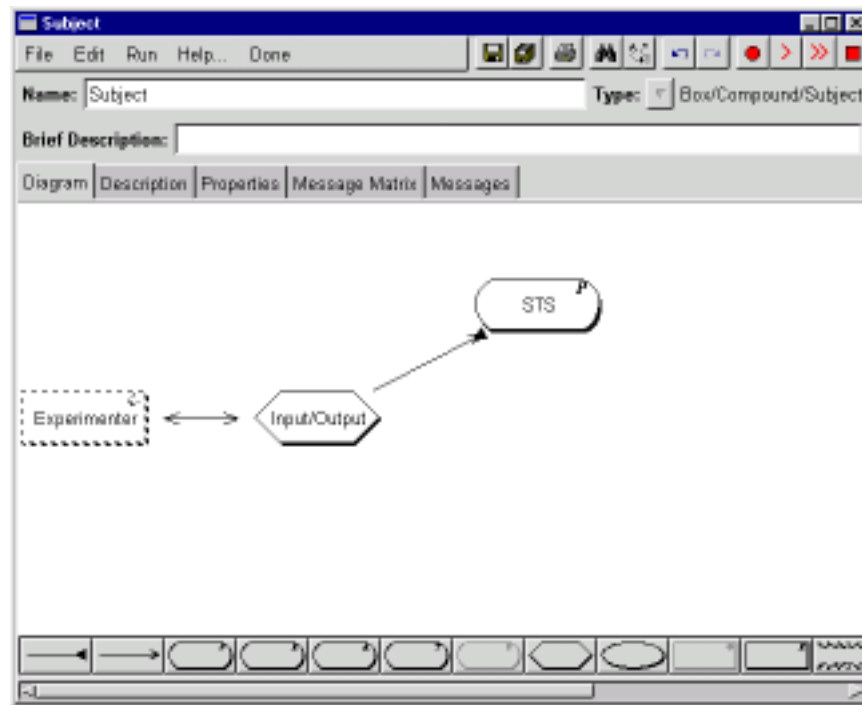
This rule will be triggered when elements of the form  $word(X)$  (where  $X$  is a variable) are received by the Input/Output process. (Recall that the Experimenter sends a series of such messages during an experimental trial.) Whenever the rule is triggered, it will add an element (whatever the  $X$  happens to correspond to) to STS.

Now you can close the rule editor window. At this point the model is actually able to do something: for each incoming word, it will be copied into the “STS” buffer. To test it, open the “STS” box, and select the Current Contents view. Now click the “Initialise” button and then the “Run” button. A list of words should appear in the window. Initialise again, and single-step for a few cycles. You will see that words are appearing (and accumulating) once per cycle. (Note that nothing will appear in this window until after a couple of cycles, because it takes some time for words to be generated and fed to the subject.)

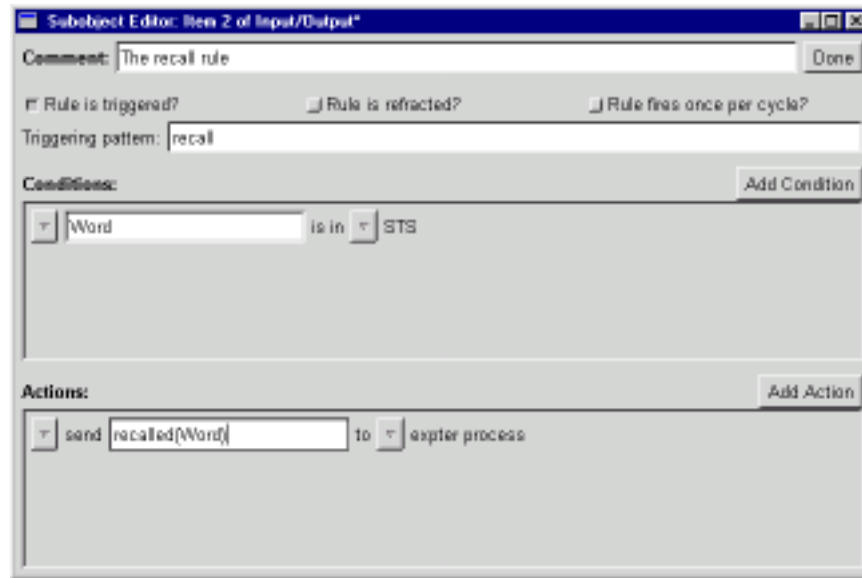
Now as was mentioned before, in the Modal Model, STS is supposed to be capacity-limited, with a span of about 7 items. Switch to the Properties view of the “STS” box — you will see a list of property names and their associated values. Find the Limited Capacity property and set it to Yes. This works in conjunction with the Capacity property, which should be set to 7. At this point, also find the On Excess property and set it to Oldest. Now return to the Current Contents view, reinitialise and step through the trial again; you will see that the number of items in the STS buffer grows until it reaches 7, after which point new additions to the buffer over-write existing elements — in this case the oldest element is always the one to be replaced. Return to the properties view, change the On excess property to Random, and again step through the trial, viewing the current contents. This time, when the capacity limit is reached, a randomly selected item is over-written by each new element.

Now we can add a recall facility to this memory system, allowing the model to perform the memory task. Return to the Subject box-arrow diagram, and add a *Read arrow* from Input/Output to STS. It should look

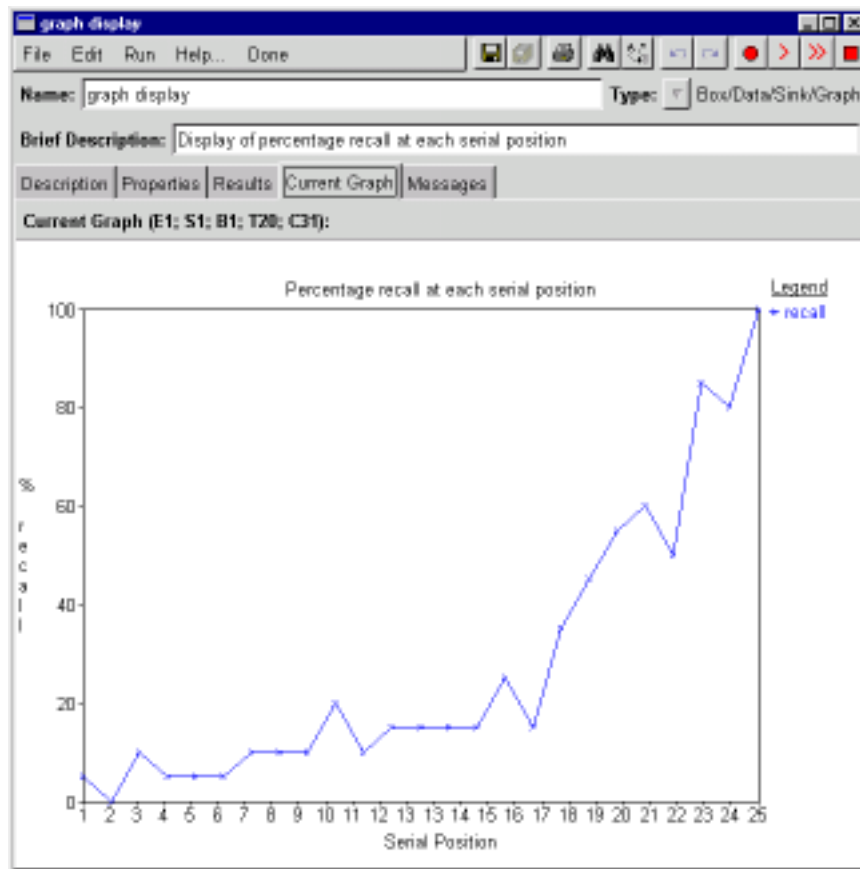
like this:



To add a rule to respond to recall requests, create a new rule in the Input/Output box and double-click it to open the rule editor. This rule will respond to the `recall` trigger, so you should enable this, as before, by checking the Rule is Triggered box and typing `recall` in the Triggering Pattern field. Then select match from the Add Condition menu. A new entry will appear in the Conditions section. In the left field, type `Word` and in the right field, select STS. Now from the Add Action menu, select send, and in the left field, type `recalled(Word)`, then in the right field, select `expter process`. Now the rule should look something like this:



Now, when triggered by the `recall` message, the rule should read the words it remembers from memory (currently, just STS), and send them to the Experimenter. (Notice that we haven't marked the rule to fire once per cycle, so the rule will send *all* words that it remembers to the Experimenter.) Close the rule editor window, and return to the graph display in the Experimenter. Click the Initialise Model button, then click the Run button. After a moment you should see a jagged pattern representing recall in a single trial. Without initialising, try clicking the Run button a few more times. Each run corresponds to a complete block of trials. You should find that the graph becomes less jagged, and if you keep clicking around 20 times you should see a curve beginning to develop. It might look something like this:

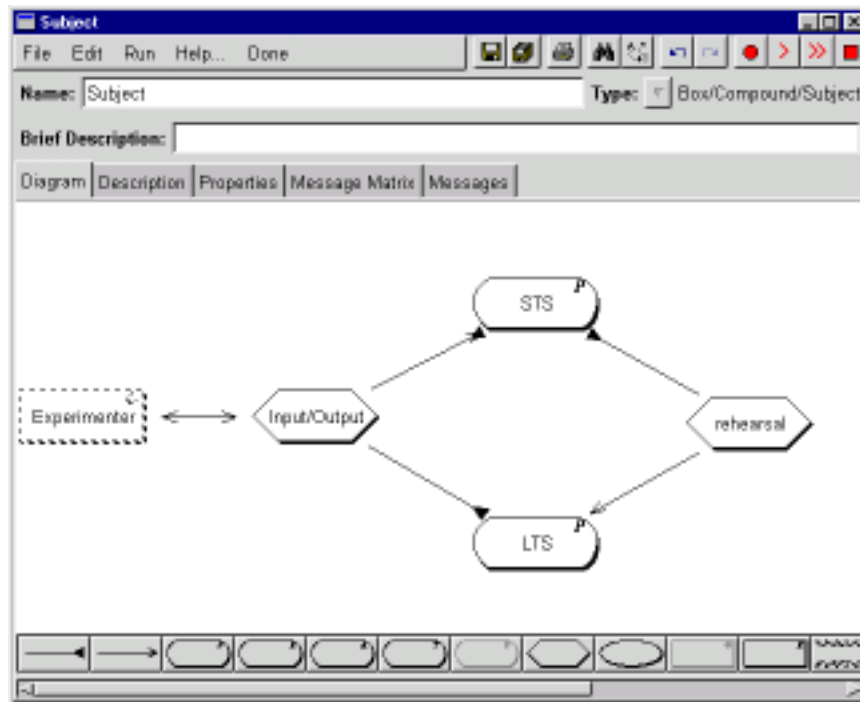


This certainly looks quite like a recency effect, but there is of course no primacy effect; the probability of recall just drops as items get older. If you have the time, you might like to experiment with this model some more before going on to the next section. Here are a few suggestions:

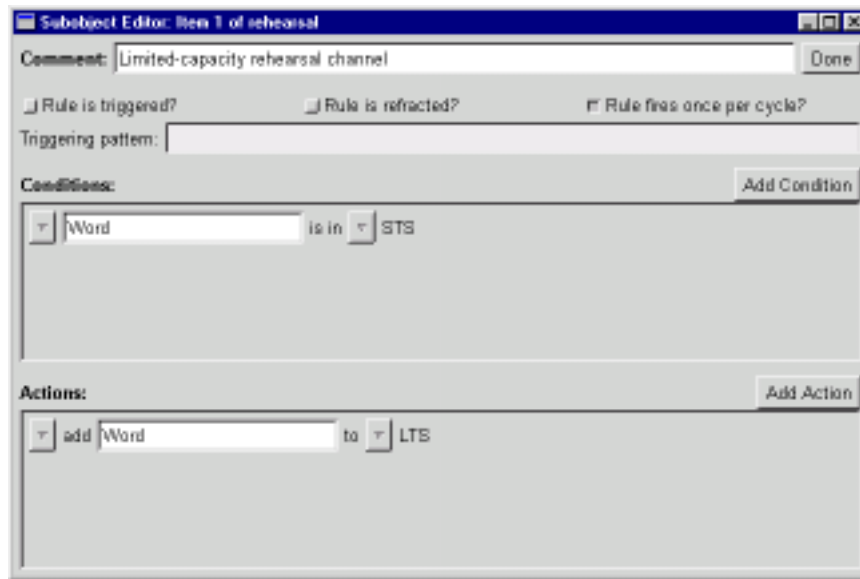
1. Keep running more trials — the curve will gradually become smoother.
2. Return to the STS box properties, and change the On excess property to another value of your choice. See what happens to the shape of the graph when you run a few trials. Explain.
3. Remember the Messages view of Input/Output? Watch what happens there now when you run (or single-step) through a trial. You can look at other boxes' Messages views too, if you like. What's going on? What do the numbers represent?

## 5 Adding the Long Term Store

We have seen that the Short Term Store alone can generate a respectable recency effect. However, to produce a primacy effect too, we need to add two more components: a Long-Term Store (LTS), which should be a Propositional Buffer, and a Rehearsal Process. The Rehearsal process should be able to read from the STS, and write to the LTS. Input/Output should be able to read LTS. Given what you've learned about editing box-arrow diagrams in the previous section, you should be able to modify the old diagram to produce such an arrangement, so go ahead and do it. Feel free to rearrange the boxes to suit your taste — you'll find that the arrows follow them around. It should end up looking roughly like this:



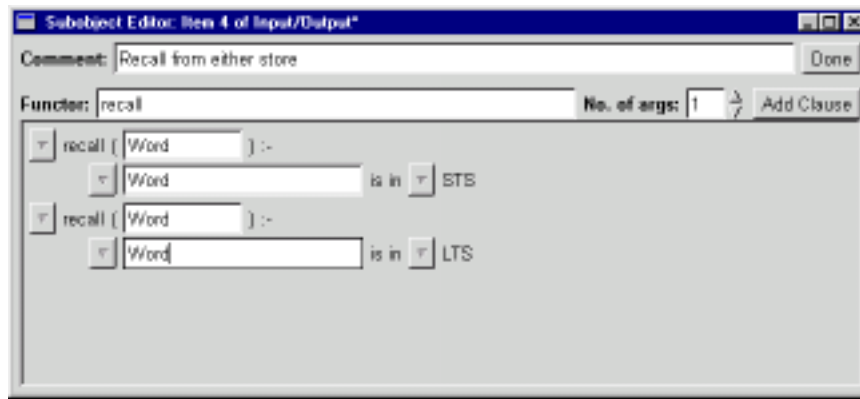
Next we need to make the Rehearsal process actually do something. Recall that its purpose is to take elements from STS and transfer them to LTS, and that it is capacity-limited. So open the Rehearsal Process window, and create a new rule in the Initial Contents view. Unlike the previous rules we have encountered, this one is not triggered — the idea is that it is supposed to operate autonomously. However, the fact that it is capacity-limited means that we must restrict its firing somehow, so check the Rule fires once per Cycle box, and uncheck the Rule is refracted box. The first of these ensures that the rule will only fire one time on each processing cycle, even if there are many possible ways the variables in the rule can be bound. (The variable-binding that is used will ultimately depend on the access properties of STS.) The second allows the rule to fire with the same variable-binding on subsequent processing cycles. (In this case allowing the same word may be rehearsed multiple times.) Now the rule must read an item from STS, and add it to LTS, so as with the recall rule, we need a match condition which reads a word from STS. Finally the action should be an Add operation, to add the word to LTS. It should end up looking like this:



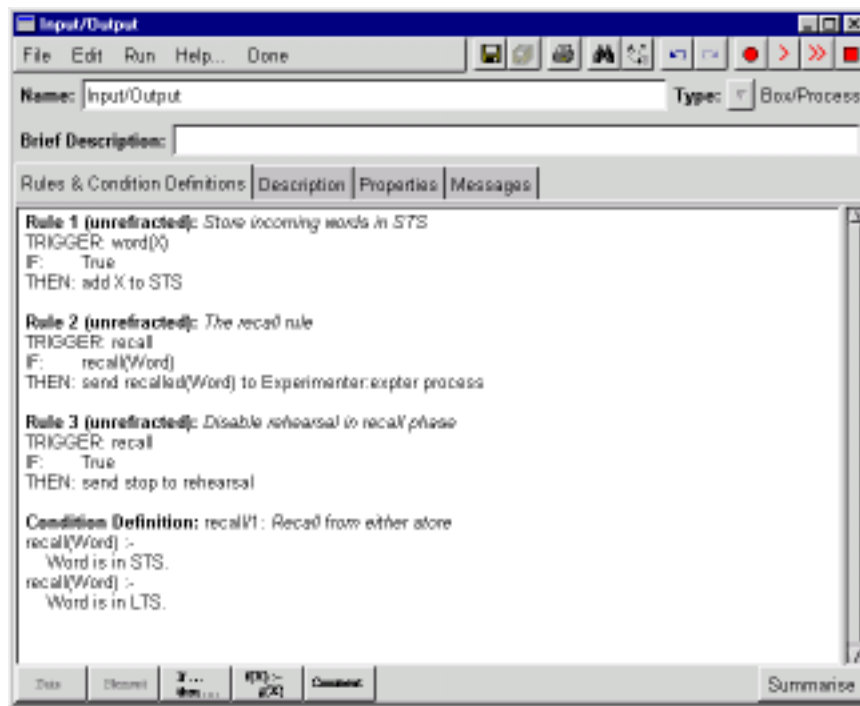
Now if you switch to the LTS box, Current Contents view, initialise and single-step through the model using the Step button, after a few steps you will see words accumulating in the LTS. But if you run a trial at this point, you will soon discover that it never stops. (Use the Stop execution button to terminate such runaway execution.) This is because the rehearsal rule is always able to fire, whatever else has happened. An *ad hoc* solution to this problem is to draw a Send arrow from the Input/Output box to the Rehearsal box, and add a new rule to Input/Output. We will send a special `stop` message via this arrow to the rehearsal process whenever the recall phase of the trial starts. Create an appropriate triggered rule in the Input/Output box. The rule should have `recall` as its trigger and `send stop` to Rehearsal as its only action. (As a somewhat advanced exercise, find a principled solution to the problem of halting execution after recall.)

While we're in the Input/Output box, we need to modify the recall procedure to recall from both memory stores. First of all, we must define a *Condition* (actually a piece of pure Prolog), with two clauses, each of which reads from a different store. So, insert a dummy condition by clicking on the button marked  $f(X) :- g(X)$  on the lower toolbar and then clicking on the canvas; a new dummy condition will appear where you clicked. Double-click on this to open the condition editor. First of all, you should change the contents of the Functor: field to a name of your choice (remember that you shouldn't use capital letters here.) We'll use the name `recall`. Then change the value in the No. of args: field to 1 — this means that the condition takes a single argument.

Now, click the Add Clause button, and a new entry will appear in the Conditions section of the window. Right-click on the button at the far left, to pull down a menu, and navigate to the Add subcondition menu and select `match`. A new `match` subcondition will appear — you should set it to read a `Word` from `STS`. Next type `Word` in the argument field in the head of the clause — this variable shares the value of the word read from the buffer, and passes it up to the recall rule. Next, click the Add Clause button again. Now define the second clause of the condition as you did the first, except this time it should read a `Word` from `LTS` instead of `STS`. In the end it should look like this:



Close the Condition Editor window. The condition that we have just defined states two ways in which a word can be recalled — it can be recalled if it is in STS, and it can be recalled if it is in LTS. We must now adjust our recall rule to use this defined condition. Go to the Input/Output window and double-click on the main recall rule; a rule editor window will open. Right-click on the button at the left of the only condition to pull down the menu, and navigate to the `Insert after condition>Prolog` submenu. You should see an entry for `recall/1` (or whatever you called the recall condition). Select this, and it should appear after the match condition in the main rule editor window. Type `Word` in the argument field, and then delete the match condition, by selecting the appropriate item on its menu. Close the rule editor window. By now the Input/Output box's Current Contents view should look like this:



At this point you can try running the model, while you monitor the graph output. If you run 20 or more trials, it should become clear from the graph display that there is now a Primacy effect as well as a Recency effect. We will improve on this behaviour in the next section, but for now you can consider the following questions:

1. You should be able to account for the Primacy Effect on the basis of the considerations in the introduction. What's going on?
2. If you monitor the Input/Output box's Messages view, you'll see that the model sometimes recalls the same word twice in the same trial. Why is this?
3. The serial position curve still doesn't look like the one in the introduction. Try to characterise the difference. Can you account for it?

## 6 Decay, Time and Rehearsal

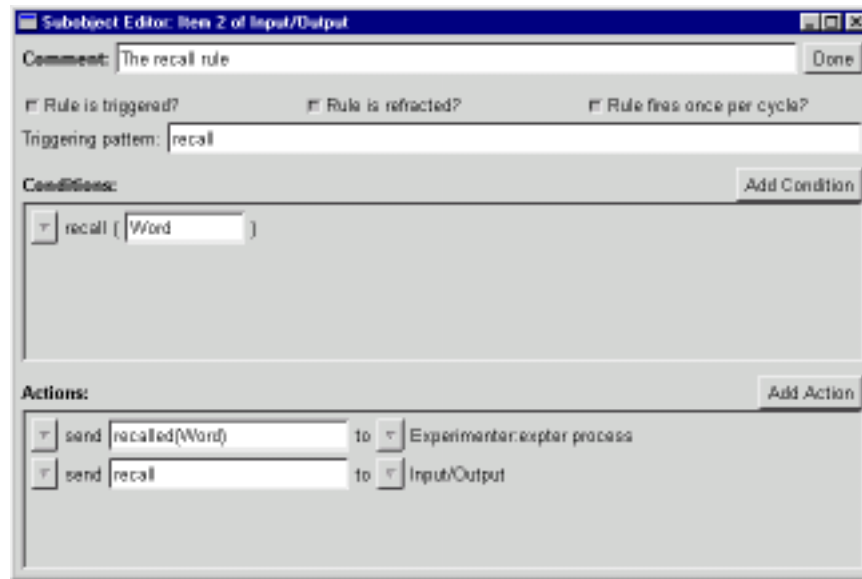
The previous sections have produced a model which shows simple Primacy and Recency effects in a serial position curve. However, there is one major feature of the theory which we still haven't dealt with, namely *Decay*. The Long-Term Store is supposed to be of unlimited capacity but less-than perfect reliability, whereas in the present model, if something gets in to LTS it stays there until the end of the trial.

COGENT supports decay of items held in buffers by means of buffer properties. Open the LTS box and switch to its Properties view. There are two properties of interest here, the Decay property, which has three possible values — None, Probabilistic and Fixed — and the Decay Constant, which is a number. The Decay Constant should be interpreted as a number of cycles. (The cycle is COGENT's basic unit of time.) Probabilistic and Fixed decay both use the value of the Decay Constant — call it  $D$  — but in different ways. In Fixed decay, items are deleted as soon as they have been in the buffer for  $D$  cycles, whereas in Probabilistic decay  $D$  specifies a half-life (as in radioactive decay), so that items may decay at any time, but the probability of their having decayed by the time they are  $D$  cycles old is 0.50.

Set the Decay property of LTS to Probabilistic, and the Decay Constant to 20, then return to the graph view and run a block of 20 trials. You will (probably) see that the Primacy Effect has been greatly reduced, or even abolished completely, whereas the Recency Effect is still strong. Now if you increase the LTS Decay Constant, making items less likely to decay in the course of a trial, the size of the Primacy Effect should increase too.

Another way you can affect the performance of the model is to change the rehearsal rate. Remember that the experimenter sends one word per cycle during the memorisation phase, and that rehearsal transfers one item per cycle to LTS. If you copy the rehearsal rule, so that there are two identical copies, you double the rehearsal rate. Try it — you should see another increase in the size of the Primacy Effect, as well as a raising of the level of the central portion of the curve. On a related note, if you like you can change the Duplicates property of LTS to the value Yes. This allows multiple copies of the same word in LTS, and should improve recall in LTS.

These considerations about time suggest another way the model can be improved. Although the experimenter sends words to the subject serially, the subject currently does not recall serially. We would like to recall words one at a time, on separate cycles, rather than in a parallel burst on a single cycle. Open the Input/Output box and edit the recall rule again. Check the Rule fires once per cycle and Rule is Refracted boxes. The first ensures that only one word will be recalled per cycle, and the second ensures that each word will be recalled only once. Then add a Send action, to send the trigger `recall` to the Input/Output box. This means that each time the rule fires, it sends a message to itself to try to fire again on the next cycle. (Note that if you can't get a rule in a given Process box to send a message to the same box, make sure that the Process' Recurrent property is set to Yes.) It should end up looking like this:



This is the end of the tutorial. For more information, see the COGENT help system and web site. There are a couple of suggestions for further experimentation with the model below, if you want to go further.

1. You have already been introduced to a range of parameters that can affect the behaviour of the model, including capacity limitations, behaviour when that capacity is exceeded, decay type and rate, and rehearsal rate. Another property of interest is the Buffer Access property, which offers the options FIFO (First-In/First-Out), LIFO (Last-In/First-Out) and Random, and controls the order in which items are read from buffers. Feel free to play with these (and other) parameters to see how they affect the model's behaviour.
2. Have a look around the Experimenter system. You'll see it's written using the same language as the Subject. Try to discover how it works.
3. If you feel confident, go on to develop the model into something substantial.

## References

- Atkinson, R. C., & Shiffrin, R. M. (1968). Human memory: A proposed system and its control processes. In Spence, K. W., & Spence, J. T. (Eds.), *The psychology of learning and motivation: Advances in research and theory*. Academic Press, Orlando, FL.
- Atkinson, R. C., & Shiffrin, R. M. (1971). The control of short term memory. *Scientific American*, 225, 82–90.
- Cooper, R., & Fox, J. (1998). COGENT: A visual design environment for cognitive modelling. *Behavior Research Methods, Instruments, & Computers*, 30(4), 553–564.
- Glanzer, M., & Cunitz, A. R. (1966). Two storage mechanisms in free recall. *Journal of Verbal Learning and Verbal Behavior*, 5, 351–360.
- Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63, 81–97.
- Young, R. M., & O'Shea, T. (1981). Errors in Children's Subtraction. *Cognitive Science*, 5, 153–177.