An Introduction to the COGENT Modelling Environment

5^{th} International Conference on Cognitive Modelling

April 9^{th}, 2003

Bamberg, Germany

Contents

Richard P. Cooper & Peter Yule

1 COGENT: Principal Features

COGENT is a computational modelling environment in which information processing models of cognitive processes may be developed and explored. The environment provides a range of functions that allow students and researchers alike to explore ideas and theories relating to cognitive processes 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 the COGENT environment are:

- A visual programming environment;
- A range of standard functional components, including rule-based processes, memory buffers, simple connectionist networks, input "sources" and output "sinks";
- An expressive, extensible, rule-based modelling language and implementation system;
- Mechanisms for the control of inter-component communication;
- Automated data visualisation tools, including tables, graphs, and animated diagrams;
- A powerful model testing environment, supporting Monte Carlo-style simulations and a generalised "experiment-based" scripting language;
- Research programme management tools, allowing related models to be encapsulated within a research programme and providing a graphical display of the relations between models within such a programme;
- Version control on models; and
- Support for documentation of both individual models and complete research programmes.

1.1 The Visual Programming Environment

COGENT (Cooper & Fox, 1998; Cooper, Yule, Fox, & Sutton, 1998; Cooper, 2002) simplifies the process of model development by providing a visual programming environment in which models may be created, edited, and tested. The visual 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). Functional modularity views a cognitive process as the product of a set of interacting sub-processes, where each sub-process has an identifiable function that contributes to the whole and the interactions between sub-processes are limited in range (see, for example, Fodor, 1983). Object-oriented design analyses complex computational systems in terms of sub-systems of different types, with the behaviour of each sub-system being determined in part by its type and the values of a set of properties that are specific to each type of sub-system (see, for example, Rumbaugh, Blaha, Premerlani, Eddy, & Lorensen, 1991; Graham, 1994).

Models are specified in COGENT by sketching their functional components using COGENT's graphical model editor. Thus, Figure 1 shows a box and arrow diagram depicting a classic theory from cognitive psychology — the Modal Model of memory (Atkinson & Shiffrin, 1968, 1971). The diagram shows five functional components, four of which are central to the Modal Model: *I/O Process* (a process that acts as an interface between the memory systems and any task to which they are applied), *STS* (a short-term store in which information is temporarily placed while it is rehearsed), *LTS* (a long-term store in which information is consolidated), and *Rehearsal* (a process that transfers information from *STS* to *LTS*). The one remaining component of the diagram — *Task Environment* — is a compound box (i.e., a box containing further internal structure) that is used to administer a task when testing the model.

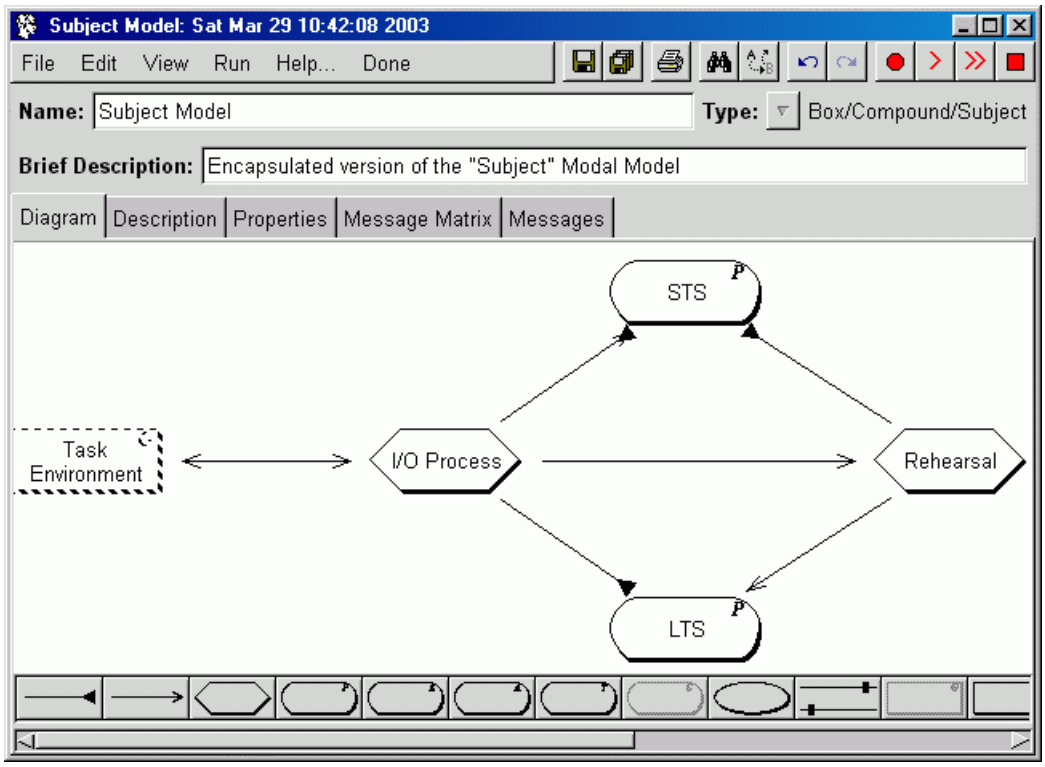

Figure 1: A box and arrow diagram of the Modal Model of memory (as developed in Section 3)

Different shaped boxes within a COGENT box and arrow diagram represent different types of component. Hexagonal boxes represent processes that transform information, rounded rectangular boxes represent buffers that store information, and rectangular boxes represent compound systems with internal structure. Similarly, different styles of arrow indicate different types of communication between the components, such as reading information from a buffer or sending messages to a process. COGENT's graphical model editor provides facilities for creating models using these and other standard types of component. Components provided in addition to the above include simple feed-forward networks, interactive activation networks and input/output devices.

Several different kinds 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, and Messages) provide access to: a text window into which notes or comments on the model may be entered; the set of properties and parameters 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 model's top-level box.

1.2 Standard Component Types

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

Rule-based processes: Rule-based processes manipulate information according to symbolic rules

specified by the user. A powerful rule language and rule interpreter allows rule-based processes to perform complex manipulations and transformations of information. Such processing may be contingent upon the contents of other components of the model.

Memory buffers: Buffers are general information storage devices. They may be used for both short term and long term storage, and different subtypes of buffer may be used to store information in different formats (e.g., propositional, tabular, and analogue). The detailed behaviour of any instance of a buffer is determined by its properties, which specify such things as 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 components).

Connectionist networks: COGENT is intended primarily for high-level symbolic modelling. Nevertheless, COGENT's generalised processing engine allows direct interface with some simple connectionist objects (two-layer feed-forward networks, associative networks, and interactive activation networks). This facility makes COGENT suitable for a variety of hybrid modelling applications. As in the case of buffers, precise network behaviour is determined by properties associated with the network. These properties govern learning rate, initialisation, the activation function, etc..

I/O sources and sinks: Specialised data source components allow data to be fed into other components in a controlled manner. Data sinks by contrast allow the collection of data from other components during model execution. Three types of data sink — text-based sinks, tabular sinks, and graphical sinks — allow a range of options for storage and presentation of model output.

Inter-module communication links: Inter-module communication is indicated within a COGENT box and arrow diagram by arrows drawn between the boxes. Two basic types of arrow are provided: read arrows and write/send arrows.

Further details of these component types are given below.

1.3 The Rule-Based Modelling Language

COGENT's rule-based modelling language allows complex processes to be specified with 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 testing the equality of data elements, as well as operations that set variables, such as matching some information stored in a buffer. Actions allow messages of various forms to be sent to other boxes. The rule language is highly expressive. While this introduces some complexities, structured rule editors simplify the process of specifying rules and other tools allow the processing of rules during model execution to be monitored.

An example rule is shown in Box 1. This rule fires when *Possible Operators* (a buffer) contains an element of the form `operator(Move, possible)`, and the condition `evaluate_operator(Move, Value)` can be satisfied. `Move` and `Value` are both variables. On firing, `Move` and `Value` are bound, the rule deletes one element from *Possible Operators* (`operator(Move, possible)`) and adds another (`operator(Move, value(Value))`).

Rules are contained within processes (the hexagonal boxes within a COGENT box and arrow diagram: see Figure 1), and may be supplemented with user-defined conditions. Such conditions may be used to provide additional control over the circumstances in which rules apply. This is illustrated by the rule in Box 1: `evaluate_operator(Move, Value)` is a call to a user-defined condition, the definition of

IF: `operator(Move, possible)` is in *Possible Operators*
`evaluate_operator(Move, Value)`
THEN: delete `operator(Move, possible)` from *Possible Operators*
add `operator(Move, value(Value))` to *Possible Operators*

Box 1: A simple rule that updates a buffer

which is specified elsewhere. The condition definition language is based on Prolog (see, e.g., Bratko, 1986; Sterling, 1986; Clocksin & Mellish, 1987), a highly expressive AI programming language which provides COGENT with substantial flexibility.

The rules and condition definitions of a process are listed in a standard format within the process' Rules and Condition Definitions view. This view also provides access to specialised rule editing facilities. Figure 2 shows this view for *Select Operators*, a process from a model of problem solving described in Chapter 4 of Cooper (2002).

1.4 Automated Data Visualisation Tools

COGENT provides a number of visualisation tools to assist in the monitoring and evaluation of a model. These tools take the form of additional types of box, and allow data to be displayed in standard tabular or graphical forms. 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, as in Figure 3. 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 for any particular table (e.g., row height, column width, row and column labels, etc.) being governed by that table's properties.

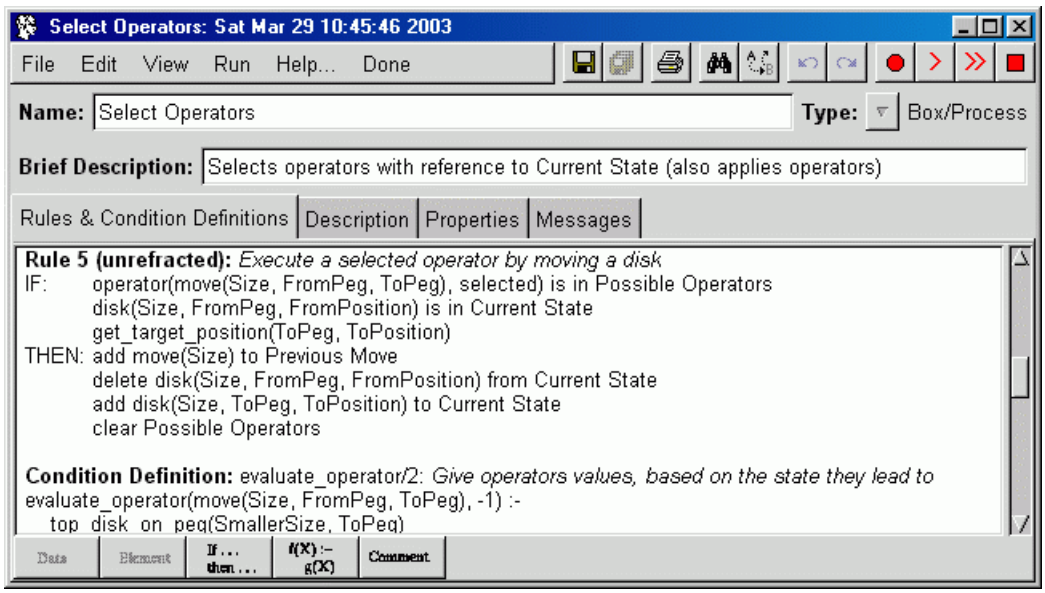

Figure 2: Some rules from the *Select Operator* process within a problem solving model

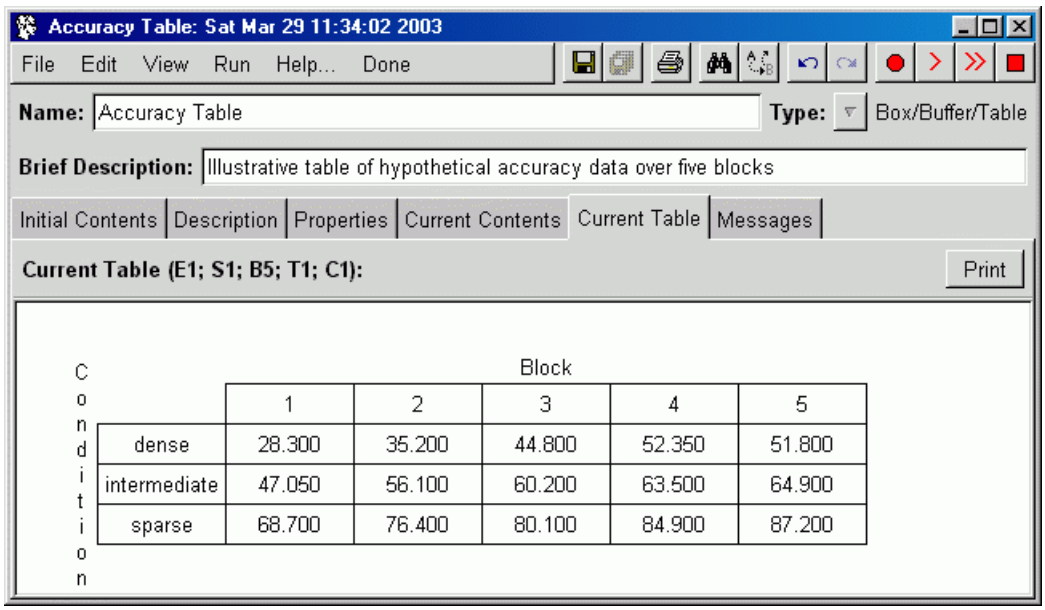

Figure 3: A table buffer, displaying data accumulated over five blocks of a task

Two types of table are provided. Output tables are write-only: data sent to such tables are displayed but cannot be inspected 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 the buffer's access properties, which allow access to be based on either temporal features of buffer elements (e.g., primacy, recency) or spatial features of the elements (e.g., left to right or right to left placement).

Graphs Data may be displayed graphically in several standard formats, including line graphs, scatter plots and bar charts. Figure 4 shows a line graph of data generated by a model of the Glanzer and Cunitz (1966) free recall task. A single graph may be used to display multiple data sets in different colours. 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 configurable properties associated with the graph.

Generalised Graphical Output Facilities are also provided for more general graphical output. Propositional buffers may be augmented with visualisation rules which map buffer elements to graphical objects (e.g., lines, shapes or text at specified coordinate positions), and these graphical objects may be stored and viewed directly within analogue buffers. To illustrate, Figure 5 shows a visualisation of the contents of a propositional buffer whose contents represent disks in the Tower of Hanoi problem. (This problem is discussed at length in Chapter 4 of Cooper (2002).) Displays associated with propositional and analogue buffers are dynamically updated whenever the contents of the buffer change.

1.5 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. Normally this involves

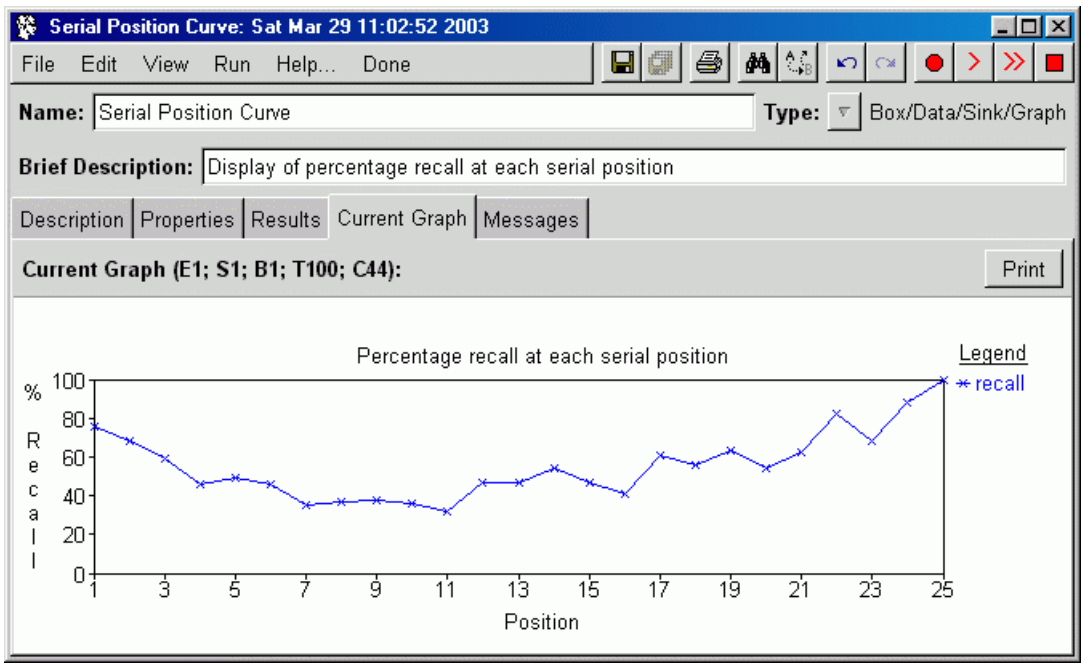

Figure 4: A graphical buffer, showing a line graph summarising output from a model of memory applied to a free recall task

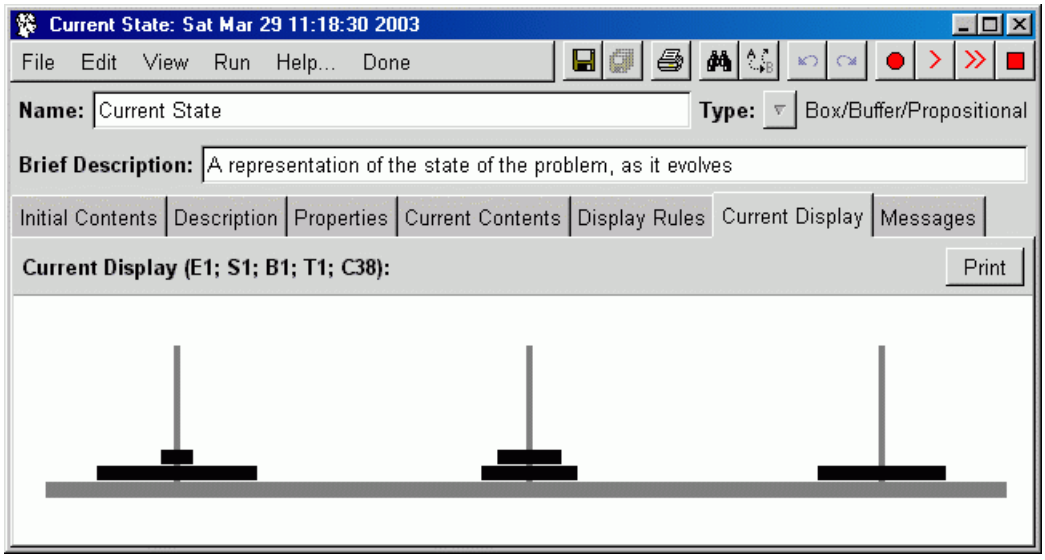

Figure 5: A visualisation of a propositional buffer's contents, showing an intermediate state of the Tower of Hanoi problem

presenting a single stimulus and gathering a single response. However, it is also possible to specify extended experimental designs, in which, for example, numerous virtual subjects are run in each of several experimental conditions, with each experimental condition involving a number of blocks and each block involving a number of trials. Such designs are constructed through a special purpose experiment script editor.

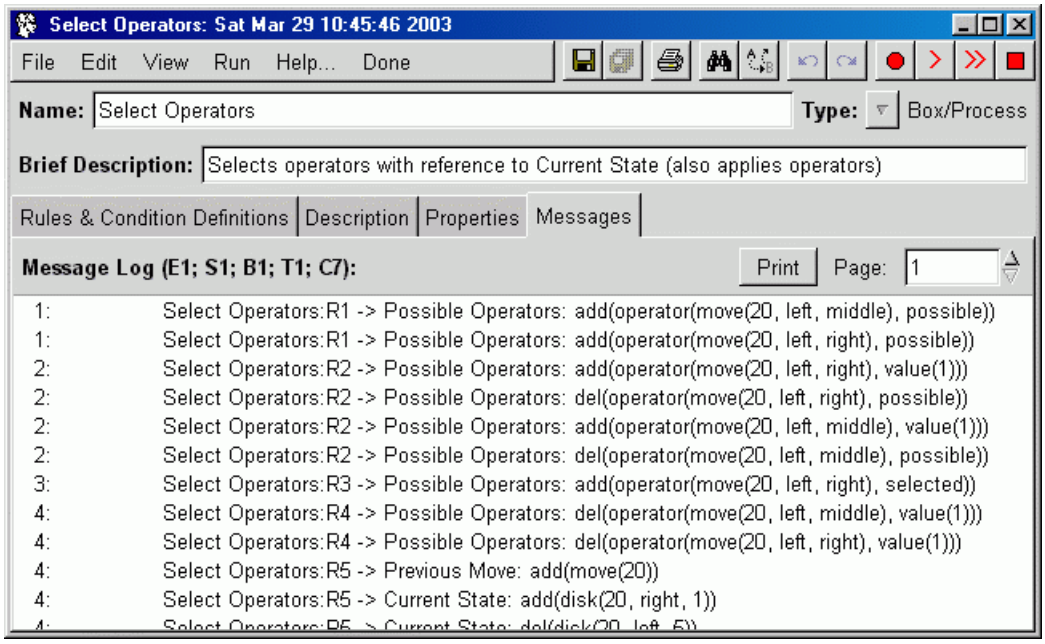

Figure 6: The Messages view of *Select Operators*

The model testing environment also provides a range of facilities for monitoring and debugging models. Monitoring is provided through the Messages view available on each component's window. This view shows all messages generated or received by a component. Thus, Figure 6 shows the messages relating to the *Select Operators* process mentioned above after 7 processing cycles. Each line shows the cycle on which the message was received or generated, the source of the message (e.g., Rule 5 of *Select Operator*), the message's destination (e.g., *Previous Move*), and the message's content (e.g., `add(move(30))`). Other facilities allow the traffic between components within a compound box to be monitored (through the box's Message Matrix view), and the execution of specific conditions within rules to be traced.

1.6 Research Programme Management

The development of a cognitive model typically takes place over an extended period of time. COGENT supports this development through tools for managing sets of models within a *research programme* (see Lakatos, 1970). These tools include 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 and archiving), documentation support, and a front-end to the graphical model editor described above.

Access to research programme management tools is through the Research Programme Manager, shown in Figure 7. The left side of the window shows all research programmes registered with COGENT. When a research programme is selected its history is displayed in the frame on the right in the form of a tree. Each node in the tree corresponds to a separate model, and double-clicking on a node opens COGENT's model editor on the corresponding model. The progress of time is represented in the history diagram along the horizontal axis, with models to the right being developed after models to the left. Links in the tree show ancestral 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.

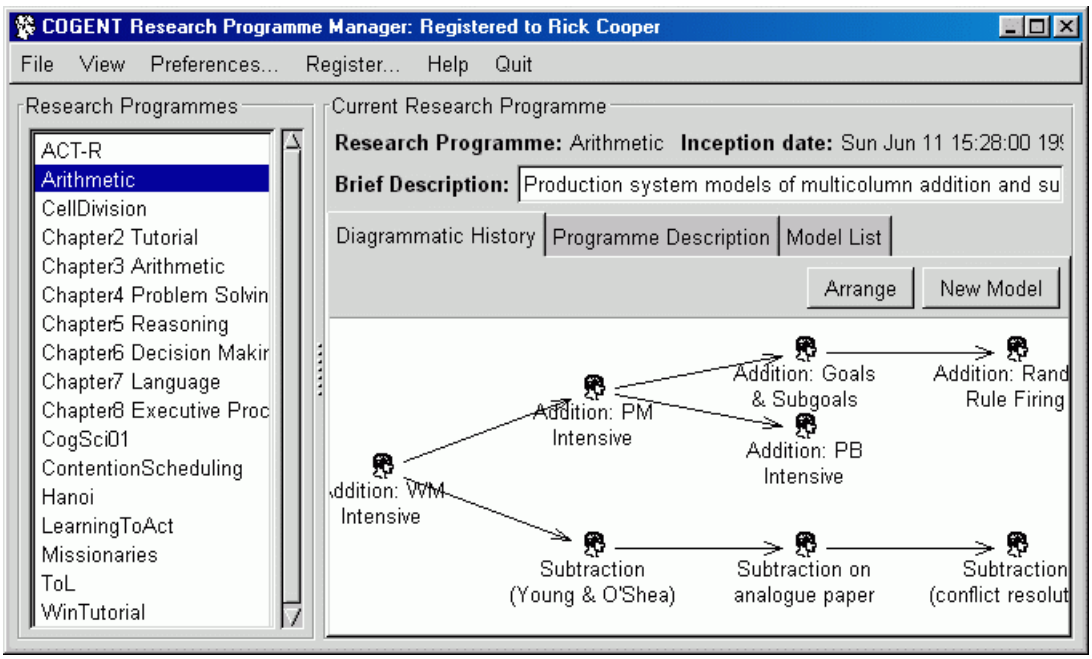

Figure 7: The research programme history view

1.7 Domains of Application

COGENT is intended to be as flexible and domain general as possible. To date, models in the following domains have been developed:

- Memory (Free recall)
- Arithmetic (Multicolumn addition and subtraction)
- Mental Imagery (Shepard's mental rotation task)
- Problem Solving (Missionaries, Hanoi, Cryptarithmetic)
- Deductive Reasoning (Syllogisms, Allen inferences)
- Categorisation/Decision Making (Medical diagnosis)
- Sentence Processing (Parsing)

Memory Early models of memory argued that primacy and recency effects in free recall resulted from the interaction of two storage devices. The COGENT model captures this interaction, and allows exploration of the computational limitations (e.g., decay, capacity) of those storage devices. COGENT can also generate serial position curves (see Section 3).

Multicolumn Subtraction This model employs an OPS-style production system interpreter. The subtraction procedure is defined by a set of production rules (Young & O'Shea, 1981). An analogue buffer is used to display the current state of the problem. Productions may be ignored to simulate errors or bugs in children's subtraction procedures.

Mental Rotation An analogue buffer is used by this model to store the mental representation of each shape. Rotation of the mental image is subject to point movement. Data on speed and accuracy is presented in a graph.

Problem Solving The missionaries and cannibals task employs a standard operator cycle: propose, evaluate, select and execute. The Tower of Hanoi/London tasks use a goal stack to focus operator selection. The cryptarithmetic model is an order of magnitude more complex. Operator selection in this task involves a choice between dozens of applicable operators. Nevertheless, the same basic mechanisms are used in this model and in the missionaries and cannibals model. In addition, an analogue buffer is used as an external representation during problem solving.

Reasoning Syllogistic reasoning models using both mental models and Euler circles have been developed. For mental models, models of the premises can be developed using a tabular buffer. For Euler circles, models of the premises can be developed using an analogue buffer. Allen inferencing models have also been developed. In these models, point movement in an analogue buffer is used to account for some empirical effects as suggested by Berendt (1996).

Medical Diagnosis Several sets of models have been developed within an extensive research programme investigating information seeking behaviour in diagnosis tasks (see Cooper & Fox, 1997; Fox & Cooper, 1997; Yule, Cooper, & Fox, 1998; Cooper & Yule, 1999; Cooper, Yule, & Fox, In press). Within each set of models, an experimenter module presents stimuli and collates results. An experiment script specifies the experimental design (e.g., number of trials per block). Visualisation components and statistical functions allow constant monitoring of the model. The environment supports comparative modelling (of different approaches to the same task) through the use of such functions.

Sentence Processing A series of sentence processing models based on left-corner chart parsing, incremental processing and backtracking have been developed (see Chapter 7 of Cooper (2002)).

COGENT has also been applied as a general simulation environment in non-cognitive domains, including the simulation of biological processes and multi-agent systems.

2 An Illustrative Task: Free Recall

The remainder of this session introduces many of the basic concepts of COGENT through a tutorial-style development of an implementation of a classic cognitive model: Atkinson & Shiffrin's so-called "Modal Model" of human memory (Atkinson & Shiffrin, 1968, 1971). The model, called "modal" because most psychologists subscribed to it at one point, is no longer current (though elements of it remain in more recent theories of memory), but it is used here because it serves as a good illustration of how COGENT can be used to implement concepts and models from the psychological literature.

The Modal Model was developed to explain the recency and primacy effects in free recall serial position curves (Glanzer & Cunitz, 1966; Postman & Phillips, 1965). 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 so that they may 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, as in Figure 8. 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*.

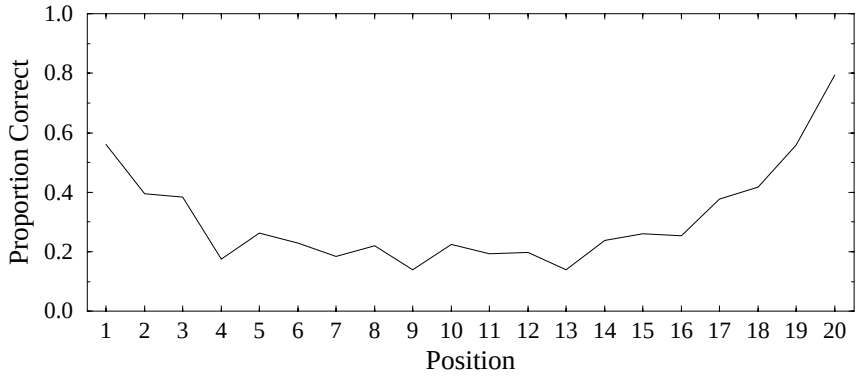

Figure 8: Typical recall accuracy as a function of serial position in the free recall task (adapted from Glanzer & Cunitz, 1966)

The recency effect can be explained by postulating 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). According to the Modal Model, 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 participants can recall from either store, one would expect a peak at the end of the curve, spanning as many items as the postulated capacity limitation of STS will allow (e.g., 7 ± 2 items: Miller, 1956).

The primacy effect requires a different explanation. In the Modal Model it is assumed to result from a 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. Assuming that more rehearsal implies better recall, they should therefore be remembered better.

3 The COGENT 'Modal Model' Model

3.1 Task Infrastructure

The box and 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 may be used not only to simulate cognitive processing, but also to implement stimulus presentation and data collection. We adopt this approach. However, because such aspects may obscure the presentation of the Modal Model, we encapsulate the "task environment" and "subject" modules in functionally distinct, pre-specified, compound boxes.

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 `FreeRecall` from COGENT's Research Programme Manager window and double-click on `Stub Model` to open it. You should see a screen like that in Figure 9.

This is the main box-arrow diagram of the model. It comprises two compound boxes, linked by arrows.

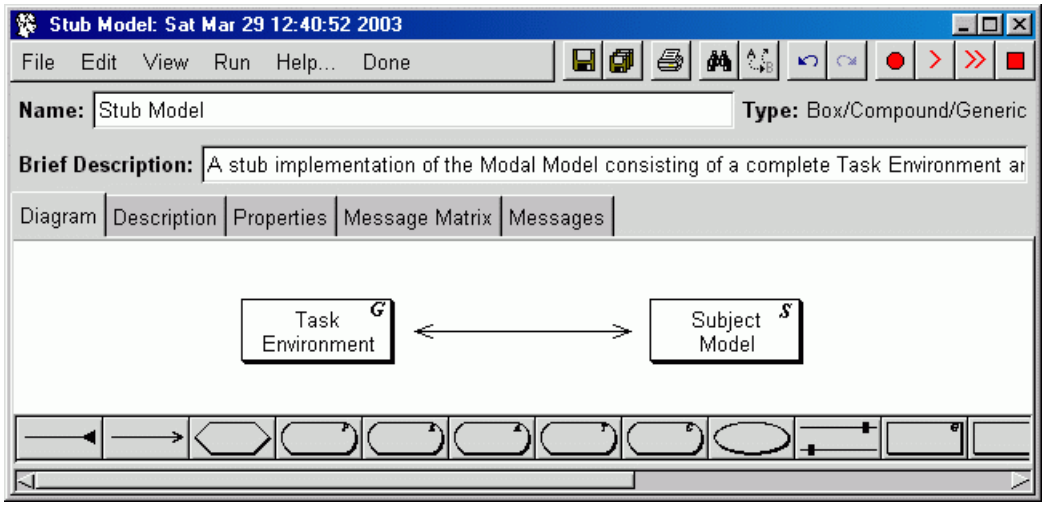

Figure 9: The top-level box-arrow diagram for the Modal Model

The box labelled *Task Environment* presents stimuli to, and collects responses from, the box labelled *Subject Model*. *Task Environment* also maintains a graphical representation of the serial position curve (as shown above in Figure 4). You can find the graph display by double-clicking on *Task Environment*, which will open a new window (Figure 10).

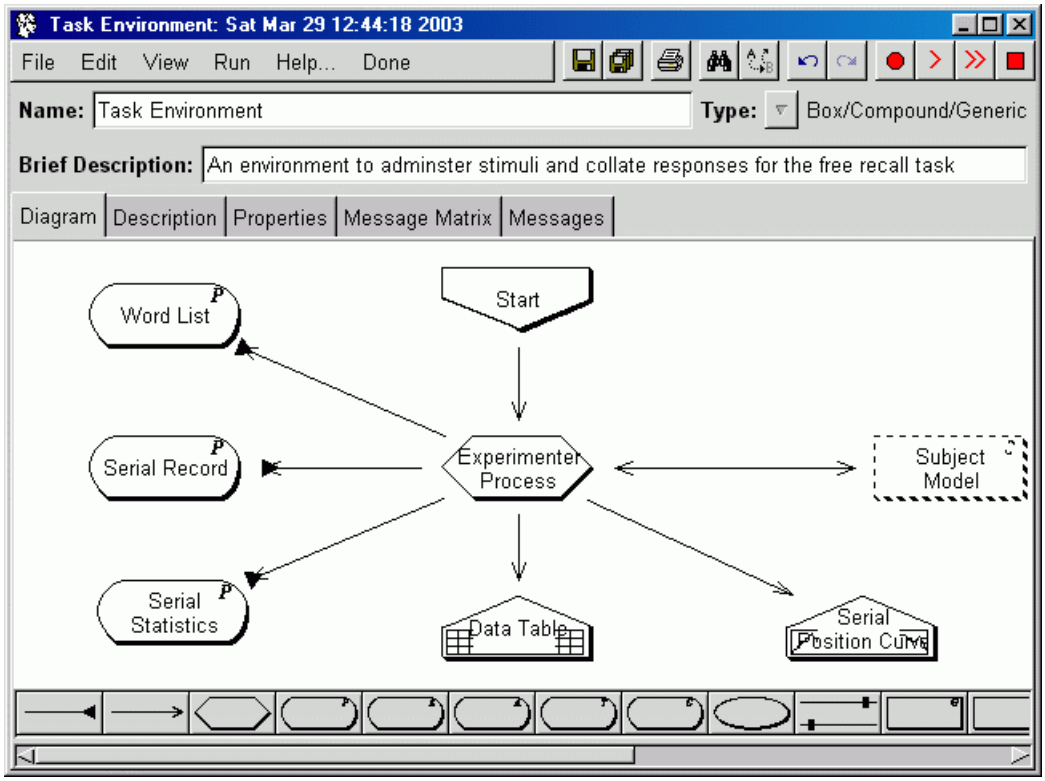

Figure 10: Internal detail of *Task Environment*

The graph display itself is opened by double-clicking the box at the bottom of the diagram labelled *Serial Position Curve*. 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 a box, you have a variety of different views available — the exact range of views depends on the type of box. Boxes that contain other boces (which they display as a box-arrow diagram) are known as *Compounds* and are represented by a rectangular box. Round-ended boxes are *Buffers*, of which there are a range of types. Hexagonal boxes are *Processes*.

Finally in this section, we introduce the *Subject Model* box and its relation to *Task Environment*. Return to the main box-arrow diagram. (You may wish to close the windows associated with *Task Environment* and *Serial Position Curve*.) Double-click on the compound box labelled *Subject Model*. 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, as shown in Figure 11.

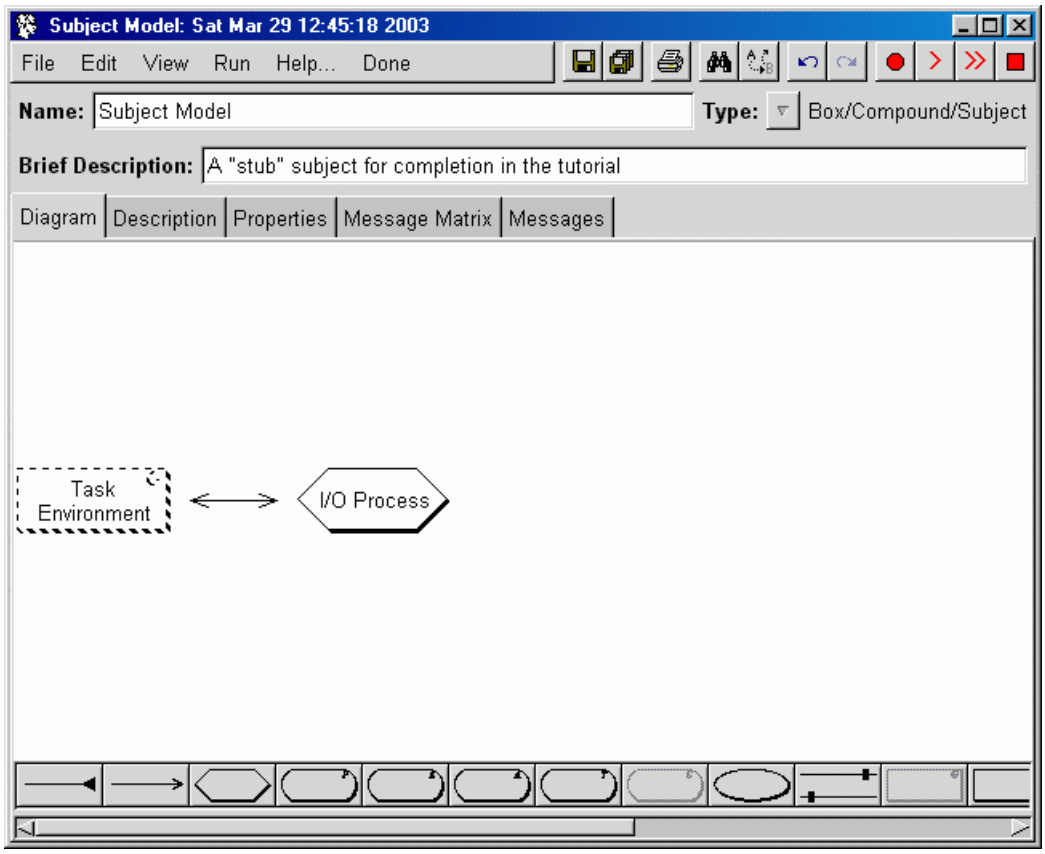

Figure 11: The "stub" *Subject Model*

The hexagonal box labelled *I/O Process* 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. This will initially display a blank screen. Now notice the row of buttons marked in red on the tool-bar at the top of the window, like this:

This is the run tool-bar, 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 *Task Environment* — in this model *Task Environment* sends a message of the form `word(Word)` to *Subject Model* on each successive cycle. When *Task Environment* has sent 25 words to *Subject Model*, it sends a final `recall` message (see Figure 12).

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

3.2 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 *Subject* 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 tool-bar 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 (see Figure 13).

Next, we need to link *I/O Process* to the new buffer, to allow *I/O Process* to store the incoming words in that buffer. For this we need a *Write* arrow. Select the appropriate arrow type (by clicking on the rightmost of the two arrow buttons on the drawing tool-bar). Now click on *I/O Process*, and, without releasing, move the pointer to the new buffer — an arrow will follow the mouse pointer — and release the button. Figure 14 shows how the resulting diagram should look.

Before continuing, you should double-click the new box to open it, and give it a name, by typing "STS" into the field near the top of the buffer's window labelled Name:.

We are now ready to add a rule to *I/O Process* that will transfer incoming words to *STS*. Double-click on *I/O Process* to open it, and view the Rules & Condition Definitions tab. You will see a tool-bar 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

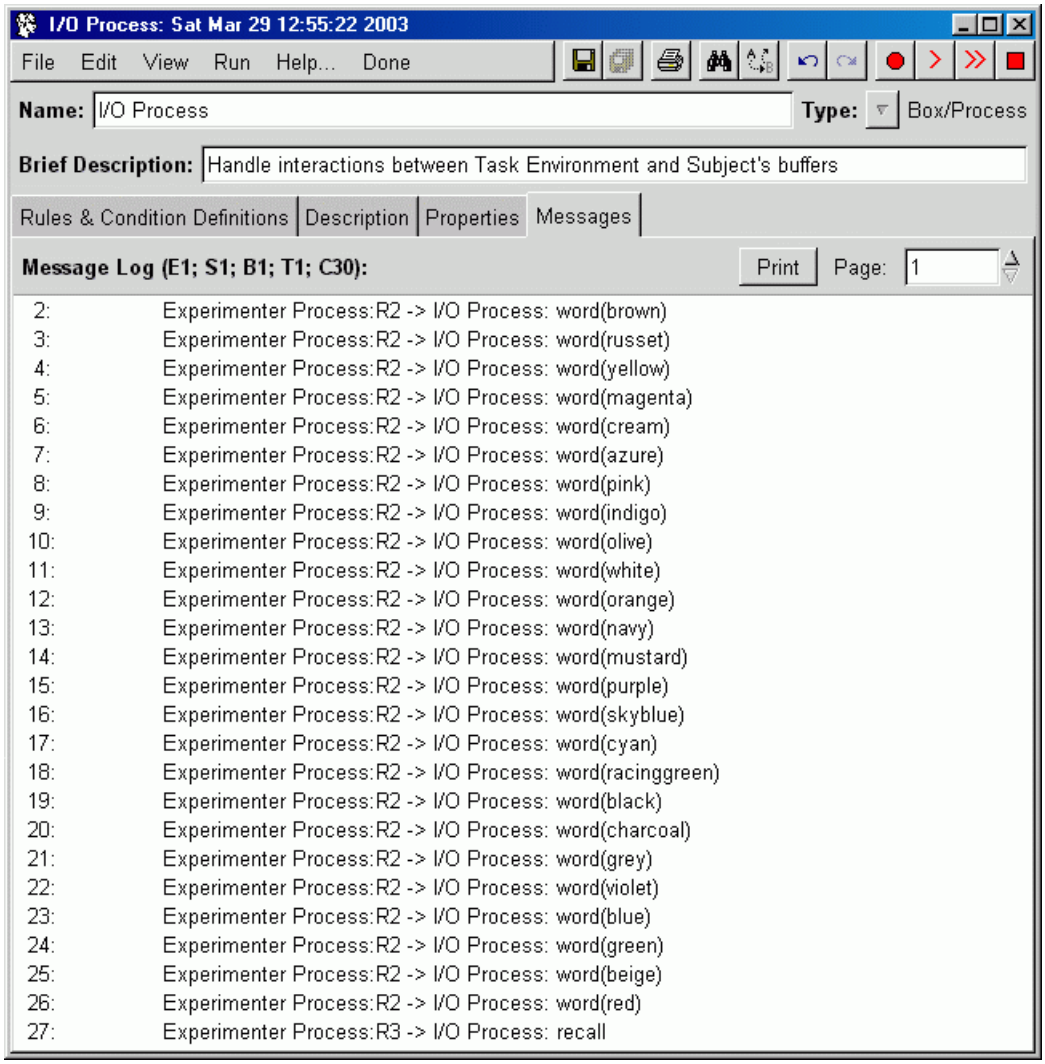

Figure 12: Messages received by *I/O Process* from *Task Environment*

new window will open. This is the *Rule editor*. It simplifies the process of writing rules, and ensures that they follow the syntax of COGENT's rule language. We are going to use the rule editor to create a rule which will write each incoming word from *Task Environment* to *STS*.

First, check the Rule is triggered check-box within the rule editor. This indicates that this rule responds to trigger messages such as those sent by *Task Environment*. The Triggering pattern field, previously grayed 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.) Recall (from Figure 12) that *I/O Process* receives messages of the form `word(X)` from *Task Environment* (where `X` is a word to be remembered). This rule will be triggered by such messages, and each time it is triggered `X` will be bound or set equal to the word in the actual triggering message. 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

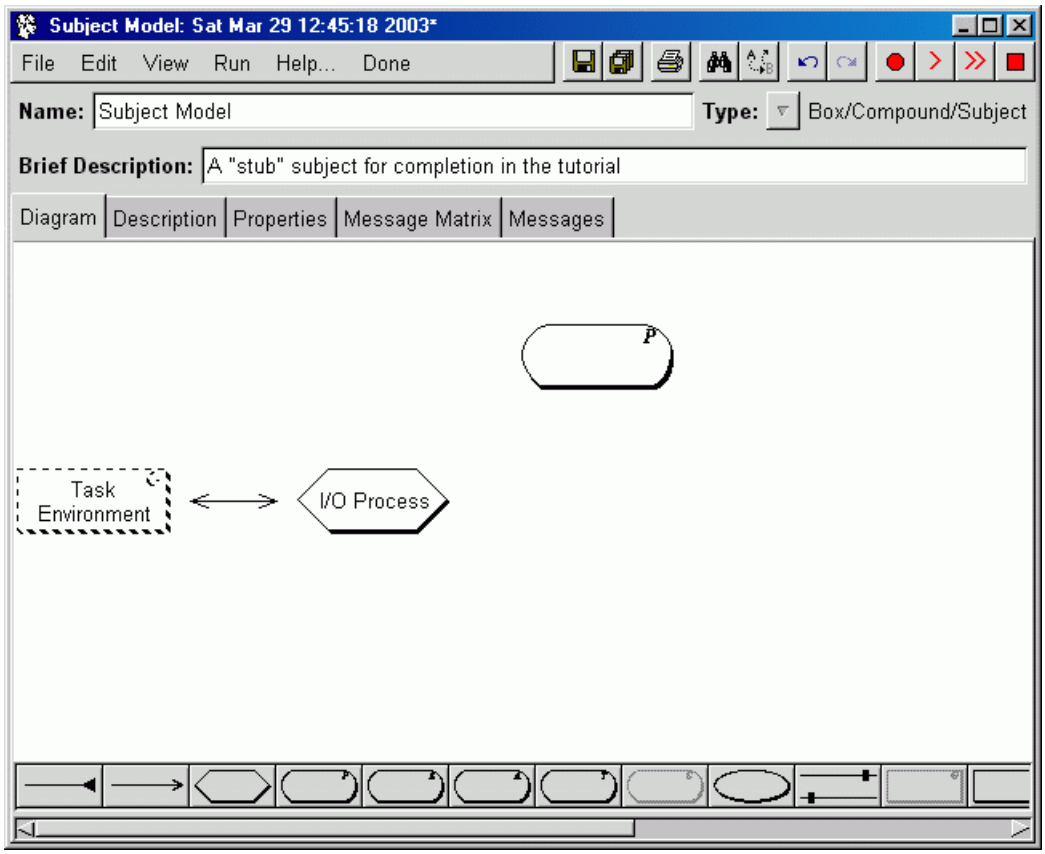

Figure 13: Building *Subject Model* (part 1)

a comment to explain what the rule does — such as Store incoming words in STS. Figure 15 shows how the rule editor window should now look.

As explained above, this rule will be triggered when elements of the form `word(X)` (where `X` is a variable) are received by *I/O Process*. *Task Environment* 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 *STS*. To test it, open *STS* and select the Current Contents tab. 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 for the first couple of cycles because it takes some time for words to be generated and fed to *Subject Model*.)

Now as was mentioned before, in the Modal Model the short-term store is supposed to be capacity-limited with a span of about 7 items. Switch to the Properties view of *STS* — you will see a list of property names and their associated values. Find the Limited Capacity property and ensure it is checked. 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, re-initialise and step through the trial again; you will see that the number of items in *STS* grows until

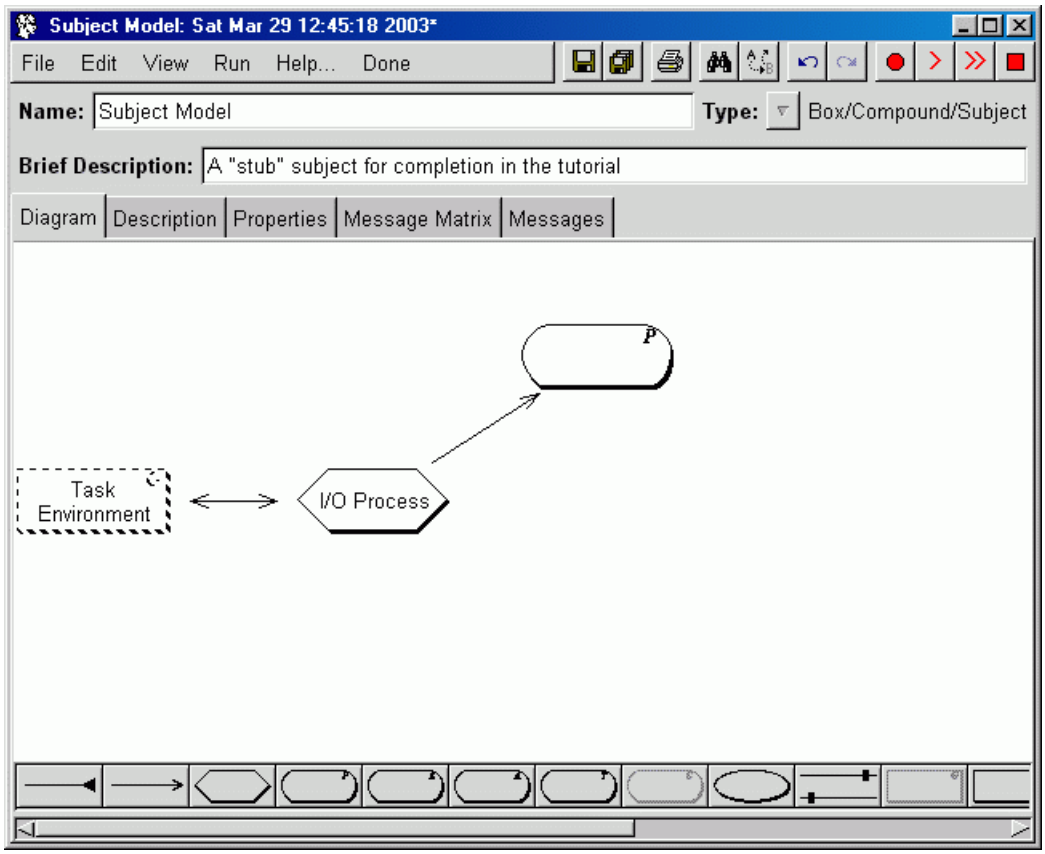

Figure 14: Building *Subject Model* (part 2)

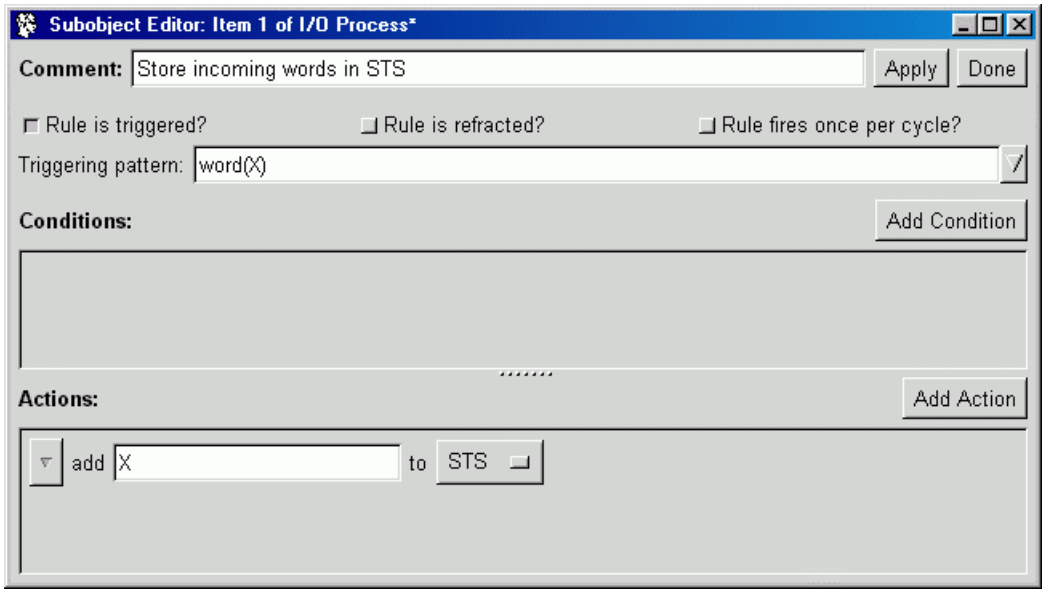

Figure 15: The rule for storing incoming words in *STS*

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 free-recall memory task. Return to the *Subject Model* box-arrow diagram, and add a *Read* arrow from *I/O Process* to *STS*. (See Figure 16.)

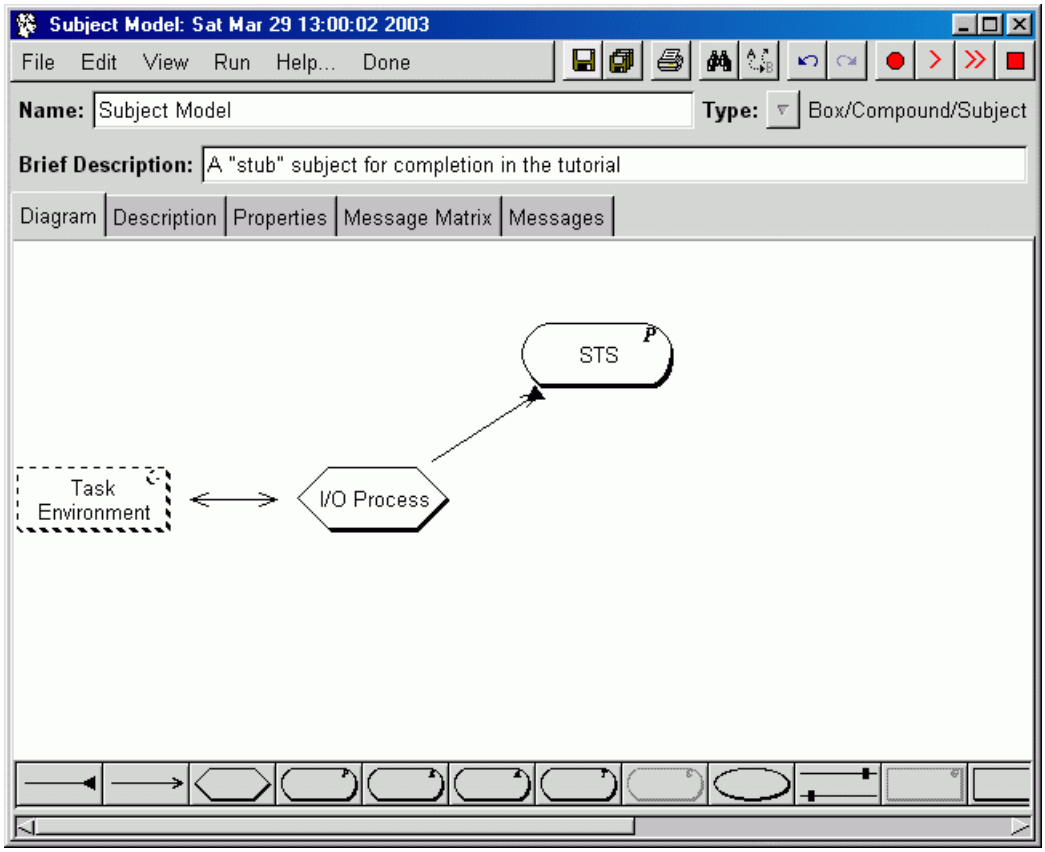

Figure 16: Building *Subject Model* (part 3)

To add a rule to respond to recall requests, create a new rule in *I/O Process* 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 of the rule editor. 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 *Task Environment: Experimenter Process*. Figure 17 shows how the rule should now appear.

Now, when triggered by the `recall` message, the rule should read the words it remembers from memory (currently, just those stored in *STS*), and send them to *Experimenter Process* within *Task Environment*. (COGENT's default behaviour is to fire all possible instantiations of rules whenever they apply. In particular, because we haven't marked the rule to fire once per cycle, it will fire multiple time on the same cycle, sending *all* words that it remembers to *Experimenter Process* on a single

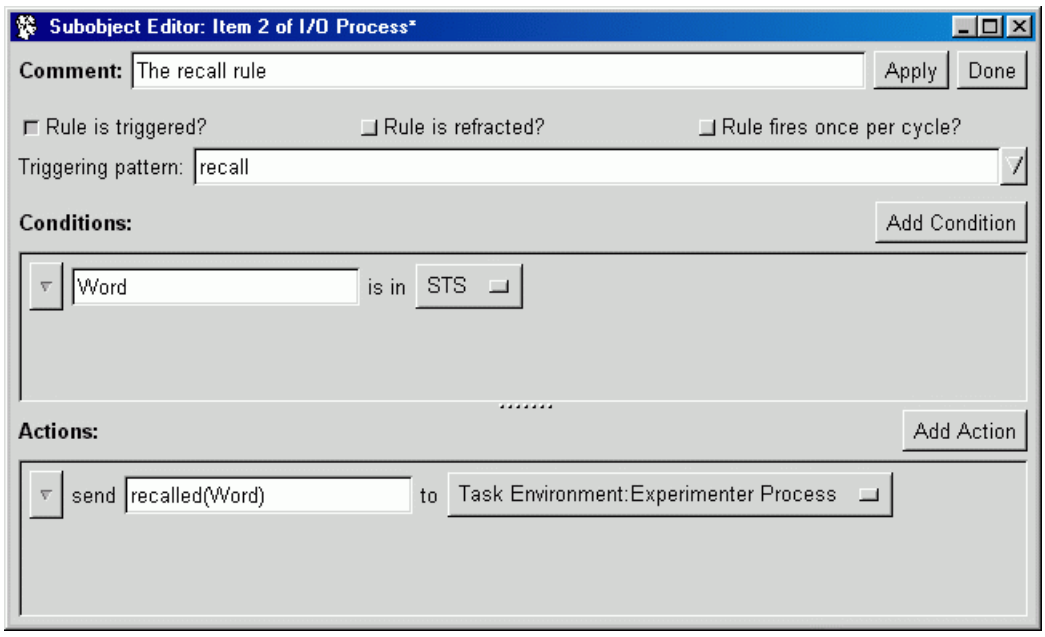

Figure 17: A rule to recall words from *STS*

processing cycle.) Close the rule editor window, and return to the *Serial Position Curve* in *Task Environment*. 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 Figure 18.

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 properties tab of *STS* 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 *I/O Process*? 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?

3.3 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 *STS*, and write to *LTS*. *I/O Process* 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

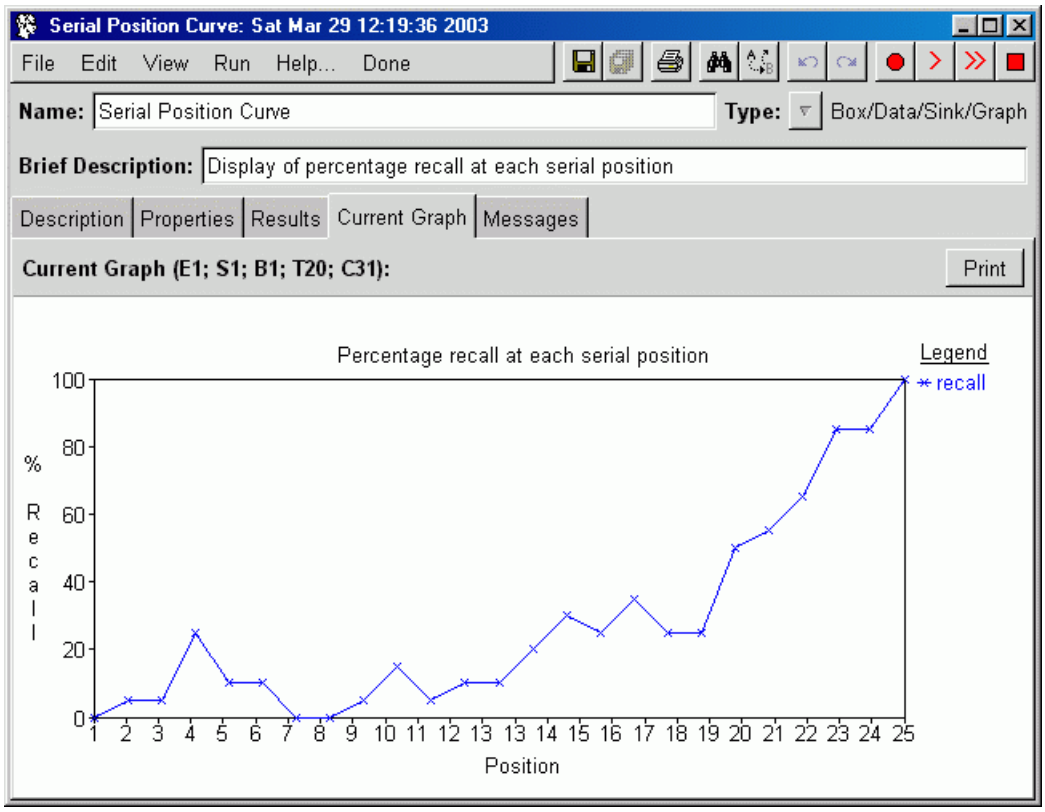

Figure 18: A recency graph generated with recall from STS only

by dragging them around on the box-arrow canvas — you'll find that the arrows follow them around. The diagram should end up looking roughly like Figure 19.

Next we need to make *Rehearsal* 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 *Rehearsal* and create a new rule in the Rules and Condition Definitions 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 to 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 Figure 20.

Now if you switch to the Current Contents tab of *LTS*, initialise and single-step through the model using the Step button, after a few steps you will see words accumulating in *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 this kind of 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 *I/O Process* to

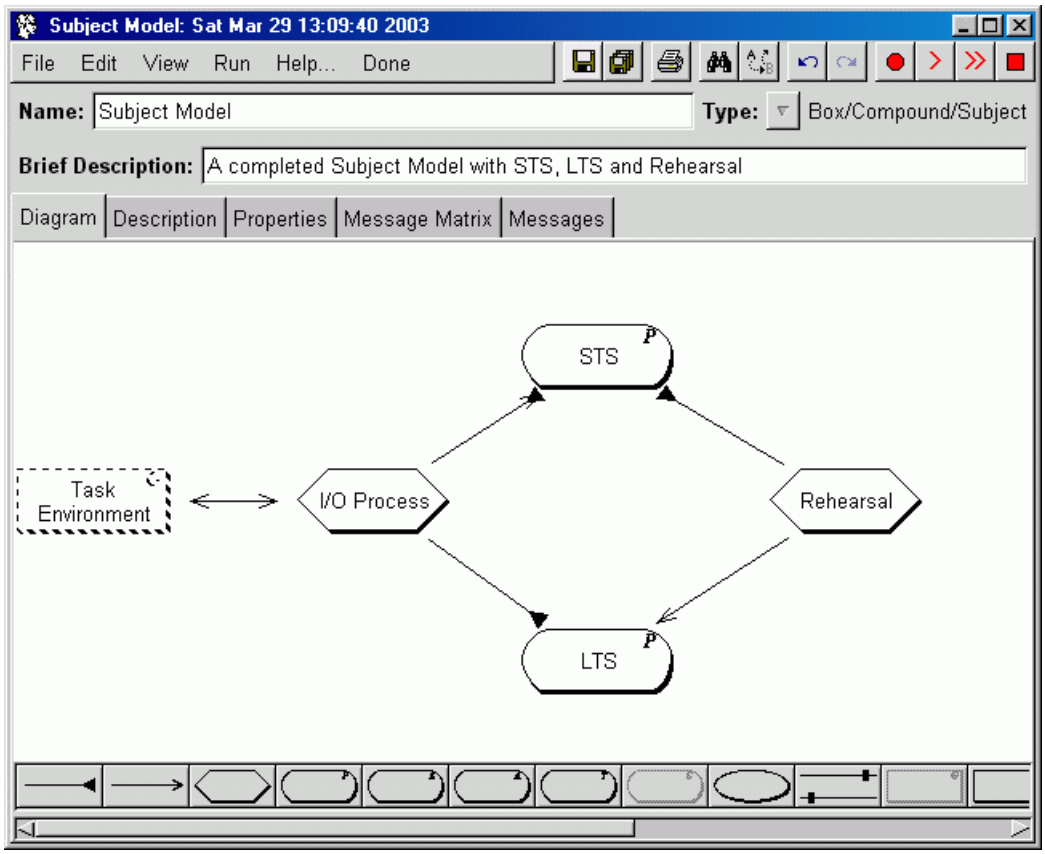

Figure 19: The extended *Subject Model* box-arrow diagram

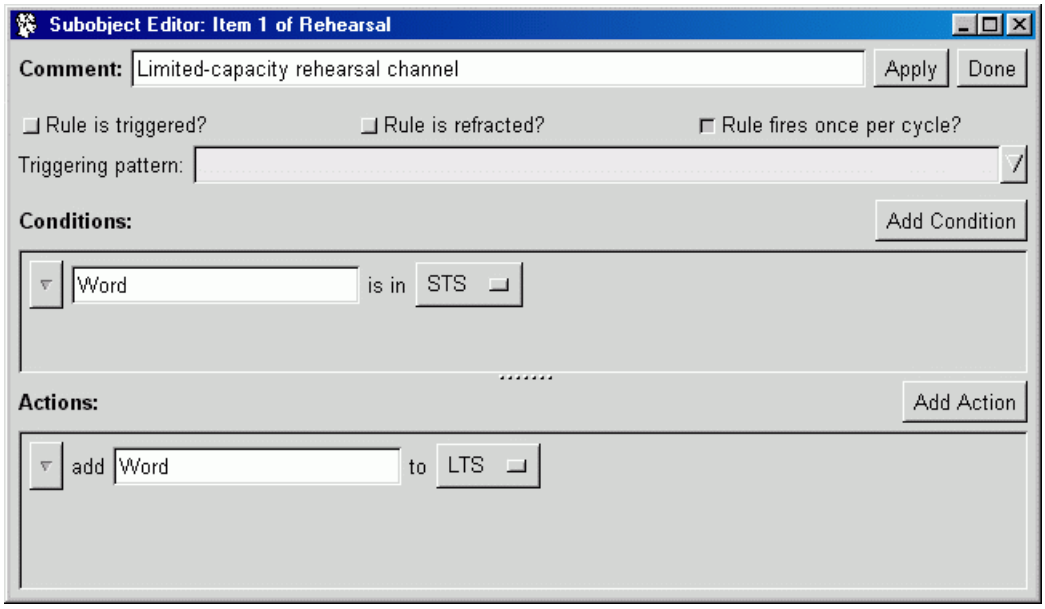

Figure 20: The rehearsal rule

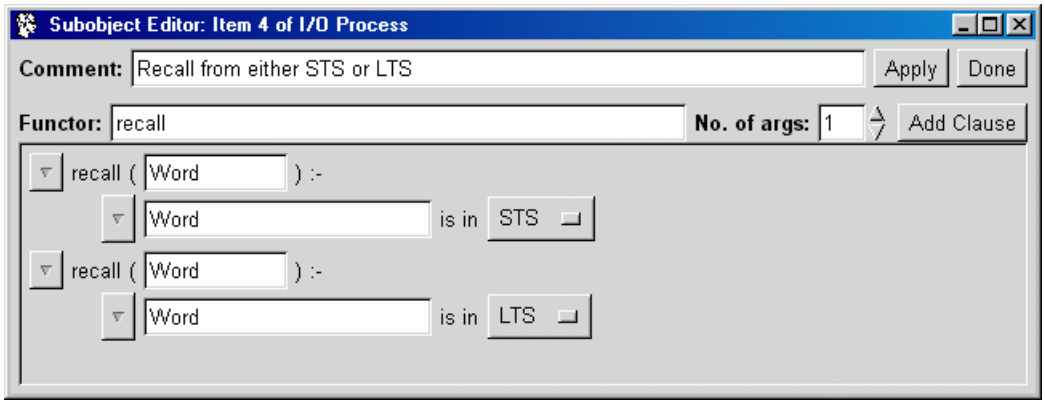

Figure 21: The recall condition

Rehearsal, and add a new rule to *I/O Process*. We will send a special `stop` message via this arrow to *Rehearsal* whenever the recall phase of the trial starts. Create an appropriate triggered rule in *I/O Process*. 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 *I/O Process*, we need to modify the recall procedure to recall from both memory stores. We could add another recall rule (like that in Figure 17, but which matches against *LTS* rather than *STS*). Instead, we're going to demonstrate the use of a *user-defined 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 tool-bar 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 `AddClause` 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 Figure 21.

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 I/O Process 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→user defined 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 Current Contents view of *I/O Process* should look like Figure 22.

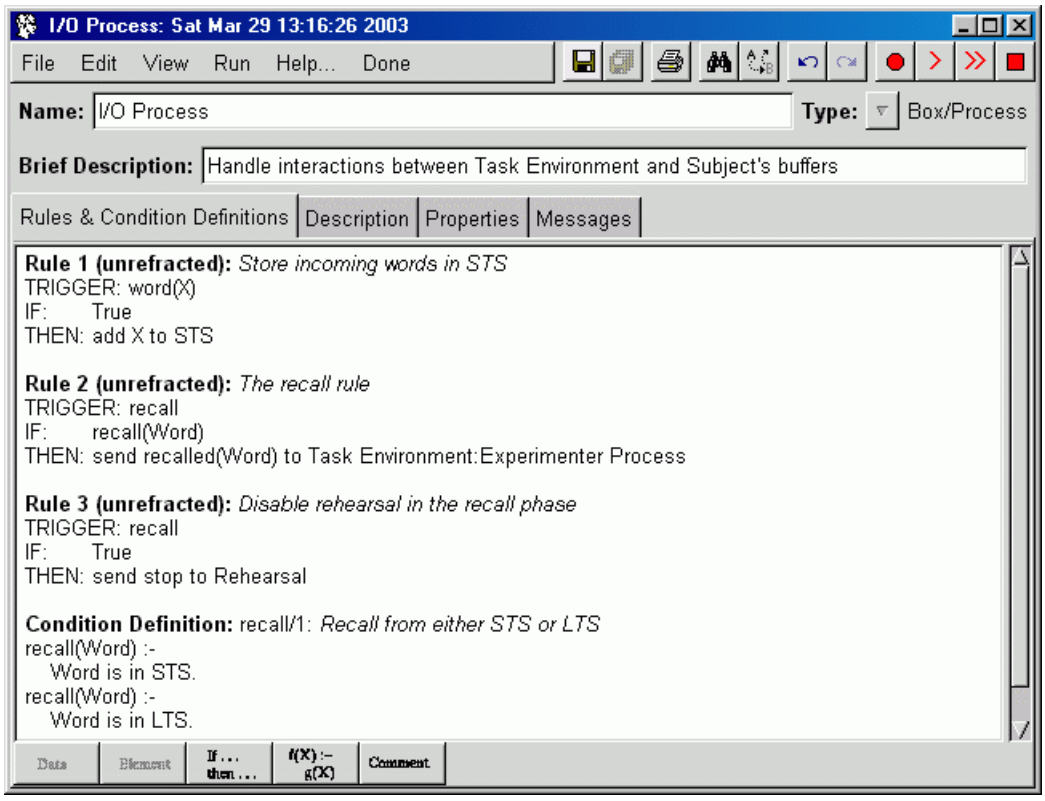

Figure 22: The complete rule set for *I/O Process*

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 Messages view of *I/O Process* you'll see that the model sometimes recalls the same word twice in the same trial. Why is this? Can you fix it so this doesn't happen?
3. The serial position curve still doesn't look like the one in the introduction. Try to characterise any differences. Can you account for them?

3.4 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. We've already seen such properties in action when considering different behaviour of limited capacity buffers. Open *LTS*

and switch to its Properties tab. There are two properties of interest here, the Decay property, which has four possible values — None, Hald Life, Linear and Fixed — and the Decay Constant, which is a number. The Decay Constant is interpreted as a number of cycles. (The cycle is COGENT's basic unit of time.) Half Life, Linear and Fixed decay all 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. In Half Life 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. In Linear decay *D* specifies a the maximum number of cycles an element may remain in the buffer, but the *a prior* probability of the element decaying on any cycle prior to this is *1 / D*.

Set the Decay property of *LTS* to Half Life, 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 *Task Environment* 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 set the Duplicates property of *LTS*. This allows multiple copies of the same word in *LTS*, and should improve recall from *LTS*.

These considerations about time suggest another way the model can be improved. Although *Task Environment* sends words to *Subject Model* serially, *Subject Model* 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 *I/O Process* 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 *I/O Process*. 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.) The final rule should end up looking like Figure 23. The rule works by recalling one word and then triggering itself to recall another. When all words that are in *STS* or *LTS* have been recalled, the rule's condition will fail, and recall will terminate.

This completes the main work of the tutorial. If you have time, consider and/or pursue the following suggestions:

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 *Task Environment* and its components. You'll see it's written using the same language as *Subject Model*. Try to discover how it works.

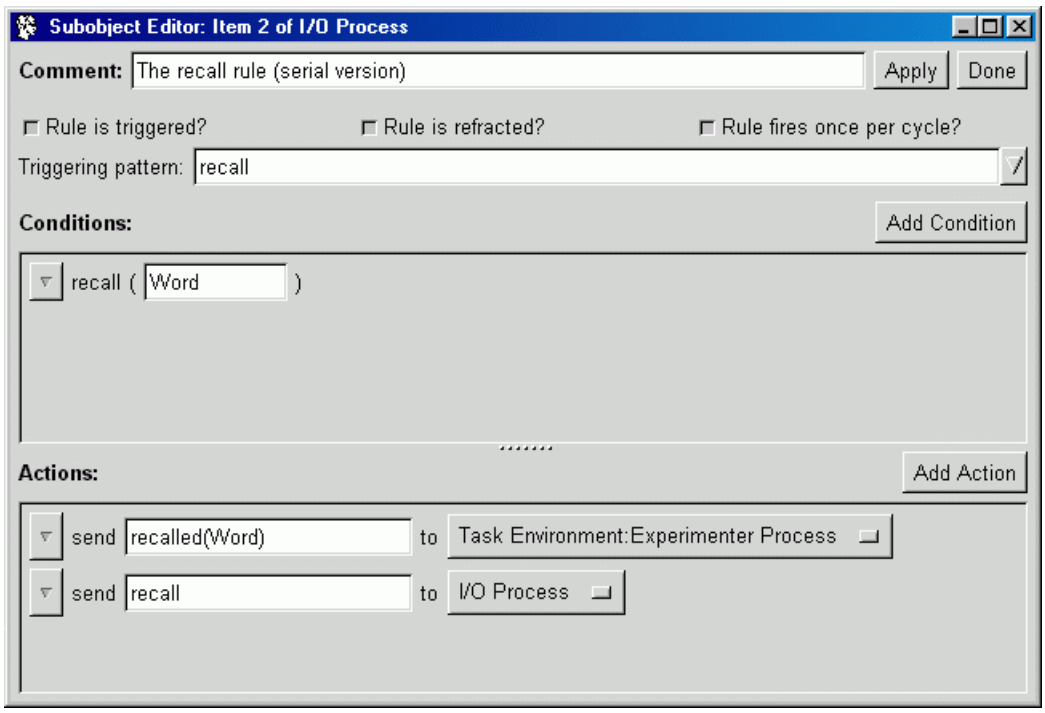

Figure 23: A rule for serial recall

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*, Vol. 2, pp. 89–195. Academic Press, Orlando, FL.

Atkinson, R. C., & Shiffrin, R. M. (1971). The control of short term memory. *Scientific American*, *225*(2), 82–90.

Berendt, B. (1996). Explaining preferred mental models in Allen inferences with a metrical model of imagery. In Cottrell, G. W. (Ed.), *Proceedings of the 18th Annual Conference of the Cognitive Science Society*, pp. 489–494. San Diego, CA. Cognitive Science Society Incorporated.

Bratko, I. (1986). *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Wokingham, UK.

Clocksin, W. F., & Mellish, C. S. (1987). *Programming in Prolog* (Third edition). Springer Verlag, Berlin.

Cooper, R. P. (2002). *Modelling High-Level Cognitive Processes*. Lawrence Erlbaum Associates, Mahwah, NJ. With contributions from P. Yule, J. Fox & D. W. Glasspool.

Cooper, R. P., Yule, P., & Fox, J. (In press). Cue selection and category learning: a systematic comparison of three theories. To appear in *Cognitive Science Quarterly*.

Cooper, R. P., & Fox, J. (1997). Learning to make decisions under uncertainty: The contribution of qualitative reasoning. In Langley, P., & Shafto, M. (Eds.), *Proceedings of the 19th Annual Conference of the Cognitive Science Society*, pp. 125–130. Stanford, CA. Cognitive Science Society Incorporated.

Cooper, R. P., & Fox, J. (1998). COGENT: A visual design environment for cognitive modelling. *Behavior Research Methods, Instruments, & Computers*, *30*(4), 553–564.

Cooper, R. P., & Yule, P. (1999). Comparative modelling of learning in a decision making task. In Hahn, M., & Stoness, S. C. (Eds.), *Proceedings of the 21st Annual Conference of the Cognitive Science Society*, pp. 120–125. Vancouver, Canada. Cognitive Science Society Incorporated.

Cooper, R. P., Yule, P., Fox, J., & Sutton, D. (1998). COGENT: An environment for the development of cognitive models. In Schmid, U., Krems, J. F., & Wysotzki, F. (Eds.), *Mind Modelling: A Cognitive Science Approach to Reasoning, Learning and Discovery*, pp. 55–82. Pabst Science Publishers, Lengerich, Germany.

Fodor, J. A. (1983). *The Modularity of Mind*. MIT Press, Cambridge, MA.

Fox, J., & Cooper, R. P. (1997). Cognitive processing and knowledge representation in decision making under uncertainty. In Scholz, R. W., & Zimmer, A. C. (Eds.), *Qualitative Aspects of Decision Making*, pp. 83–106. Pabst Science Publishers, Lengerich, Germany.

Glanzer, M., & Cunitz, A. R. (1966). Two storage mechanisms in free recall. *Journal of Verbal Learning and Verbal Behavior*, *5*(4), 351–360.

Graham, I. (1994). *Object Oriented Methods*. Addison-Wesley, Wokingham, UK.

Lakatos, I. (1970). Falsification and the methodology of scientific research programmes. In Lakatos, I., & Musgrave, A. (Eds.), *Criticism and the Growth of Knowledge*, pp. 91–196. Cambridge University Press, Cambridge, UK.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, *63*(2), 81–97.

Postman, L., & Phillips, L. W. (1965). Short-term temporal changes in free recall. *Quarterly Journal of Experimental Psychology*, *17*(2), 132–138.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ.

Sterling, L. (1986). *The Art of Prolog*. MIT Press, Cambridge, MA.

Young, R. M., & O'Shea, T. (1981). Errors in children's subtraction. *Cognitive Science*, *5*(2), 153–177.

Yule, P., Cooper, R. P., & Fox, J. (1998). Normative and information processing accounts of medical diagnosis. In Gernsbacher, M. A., & Derry, S. J. (Eds.), *Proceedings of the 20th Annual Conference of the Cognitive Science Society*, pp. 1176–1181. Madison, WI. Cognitive Science Society Incorporated.