An Introduction to the COGENT Modelling Environment (with a focus on models of sentence processing)

Richard Cooper & Peter Yule

23rd Annual Conference of
the Cognitive Science Society,
Edinburgh, UK

August 1st, 2001

http://cogent.psyc.bbk.ac.uk/

Contents

1 Tutorial Aims and Overview

This tutorial aims to introduce attendees to the COGENT modelling environment and provide them with sufficient familiarity to assess whether COGENT may be of use for their own teaching and/or research. This will be achieved by developing a series of progressively more complex models of human sentence processing. COGENT is a general modelling environment that has been applied in a range of domains (including problem solving, reasoning, decision making and memory). The focus on sentence processing in this tutorial is a tribute to the Cognitive Science heritage of Edinburgh. COGENT is not a system designed specifically for models of sentence processing, and previous tutorials have focussed on other domains. Notes from these tutorials may be downloaded from the COGENT web site.

Much of the material included in these notes is extracted from *Modelling High Level Cognitive Processes*, to be published be Lawrence Erlbaum Associates in 2002. The notes consist of five principal sections. Section 2 provides a brief overview of COGENT. It describes and illustrates much of the basic functionality of the environment. Section 3 then presents some background in linguistics and sentence processing. This is provided to ensure that the tutorial notes are self contained. Readers with relevant background can safely ignore this section. Section 4 develops a basic COGENT model of sentence processing. The model builds sentence structure using a bottom-up parallel strategy. In Section 5 the model is modified to use both top-down and bottom-up information. The model is modified further in Section 6, where it is converted to a serial back-tracking model. The resultant model has a modest level omf psychologically plausibility.

This tutorial is deliberately long. It is unlikely that attendees will complete all of the material provided within the session. Rather, it is hoped that attendees will return to the material in the own time after the tutorial, and continue working through the models. A comprehensive reference section is included following the main tutorial notes to encourage this.

2 COGENT: Principal Features

COGENT is a computational modelling environment that provides a flexible system within which information processing models of cognitive processes may be developed and explored. The system 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 memory buffers, rule-based processes, simple connectionist networks, input sources and output sinks;
- Mechanisms for the control of inter-component communication;
- An expressive, extensible, rule-based modelling language and implementation system;
- Automated data visualisation tools, including tables, graphs, and animated diagrams;
- A powerful model testing environment, supporting monte carlo-style simulations and an 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 at both the model and research programme levels.

2.1 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 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

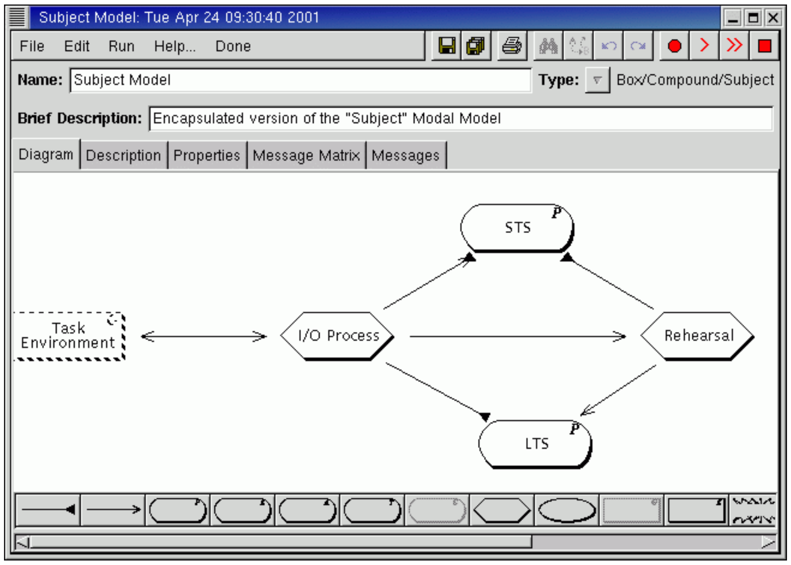

Subject Model: Tue Apr 24 09:30:40 2001
File
Edit
Run
Help...
Done
Name: Subject Model
Type: Box/Compound/Subject
Brief Description: Encapsulated version of the "Subject" Modal Model
Diagram
Description
Properties
Message Matrix
Messages
STS
Task Environment
I/O Process
Rehearsal
LTS

Figure 1: A box and arrow diagram of the Modal Model of memory

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 (cf. Rumbaugh, Blaha, Premerlani, Eddy, & Lorensen, 1991).

Models are specified in COGENT by sketching their functional components using the 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 it 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.

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 arrows indicate different types of communication between the components, such as reading information from a buffer or sending messages to a process. The 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.

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, 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 top-most box.

2.2 Standard Component Types

COGENT provides a library of standard configurable components. Models are constructed by assembling these components (in the form of 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 user-specified symbolic rules. 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 COGENT objects.

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 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 component). Different subtypes of buffer may be used to store information in different formats (e.g., propositional, tabular, and analogue).

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 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 COGENT by arrows drawn between components within a COGENT box and arrow model. Two basic types of arrow are provided: read arrows and write/send arrows.

Further details of these component types are given in the appendix (see especially Section C).

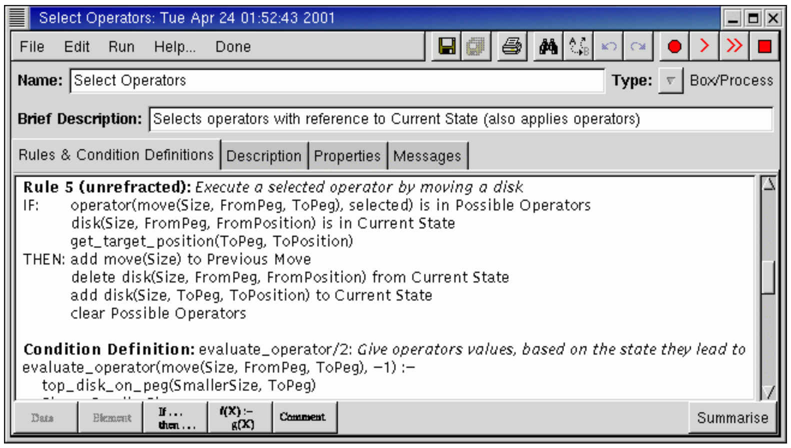
2.3 The Rule-Based Modelling Language

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 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.

An example rule is shown in Figure 2. 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. On firing, the rule deletes one element from *Possible Operators* (`operator(Move, possible)`) and adds another (`operator(Move, value(Value))`).

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*

Figure 2: A simple rule that updates a buffer



Name: Select Operators
Type: Box/Process
Brief Description: Selects operators with reference to Current State (also applies operators)

```
Rule 5 (unrefracted): Execute a selected operator by moving a disk
IF:     operator(move(Size, FromPeg, ToPeg), selected) is in Possible Operators
        disk(Size, FromPeg, FromPosition) is in Current State
        get_target_position(ToPeg, ToPosition)
THEN: add move(Size) to Previous Move
        delete disk(Size, FromPeg, FromPosition) from Current State
        add disk(Size, ToPeg, ToPosition) to Current State
        clear Possible Operators

Condition Definition: evaluate_operator/2: Give operators values, based on the state they lead to
evaluate_operator(move(Size, FromPeg, ToPeg), −1) :−
   top_disk_on_peg(SmallerSize, ToPeg)
```

Figure 3: Some rules from the *Select Operator* process within a problem solving model

COGENT's rule language is both flexible and powerful. The flexibility and power introduce some complexities, but special tools simplify the process of specifying rules and to allow monitoring of the processing of rules during model execution.

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 Figure 2: `evaluate_operator(Move, Value)` is a call to a user-defined condition, the definition of which is specified elsewhere in the process. The condition definition language is based on Prolog (cf. 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 special purpose editing facilities. Figure 3 shows this view for *Select Operators*, a process from a model of problem solving.

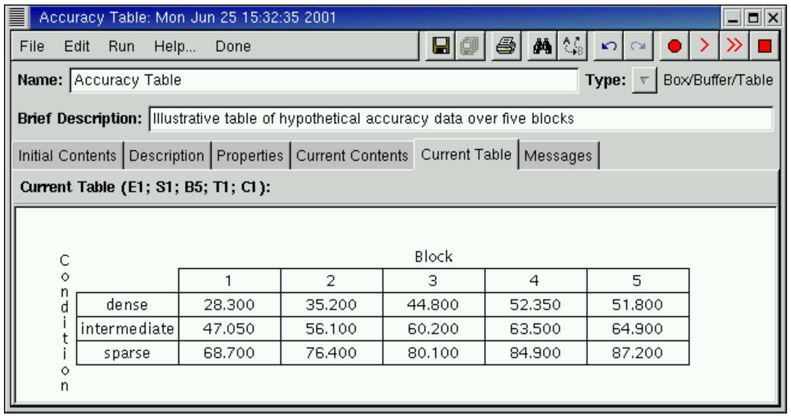
2.4 Automated Data Visualisation Tools

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 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 for any particular table (e.g., row height, column width, row and column labels, etc.) being governed by that table's properties.

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) of spatial features of the elements (e.g., left/right, right/left).

Graphs Data may be displayed graphically in several standard formats, including line graphs, scatter plots and bar charts. Figure 5 shows a line graph of data generated by a model of free recall. 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



Accuracy Table: Mon Jun 25 15:32:35 2001

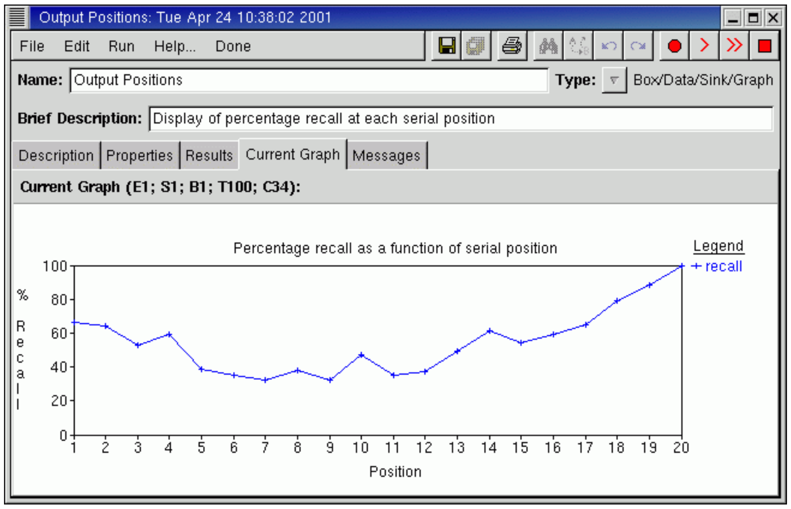
Name: Accuracy Table **Type:** Box/Buffer/Table

Brief Description: Illustrative table of hypothetical accuracy data over five blocks

Current Table (E1; S1; B5; T1; C1):

Condition	Block 1	Block 2	Block 3	Block 4	Block 5
dense	28.300	35.200	44.800	52.350	51.800
intermediate	47.050	56.100	60.200	63.500	64.900
sparse	68.700	76.400	80.100	84.900	87.200

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



Output Positions: Tue Apr 24 10:38:02 2001

Name: Output Positions **Type:** Box/Data/Sink/Graph

Brief Description: Display of percentage recall at each serial position

Current Graph (E1; S1; B1; T100; C34):

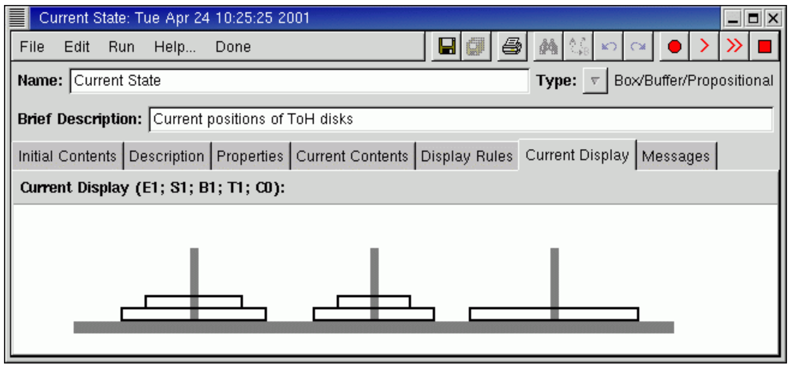
Figure 5: A graphical buffer, showing a line graph summarising output from a model of memory applied to a free recall task

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 6 shows a visualisation of the contents of a propositional buffer whose contents represent disks in the Tower of Hanoi problem. Displays associated with propositional and analogue buffers are dynamically updated whenever the contents of the buffer change.

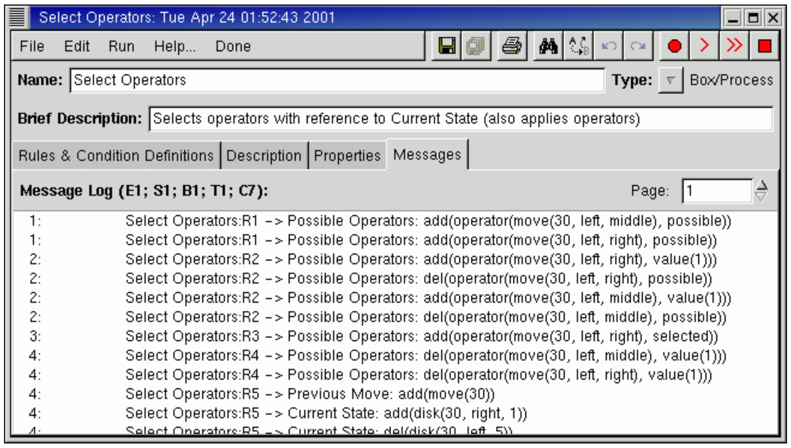
2.5 The Model Testing Environment

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 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



Name: Current State
Type: Box/Buffer/Propositional
Brief Description: Current positions of ToH disks
Current Display (E1; S1; B1; T1; C0):

Figure 6: A visualisation of a propositional buffer's contents, showing an intermediate state of the Tower of Hanoi problem


Select Operators: Tue Apr 24 01:52:43 2001
File
Edit
Run
Help...
Done
Rules & Condition Definitions
Description
Properties
Messages

Name: Select Operators
Type: Box/Process
Brief Description: Selects operators with reference to Current State (also applies operators)
Message Log (E1; S1; B1; T1; C7):
Page: 1

```
1:    Select Operators:R1 -> Possible Operators: add(operator(move(30, left, middle), possible))
1:    Select Operators:R1 -> Possible Operators: add(operator(move(30, left, right), possible))
2:    Select Operators:R2 -> Possible Operators: add(operator(move(30, left, right), value(1)))
2:    Select Operators:R2 -> Possible Operators: del(operator(move(30, left, right), possible))
2:    Select Operators:R2 -> Possible Operators: add(operator(move(30, left, middle), value(1)))
2:    Select Operators:R2 -> Possible Operators: del(operator(move(30, left, middle), possible))
3:    Select Operators:R3 -> Possible Operators: add(operator(move(30, left, right), selected))
4:    Select Operators:R4 -> Possible Operators: del(operator(move(30, left, middle), value(1)))
4:    Select Operators:R4 -> Possible Operators: del(operator(move(30, left, right), value(1)))
4:    Select Operators:R5 -> Previous Move: add(move(30))
4:    Select Operators:R5 -> Current State: add(disk(30, right, 1))
4:    Select Operators:R5 -> Current State: del(disk(30, left, 5))
```

Figure 7: The Messages view of *Execute Process*

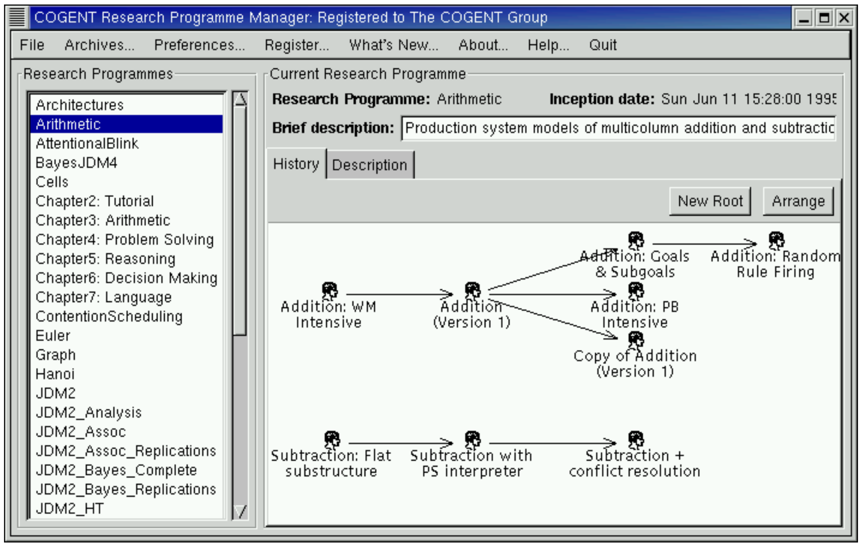
conditions, with each experimental condition involving a number of blocks and each block involving a number of trials. Such designs are constructed through a special purpose experiment script editor.

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 by or received by a component. Thus, Figure 7 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 elements within rules to be traced.

2.6 Research Programme Management

The development of a cognitive model typically takes place over an extended period of time. COGENT supports this development through special tools for managing sets of models within a *research programme* (cf. Lakatos, 1970), including: 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 8. 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



Research Programme: Arithmetic **Inception date:** Sun Jun 11 15:28:00 1995

Brief description: Production system models of multicolumn addition and subtractic

Figure 8: The research programme history view

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.

The research programme manager includes facilities for creating and unpacking archive files, allowing whole research programmes to be stored and transferred in a convenient, single file. You may wish to use this facility if you want to keep the models you develop in this tutorial.

3 Sentence Processing: Background

3.1 Major Topics in the Psychology of Language

The ability to acquire and use language is frequently quoted as one of the central characteristics of our species. Language is also frequently quoted as central to a number of higher cognitive faculties, such as reasoning and problem solving. For this reason the computational processes that support language and its use are of special interest within Cognitive Science. The domain of language is also of special historical significance, for it is generally agreed that Chomsky's arguments concerning the inadequacies of behaviourist accounts of language (Chomsky, 1959) played a central role in shifting psychology's emphasis from behaviourist to cognitivist accounts of mental processes.

The study of the psychology of language is typically divided into a number of sub-disciplines, including phonology, syntax, semantics and pragmatics. Phonology concerns the individual sounds (phonemes) that make up speech. There is a substantial body of empirical research on, for example, factors that influence phoneme perception. Syntax concerns the sequential order of words and affixes in well-structured language. There are complex constraints or rules that govern meaningful orderings of sentence constituents, and these constraints have been studied from both linguistic and psychological perspectives. Semantics relates to the literal meaning of language. This has been studied both at the word level (in which case it is closely related to the study of concepts) and at the level of phrases or sentences. Lastly, pragmatics concerns the use of language in context. This involves the study of the intended (as distinct from literal) meaning of language, and can be distinguished from semantics by considering utterances such as "Can you wipe your feet?" (said to someone with muddy boots about to walk on a clean floor) which, semantically is a question having a yes/no answer, but pragmatically is a request for the person to wipe his/her feet.

A second dimension of language that cuts across these four sub-disciplines relates to the distinction

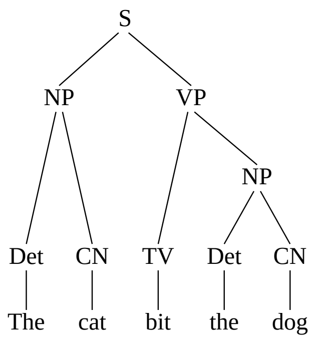


Figure 9: The syntactic structure of *The cat bit the dog*

between generation and reception. There is no reason in principle why the processes involved in, for example, production of well-formed sentences should be the same as those involved in comprehension of the same sentences. Indeed, there is neuropsychological evidence from the breakdown of language following brain injury that demonstrates that language production may be compromised without affecting language comprehension (Coltheart, Sartori, & Job, 1987).

Clearly the cognitive psychology of language is a vast subject. Within the above context, this tutorial will focus on the reception of syntax. A number of approaches to the parsing problem — the problem of testing well-formedness of a sequence of words and assigning an underlying phrasal structure to the sequence — will be developed, culminating in a sophisticated model informed by some well-attested empirical regularities.

3.2 Syntactic Structure

Consider a simple sentence, such as *the cat bit the dog*. Like many sentences of English, this sentence can be analysed as consisting of a noun phrase (*the cat*) followed by a verb phrase (*bit the dog*). This simple fact of the structure of English sentences may be expressed by a *phrase structure* rule:

S → NP VP

where S represents sentence, NP represents noun phrase, and VP represents verb phrase. The rule states that a sentence may be composed of a noun phrase followed by a verb phrase.

The two principal constituents of the above sentence may also be decomposed. The noun phrase consists of a determiner (i.e., a word such as *the, a, every, some*, etc.) followed by a common noun. The verb phrase consists of a transitive verb i.e., a verb that relates two things) and a second noun phrase. These facts may also be expressed as phrase structure rules:

NP → Det CN
VP → TV NP

In order to analyse the sentence completely, it is also necessary to know the syntactic categories of the words involved — that *the* is a determiner, *cat* and *dog* are common nouns, and *bit* is a transitive verb.

It is common to represent the structure of sentences pictorially using a tree. Figure 9 shows the tree representation for *the cat bit the dog*. The nodes in the tree directly corresponding to words are referred to as terminal nodes. The other, higher, nodes are referred to as non-terminal nodes. The existence of such nodes is inferred from regularities across sentence types. Thus, even the previous three sentences in this paragraph can be analysed in terms of the simple S → NP VP rule, although the NPs and VPs involved are relatively complex. (In the first case, the NP is *the nodes in the tree directly corresponding to words*, in the second case the NP is *the other, higher, nodes*, and in the third case the NP is *the existence of such nodes*.)

Many more phrase structure rules are obviously required to describe English, or any other naturally occuring human language. In the case of English, we might start with:

NP → Pro
NP → PN
NP → NP PP

PP $\rightarrow$ Prep NP

These rules state that a noun phrase may consist of a pronoun (e.g., *him*, *her*, *you*, *me*), a proper name (e.g., *John*, *Mary*, *London*) or a noun phrase followed by a prepositional phrase (e.g., *the cat on the mat*), where a prepositional phrase consists of a preposition (e.g., *in*, *on*, *at*) followed by a noun phrase.

Similarly, there are additional types of verb phrase, including:

VP $\rightarrow$ IV
VP $\rightarrow$ DV NP NP
VP $\rightarrow$ DV NP PP(to)
VP $\rightarrow$ VP Adv
VP $\rightarrow$ VP PP

These rules state that a verb phrase may consist of a single intransitive verb (e.g., *runs*, *sleeps*), a ditransitive verb followed by two noun phrases (e.g., *gave Mary the book*), a ditransitive verb followed by a noun phrase and prepositional phrase in the "to" form (e.g., *gave the book to Mary*), a verb phrase followed by an adverb (e.g., *runs quickly*), or a verb phrase followed by a prepositional phrase (e.g., *slept in the park*).

Three more types of verb phrase are common, and will be used in examples later in the tutorial:

VP $\rightarrow$ V_{inf1} VP(inf)
VP $\rightarrow$ V_{inf2} NP VP(inf)
VP(inf) $\rightarrow$ INF VP
VP $\rightarrow$ V_{comp} S(comp)
S(comp) $\rightarrow$ Comp S

These rules cover verb phrases such as *wants to wash the dishes*, *promised Mary to wash the dishes* and *thought that Mary liked him*. The first states that a verb phrase may consist of an infinitival-complement verb (such as *wants*) followed by a verb phrase in the infinitive form (e.g., *to wash the dishes*). The second states that a verb phrase may consist of a second type of infinitival-complement verb (such as *asked*, *promised*, *pretended*) followed by a noun phrase (e.g., *Mary*) and a verb phrase in the infinitive form (e.g., *to wash the dishes*). The third states that a verb phrase in the infinitive form consists of an infinitive (i.e., *to*) followed by a standard verb phrase. The fourth states that a verb phrase may consist of a that-complement verb (e.g., *thought*, *believed*) followed by a sentence in the comp form, and the fifth states that a sentence in the comp form consists of a complementiser (e.g., *that*) followed by a sentence.

This analysis begins to show the complexity of the structure of English, yet it hardly scratches the surface: it does not consider, for example, number agreement between sentence components (e.g., *The cats chases the dog* is ungrammatical), conjunction (e.g., *John took the bus but Mary cycled*), tense and aspect (e.g., what is the structure of *John will be taking the bus*?), so-called movement (as in *Which bus did John take?*) or "it-cleft" sentences such as *It was John who stole the bus*. Space does not permit a proper treatment of such issues, and the interested reader is directed to standard texts on linguistic structure (e.g., Burton-Roberts, 1986).

Exercise 1: Using the above phrase structure rules, draw trees showing the syntactic structure of the following sentences:

(1) a. John gave Mary a kitten
b. John ate the cake quickly
c. John asked Mary to wash the car
d. Mary persuaded John to wash the car
e. John saw the kitten in the park

Notice how the last sentence is ambiguous. John may have seen the kitten that was in the park, or John may have been in the park when he saw the kitten. You should be able to draw two distinct trees, corresponding to the two distinct meanings, for this sentence. ◇

3.3 Parsing and Syntactic Structure

As English speakers we are able to detect when a putative sentence does or does not conform to the rules of English. While testing grammaticality is not normally something that we are aware of, the fact that it appears to be done by our language processing system imposes certain computational requirements on that system.

The process of determining the syntactic structure of a sentence or sentence fragment is referred to as parsing. Much early work in the cognitive science of language assumed that the linguistic capabilities of humans were embodied within a system (sometimes referred to as the Human Sentence Processing Mechanism: HSPM) that took as input sequences of words and generated from those words a meaning. Many researchers further assumed that a key stage (possibly the first stage) of processing within the HSPM was parsing of the input word sequence.

This view of the processing of language, and the assumption that language is rule-governed (i.e., syntactic structure is determined by a set of phrase structure rules, such as those above), was reinforced by the fact that there are many example word sequences that are, according to most phrase structure formulations of English, well-formed syntactically, but (to the untrained, at least) not obviously grammatical. One often cited case is centre-embedding. Consider Example 2a below. Simplifying somewhat, it is a noun phrase constructed from a rule that might be represented as: NP $\rightarrow$ Det CN COMP NP TV. If we apply this rule twice, we find that Example 2b is also a noun phrase. Adding a verb phrase (e.g., consisting of the intransitive verb *squealed*) to Example 2a yields Example 2c, which to most English speakers is an acceptable sentence. Doing the same to Example 2b yields Example 2d, which to most English speakers is not acceptable.

(2) a. the mouse that the cat chased
b. the mouse that the cat that the dog bit chased
c. the mouse that the cat chased squealed
d. the mouse that the cat that the dog bit chased squealed

Cases such as these led Chomsky (1965) to distinguish between linguistic competence and linguistic performance. Competence in any domain is the idealised knowledge upon which processing in that domain is based. Performance is concerned with the results of using that knowledge. Use of knowledge may be constrained, for example, by memory limitations, and so competence may exceed performance.

In many ways the competence/performance distinction is intuitively plausible. Many cognitive processes appear to be resource-bound (by, for example, limited short-term memory). If language is rule-based and any part of linguistic processing is resource-bound, then there are likely to be sentences that are grammatically well-formed but not parsable by the HSPM.

However, the competence/performance distinction also raises problems concerning the precise limits of grammaticality: should the unacceptability of a putative question such as *Who did Mary give the book that John borrowed from to Bill?*, where the *who* refers to the person John borrowed from, be attributed to a violation of the grammar rules or a performance violation? For a given theory of syntactic structure, the unacceptability of a sentence that is grammatical according to the theory but unacceptable according to human participants may be attributed to performance factors. Hence, a grammatical theory cannot be falsified through over-prediction (i.e., through predicting that unacceptable sentences are grammatical). In this way, the competence/performance distinction undermines the status of grammatical theory.

The above is only true, however, if the grammatical theory is stated in the absence of a performance theory. Performance, which should be understood at Marr's level two, is the product of a sub-optimal process operating with correct and complete information (e.g., phrase structure rules). A well-specified theory of linguistic performance, tied to a theory of linguistic competence, should account for all and only acceptable sentences of the language under consideration. This view led a number of researchers in the 1970s and 1980s (e.g., Kimball, 1973; Frazier & Fodor, 1978; Wanner, 1980) to focus on principles of the parsing process that might give rise to (and hence explain) data from human linguistic performance.

Several types of performance data are relevant to this enterprise. First and foremost are grammaticality (or acceptability) judgements, where participants or "native speakers" agree that word sequences are acceptable or unacceptable fragments of their language. A second type of performance data relates to sentences that are agreed to be ambiguous. In such cases the competence theory will typically allow multiple syntactic structures for a single word sequence (recall Example 1e). Ambiguous sentences often have preferred readings, where native speakers show a strong preference for one reading over other readings. This constitutes another type of performance data: in such cases a performance theory should predict the preferred reading.

A further type of performance data relates to sentences that are acceptable only after a second reading. Sentences such as those in example 3 illusrate this phenomenon.

(3) a. The horse raced parsed the barn fell
    b. Since Jay jogs a mile seems a small distance to him
    c. Teachers taught by the Berlitz method passed the test
    d. The officer told the criminal that he was arresting her husband

These sentences are known as garden path sentences (cf. Bever, 1970; Frazier & Rayner, 1982; Crain & Steedman, 1985). They arise when multiple partial syntactic structures are possible at some point in the left-right sequence of processing a sentence (i.e., the sentence is locally ambiguous), when one of those partial syntactic structures is preferred, and when the actual syntactic structure corresponding to the complete sentence is not based on the preferred one. The reality of garden path effects has been demonstrated through sophisticated psycholinguistic experiments (see, e.g., Frazier & Rayner, 1982; Crain & Steedman, 1985; Altmann & Steedman, 1988). Performance theories must therefore account for why the HSPM initially fails in attempting to parse such sentences.

3.4 Some Dimensions of Parsing

A number of key issues have arisen in computational approaches to natural language parsing. Wanner phrases some of these issues as "dimensions of the parsing problem" (Wanner, 1988, p. 80). One dimension concerns the degree to which parsing is *incremental*. Incremental parsers incorporate each successive word into some structure or sentence frame as each word is encountered. Non-incremental parsers operate on complete sentences. A second dimension concerns the direction in which the syntactic representation is constructed — from terminal word nodes to a non-terminal sentence node, or *vice versa*. *Bottom-up* parsers group words in the input into phrases, and then those phrases into larger phrases, and so on. *Top-down* parsers hypothesise that the word sequence is a sentence, and that it consists of a noun phrase followed by a verb phrase, and so on, decomposing each category until a match with the sentence being parsed is achieved. There is clear evidence that both top-down and bottom-up processes are involved at different stages in human sentence processing.

Wanner's third dimension concerns the degree of parallelism involved in the parsing process. Serial parsers build and maintain one syntactic structure at a time, even when multiple structures are possible (as in sentences containing local ambiguities). If that structure proves to be incorrect, the serial parser must attempt to reanalyse the sentence using a different possible syntactic structure. Parallel parsers, in contrast, maintain multiple possible syntactic structures at points of local ambiguity. Consequently parallel parsers do not need to reanalyse a sentence if one structure proves incorrect. Garden path sentences suggest that human parsing may be serial, but it may still be the case that some aspects of human parsing involve the simultaneous maintenance of multiple syntactic structures in some short-term linguistic store.

A fourth dimension of parsing (not discussed by Wanner (1988)) concerns the role of semantics or meaning and context on parsing. Most parsers rely purely on syntactic information when constructing the syntactic structure associated with a sentence, but Crain and Steedman (1985) point out that this need not be the case. The human parser may be influenced by semantics or context during the parsing process, and this influence may lead to different syntactic structures being favoured in different semantic contexts. Evidence for such an interaction between syntax and linguistic context is presented by Crain and Steedman (1985) (see also Altmann & Steedman, 1988). Tanenhaus, Spivey-Knowlton, Eberhard, and Sedivy (1995) have further shown, with eye movement studies, that sentence processing can also be influenced by visual, non-linguistic, context.

These dimensions provide a context within which specific parsing models may be compared. Two dimensions — the bottom-up/top-down dimension and the parallel/serial dimension — are illustrated in depth by the models developed later in this tutorial.

4 Sentence Processing: A First Model

4.1 An Incremental Input Module

Human sentence processing is generally considered to involve the presentation of linguistic input to the HSPM one word or phrase at a time. This presentation may be modelled within COGENT by an experimenter module that, on each trial, selects an input word sequence from a set of stimuli and then feeds that sequence, one word at a time, to a subject module (which, it is assumed, attempts to parse the input). The experimenter module may, on successive trials, feed successive word sequences to the subject module, allowing the subject module to be tested on a variety of inputs.

A minimal experimenter module that performs the above task consists of: one propositional buffer, *Stimuli*, that initially contains the full set of stimuli (word lists) and that is reinitialised only at the start of a subject; one process, *Present Stimuli*, that presents one stimulus from *Stimuli* on each trial, deleting the stimulus as it is presented; and another propositional buffer, *Current Stimulus*, that is used for temporary storage of the current stimulus during the trial on which it is being presented. Within the minimal experimenter module, *Present Stimuli* must be able to read from and write to both propositional buffers. It should also send to an input/output process located within the subject module.

Exercise 2: Create a new research programme and a new model within that research programme. Draw a box and arrow diagram within the new model consisting of two compound boxes, *Experimenter* and *Subject*, corresponding to an experimenter module and a subject module. Populate the two modules with boxes as described in the previous paragraph and link the boxes with appropriate arrows. ◇

Experimenter requires two rules: one to select a test sentence for the current trial and transfer it from *Stimuli* to *Current Stimulus*, and one to drip-feed words from *Current Stimulus* to the subject module. The two rules in Figure 10 perform these functions. Rule 1 assumes that *Stimuli* contains elements that are lists of words (the putative sentences which are to be parsed on successive trials). Rule 2 is triggered by `system_quiescent`. This ensures that it only fires when all other processing within the system is complete. This means that the subject module may perform arbitrary processing on receipt of each word, and only when a word has been fully processed will the next word be fed to the subject. This will allow maximal exploration of incremental left-to-right processing within the parsers described later in the tutorial.

Rule 1 (refracted): *Select a sentence to parse from the Stimuli buffer*
IF: the current cycle is `1`
  once `WordList` is in *Stimuli*
THEN: delete `WordList` from *Stimuli*
  add `words(WordList)` to *Current Stimulus*

Rule 2 (unrefracted): *When quiescent, feed one more word to the subject*
TRIGGER: `system_quiescent`
IF: `words([Head|Tail])` is in *Current Stimulus*
THEN: delete `words([Head|Tail])` from *Current Stimulus*
  add `words(Tail)` to *Current Stimulus*
  send `word(Head)` to *Subject:Input/Output*

Figure 10: Incremental input rules from *Present Stimuli*

Exercise 3: Add the rules from Figure 10 to *Present Stimuli*. Also add a range of putative sentences to *Stimuli*. Each sentence should be expressed as a single buffer element consisting of a list of words. Ensure that the initialise properties of both buffers are set as described above, and test *Experimenter* by monitoring the messages sent to *Input/Output* when the model is run. ◇

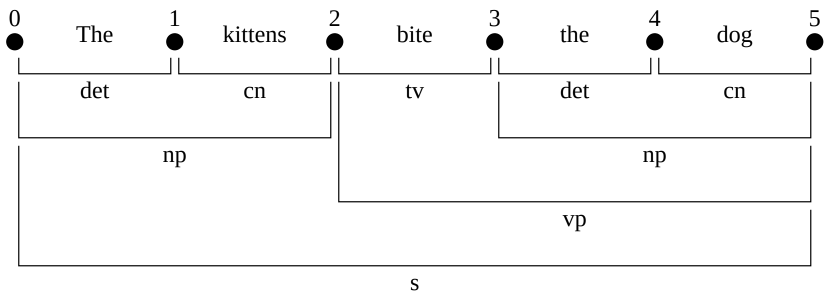


Figure 11: A completed chart for *the kittens bite the dog*

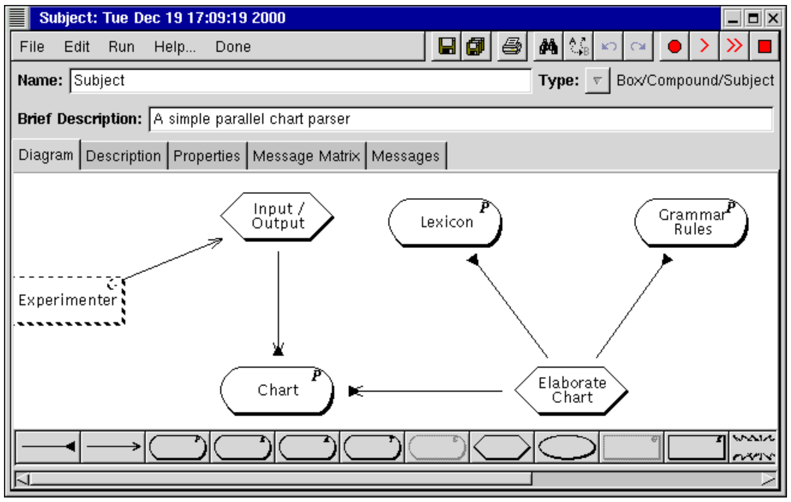
4.2 A Bottom-Up Parallel Parser

As noted above, there are several approaches to the problem of verifying that a sequence of words is a well-formed sentence, and, if so, deriving the sentence's syntactic structure. One simple approach involves constructing a grid or chart as in Figure 11. The numbered disks are referred to as vertices or nodes, and the lines spanning the vertices are referred to as edges or arcs. The goal of chart parsing is to construct an edge that spans the entire sequence of vertices. If the word sequence is intended to be a sentence, then the edge should be labelled "s" (as in Figure 11).

Charts may be constructed in a variety of ways, corresponding to the dimensions of parsing outlined in Section 3.4. Thus, the chart may be constructed bottom-up or top-down, through exploring possible edges in sequence or in parallel, etc. In all cases, the general method is referred to as chart parsing.

4.2.1 Functional Components

A simple chart parser within COGENT consists of: a buffer in which to construct the chart, a process that enters words onto that chart, buffers containing lexical knowledge (i.e., the syntactic categories of words) and grammatical knowledge (i.e., the phrase structure rules of the grammar), and a process that uses lexical and grammatical knowledge to enter edges onto the chart. Figure 12 shows the components and their interconnections.



Name: Subject
Type: Box/Compound/Subject
Brief Description: A simple parallel chart parser

Figure 12: The box and arrow structure of the parallel chart parser

```
Rule 1 (unrefracted): Add a word to the first position of the chart
TRIGGER: word(W)
IF:      not edge(_, _, _, _) is in Chart
THEN: add edge(0, 1, word(W), 0) to Chart

Rule 2 (unrefracted): Add a word to the next position of the chart
TRIGGER: word(W)
IF:      edge(N0, N1, word(W1), Y) is in Chart
         not edge(N1, N2, word(W2), Y) is in Chart
         N2 is N1 + 1
THEN: add edge(N1, N2, word(W), Y) to Chart
```

Figure 13: Rules to enter words onto the chart (from *Input/Output*)

4.2.2 Input/Output

Recall that *Experimenter* was designed to deliver one word at a time to *Input/Output*, and that *Input/Output* must enter words as they are presented onto the chart. This may be achieved with two rules: one to enter the first word spanning vertices 0 to 1, and a second to enter each successive word to the right of the previous word. The rules in Figure 13 accomplish the above tasks. They represent chart edges by terms of the form:

`edge(LeftVertex, RightVertex, Content, Level)`

where `LeftVertex` and `RightVertex` are integer vertex labels, `Content` represents the content of the edge (e.g., `word(cat)`), and `Level` is an integer that will be used for formatting the visual representation of the chart. Given this propositional representation, display rules may map the contents to *Chart* to a visual, chart-like, representation. Figure 14 shows two suitable rules.

```
Display Rule 1: Draw arcs for all categories entered on the chart
IF:      edge(N0, N1, C, L) is in Chart
         X is (N0 + N1) * 50
         Y is L * 25 + 50
THEN: show text(chart, C aligned (c, c) at (X, Y), [colour(black)])

Display Rule 2: Draw arcs for all categories entered on the chart
IF:      edge(N0, N1, C, L) is in Chart
         X0 is N0 * 100 + 5
         X1 is N1 * 100 − 5
         Y0 is L * 25 + 40
         Y1 is Y0 − 5
THEN: show line(chart, (X0, Y0) to (X1, Y0), [colour(black)])
      show line(chart, (X0, Y0) to (X0, Y1), [colour(black)])
      show line(chart, (X1, Y0) to (X1, Y1), [colour(black)])
```

Figure 14: Display rules for *Chart*

Exercise 4: Draw the box and arrow diagram of the chart parser and add the rules given in Figure 13 to *Input/Output* so that words are appropriately added to the chart as they are received. Also add the display rules given in Figure 14 to *Chart*. Test the system by opening *Chart* in Current Display view and running the model over several trials. On each trial, the words making up one stimulus sentence should appear on the chart, together with edges corresponding to each word. ◇

4.2.3 Lexical Look-Up and Creating Chart Edges

Once a word is entered on the chart, it is straightforward to enter its corresponding grammatical category by looking up that category in a lexicon (i.e., a list of words and their properties). If *Lexicon* contains the syntactic category of all known words, then this may be achieved by a single rule within *Elaborate Chart*, as in Rule 1 of Figure 15. The rule matches an element in *Chart*, looks up its category in *Lexicon*, and adds a category edge at the next level to *Chart*. The rule assumes that *Lexicon* contains terms of the form:

```
category(the, det)
category(kittens, cn)
category(dog, cn)
```

Rule 1 (refracted): *Lexical look-up*
IF: `edge(N0, N1, word(W), L1)` is in *Chart*
`category(W, C)` is in *Lexicon*
`L` is `L1` + `1`
THEN: add `edge(N0, N1, cat(C), L)` to *Chart*

Rule 2 (refracted): *Apply Unary Grammar Rules*
IF: `edge(N0, N1, cat(C1), L1)` is in *Chart*
`rule(C, [C1])` is in *Grammar Rules*
`L` is `L1` + `1`
THEN: add `edge(N0, N1, cat(C), L)` to *Chart*

Rule 3 (refracted): *Apply Binary Grammar Rules*
IF: `edge(N0, N1, cat(C1), L1)` is in *Chart*
`edge(N1, N2, cat(C2), L2)` is in *Chart*
`rule(C, [C1, C2])` is in *Grammar Rules*
`L` is `max(L1, L2)` + `1`
THEN: add `edge(N0, N2, cat(C), L)` to *Chart*

Figure 15: Rules for adding edges to *Chart* (from *Elaborate Chart*)

Exercise 5: Add Rule 1 to *Elaborate Chart* to perform lexical lookup. Also add some lexical entries to *Lexicon*. Test the partial model by running it on various test sentences. Do this by specifying the test sentences in the *Experimenter* module. ◇

4.2.4 Building Phrases

It is intended that phrase structure rules (such as S → NP VP) are represented in *Grammar Rules*, and that this buffer is accessed when applying phrase structure rules. *Grammar Rules* might therefore contain terms such as:

```
rule(s, [np, vp])
rule(np, [det, cn])
rule(np, [pn])
```

Here, the first argument of `rule`/2 specifies the syntactic category on the left side of the phrase structure rule and the second argument is a list specifying the categories (in order) on the right side of the phrase structure rule. Unary phrase structure rules (i.e., rules of the form NP → PN, where there is a single category on the right side of the rule) may then be applied with a COGENT rule such as Rule 2 in Figure 15. Similarly, binary phrase structure rules (e.g., S → NP VP) may be applied with a rule such as Rule 3. Rule 3 extends the use of terms of the form `edge(N0, N1, cat(det), L)` within the chart, such that edges may span sequences of words (e.g., `edge(0, 2, cat(np), 2)`).

Exercise 6: Add appropriate phrase structure rules to *Grammar Rules*. Also add the above rules to *Elaborate Chart*. Test the model on a range of sentences. The model should now be able to construct complete charts for all sentences specified by the grammar. ◇

4.2.5 Discussion of the Model

The chart-parser described above is extremely simple, yet it is complete. It is bottom-up because parsing is driven by the addition of words to the chart. The chart is then expanded "upwards" (from lexical categories to phrasal categories) as far as possible. If at any stage in processing no further additions to the chart are possible, the system quiesces (i.e., no more rules fire), and *Experimenter* sends in the next word. The system is parallel because COGENT's default behaviour is parallel. If multiple additions to the chart are simultaneously possible, then those additions will be made simultaneously. In particular, if a word is syntactically ambiguous (i.e., has two possible syntactic categories), both categories will be added to the chart simultaneously, and analyses based on both categories will be explored in parallel. While this results in a messy chart with alternative analyses appearing to over-write each other, it does not affect the performance of the parser. Subsequent sections of this tutorial will explore modifications to both the bottom-up and parallel nature of this system.

Exercise 7: Extend the lexicon and grammar rules. For a particular challenge, try to add phrase structure rules that allow sentences such as *The kitten that chases the dog bites the man* and *The kitten that the dog chases bites the woman*. For present purposes, restrict attention to sentences in the simple present tense. ◇

Exercise 8: The COGENT rules for applying unary and binary phrase structure rules contain some redundancy. They are also insufficient for cases involving ternary phrase structure rules (and other higher order phrase structure rules). Such rules appear to be implicated in a range of grammatical constructions, such as those involving verbs with multiple complements (e.g., dative verbs, such as *give*, and some verbs with infinitival complements, such as *promise* and *persuade*). Merge the two COGENT rules for building phrases into a single rule that can apply irrespective of the number of elements on the right side of the phrase structure rule. It may be useful to approach this exercise by creating a user-defined condition (e.g., `spans`/3) that takes as arguments two integers (representing vertices) and a list of syntactic categories and succeeds if the list of syntactic categories spans the two vertices. Another user-defined condition may be required to determine the level of spanning elements on the chart. ◇

4.3 Extending the Grammar: Agreement

One difficulty with the parser developed above is that, if given appropriate lexical entries, it will successfully parse ill-formed sentences such as the following:

(4) a. John chase Mary
    b. Every kitten bite the dog
    c. A kittens run

This is actually a difficulty with the grammar (i.e., the rules of syntax), and not the parser. For example, noun phrases within English cannot consist of any determiner followed by any common noun: the determiner and common noun must both be either singular or plural. Thus, *a kittens* is an ill-formed noun phrase because the determiner (*a*) is singular but the common noun (*kittens*) is plural.

One way in which number agreement may be correctly taken into account is by elaborating the lexicon with number information and adding agreement specifications to the grammar rules. The former may be achieved by replacing lexical entries such as:

```
category(a, det)
category(the, det)
category(kittens, cn)
```

with:

Table 1: Pronouns and their person/number/case features

		Number/Case			
		sing/nom	sing/acc	plural/nom	plural/acc
Person	1	I	me	we	us
	2	you	you	you	you
	3	he/she	him/her	they	them

```
category(a, det(sing))
category(the, det(_))
category(kittens, cn(plural))
```

Note that because *the* can be singular or plural, its number specification is left as a variable (the underscore character).

The latter may be achieved by replacing grammar rules such as:

```
rule(np, [det, cn])
```

with:

```
rule(np(Num), [det(Num), cn(Num)])
```

which indicate that a noun phrase with number specification `Num` may consist of a determiner with number specification `Num` followed by a common noun with the same number specification.

Similarly, the rule that licenses verb phrases headed by transitive verbs:

```
rule(vp, [tv, np])
```

must be elaborated with number agreement specifications:

```
rule(vp(Num), [tv(Num), np(_)])
```

The underscore character is used in this rule to indicate that the number feature of the noun phrase following a transitive verb is independent of the number feature of the transitive verb or the verb phrase as a whole. Thus, *chases the kitten* and *chases the kittens* are both singular verb phrases, because they can both be preceded by a singular noun phrase (e.g., *John*) to yield a well-formed sentence. Neither can be legitimately preceded by a plural noun phrase (e.g., *the dogs*) to yield a well-formed sentence. In contrast, *chase the kitten* and *chase the kittens* are both plural, as they can both be preceded by a plural noun phrase to yield well-formed sentences.

Exercise 9: Extend the lexicon and the grammar rules with number information. Check that the parser now correctly discriminates between sentences with correct number agreement and sentences with incorrect number agreement by providing a range of grammatical and ungrammatical examples in *Stimuli*. ◇

Two other forms of agreement of relevance to English noun and verb phrases concern person and case. Person may be first (e.g., *I*, *me*, *we*), second (e.g., *you*) or third (e.g., *he*, *she*, *it*, *him her*, *them*). Case may be nominative (e.g., *I*, *he she*) or accusative (e.g., *me*, *him her*).

Person and case agreement are seen most clearly in noun phrases consisting of pronouns. Thus, Table 1 shows standard pronouns for the various combinations of person, case and number. Person and number features also apply to verbs. Hence a table similar to Table 1 could also be constructed for each verb.

Grammaticality requires that noun phrases in subject position must be nominative case and the person feature of the subject noun phrase must agree with the person feature of the following verb phrase. Noun phrases in object position (i.e., following a transitive verb within a verb phrase based on that verb) must be accusative case. To ensure agreement of all features (number, person and case), all phrase structure rules relating to noun phrases and verb phrases must be revised to include number, person and case features. For example:

S → NP(`Number`, `Person`, nom) VP(`Number`, `Person`)
VP(`Number`, `Person`) → TV(`Number`, `Person`) NP(_, _, acc)

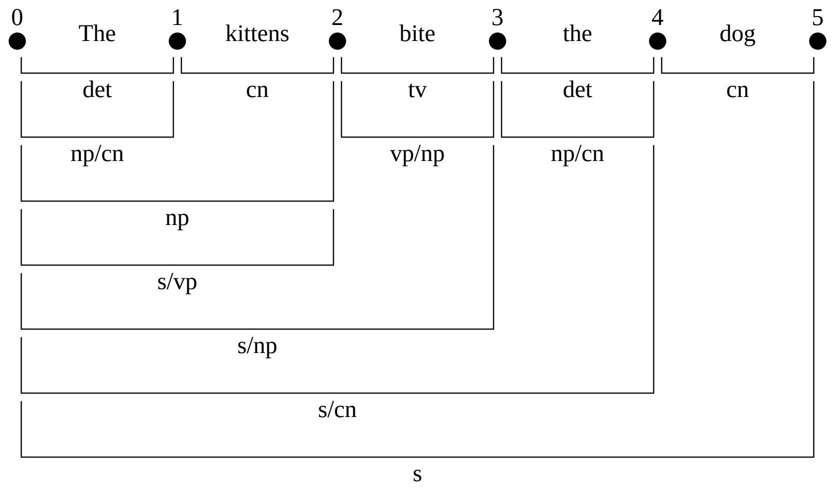


Figure 16: A fully incremental chart for *the kittens bite the dog*

NP(`Number`, 3, _) → Det(`Number`) CN(`Number`)
NP(`Number`, `Person`, `Case`) → PN(`Number`, `Person`, `Case`)

In these rules `Number`, `Person` and the underscore character represent variables.

Exercise 10: Extend the lexicon and grammar to include agreement on number, person and case, as described above. Test your solution with sentences containing pronouns in a variety of forms. ◇

5 Sentence Processing: Towards Incremental Interpretation

It is generally agreed that sentence processing is incremental, in the sense that words in an input stream are incorporated into mental structures as they are encountered, and not at, for example, the end of a phrase or sentence (cf. Mitchell, 1994; Crocker, 1999). On this count, the bottom-up parallel chart parser described above lacks psychological plausibility. When parsing *the kittens bite the dog*, for example, the parser must parse the complete verb phrase (*bite the dog*) before combining it with the noun phrase. Worse, after processing *the kittens* the parser has no expectations about the next word or phrase. English speakers, in contrast, know that the next constituent is likely to be a verb phrase (or, in certain situations, a noun phrase modifier such as *in the garden*).

5.1 Left Corner Parsing

The bottom-up parsing algorithm can be modified to incorporate incremental processing and prediction by the introduction of "active" chart edges. An active edge is one that represents an incomplete constituent. Thus, in Figure 16, the edge labelled np/cn, and spanning the first instance of *the*, is an active edge corresponding to an incomplete noun phrase. The notation np/cn indicates that the edge must be merged with an edge labelled cn (and hence spanning a common noun) to yield a complete noun phrase. The np/cn edge is licensed by the grammar rule that states that a noun phrase consists of a determiner followed by a common noun.

In the figure the edge labelled np/cn and spanning *the* is followed by an edge labelled cn (and spanning *kittens*). These edges together license a further edge, labelled np and spanning *the kittens*. The grammar rule that states a sentence consists of a noun phrase followed by a verb phrase allows another active edge spanning *the kittens* to be added to the chart. This edge is labelled s/vp, indicating that *the kittens* is a sentence that is missing a verb phrase.

The transitive verb *bites* may similarly be spanned by an edge labelled vp/np (because a transitive verb is equivalent to a verb phrase missing a noun phrase). Incremental processing may be maintained by adding an edge spanning the sentence initial fragment (labelled s/vp) and the verb phrase initial fragment (labelled vp/np). Together, the fragments yield a sentence missing a noun phrase, so the new edge is labelled s/np. Similar principles license an edge incorporating the second determiner (labelled s/cn) and the final edge spanning the entire sentence. Figure 16 shows the complete chart resulting from this approach. The chart is fully incremental because each successive word may be incorporated into the structure to its left.

The incremental technique depends upon three augmentations to the earlier approach:

1. allowing projection of edges corresponding to complete constituents to active edges spanning those constituents (e.g., projection from a complete Y constituent to an incomplete X/Z constituent spanning the same material, for each grammar rule of the form $X \rightarrow Y\ \ Z$);
2. allowing an edge labelled X/Y followed by an edge labelled Y to be spanned by an edge labelled X; and
3. allowing an edge labelled X/Y followed by an edge labelled Y/Z to be spanned by an edge labelled X/Z.

The first two of these augmentations characterise the “left corner” family of parsing algorithms. (So named because, in augmentation 1, the constituent Y is the left corner of the phrase X, so phrases are introduced when their left corners are encountered.) The third augmentation, which is sometimes referred to as composition, is not essential for successful left-corner parsing but allows greater incrementality.

5.2 A Parallel Model of Left Corner Parsing

Conversion of the parser developed in Section 4.2 to a left corner parser requires modification only of the representations of syntactic categories and the rules within *Elaborate Chart* for creating edges on the chart. No modifications are required to the *Lexicon* or *Grammar Rules*.

Full generality requires that it be possible to represent several different kinds of syntactic category. The simplest are those corresponding to complete constituents (with no missing constituents), such as np, vp, and s. Incomplete constituents lacking one sub-constituent were illustrated above, but in general incomplete constituents may lack multiple sub-constituents, as illustrated by dative verbs (such as *give*), which require a noun phrase and a prepositional phrase to form a complete verb phrase. (This corresponds to the fact that dative verbs are introduced by ternary phrase structure rules.) For full generality it is therefore appropriate to represent syntactic categories as structures of the form `atom/list`, where `atom` is the main category, and `list` is a list of the categories of missing constituents. This approach allows all categories to be represented in a uniform manner, for example:

`np/[ ]`:	*the kittens*
`s/[vp]`:	*the kittens*
`s/[ ]`:	*the kittens bite the dog*
`np/[cn]`:	*the*
`vp/[np]`:	*bite*
`vp/[np, pp]`:	*gave*
`vp/[cn, pp]`:	*gave a*
`s/[cn, pp]`:	*the kittens gave a*

Given this representation the action of the rule for lexical look-up (Rule 1 from Figure 15) must be modified to specify that lexical items correspond to complete constituents, as in Rule 1 of Figure 17.

A second rule is required to apply phrase structure rules by projecting up from their left corners. Rule 2 of Figure 17 applies when *Chart* contains an edge corresponding to a complete constituent and *Grammar Rules* contains a phrase structure rule with that constituent as its left corner. The list representations used for missing constituents ensure that Rule 2 may apply for all phrase structure rules, including unary phrase structure rules (e.g., projecting an intransitive verb (`iv/[ ]`) up to a complete verb phrase (`vp/[ ]`)), binary phrase structure rules (e.g., projecting a determiner (`det/[ ]`) up to an incomplete noun phrase (`np/[cn]`)), and higher-order phrase structure rules (e.g., projecting a dative verb (`dv/[ ]`) up to an incomplete verb phrase (`vp/[np, pp]`)).

Given Rules 1 and 2, a third rule (Rule 3 in Figure 17) is required to merge edges. Rule 3 applies when a partial constituent is followed by a complete constituent of an appropriate type. Its action is to add an edge spanning those two constituents.

Rule 1 (refracted): *Lexical look-up*

```
IF:     edge(N0, N1, word(W), L1) is in Chart
        category(W, C) is in Lexicon
        L is L1 + 1
THEN:   add edge(N0, N1, cat(C/[]), L) to Chart
```

Rule 2 (refracted): *Project complete constituents*

```
IF:     edge(N0, N1, cat(C1/[]), L1) is in Chart
        rule(C, [C1|Tail]) is in Grammar Rules
        L is L1 + 1
THEN:   add edge(N0, N1, cat(C/Tail), L) to Chart
```

Rule 3 (refracted): *Merge active edges with following edge*

```
IF:     edge(N0, N1, cat(C/[H|T]), L1) is in Chart
        edge(N1, N2, cat(H/[]), L2) is in Chart
        L is max(L1, L2) + 1
THEN:   add edge(N0, N2, cat(C/T), L) to Chart
```

Figure 17: Rules for parallel left-corner parsing

Exercise 11: The three rules in Figure 17, together with the triggered rules for adding initial edges to the chart (Rules 1 and 2 in Figure 13), constitute a complete, parallel, left-corner parser. No user-defined conditions are required. Create a new chart parser model (based on the model developed earlier in the tutorial) and convert it to a left-corner parser by replacing the relevant contents of *Elaborate Chart* with the above rules. Test the parser on a range of sentences. ◇

Exercise 12: Rule 3 from Figure 17 only allows the combination of edges when the second edge spans a complete constituent. The rule therefore does not allow composition. Many constructions licensed by the phrase structure rules given earlier in the tutorial cannot therefore be parsed in a fully incremental fashion by a parser based only on the rules in Figure 17. Modify Rule 3 to overcome this restriction. ◇

6 Sentence Processing: Serial Parsing

All parsers developed thus far have operated in parallel. At each stage of processing, all possible parse structures have been generated, and, in the case of constituents with multiple possible structures (including both local and global ambiguity) all structures have been pursued in parallel. COGENT is well suited to the development of parallel parsing systems, but garden path sentences (as in Example 3) provide clear evidence that the HSPM does not work this way, for if it did such sentences would present no problem: all parse structures would be explored in parallel and the incorrect garden path structure would not prevent or impact upon the simultaneous generation of the correct and complete structure.

Theorists have adopted different interpretations of garden path and related phenomena (see Mitchell (1994) for a review). On some accounts, parsing is a parallel process similar to that described above, but alternative parse structures are ranked in order of preference (e.g., MacDonald, Pearlmutter, & Seidenberg, 1994; Trueswell & Tanenhaus, 1994; Vosse & Kempen, 2000). Garden path sentences are those that require a high-ranking parse to be discontinued and a low-ranking parse to be shifted to the top of the rank order. On other accounts, parsing is a strictly serial process in which a single structure, corresponding to the "preferred reading", is constructed during processing (e.g., Frazier & Fodor, 1978; Abney, 1989).

6.1 Computational Requirements of Serial Parsing

Within serial parsing models, at each point of potential ambiguity the parser selects one structure and ignores all others. If this structure proves to be incorrect (as in a garden path), then the parser retraces its steps (or backtracks) to a previous point of ambiguity, disassembling incorrectly parsed constituents on the way, and selects a different structure to pursue. On this view, garden path sentences cause the parser to backtrack. In severe cases this backtracking may fail, yielding grammatical sentences that cannot be parsed (i.e., a dissociation between linguistic competence and linguistic performance).

Serial linguistic processing therefore has several computational requirements. First, at each stage of processing the parser must select between elaborating existing memory structures (i.e., in the current context, adding a new edge spanning existing chart elements), processing the next word, or backtracking to some previous state of processing. If multiple sources of information deriving from, for example, semantic context, syntactic processing preferences, or statistical biases, may contribute to this selection — and some empirical work suggests that this is the case — then the selection process bears some similarity to the process of operator selection in problem solving; see also Newell (1990, pp. 440–459) and Lewis (1993)). Recall that operator selection is a multi-step process, in which operators are first proposed and then evaluated. This allows multiple sources of information to contribute to an operator's evaluation. When all operators have been evaluated, the operator with the highest evaluation is selected and applied. The process then repeats.

The second computational requirement of serial parsing is that, at least in the context of chart parsing, processing should involve depth-first rather than breadth-first search of the parse space. That is, parsing should involve building a structure representing one possible parse, and abandoning this only if and when it proves counter-productive. Chart parsing as described in the previous sections tends to build a chart in which all edges (representing all possible parses) are present. Serial parsing within the chart parsing framework therefore requires inhibiting the production of some edges. Specifically, serial parsing requires that once an edge has been spanned (i.e., covered by a new edge), it should become unavailable for further processing.

A third computational requirement is that previous states of the parser must be recoverable when backtracking is required. Within the context of chart parsing previous states of the parser are represented by edges on the chart that are spanned by other, newer, edges. Backtracking therefore consists primarily of removing edges from the chart, thus exposing edges added earlier, and making them once again available for processing. Backtracking also requires that, when reparsing a constituent, the system yields an alternative parse. Thus, when multiple ways of combining constituents are available, the parser must differentiate between those that have been pursued and found to be wanting, and those that remain to be pursued.

In sum, serial chart parsing may be achieved with a system based on operator proposal, selection, and application, where the principal parsing operator is "add an edge", where backtracking (which involves the removal of edges) may be triggered if no operators are available, and where a mechanism is included to allow the system to recover from backtracking with an alternative parse.

6.2 A Serial Model of Left Corner Parsing

6.2.1 The Basic Architecture

Figure 18 shows a box and arrow diagram for an operator-based serial chart parser. The diagram differs from the parallel non-operator-based version (Figure 12) in two principal ways. First, the *Elaborate Chart* process has been replaced by three components: *Propose Operators* (a process containing rules that inspect *Chart* and propose possible operators and evaluations), *Operators* (a buffer in which proposed operators are stored), and *Apply Operator* (a process that selects the operator in *Operators* with the greatest evaluation and applies it to *Chart*). Second, the *Input/Output* process has been deleted. In its place, *Propose Operators* has read access to *Paper*. This configuration of components assumes that the reading of words is achieved through selection and application of an appropriate operator (rather than through a separate input/output process).

The use of an operator to read the next word requires a rule within *Propose Operators* to propose the operator at the appropriate time (i.e., when no other operators are being processed), and a rule within *Apply Operator* to apply such an operator if it is selected.

Incremental processing suggests that operator proposal should involve proposing an operator to read the next word at each stage in processing, but that the operator should have a low evaluation (so that processing of previously read words is preferred). Rule 1 of Figure 19 is appropriate. This rule will only fire when *Operators* is empty. If there is an unread word, an operator to read that word will be proposed (i.e., added

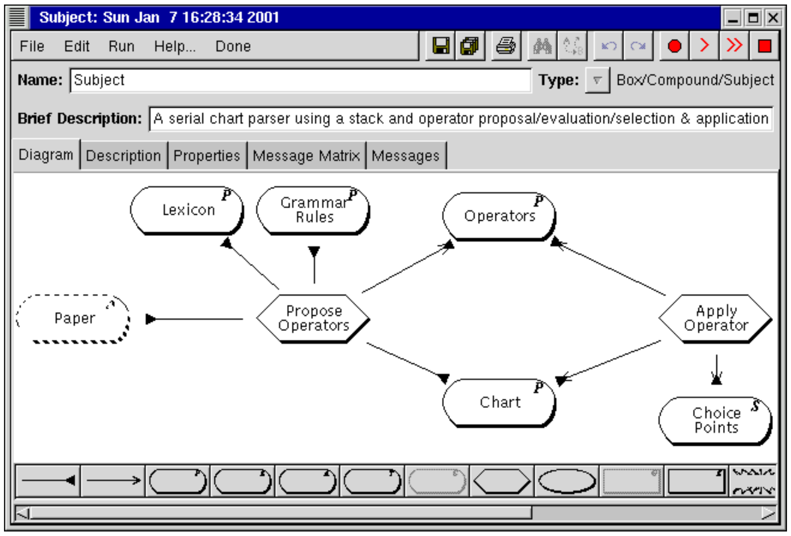

Subject: Sun Jan 7 16:28:34 2001
File Edit Run Help... Done
Name: Subject
Type: Box/Compound/Subject
Brief Description: A serial chart parser using a stack and operator proposal/evaluation/selection & application
Diagram Description Properties Message Matrix Messages
Lexicon
Grammar Rules
Operators
Paper
Propose Operators
Apply Operator
Chart
Choice Points

Figure 18: The box and arrow structure of the serial chart parser

to *Operators*), and given an evaluation of zero.

Rule 1 (unrefracted): *Propose reading of next word*
IF: not `AnyOperators` is in *Operators*
`next_unread_word(Word)`
THEN: add `operator(add_word(Word), value(0))` to *Operators*

Condition Definition: `next_unread_word/1`: *Get the next word from the paper*
`next_unread_word(Word) :-`
not `edge(_, _, word(_), _, _)` is in *Chart*
`word(0, Word)` is in *Paper*
`next_unread_word(Word) :-`
`edge(_, N, word(_), _, _)` is in *Chart*
not `edge(N, _, word(_), _, _)` is in *Chart*
`word(N, Word)` is in *Paper*

Figure 19: An operator proposal rule for reading the next word

Exercise 13: Create a copy of the previous version of the left corner parser and perform the modifications to both the top-level and the *Subject* box and arrow diagrams necessary for an operator-based approach (cf. Figure 18). Add Rule 1 and the definition of `next_unread_word/1` from Figure 19 to *Propose Operators*. (Hint: The condition should compare the words on *Paper* with the words entered in *Chart*. If there are no words on *Chart*, the argument should be unified with the first word on *Paper*. If there are some words on *Chart*, the index of the right-most word should be used to identify the next word in *Paper*, and the argument should be unified with this.) Test the modifications. The system should begin by proposing an operator to add the first input word to the chart. ◇

With the operator proposal rule of Figure 19, only one operator will be proposed at a time, and that operator will always have an evaluation of zero. In general, however, multiple operators with different evaluations may be proposed during the parsing process. *Apply Operator* requires a rule for selecting from the available operators one operator with the highest value. The rule should fire whenever *Operators* contains evaluated operators, and should change the status of one of those operators (the selected operator) to `selected`. This effect is achieved by Rule 1 of Figure 20.

```
Rule 1 (unrefracted; once): Select an operator
IF:    operator(Operator, value(Score)) is in Operators
       not operator(AnyOp, selected) is in Operators
       not operator(OtherOp, value(OtherScore)) is in Operators
           OtherScore is greater than Score
THEN:  delete all operator(AnyOp, value(AnyVal)) from Operators
       add operator(Operator, selected) to Operators

Rule 2 (refracted): Add a word to the next position of the chart
IF:    operator(add_word(W), selected) is in Operators
       get_word_position_parameters(N0, N1)
THEN:  add edge(N0, N1, word(W), W, 0) to Chart
```

Figure 20: Selected operator selection/application rules for *Apply Operator*

Rules are also required to apply selected operators. Thus, if an `add_word`/1 operator is selected, a new word edge should be added to the chart as in Rule 2 of Figure 20. Other operators (as introduced below) will require additional operator application rules. Rule 2 assumes that `get_word_position_parameters`/2 determines the vertices of the edge. If the chart is empty, this condition should bind its arguments to `0` and `1`. Otherwise the vertices should be determined from the values of the right-most word on the chart.

A further rule (not presented here) is required in *Apply Operator* to remove selected operators once they have been applied. While this could be achieved by adding an appropriate delete action to Rule 2 above, the action is required for all operator application rules, and so is better performed by a single general purpose rule that captures this general fact.

Exercise 14: Add the operator selection and application rules as described above (including a rule for deleting selected operators). Test your solution. The system should now propose and apply a series of operators, resulting in all words in the stimulus input for the current trial being added, in an incremental fashion, to the chart. ◇

Exercise 15: The remainder of the parsing model may now be converted to a serial, operator-based, left corner, parser. Further operator proposal rules are required for the three edge addition rules previously defined in *Elaborate Chart* (for lexical look-up, projection from the left-corner of a phrase structure rule, and composition of consecutive constituents). Each operator proposal rule should, like the operator proposal rule above, only fire when *Operators* is empty (i.e., when processing of the previous operator is complete), and should add operator specifications (i.e., elements representing edges to be added) to *Operators*. These operators should always be preferred to the `add_word`/1 operator, but as yet we have no justification for favouring one over the other. Therefore give the operators generated by each rule equal positive evaluations. For present purposes do not attempt to restrict operators to edges that are not otherwise spanned, and do not attempt to include backtracking. (This version of the parser will therefore not attempt depth-first parsing, but will perform a serial version of breadth-first parsing.) ◇

Exercise 16: The following grammar fragment licenses garden path sentences such as *the horse raced past the barn fell*:

NP → PN	PN: john, mary
NP → Det(Num) CN(Num)	Det(sing): a, every
CN(Num) → CN(Num) RedRel	Det(_): the
RedRel → VP	CN(sing): horse, barn
VP → IV	IV: fell
VP → TV(Comp) Comp	TV(PP(move)): raced
PP(X) → Prep(X) NP	Prep(move): through, past
S → NP VP	CN(plural): horses, barns

The garden path effect arises in the grammar because the clause *raced parsed the barn* may be analysed as a constituent of type RedRel (a reduced relative clause, that may modify a common noun) or a constituent of type VP. Note also that the grammar includes the rule:

$$\text{VP} \rightarrow \text{TV(Comp)} \;\; \text{Comp}$$

This is a "meta-rule" that licenses a variety of transitive verb-type constructions. In the grammar the lexical category of *raced* is defined as TV(PP(move)) (i.e., a transitive verb taking a single complement of category PP(move). The meta-rule therefore licenses phrases such as *raced past the barn* (because other rules license *past the barn* as a constituent of category PP(move)). If the grammar also included a lexical entry such as *chased*, of category VP(NP), the same meta-rule would license VP phrases such as *chased the horses*.

Replace the existing lexicon and grammar rules with the above, and substitute appropriate test sentences (involving garden path and non-garden path sentences licensed by the grammar) in the *Experimenter* module. Test the parser. It should provide correct parses for both garden path and non-garden path sentences of the grammar. It should also provide unusual parses for various constituents. For example, *the horse fell* should be parsable as both a complete sentence and as a noun phrase (via the reduced relative construction). (In fact, *the horse fell* is not a grammatical noun phrase of English. It is only licensed as such by this grammar because the grammar lacks features marking aspect. Specifically, reduced relative constructions require a verb phrase in progressive form (e.g., *raced past the barn* or *felled*), rather than one in the simple past (e.g., *fell*).) ◇

6.2.2 Restricting Operator Proposal

As noted above, serial (depth-first) parsing implies that there is one syntactic structure under construction at a time, and that, when adding edges to the chart, only operators that extend this structure are proposed. Operators that contribute to other structures or that reanalyse existing sub-structures are not proposed. This may be achieved by adding extra conditions to the operator proposal rules that prevent those rules from applying to edges that have already been spanned. To illustrate, consider the definition of `edge_is_spanned`/3 in Figure 21. The condition is true of an edge between `W0` and `W1` and at level `Y1` if and only if there is another edge on the chart spanning at least the same vertices, but at a higher level. Any such higher-level edge should prevent the lower-level edge being used to trigger the addition of another edge.

Rule 2 of Figure 21 shows how `edge_is_spanned`/3 may be integrated into lexical look-up. With the addition of the condition to check for spanning edges, `add_edge`/4 operators deriving from lexical look-up will only be proposed when a word is not otherwise spanned.

Rule 2 (unrefracted): *Propose lexical look-up*

```
IF:    not AnyOperator is in Operators
       edge(A, B, word(W), L) is in Chart
       not edge_is_spanned(A, B, L)
       category(W, C) is in Lexicon
       L1 is L + 1
THEN: add operator(add_edge(A, B, C/[ ], L1), value(5)) to Operators
```

Condition Definition: `edge_is_spanned`/3: *Is an edge already spanned?*

```
edge_is_spanned(W0, W1, L1) :-
    edge(N0, N1, C, L) is in Chart
    W0 is not less than N0
    W1 is not greater than N1
    L is greater than L1
```

Figure 21: A rule to restrict lexical lookup (from *Propose Operators*)

Exercise 17: Add the definition of `edge_is_spanned`/3 from Figure 21 to *Propose Operators* and modify all operator proposal rules to block the proposal of operators that involve edges that have already been

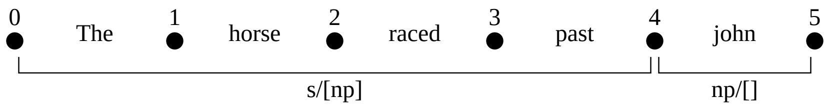


Figure 22: A partial chart for *the horse raced past john*

spanned. Test the model. It should now only build one parse for any constituent. That is, it should now be both incremental and sequential. Note however that this version of the parser will not be able to backtrack or recover from incorrect parsing attempts: once a parse has been established for a constituent, that parse cannot be altered or decomposed. ◇

6.2.3 Simplified Operator Evaluation

In the model developed thus far there are three situations that may lead to an edge being added to the chart: lexical look-up, projection from the right-corner of a phrase to its parent phrase, and composition of consecutive edges. With random operator selection (as above), there will be no bias towards the addition of any particular type of edge when multiple edges resulting from the different situations might be added. To illustrate, consider the partial chart in Figure 22. Two possible edges might be added at this point (assuming the grammar introduced in Exercise 16):

1. The two constituents might be merged through composition, yielding an `s/[]` constituent
2. The `np/[]` constituent (john) could be projected up to `s/[vp]` (via the sentential rule S $\rightarrow$ NP VP)

In the situation shown, the first of these options should be favoured, and a general parsing strategy might consist of always favouring composition of edges over projection of left-corners. Within the current framework, this strategy may be enforced by giving composition operators higher evaluations than projection operators.

Exercise 18: Alter the operator proposal rules to give higher numeric evaluations to composition operators than projection operators. Ensure that this results in appropriate operator selection when parsing the example above. ◇

The above operator evaluation strategy is naïve in many ways. For example, given a sentence ending in a noun phrase followed by a prepositional phrase (e.g., Example 1e), it will result in the noun phrase being attached to the verb phrase and the sentence before processing of the prepositional phrase. The prepositional phrase will then act as a sentence modifier, rather than a noun phrase modifier. The strategy therefore conflicts with that of *late closure*, identified by Kimball (1973). Late closure demands that clause modifiers are attached if possible before the clause is "closed". In the prepositional phrase modifier case alluded to above, this requires that the prepositional phrase is attached to the noun phrase, and not to the sentence. Kimball (1973) argues that late closure captures human ambiguity resolution preferences. However, violations of late closure do occur. For example, Crain and Steedman (1985) and Altmann and Steedman (1988) have demonstrated that different prior contexts can bias sentence processing, leading to early closure in some situations and late closure in others. In essence, they propose that a noun phrase is closed (i.e., incorporated into the sentential structure, through an operation such as composition) precisely when it identifies an appropriate referent. Thus, in a context with multiple cats, a fragment beginning *the cat* will lead to an expectation of a modifier of the initial noun phrase (such as a restrictive relative clause or a prepositional phrase), but if there is only one cat in the context, the same fragment will be parsed as a complete noun phrase. Incorporation of the proposal of Crain, Altmann & Steedman into the current parser requires inclusion of a representation of context, and use of this representation to bias operator evaluation. Such extensions are beyond the scope of this tutorial.

Operator evaluation might also be biased towards preferential application of certain phrase structure rules. For example, noun phrase modification by a prepositional phrase or a relative clause may be preferred over noun phrase modification by a reduced relative construction. Such preferences may be incorporated into operator evaluation by tagging all phrase structure rules with a numeric strength, and basing the evaluation of an operator on the strength of the rule that licenses the operator. The strengths of phrase structure rules might then be modified with use (e.g., adopting the laws of exercise and effect (Thorndike, 1911), such that rules are strengthened when used successfully and weakened when used unsuccessfully).

6.2.4 Backtracking and Reanalysis

In order to allow backtracking it is necessary to record points in processing where multiple operators might apply. These points are known as choice points. If parsing fails, backtracking may then be achieved by undoing all processing since the last choice point, and selecting a different operator from those possible at that choice point. Multiple choice points may occur during normal processing. Full generality (i.e., completeness at the competence level) requires that all such choice points are recorded and that backtracking always involves returning to the most recent choice point that hasn't been fully explored. It is therefore appropriate to record choice points (and the context in which they occurred) on a stack. When choice points are encountered, their details may be pushed onto the stack. When backtracking is triggered the top-most choice point may be popped from the stack and its context restored.

To illustrate backtracking, consider the processing that ensues after parsing *the* as an `np/[cn]` and *horse* as a `cn/[ ]`. Several operators are applicable, including composing the constituents to give an `np/[ ]` constituent and projecting *horse* to a `cn/[rr]` (where `rr` denotes a reduced relative construction). This is a choice point because more than one operator is applicable. Processing should therefore involve selecting one operator to apply and pushing the other operator onto the choice point stack. Given the naïve evaluation strategy above, the former will be selected. If this selection turns out to be inappropriate (as in the case of garden path sentences), backtracking requires that 1) all edges added to the chart after the choice point are removed, 2) the second operator is selected and 3) processing resumes from the choice point.

Extending the model developed thus far to include backtracking requires addition of a stack buffer (*Choice Points*) to *Subject* that is readable and writable by *Apply Operator*. The original operator selection rule (Rule 1 from Figure 20) may be retained, but additional rules are required to push choice points onto the stack when they occur. This is achieved by Rules 2 and 3 in Figure 23.

Rule 2 (unrefracted): *Apply the selected operator, remove all others*
IF: `operator(Operator, selected)` is in *Operators*
THEN: delete all `operator(_, _)` from *Operators*
  add `operator(Operator, apply)` to *Operators*

Rule 3 (unrefracted): *Push unselected operators onto the stack*
IF: `operator(Operator, selected)` is in *Operators*
  `Ops` is the list of all `operator(O, value(V))` such that
    `operator(O, value(V))` is in *Operators*
    `V` is greater than `0`
  `Ops` is distinct from [ ]
  `get_context(Context)`
THEN: send `push(choices(Context, Ops))` to *Choice Points*

Condition Definition: `get_context`/1: *Get the current context id*
`get_context(0) :-`
  not `Anything` is in *Choice Points*
`get_context(C1) :-`
  `choices(C, _)` is in *Choice Points*
  `C1` is `C + 1`

Figure 23: Operator selection rules from *Apply Operator*, modified to use a context and record choice points

Rule 2 fires after an operator has been selected. It changes the state of the selected operator to `apply` (signaling application of the operator by additional rules) and removes all other operators from *Operators*.

Rule 3 fires only when multiple operators are applicable (i.e., when one operator has been selected and when other, unselected, operators with positive evaluations remain). Several aspects of the rules are critical. First, if the rule fires, it will fire in parallel with Rule 2 (i.e., it will be triggered before Rule 2 removes all unselected operators). Second, if it fires it will push a representation of the current choice point onto *Choice Points*. That representation consists of a `choices`/2 term. The first argument of this term is a representation of the context. The second is the list of unselected operators with positive evaluations. This means that if there are several unselected but applicable operators, they will all be pushed onto the stack within a single choice point. The rationale for this is described below.

The representation of context is determined by `get_context`/1, a definition for which is also given in Figure 23. This definition gives a context value of `0` if the choice point stack is empty and `C + 1` if the top-most element of the choice point stack was created in context `C`. Thus, if the top-most element of the choice point stack was created in context `4`, a call to `get_context`/1 will bind its argument to `5`.

Operator application rules follow the format set by Rule 2 of Figure 20, with two exceptions:

1. a rule is required to delete applied operators, and
2. edges must include a specification of the context in which they were added.

Rule 4 in Figure 24 addresses deletion of applied operators. Rule 5 illustrates the addition of a context specification to chart edges (cf. Figure 20).

Rule 4 (unrefracted): *Remove applied operators*
IF: `operator(Operator, apply)` is in *Operators*
THEN: delete `operator(Operator, apply)` from *Operators*

Rule 5 (unrefracted): *Add a word to the next position of the chart*
IF: `operator(add_word(W), apply)` is in *Operators*
`get_word_position_parameters(N0, N1)`
`get_context(TS)`
THEN: add `edge(N0, N1, word(W), 0, TS)` to *Chart*

Figure 24: Selected operator application rules from *Apply Operator*, modified to use a context

Exercise 19: Create a new model based on the previous no-backtracking serial model, add the *Choice Point* stack to the *Subject*, and modify the rules in *Apply Operator* as described above. (Be sure to include a rule for applying `add_edge`/4 operators. The rule is not given above.) Test the model. It should behave much as before, except that as processing proceeds *Choice Points* should grow. ◇

Utilisation of choice point information requires detection of parsing failure and, on such failure, popping the top element of *Choice Points* and restoring the context prior to pushing that element. The three rules in Figure 25 perform the necessary operations. Also given in the figure is a definition for the user-defined condition `parse_successful`/0, which is true when and only when the input has been successfully parsed.

Rule 7 restores the context prior to the previous choice point by removing all edges from the chart whose context (or time stamp) is after that of the choice point. Rule 8 restores all operators (and valuations) which were not applied when the choice point was first encountered. This allows processing to resume from the selection phase of the operator processing cycle. Rule 9 removes the choice point from the stack. All of these rules will fire in parallel when backtracking is necessary. However, Rule 7 will have multiple instantiations if multiple chart edges were added in the previous context, and Rule 8 will have multiple instantiations if several operators were possible at the popped choice point.

With the given definition, `parse_successful`/0 will be true if and only if there is an edge of category `s/[ ]` (i.e., a complete sentence) spanning the entire word sequence. This could be improved but is sufficient for present purposes.

6.2.5 Discussion of the Model

The model should now be able to parse both *the horse raced past john* and *the horse raced past john fell*, but in the latter case parsing will involve extensive backtracking and reanalysis, and will only be possible if the choice point stack's capacity is sufficient. The model therefore captures some of the principal facts

Rule 7 (unrefracted): *Backtrack (step a: remove unwanted edges)*
TRIGGER: `system_quiescent`
IF: not `parse_successful`
`choices(TS, Operators)` is in *Choice Points*
`edge(N0, N1, C, L, TS1)` is in *Chart*
`TS1` is greater than `TS`
THEN: delete `edge(N0, N1, C, L, TS1)` from *Chart*

Rule 8 (unrefracted): *Backtrack (step b: unstack stacked operators)*
TRIGGER: `system_quiescent`
IF: not `parse_successful`
`choices(TS, Operators)` is in *Choice Points*
`Operator` is a member of `Operators`
THEN: add `Operator` to *Operators*

Rule 9 (unrefracted): *Backtrack (step c: pop the stack)*
TRIGGER: `system_quiescent`
IF: not `parse_successful`
`choices(TS, Operators)` is in *Choice Points*
THEN: send `pop` to *Choice Points*

Condition Definition: `parse_successful`/0: *Has the parse succeeded?*
`parse_successful :-`
`edge(0, N, cat(s/[]), _, _)` is in *Chart*
not `edge(N, _, word(_), _, _)` is in *Chart*

Figure 25: Rules from *Apply Operator* for processing choice points

surrounding garden path sentences. Nevertheless, many aspects of the model are clearly highly simplified, including the detection of parsing failure and the evaluation of operators.

Consider first the detection of parsing failure. This is crucial in triggering backtracking, but the condition definition suggested above depends on the parser being given self contained sentences. At first glance this may seem to require a further parser to segment the words in the input into sentences. However, both written and spoken language contain cues that may assist in determining when a phrase or sentence is complete (and hence whether parsing has been successful). In the case of written language, the cues take the form of punctuation. In the case of spoken language the cues are embedded in the prosody, or changing tone and rhythm, of the input.

Operator evaluation is perhaps the least satisfactory aspect of the model. Giving operators fixed evaluations means that the parser cannot adjust its performance in an attempt to minimise backtracking, and that the parser is insensitive to the greater context in which a sentence may occur. Given this, the model is perhaps better viewed as a framework within which different approaches to operator evaluation may be explored.

The construction of choice points also requires comment. Rule 3 in Figure 23 builds a choice point from the set of all applicable but non-selected operators with positive evaluations. Operators with zero or negative evaluations are therefore excluded from the choice point, and, because of the mechanism for recovering when backtracking occurs, cannot be selected on backtracking. This is primarily because of the mechanism used for reading successive words. At almost every stage in processing, an `add_word`/1 operator is proposed, so that the parser may proceed to the next word if processing of the previously read words appears complete. If such operators were viewed as on a par with legitimate operators for projecting words or combining categories, then every operator selection step would become a choice point. (To verify this, try altering the rules so that operators with zero evaluation are included in the choice point.)

The second critical feature relevant to the construction of choice points concerns the situation when multiple unselected positively evaluated operators exist. All such operators are included in the choice point. When backtracking occurs, the full set of such operators is recreated in *Operators*, allowing the standard operator selection process to resume. If after selection there still remain multiple unselected positively evaluated operators, then the choice point will be re-established, but with only the remaining unselected operators. In this way, backtracking will if necessary progress in sequence through all possible operators, irrespective of the number of possible operators. An alternative approach would be to keep track of those operators that had been unsuccessfully applied (and the context in which they had failed), and then allow regeneration of all operators when backtracking occurs. Failed operators could then be given negative evaluations.

Exercise 20: The model as developed above is a competence model in the sense that, although it is serial and may perform complex backtracking while attempting to parse a sequence of words, it is able to parse all (and only) sentences of the grammar defined by its lexicon and phrase structure rules. One way in which performance factors may impinge upon the model's behaviour is through the capacity of the choice point stack. Explore the effect of restricting the stack's capacity (through the Properties panel of *Choice Points*). How does a limited stack affect the range of sentences that may be parsed? ◇

7 Conclusion

This completes the tutorial. To recap, COGENT has been introduced and used to develop a series of models of sentence processing. The models illustrate many aspects of the COGENT approach, including: research programme management, the use of box and arrow diagrams at the model level, and the division of a model into experimenter and subject modules. They also illustrate COGENT's flexibility, with both parallel and serial models, and with models using information in both top-down and bottom-up ways. The tutorial has not attempted to illustrate COGENT's generality. As noted in the introduction, COGENT models have been developed in a range of cognitive domains, and COGENT is well-suited to both teaching and research.

Reference Manual

A Configurable Preferences

A range of user-specifiable preferences allow the behaviour and appearance of COGENT to be tailored to individual users' tastes. These preferences are specified through a popup window accessible either from the research programme manager's Preferences... button or from the Preferences... item on the Edit menu of any box window. Preferences relating to different aspects of COGENT's functioning are shown on separate pages of a notebook, under the headings described below. Not all options are available in all versions of COGENT — some are specific to the Windows version whereas others are specific to the UNIX version. In addition, some options are only available to registered users.

If a preference is altered the alteration takes immediate effect for the current session. If alterations are intended to be permanent then they must be explicitly saved using the Save button at the top of the preference window. Note that this button is context sensitive: it only saves the preferences on the currently visible page. Under Windows, preferences are saved in the user's regsitry entry. Under UNIX, preferences are saved in a file named `.cogentrc` the user's home directory.

A.1 Folders

These preferences specify locations for some critical system and user files.

Project directory: This is the folder or directory in which COGENT stores research programmes. Normally each user will keep all his/her research programmes in his/her own file space, but alternate configurations are possible. Users working within a group might, for example, which to share research programmes. In such cases all members of the group may share a project directory.

Script directory: A COGENT script is a file containing instructions that control model execution. (See Section I.) Several default scripts are provided in the standard distribution. These should be located in the script directory. Whenever a model is created, copies of the contents of script directory are assigned to the new model. Under normal functioning the default value of this preference will not need to be altered.

A.2 Help

COGENT provides a web-based help system consisting of a network of hypertext files. These files may be viewed by a standard web browser, and accessed either online (e.g., from the COGENT web site or from some local alternative) or offline (i.e., from your machines hard disk)

Use online help? If this preference is set COGENT will attempt to use online help (see **Online help URL**). If the preference is not COGENT will attempt to use offline help (see **Offline help directory**).

Online help URL: If online help is being used, this preference specifies the URL location of the help system.

Offline help directory: If offline help is being used, this preference specifies the directory or folder containing the help system. If COGENT has been installed correctly, the default value will be correct.

Show help with index frames? If this preference is set, and your system is capable, help pages will be displayed within an index frame. This simplifies navigation through the help system. Web browsers and platforms that support this option include netscape and mozilla on UNIX.

Help browser? This preference (only available under UNIX) specifies the web browser to use when viewing help files. A number of built-in values are provided on a drop-down menu, but any value may be typed in. Under Windows, the default web browser is used.

Help browser accepts remote commands? This preference, which is only available under UNIX, specifies whether the web browser should be accessed using the "remote" command. Certain browsers function better when opened with these commands (e.g., netscape and mozilla). Other browsers (e.g., opera) function well without using remote commands. Others (e.g., lynx) are not able to function remotely.

A.3 OOS

OOS (Object Oriented Sceptic) is name of COGENT's model execution language. Several preferences control its behaviour, though if standard installation is followed all default values should be acceptable.

OOS executable: This preference specifies the location of the OOS executable.

OOS class directory: This specifies the location of the OOS class files. These files define the behaviour of the various types of box.

OOS library directory: This specifies the location of various OOS library files (including operator declarations and, in some versions of COGENT, support for optimisation).

OOS I/O directory: This is the name of the folder or directory in which files containing model output (e.g., from data sinks) are created.

Show debugging information: If this preference is set debugging information is printed to the OOS output trace window.

Save message logs: If this preference is set all messages that pass between boxes are stored (and hence viewable within each box's Message panel). If message logs are saved, they can be valuable for debugging purposes. However, saving message logs significantly slows model execution, and may eat up large amounts of disk space.

Optimise: If this preference is set OOS will use optimised versions of various internal functions. This may result in a substantial increase in speed of model execution.

Show console: If this preference (which is only available under Windows) is set, OOS will run in its own window. Normally this window is suppressed, and any output destined for the window is forwarded to the OOS output trace.

A.4 Printing

A number of preferences allow control over the content, form, and appearance of printed output. These preferences may be set either through the appropriate page on the Preferences window or on the dialogue window that pops up when a print command is issued.

Print to file: If set, printer output will be redirected from the printer to a standard file. Further preferences as described below allow the format of that file to be specified.

Print directory: If Print to file is set, this preference specifies the directory or folder in which the print file will be written.

Print file: If Print to file is set, this preference specifies the name of the print file (within the print directory).

Printer command: This preference is only available under UNIX. If Print to file is not set, the preference specifies the printer command to use to print the output. Useful values of this preference are `lp -c` (for SOLARIS and IRIX 5.X), `lpr` (for SUN OS 4.X and LINUX), and `ghostview` to preview PostScript output without printing it.

Output format: This preference specifies the format in which print output should be generated. It is of most use when Print to file is set. Possible formats include: PostScript, Plan Text, HTML, and LaTeX.

Output page style: This preference controls the orientation and style of output in PostScript output. Possible values are Portrait, Landscape and Two Up. In Portrait style the long edge of the paper is vertical. In Landscape style the long edge is horizontal. Two Up page style consists of two logical pages per sheet of paper. The pages are oriented in portrait style, but reduced by a factor of 1.414 and printed side by side with the long edge of the paper horizontal.

Printer font size: This preference controls the font size used in PostScript printouts. It is ignored for other output formats.

Print new box on new page? When this preference is set, a page break will be generated prior to printing details relating to each box within a model. If the preference is not set, page breaks will only be generated where necessary.

Show initial contents? If set, the initial contents of boxes will be included in the printout.

Show user-defined conditions? If set, definitions of all user-defined conditions (contained in process boxes) will be included in the printout.

Show current contents? If set, the current contents of boxes will be included in the printout.

Show messages? If set, the list of messages sent to and from boxes will be included in the printout.

Show buffer mappings? If set, then any mapping rules that exist for buffers included in the printout will also be printed.

Recurse through subobjects? If set, the details of all subboxes of any compound boxes will be included in the printout.

A.5 Archive

COGENT provides an integrated archive management system. This allows complete research programmes to be packaged into a single file, either for backup or for exchange with other users. The process of packaging a research programme is referred to as archiving. The process of recovering a research programme from an archive is known as extracting.

The behaviour of the archive system is controlled by a number of preferences. Each can be set either from the appropriate page of the preference window, of through the archive browser.

Archive directory: This is the name of the folder or directory in which archives of COGENT research programmes are stored.

Archive format: This species the format in which archive files will be created. Several formats are supported, through COGENT's native format (the "car" file) is recommended.

Sort archives by: This preferences specifies the sort key (name, format, creation date or size) used to sort archives in the archive browser.

Retain research programme after archiving? When set, archiving a research programme will not make the research programme inaccessible from the research programme manager. When not set, archiving a research programme will remove it from the research programme manager.

Retain archive after extracting? When set, extracting an archive will not make the archive inaccessible from the archive browser. When not set, extracting an archive will remove it from the archive browser.

A.6 Fonts

The Fonts panel allows modification of some fonts used on COGENT's windows. Alternative fonts may be used to give COGENT a personalised look, or if something bigger or smaller is required (e.g., for demonstrations). Note that different computers (and in the case of UNIX, different displays) may support different fonts.

Canvas font: This preference specifies the font used for drawing model names on the main history canvas and box names on box and arrow diagrams.

Output font: This preference specifies the font used when displaying the current states of objects.

A.7 Warnings

Several preferences control whether COGENT generates warning messages in unexpected situations. By default all but the first of these is set.

Reassure on success? If set, display a reassuring message whenever a file operation is successfully completed.

Warn on file access errors? If set, display a warning message whenever a file read/write fails.

Warn on syntax errors? If set, display a warning message whenever a syntax error is detected in a term within a model.

Warn on invalid model specification? If set, display a warning message whenever something odd is detected in a model specification file.

Warn on memory allocation failure? If set, display a warning whenever memory allocation fails. Such warnings typically indicate that there is insufficient RAM on your machine to continue processing.

Warn on miscellaneous errors? If set, and any unexpected events not covered by the above occurs, display a warning describing the event.

A.8 Confirmations

The degree to which COGENT seeks confirmation before irreversible acts can be controlled through a number of confirmation preferences.

Confirm discard of model edits? If set, query the user before any unsaved edits are discarded.

Confirm project/model deletion? If set, query the user before models or entire research programmes are deleted.

Confirm overwrite file? If set, query the user before overwriting any files.

Confirm before printing PostScript? If set, and printing in PostScript format under UNIX, query the user before the file is printed. The query will announce the number of pages to be printed, thus allowing large print jobs to be aborted.

Confirm save of model prior to running? If set, query the user before saving whenever a model with unsaved edits is run.

A.9 Miscellaneous

A number of further preferences control other aspects of COGENT's operation:

Double click interval: This preference specifies the maximum number of milliseconds that can elapse between two mouse clicks if they are to be interpreted as a double click.

Grid size: This preference determines the size of the grid (in pixels) to which boxes drawn in a box and arrow diagram will automatically snap.

Display Precision: This value specifies the number of significant digits that COGENT uses when presenting real numbers. (Maximum precision is used for internal calculation.)

Remember window positions? If set, record the size and position of all windows associated with each model and use this information to recreate the display state (i.e., exact window sizes and positions) of that model when it is re-opened.

X Drift: If Remember window positions is set, this specifies the horizontal correction (in pixels) that may be necessary to ensure windows reposition themselves correctly. Some UNIX window managers require small non-zero values for this preference.

Y Drift: As for X Drift, but for vertical correction.

Enable virtual window management? If set, allow windows to be located on multiple virtual desktops. This is extremely useful if COGENT is being run under a window manager that supports multiple virtual desktop. Such facilities are common within UNIX but rare within Windows.

Summarise box contents? If set, show non-compound box contents in a summarised form. (A button on each box window allows toggling between summarised and expanded form.)

Edit on create? If set, automatically open an appropriate editor whenever a box or box element is created.

Scale history? If set, the time-line display of research programme histories will use the creation time of models to scale the horizontal axis of the display. The horizontal distance between models on the display will then represent the difference in their creation times. If the preference is not set, models will be spaced equally on the history display.

Smooth scrolling of text elements? If set, use smooth scrolling for Initial Contents windows. This looks nice but is impractical on slow machines where the alternative, jump scrolling, is better.

Position edit palette above canvas? If set, position the palette of edit buttons for all box windows above the main canvas. Otherwise position the edit palette below the canvas.

Show scrollbars on canvases? If set, display horizontal and vertical scrollbars on all box and arrow diagrams.

Show toolbars on box windows? If set, display toolbars providing shortcuts to common functions on all box windows.

B Research Programme Management

The research programme manager consists of a window with three parts: a row of menus and buttons for accessing a variety of functions, a list of the user's research programmes, and a display of the currently selected research programme.

The research programme manager supports the following operations:

New: Create a new research programme. The user will be prompted for a name for the programme.

Delete: Delete the currently selected research programme.

Rename: Change the name of the currently selected research programme. The user will be prompted for a new name for the programme.

Copy: Create a copy of the currently selected research programme. The user will be prompted for a name for the copied programme.

Print: Print the currently selected research programme. A print dialogue window will appear allowing the user to select which aspects of the research programme should be included in the printout.

In addition, archive functions (e.g., for backing up a research programme) are available through the archive browser.

A research programme may be opened by selecting its name within the list of research programmes on the left of the research programme manager. When a research programme is opened, the research programme manager provides access to a graphical History view and a text-based Description view of the research programme. The History view represents the research programme diagrammatically, with nodes corresponding to individual models and lines between nodes showing ancestral relations between related models. A model may be opened by double-clicking on its node within the History view. The Description view is intended for any notes the user may which to associate with the research programme.

C The Hierarchy of Object Classes

Each box within a COGENT box and arrow diagram has a class, some properties, and some contents. Different classes of box have different functionality (e.g., buffers function as storage devices whereas processes function as information transformation devices), different properties that govern aspects of that functionality (e.g., buffers have properties that govern decay), and different types of content (e.g., buffers contain elements, whereas processes contain rules). This section describes the basic functionality, type of content, and configurable properties of each box class. The full range of available classes are shown in Figure C.

C.1 Compound

Compound boxes are boxes that contain other boxes. They provide a bracketing mechanism that allows a set of related boxes to be grouped together into a single functional unit. Thus, a complex functional component within a box and arrow diagram may be implemented as a compound, whose internal structure is specified by a separate box and arrow diagram embedded within the compound. This box and arrow diagram may be accessed by opening the compound (e.g., by double-clicking on it).

Arrows drawn to or from a compound box allow communication between boxes embedded within the compound and boxes at the higher level. The addition of such an arrow between, for example, box A and box B, where box B is a compound, will make box A accessible to boxes embedded within box B.

Since compound boxes are basically bracketing devices, they have no specific properties governing their behaviour. There are, however, two subclasses of compound box: Compound/Generic and Compound/Subject. Boxes corresponding to these subclasses are distinguished diagrammatically by the letter *G* or *S* in the top-right corner of the compound icon. Generic compounds may contain boxes of any type, and place no restrictions on the contents of those boxes. Subject compounds may only contain boxes that make sense within a model of a subject. For example, data boxes and generic compounds may not occur within a subject compound, because such boxes may have access to the internal state of COGENT in a way that is unrealistic for a subject. Certain built-in conditions are also not accessible from subject compounds (e.g., statistical functions). Subject compounds are meant to provide a small degree of constraint

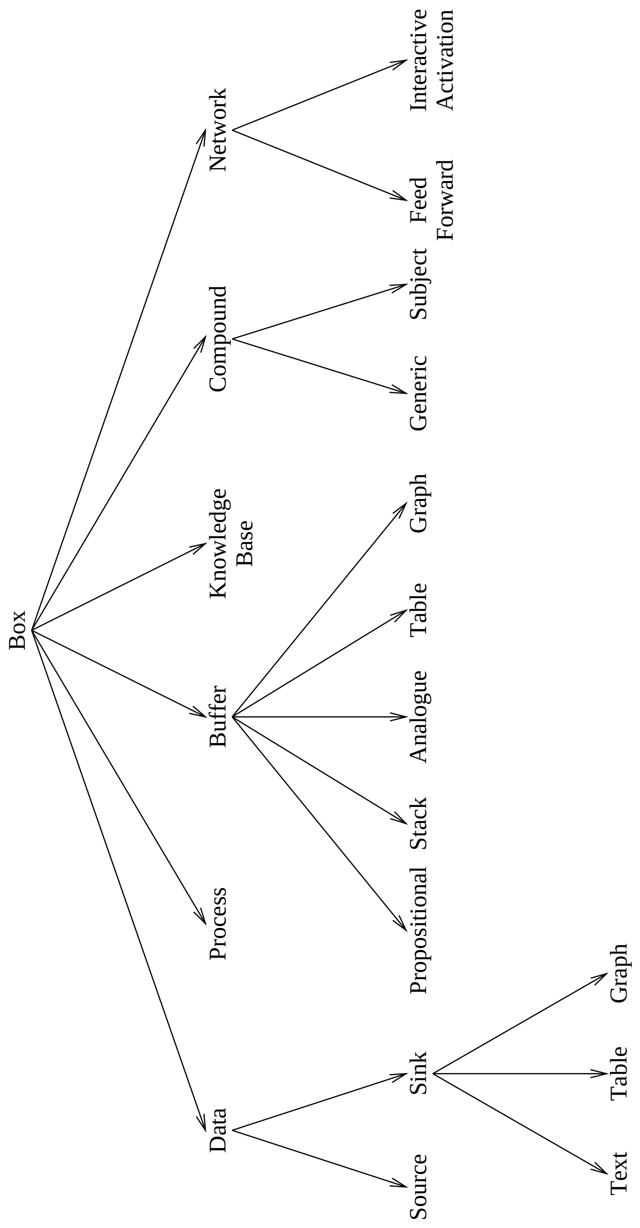


Figure 26: COGENT's hierarchy of box classes

on the development of cognitive models within COGENT. Users who find such restrictions disagreeable or under-motivated may ignore subject compounds and use generic compounds in preference.

C.2 Buffer

A buffer is an information store — a place where information can be placed for later retrieval. Buffers may be configured through subclasses and properties so as to behave in a variety of different ways (allowing, for example, decay of elements or capacity restrictions). This flexibility means that buffers may be used for both short term storage (as when modelling aspects of working memory, for example) and long term storage (as when modelling storage and retrieval of knowledge acquired over an extended time period).

Buffers are typically initialised with a set of buffer element. During processing, existing buffer elements may be matched or deleted, and additional buffer elements may be created. Matching an element in a buffer is achieved through use of the match condition within a rule. The condition might appear within a rule as:

> $\mathtt{term}(\mathtt{Arg1}, \mathtt{Arg2})$ is in *Sample Buffer*

where *Sample Buffer* is the name of a buffer that is readable by the process containing the rule and $\mathtt{term}(\mathtt{Arg1}, \mathtt{Arg2})$ is a template for the element being matched. Such a condition will be satisfied if there is an element in *Sample Buffer* that unifies with the template, in which case any uninstantiated variables within the template will be bound to the values they take in the term matched in the buffer. If a buffer contains multiple elements that match the template, the order in which elements are retrieved is determined by the buffer's Access property (see below).

Buffer contents may be altered by sending an appropriate message to the buffer. This requires that the box doing the sending has a write arrow leading to the buffer. Four types of message are relevant:

> add $\mathtt{term}(\mathtt{arg1}, \mathtt{arg2})$ to *Sample Buffer*
> clear *Sample Buffer*
> delete $\mathtt{term}(\mathtt{arg1}, \mathtt{arg2})$ from *Sample Buffer*
> delete all $\mathtt{term}(\mathtt{arg1}, \mathtt{X})$ from *Sample Buffer*

The first of these adds a single element to *Sample Buffer* (subject to the buffer's capacity limitations: see below). The second clears all elements from the buffer. The third deletes a single element from the buffer (provided that element matches the template given for deletion). The fourth deletes from the buffer all elements that unify with the given template.

There are five types of buffer, each having different behaviour. They all share the following six properties:

Initialise [Each Trial/Block/Subject/Experiment/Session; default: Each Trial]: The timing of buffer initialisation. If the value is Each Trial, the buffer will reload its initial contents at the beginning of each trial. If the value is Each Block, buffer contents will be preserved across trials, but re-initialised at the beginning of each block. Analogous behaviour is achieved with the other settings.

Decay [None/Half Life/Linear/Fixed; default: None] The decay function (if any) that should operate over the buffer's contents. When the value is None, elements will remain in a buffer until they are explicitly deleted, forced out by the buffer overflowing, or the buffer is re-initialised. When the value is Half Life, elements decay in a random fashion, but with the probability of decay being constant on each cycle. This probability is specified in terms of a half life (specified by the Decay Constant property). The half life is the number of cycles it takes on average for half of the elements to decay. The buffer's contents also decay in a probabilistic manner when the value is Linear, but in this case the probability of an element decaying within n cycles of entering the buffer is directly proportional to n, with the rate of decay again determined by the Decay Constant property. When the value is Fixed, elements remain in the buffer for a fixed number of cycles (specified by the Decay Constant property).

Decay Constant [1–9999; default: 20]: The decay rate (if Decay is not None). If the preference's value is r, then: for Half Life decay the probability that a given element decays on a given cycle is constant, and the probability of decaying within r cycles is 0.5; for Linear decay the probability that a given element decays within n cycles of being added is $\frac{n}{r}$ (for $n \leq r$); for Fixed decay the probability that a given element decays within n cycles of being added is 0 unless $n = r$, in which case it is 1. In all cases, the larger the constant the slower the decay.

Limited Capacity [No/Yes; default: No]: A switch that determines if the buffer's capacity should be limited. If Yes, the maximum number of elements that may be stored in the buffer at any time is specified by the Capacity property (below) and the behaviour when the buffer overflows is specified by the On Excess property. If No, the Capacity and On Excess properties are ignored.

Capacity [1–9999; default: 7]: Specifies the capacity of a buffer in terms of the number of items it may hold. This property has no effect if Limited Capacity is No.

On Excess [Random/Youngest/Oldest/Ignore; default: Random]: Determines the buffer's behaviour when its capacity is reached and an attempt is made to add a new element. If set to Ignore, new elements will be ignored (until the buffer's capacity decreases). If set to Random, a random element will be deleted from the buffer to make way for the new element. If set to Youngest, the most recently added element will be deleted to make way for the new element. If set to Oldest, the least recently added element will be deleted to make way for the new element. This property has no effect if Limited Capacity is No.

C.3 Buffer/Propositional

Propositional buffers are a subclass of buffers that store propositions or terms, with each term corresponding to a distinct buffer element. There are no restrictions on the kinds of terms that may be stored in a propositional buffer. As such, they may be used for general purpose storage.

Propositional buffers inherit all of the properties of the parent buffer class (Initialise, Decay, Decay Constant, Limited Capacity, Capacity and On Excess). They have two additional properties:

Access [Random/FIFO/LIFO; default: Random]: The order in which the buffer's elements are accessed by match operations. If access is Random, then match will instantiate its argument to an element from the buffer selected at random (provided that element unifies with the argument of match). If access is FIFO (First In, First Out), then match will return the oldest element that unifies with its argument. If access is LIFO (Last In, First Out), then match will return the youngest element that unifies with its argument.

Duplicates [No/Yes; default: No]: A switch that determines the buffer's behaviour when it receives an add message for an item already in the buffer. If Yes, the item is simply added to the buffer according to the usual rules (which depend on the other properties of the buffer). If No, the item is not added, but its previous occurrence will be "refreshed". (This is actually effected by deleting the previous occurrence before adding the new occurrence.) The procedure that tests if an element is already in the buffer requires that the given element match term for term and variable for variable an existing term. It is not simple term unification. Hence for the purpose of duplicate testing the elements `item(X)` and `item(cat)` are different, but the elements `item(X)` and `item(Y)` are the same.

Propositional buffers may also be augmented with display rules. These rules play no part in model performance, but allow propositional buffer elements to be rendered in a graphical form. This may simplify interpretation of the model's behaviour. Display rules are similar to standard process rules (see Section C.9), except that their conditions can match only elements in the buffer to which the rules are attached, and their actions must all take the form:

show `X`

where `X` is specifies a two-dimensional graphical element (e.g., a line, box, circle, etc.) Thus, a display rule may take the form:

Display Rule 2: *Represent buffer elements graphically*
IF: `edge(N0, N1, C, L)` is in *Chart*
`X0` is $\mathtt{N0} * 100 + 5$
`X1` is $\mathtt{N1} * 100 - 5$
`X` is $\mathtt{X0} + \mathtt{X1}/2$
`Y` is $\mathtt{L} * 25 + 40$
THEN: show `text(chart, C aligned (c, c) at (X, Y), [colour(black)])`
show `line(chart, (X0, Y) to (X1, Y), [colour(black)])`

Section C.5 provides details of the full set of possible graphical elements understood by COGENT. Any of these may be used within the show action of a display rule.

C.4 Buffer/Stack

A stack buffer is a kind of buffer that automatically orders its elements in terms of recency and limits access to the most recent (or "top-most") element. There are three operations that may be performed on a stack: the top-most element may be matched; a new element may be "pushed" onto the top of the stack (thus making the previous top-most element temporarily inaccessible); and the top-most element may be "popped" off of the stack (thus revealing a previously inaccessible element). The first of these is achieved through the standard match condition. The other two are achieved by sending special messages to a stack. Other buffer operations, such as add, delete and clear, may not be used with stacks.

An element may be pushed onto a stack by sending the stack a message of the form `push(Element)`. The top-most element may be popped from a stack by sending the stack a message of the form `pop`. Thus, the actions of a rule which pops the top element off *Goal Stack* and replaces it with `subgoal` might look like:

> send `pop` to *Goal Stack*
> send `push(subgoal)` to *Goal Stack*

Normally, the ordering of rule actions does not affect the order of execution of those actions: actions are normally effectively performed in parallel. However, messages sent to stacks are processed differently. If multiple `push`/1 and/or `pop`/0 messages are sent to the same stack from the action list of one rule, then those messages are processed in the order in which they appear on the action list. If multiple `push`/1 and/or `pop`/0 messages are sent to the same stack from multiple rules, then messages from the same rule will be processed in order, but messages from different rules may be processed in any order.

The current contents of stack buffers may be viewed in text mode or in a graphical mode. In text mode, the elements in the stack are displayed with the top-most element at the top of the text view. In graphical mode, the stack is displayed pictorially, with an arrow pointing to the top-most element. Both views may be printed by selecting Show current contents? on the print popup window.

Stack buffers inherit all of the properties of the parent buffer class (Initialise, Decay, Decay Constant, Limited Capacity, Capacity and On Excess), but have no additional properties.

C.5 Buffer/Analogue

Analogue buffers are specialised buffers whose elements may be interpreted as representing graphical objects. They may be displayed graphically, but they may also be accessed and modified with standard COGENT rules and conditions. They also have properties that relate specifically to the graphical or imagistic representation. They are probably most useful in models involving conceptions of time or imagery processes.

Analogue buffers may be either one-dimensional or two-dimensional. (Dimensionality is set via a property.) In both cases, elements are rendered graphically by a special purpose viewer, which is displayed by selecting the analogue buffer's Current Image view. By default the viewer colour-codes the objects, and displays a list of colour/object correspondences.

Analogue buffer elements take the following general form:

> `Type(Name, Geometry)`

where `Type` specifies the type of element (e.g., `point`, `line` or `circle`), `Name` is an identifier for the element, `Geometry` specifies the element's position.

For one-dimensional analogue buffers, elements must take one of the following forms:

`point(Name, Coord)`: A point at the specified coordinate. This, `point(foo, 15)` will be displayed as a point, 15 units (along the horizontal) from the zero position.

`x_mark(Name, Coord)`: The same as point but displayed as a small x.

`text(Name, Text aligned Alignment at Coord)`: A text marker at the specified coordinate. The alignment specifier, which is optional, may have one of the values `l` (left), `c` (centre) or `r` (right), and if omitted, defaults to left alignment.

`interval(Name, Start to End)`: An interval on the dimension starting at `Start` and ending at `End`. `Start` is constrained to be less than or equal to `End`. The interval is drawn as a line.

`line(Name, Coord1 to Coord2)`: A line between the coordinates. No restrictions on relative positions are imposed.

For two-dimensional analogue buffers elements must take one of the forms listed below. In all cases the X axis is horizontal, with zero at the left and increasing to the right. The Y axis is vertical with zero at the top and increasing to the bottom.

`point(Name, (X, Y))`: A point at the specified `(X, Y)` coordinates. For example, `point(foo, (15, 7))` draws a point 15 units from the left and 7 units from the top.

`x_mark(Name, (X, Y))`: The same as point but displayed as a small x.

`text(Name, Text aligned (Xalign, Yalign) at (X, Y))`: A text marker at the specified coordinates. The alignment argument, which is optional, specifies horizontal (`l`, `c`, `r`) and vertical (`b`, `c`, `t`) alignment of the text with respect to the coordinates. Thus, `text(foo, blah aligned (c, c) at (21, 7))` draws the string `blah`, centred horizontally and vertically, at the point (21,7).

`box(Name, (W, H) aligned (Xalign, Yalign) at (X, Y))`: A rectangle of width `W` and height `H` at the specified coordinates. The alignment argument is optional. Thus, `box(foo, (5, 10) aligned (l, t) at (21, 7))` draws a box 5 units wide and 10 high, whose left, top corner is located at the point (21,7).

`circle(Name, Radius at (X, Y))`: A circle with the given radius centred at the given `(X, Y)` location.

`line(Name, (X1, Y1) to (X2, Y2))`: A line between the coordinates.

`polyline(Name, ListOfPoints)`: A structure of linked lines joining the (X,Y) points in `ListOfPoints` in sequence. Thus, `polyline(foo, [(34, 27), (59, 70), (83, 36)])` draws one line between (34,27) and (59,70) and a second line between (59,70) and (83,36).

`polygon(Name, ListOfPoints)`: Like polyline except the last and first points in the list are joined to give a closed figure. Thus `polygon(foo, [(34, 27), (59, 70), (83, 36)])` draws a triangle.

Note that for one-dimensional analogue buffers the coordinate argument should be a single real number, but for two-dimensional analogue buffers the coordinate argument should be a pair of real numbers.

Both one-dimensional and two-dimensional graphical objects may be given a third, optional, style argument, consisting of a list of secondary property specifications. For example, a circle can be coloured red and filled by specifying it as follows:

```
circle(name, 50 at (100, 100), [colour(red), filled(yes)])
```

The following secondary properties may occur in the style list:

`colour(Colour)`: `Colour` may be any of the atoms `black`, `blue`, `green`, `brown`, `violet`, `grey`, `red`, `orange`, `yellow` or `white`, or an integer from 0–26. The object will be drawn using the specified colour, and will not be listed in the buffer's list of colour/object correspondences.

`filled(Boolean)`: (2D objects only) If `Boolean` is `true` or `yes` the object will be drawn filled; otherwise, it will be drawn as outline.

`style(Style)`: If `Style` is `dash`, `dashed` or `dashes`, lines within the object will be dashed. Otherwise they will be drawn as solid lines.

`weight(Weight)`: If `Weight` is a positive integer, lines will be drawn `Weight` pixels wide. The default `Weight` is 2.

Analogue buffers inherit all of the properties of the parent buffer class (Initialise, Decay, Decay Constant, Limited Capacity, Capacity and On Excess). They have several additional properties:

Dimensionality [1D/2D; default: 2D]: The number of dimensions available within the buffer. This determines the types of objects allowed, and their graphical interpretations.

Access [Random/FIFO/LIFO/Left→Right/Right→Left/Top→Bottom/Bottom→Top; default: Random]: The order in which the buffer's elements are accessed by match operations. Random, FIFO and LIFO have effects analogous to propositional buffers (see Section C.3). The directional options such as Left→Right and Top→Bottom will return elements in orders dependent on their spatial layout. Objects are accessed in the order in which they would be encountered, by scanning the geometrical space in the given direction. So for example, by scanning left to right, object A would be retrieved before object B if object A's leftmost point is farther left than object B's. The effects of directional scanning access are dependent on the buffer's dimensionality: in the 2D case, the directions correspond to the directions visible on the analogue buffer viewer's canvas, whereas in the 1D case, Left→Right and Top→Bottom are equivalent, and give the same results. Essentially, in the 1D case, left and top correspond to low values, whereas right and bottom correspond to high values.

Duplicates [No/Yes; default: No]: A switch that determines the buffer's behaviour when it receives an add message for an item already in the buffer. The possible behaviours are identical to those of propositional buffers (see Section C.3).

Continuity [No/Yes; default: No]: If Yes, the space is treated as being continuous in all dimensions. If No, the space is treated as being granular in all dimensions, and all coordinate values of all buffer elements are rounded to multiples of the value of Granularity.

Granularity [any positive real number; default: 1.0]: The grain size used for rounding when Continuity is No. For example, objects may be constrained to integer coordinates by setting Continuity to No and Granularity to 1.

Point Movement [No/Yes; default: No]: When point movement is selected, locations are adjusted with small random deviations on each processing cycle, so that over time, the contents of the buffer tend progressively to distort. This property reflects one view of an imagistic representational medium that is unable to represent location accurately. More generally, however, it may be used to provide a source of random variance in the graphical/imagistic domain. The amount of point movement is controlled by the Variance parameter.

Variance [any positive real number; default: 1.0]: The variance of the distribution of random point movement (assuming Point Movement is Yes). With Variance set to 1.0, approximately 68% of deviations will be less than or equal to 1 scale unit in magnitude. The effects of point movement interact with those of granularity, since with small variance values relative to the granularity values, the probability that a point will not deviate enough to reach the next grain is increased. As well as applying to all locations explicitly represented in objects in 1D or 2D buffers, point movement also applies to circles' radii, causing circles to change in size as well as to move.

Colour [True/False; default: True]: If True, display will be in colour, with a colour key showing object/colour correspondences. If False, all objects will be shown in black, and colour specifications via secondary properties will be ignored. It does not affect the way in which analogue buffer contents are processed.

X Scale [0.0–10.0; default: 1.0]: A scaling factor used purely for display purposes that stretches or compresses the display of the horizontal dimension. It does not affect the way in which analogue buffer contents are processed.

Y Scale [0.0–10.0; default: 1.0]: A scaling factor used purely for display purposes that stretches or compresses the display of the vertical dimension. It does not affect the way in which analogue buffer contents are processed.

C.6 Buffer/Table

Table buffers are a subtype of buffer in which elements represent entries in a two-dimensional table (see also Section C.13: Data/Sink/Table). All elements of a table buffer should consist of `data`/3 terms:

$$\texttt{data}(\texttt{Row}, \texttt{Col}, \texttt{Value})$$

Such terms are interpreted as indicating that the value of the cell within the table that is indexed by the values of `Row` and `Col` is `Value`. Each cell within a table may have at most one value. Thus, when a `data`/3 term is added to a table buffer, it over-writes any previous element with the same indices (i.e., the same values of `Row` and `Col`). Special purpose viewers are provided to display the contents of table buffers in tabular form, and this form may be printed by selecting Show current contents? on the print popup window.

Properties control aspects of table layout. Table buffers inherit all of the properties of the parent buffer class (Initialise, Decay, Decay Constant, Limited Capacity, Capacity and On Excess). They have five additional properties:

Access [Random/FIFO/LIFO/Left→Right/Right→Left/Top→Bottom/Bottom→Top; default: Random]: The order in which the buffer's elements are accessed by match operations. Random, FIFO and LIFO have effects analogous to propositional buffers (see Section C.3). The directional options such as Left→Right and Top→Bottom will return elements in orders dependent on the `Column` and `Row` indices respectively. So, for example, with a setting of Top→Bottom, elements from the top of the table will be returned before those from the bottom, and so on. In the spatial layout, rows and columns are ordered alphabetically by row/column label.

Column Label [an arbitrary sequence of letters; default: "Columns"]: The text used to label the table's columns when the table is viewed in Current Table mode.

Row Label [an arbitrary sequence of letters; default: "Rows"]: The text used to label the table's rows when the table is viewed in Current Table mode.

Cell Width [a positive integer; default: 70]: The width, in pixels, of each column when the table is viewed in Current Table mode.

Cell Height [a positive integer: default: 25]: The height, in pixels, of each row when the table is viewed in Current Table mode.

Sort : [Alpha/Reverse Alpha/Primacy/Recency]: The order in which rows and columns are displayed in the Current Table view. In the case of Alpha and Reverse Alpha the row and column headings are used to sort the rows and columns before the table as drawn. In the case of Primacy and Recency the time at which rows and columns were altered is used to sort them before the table as drawn.

C.7 Buffer/Graph

Graph buffers provide the functionality of a buffer with an option to view and print the buffer contents in graphical form. Elements of a graph buffer may take two forms: $\texttt{type}(\texttt{DataSet}, \texttt{Type}, \texttt{Properties})$ or $\texttt{data}(\texttt{DataSet}, \texttt{X}, \texttt{Y})$, where `DataSet` identifies a particular data set, `Type` specifies the type of graph required for that data set (see below), `Properties` is a list of secondary properties specifying aspects of the appearance of the particular data set (see below), and `X` and `Y` are numeric values for the data set. Graph buffer elements may be matched, added or deleted, in a way analogous to elements from other classes of buffer.

There is no limit to the number of data sets that may be stored in one graph buffer. Normally there will be one `type`/3 element and several `data`/3 elements for each data set. Multiple data sets within a single graph buffer are treated independently, with the limitation that the graph axes and labels are determined by graph properties (see below), and are hence shared by all data sets within a graph.

A `type`/3 term specifies the style or type of graph to be drawn for a given data set. The second argument must be one of `scatter`, `line`, or `bar`. The third argument species graph properties such as colour and marker style, in the form of a list of secondary properties (see Section C.5). Thus, the following `type`/3 element:

$$\texttt{type}(\texttt{frequency}, \texttt{bar}, [\texttt{colour}(\texttt{blue}), \texttt{fill}(\texttt{true})])$$

specifies that the graph associated with `frequency` should be a bar-chart drawn with blue filled bars. This could be changed to a line graph using red filled square markers (and a red line) by replacing the above element with:

$$\texttt{type}(\texttt{frequency}, \texttt{line}, [\texttt{colour}(\texttt{red}), \texttt{fill}(\texttt{true}), \texttt{marker}(\texttt{square})])$$

Three kinds of secondary property are recognised: `colour`, `fill`, and `marker`. There are four marker types: `square`, `circle`, `cross` and `plus`.

Graph buffers inherit all of the properties of the parent buffer class (Initialise, Decay, Decay Constant, Limited Capacity, Capacity and On Excess). They also have several additional properties that control access and appearance:

Access [Random/FIFO/LIFO; default: Random]: The order in which the buffer's elements are accessed by match operations. Interpretation of the property's values follows that of propositional buffers (see Section C.3).

Title [an arbitrary character string; default: "Title"]: The graph's title, which is centred above the graph when viewed in Current Graph mode or when the graph is printed.

X Label [an arbitrary character string; default: "X"]: The label drawn beside the graph's horizontal axis.

X Min [a real number; default: 0.0]: The minimum value of the horizontal coordinate of the graph.

X Max [a real number; default: 10.0]: The maximum value of the horizontal coordinate of the graph.

X Units [a positive integer: default: 5]: The number of units into which the horizontal axis of the graph is divided.

Y Label [an arbitrary character string; default: "Y"]: The label drawn beside the graph's vertical axis.

Y Min [a real number; default: 0.0]: The minimum value of the vertical coordinate of the graph.

Y Max [a real number; default: 100.0]: The maximum value of the vertical coordinate of the graph.

Y Units [a positive integer: default: 5]: The number of units into which the vertical axis of the graph is divided.

C.8 Knowledge Base

Knowledge bases are similar to buffers, except that they are not subject to decay or capacity limitations, and they may, if their properties are set appropriately, be accessed from other boxes without appropriate arrows. Their behaviour is determined by four properties:

Initialise [Each Trial/Block/Subject/Experiment/Session; default: Each Trial]: The timing of knowledge-base initialisation. The effects of the various values of this property on knowledge bases are the same as the identical values on buffer initialisation (see Section C.2).

Access [Random/FIFO/LIFO; default: Random]: The order in which knowledge base elements are accessed by match operations. The effects of the various values of this property on knowledge bases are the same as the identical values on propositional buffer matching (see Section C.3).

Globally Readable [True/False; default: False]: It set, the knowledge base can be read by any box (or subbox) of its parent compound, without the requirement of a read arrow from the box doing the reading to the knowledge base.

Globally Writeable [True/False; default: False]: It set, the knowledge base can be written to by any box (or subbox) of its parent compound, without the requirement of a write arrow from the box doing the writing to the knowledge base.

Knowledge base elements take the same form as buffer elements, and they are edited with the standard buffer element editor. However, it is anticipated that future versions of COGENT may include knowledge engineering tools for maintaining knowledge bases.

C.9 Process

A (rule-based) process is a box that manipulates and transforms information according to a set of rules and condition definitions. The rules contained within a process generate messages (often in response to messages received by the process), and send those messages to other boxes, triggering processing elsewhere in a model. A process' behaviour is determined by its contents (i.e., the rules and condition definitions within it) and the values of its three properties:

Initialise [Each Trial/Block/Subject/Experiment/Session; default: Each Trial]: The timing of process initialisation. Initialisation determines the scope of refractory tests, such that refracted rules fire at most once for any particular binding of variables within the specified initialisation level (trial, block, subject, etc.).

Recurrent [No/Yes; default: No]: A switch that specifies if a process can send messages to itself. By default processes receive information, process it, and send it on. Recurrent processes are able to feed their output back into themselves.

Firing Rate [0.0–1.0; default: 1.0]: The probability of a process' rules firing when their conditions are satisfied. By default a rule will fire whenever its conditions hold, but a process may be degraded by setting its firing rate to (for example) 0.8, meaning that rules will only fire on 80% of occasions on which their conditions are satisfied.

COGENT's rule language is described further in Section E.

C.10 Data/Source

Data sources provide a very simple means of feeding a fixed sequence of messages to other boxes within a model at fixed points during processing. Their usual function is to act as input devices, where they can feed a stream of input to the model. Any Prolog term can be passed as a message, so data sources can be used to feed information of arbitrary complexity into a model. They may also send multiple messages on a single cycle (including messages to different boxes). Thus, they can be used to trigger rules in a process, or to add elements to a buffer.

The behaviour of data sources is governed by one property:

Initialise [Each Trial/Block/Subject/Experiment/Session; default: Each Trial]: The timing of data source initialisation. Initialisation resets the sequence of input messages. This reset may occur at the beginning of each trial, block, subject, etc..

Data sources are of limited use because of the fixed nature of the input that they generate. For models with more sophisticated input demands, a compound box containing a propositional buffer and associated process may be more appropriate.

C.11 Data/Sink

Data sinks are the output equivalent of data sources. They may be used to collect and view model output. There are three forms of data sink: text data sinks, tabular data sinks and graph data sinks. There is no limit to the number of data sinks you can have in one model, and multiple data sinks behave as independent devices. The details of their behaviour are governed by three properties:

Initialise [Each Trial/Block/Subject/Experiment/Session; default: Each Trial]: The timing of data sink initialisation. The value specifies the point during model execution when existing data sink elements should be cleared.

File [an arbitrary character string; default "data.current"]: The name of the output file in which data should be stored.

Location [Local/IO Directory; default: Local]: The location of the output file (named in the above property). Sink output is normally stored locally (with the definition of the sink), but it may also be directory to the standard I/O directory (see Section A.1), from where it may be retrieved, e.g., for inclusion in a document.

C.12 Data/Sink/Text

Text data sinks are used for collecting text-based output from a model. Messages sent to a text data sink are saved in a file (as specified by the sink's properties) and displayed as text. Text data sinks inherit all of the properties of the parent data sink class (Initialise, File and Location), but have no additional properties.

C.13 Data/Sink/Table

Tabular data sinks accumulate output (in the same way as other data sinks) and allow that output to be displayed (and printed) as a table. The contents of messages sent to a graphical data sink are interpreted by the table viewer in the same way as elements in a tabular buffer (see Section C.6): they should all be of the form `data(Row, Col, Value)`, where `Row` and `Col` identify a particular cell within the table and `Value` specifies the value in that cell. If two messages are received by a tabular data sink that specify different values for the same cell then the second value will over-write the first.

Tabular data sinks inherit all of the properties of the parent data sink class (Initialise, File and Location), and several additional properties (Column Label, Row Label, Cell Width and Cell Height) that govern the appearance and layout of the table view. The interpretation of these layout properties is identical to that of the equivalent properties in tabular buffers (see Section C.6).

C.14 Data/Sink/Graph

Graphical data sinks accumulate output (in the same way as other data sinks) and allow that output to be displayed (and printed) as a graph. The contents of messages sent to a graphical data sink are interpreted by the graph viewer in the same way as elements in a graphical buffer (see Section C.7): they should be either of the form `type(DataSet, Type, Properties)` or `data(DataSet, X, Y)`, where `DataSet` identifies a particular data set, `Type` specifies the type of graph required for that data set, `Properties` is a list of secondary properties specifying aspects of the appearance of the particular data set, and `X` and `Y` are numeric values for the data set.

Graphical data sinks inherit all of the properties of the parent data sink class (Initialise, File and Location), and several additional properties (Title, X Label, Y Label, X Min, X Max, X Units, Y Min, Y Max and Y Units) that govern the appearance and layout of the graph view. The interpretation of these layout properties is identical to that of the equivalent properties in graphical buffers (see Section C.7).

C.15 Network/Feed-Forward

Feed-forward network consists of a set of input nodes and a set of output nodes, together with a set of weighted connections between the nodes. Feed-forward networks map input vectors to output vectors, and are able to learn input/output correspondences (subject to the well-known perceptron learning limitations: Minsky & Papert, 1988). Properties govern aspects of feed-forward networks' behaviour, such as the number of input and output units, the learning rule and rate, etc.

Feed-forward networks may be used in two ways. They may be trained (by sending appropriate `train`/2 messages), and they may be tested (by matching with a `test`/2 term). The following match condition may be used to test a feed-forward network:

`test(InVector, OutVector)` is in *Sample Net*

This condition, which may occur within a rule contained in a process, will succeed if `InVector` is a list of N numbers and N is the width of the input layer of *Sample Net*. It will bind `OutVector` to a list of M numbers where M is the width of the output layer of *Sample Net* and `OutVector` is generated by feeding `InVector` to the weight matrix of *Sample Net*.

A feed-forward network may be trained by sending it a training message:

send `train(InVector, OutVector)` to *Sample Net*

where `InVector` is a list of N numbers and N is the width of the input layer of *Sample Net* and `OutVector` is a list of M numbers and M is the width of the output layer of *Sample Net*. Such a message will result in the feed-forward network's weight matrix being adjusted such that the next time it is tested with `InVector` it will generate an output that more nearly approximates `OutputVector`. If a network receives several training messages on the same processing cycle, then all such messages are processed (in pseudo-parallel) and a single weight adjustment (corresponding to the mean of the individual adjustments) will be made.

The properties that control feed-forward network behaviour are:

Initialise [Each Trial/Block/Subject/Experiment/Session; default: Each Trial]: The timing of network initialisation. This parameter determines the scope of learning. Thus, if set to Each Block, learning will be confined to within block, but not between block.

Initial Weights [uniform/normal: default: uniform]: The shape of the initial weight distribution function. If set to uniform, then on initialisation weights will be randomly selected from a uniform (i.e., flat) distribution, with lower and upper limits determined by Weight Parameter A and Weight Parameter B. If Initial Weights is set to normal, then on initialisation, weights will be randomly selected from a normal distribution, with mean and standard deviation determined by Weight Parameter A and Weight Parameter B.

Weight Parameter A [any real number; default: −1.00]: If Initial Weights is set to uniform, this specifies the lower limit of the weight distribution. It Initial Weights is set to normal, this specifies the mean of the weight distribution.

Weight Parameter B [any real number; default: 1.00]: If Initial Weights is set to uniform, this specifies the upper limit of the weight distribution. It Initial Weights is set to normal, this specifies the standard deviation of the weight distribution.

Input Width [a positive integer; default: 10]: The number of nodes in the network's input layer.

Output Width [a positive integer; default: 10]: The number of nodes in the network's output layer.

Connectivity [0.0–1.0; default: 1.00]: The proportion of possible connections between input and output nodes that are actually present in the network. Thus, a connectivity of 0.50 means that, on average, each input node is connected to only one half of the output nodes.

Act Function [linear/sigmoidal; default: linear]: The form of activation function that is applied to weighted inputs to determine network output. Two basic functions are available: linear, which employs a piecewise linear squashing function, such that 1) inputs below one threshold are mapped to the minimum activation, 2) inputs greater than a second threshold are mapped to the maximum activation, and 3) inputs between the thresholds are mapped linearly between the activation limits; and sigmoidal, which employs a non-linear function based on the sigmoid or logistic equation, and mapping 1) low inputs to values near the minimum activation, 2) high inputs to values close to the maximum activation, and 3) intermediate inputs non-linearly between the activation limits.

Max Act [any real number; default: 1.00]: The maximum activation that an output node may achieve (as above).

Min Act [any real number; default: −1.00]: The minimum activation that an output node may achieve (as above).

Act Midpoint [any real number; default: 0.00]: The weighted sum of inputs that should map to half way between Min Act and Max Act. This property partially determines the activation function, but is independent of the form of that function (i.e., linear or sigmoidal).

Act Slope [any real number; default: 1.00]: The gradient of the activation function when the input to that function is Act Midpoint. This, together with the above parameters, fully specifies the activation function in such a way that selecting different values for Act Function will preserve the basic characteristics of that function (i.e., its steepness and its effects on extreme inputs).

Learning Rule [delta/Hebbian; default: delta]: The learning rule to use in weight adjustment. Learning may be either through application of the delta-rule or through Hebbian weight adjustment. In each case the learning rate is determined by a separate parameter.

Learning Rate [any real number; default: 0.10]: The rate of learning. In general, a high learning rate will mean that the weight matrix responds more quickly to input-output training pairs, but may result in the network being insufficiently sensitive to its past training history.

Weight Decay [True/False; default: False]: If set, weights will gradually "decay" in the absence of learning through the addition of random noise.

Decay Parameter [any real number; default: 0.00]: If Weight Decay is True, this parameter specifies the standard deviation of noise added to each weight on each processing cycle.

C.16 Network/Interactive Activation

Interactive activation network boxes contain nodes with associated activation values. Nodes may be created, deleted, excited or inhibited, and their activation values may be queried.

The initial contents of an interactive activation network are the set of names of nodes initially contained in that network. Properties determine the distribution of initial activation values. Further nodes may be added through the use of `add` actions:

add `NodeName` to *Network*

Similarly, nodes may be deleted through the use of `clear` actions, `delete` actions, and `delete all` actions:

clear *Network*

delete `NodeName` from *Network*
delete all `NodeName` from *Network*

The first of these removes all nodes from a network. The second deletes one node (whose name unifies with `NodeName`). The third deletes all nodes whose names unify with `NodeName`.

Interactive activation networks will normally co-exist with a process that sends appropriate `excite` messages to control the node activations:

send `excite(NodeName, Excitation)` to *Network*

On each processing cycle, the net excitation to each node within the network is summed. This net excitation is then used, along with the current activation and network properties, to determine the new activation of each node.

In addition to external input, node activations may be subject to internal competition from lateral inhibition and self activation. Each of these can be configured by additional properties, allowing them to be enabled separately, scaled appropriately, and to use alternative baselines. In addition, lateral inhibition can be configured to use any of several different functions, and can be calculated over the whole network or within sub-networks that partition the set of nodes in the box into several effectively separate networks.

If sub-network competition is enabled, `NodeName` must be specified as a structure containing a slash operator ("/"), in which the term prior to the slash is the name of the node, and term after the slash is the name of the sub-network, e.g. `mynode/subnet1`. This node will then only compete with others belonging to the `subnet1` sub-network.

Node activations within an interactive activation network may be queried by matching against node names:

`node(Name, Activation)` is in *Network*

In the above condition, `Activation` will normally be an uninstantiated variable, and execution of the condition will bind that variable to the activation of the node whose name is bound to `Name`.

The properties that govern the behaviour of interactive activation networks may be broken into several categories:

C.16.1 General Properties

Max Act [any real number; default: 1.00]: The maximum activation that any node may obtain.
Min Act [any real number; default: −1.00]: The minimum activation that any node may obtain.
Rest Act [any real number; default: 0.00]: The rest activation to which nodes revert in the absence of excitation.
Persistence [any real number; default: 0.90]: The degree to which activation values persist in the absence of excitation. A persistence of 1.00 will lead to nodes maintaining their current activation in the absence of any excitation. A persistence of 0.00 will lead to nodes reverting immediately to rest activation in the absence of excitation.
Noise [any real number; default: 0.00001]: The variance of normally distributed noise affecting the net excitation of all nodes.
Update Function [MR/GH/CS; default: MR]: The activation function used to calculate the result of excitation and inhibition on each node on each cycle. Several update functions used in the literature are available, including:

- MR: McClelland & Rumelhart (1981)
- GH: Houghton (1990)
- CS: Cooper & Shallice (2000)

Each of these functions is calculated with respect to Min Act, Max Act, Rest Act and Persistence.

C.16.2 Initialisation Properties

Initialise [Each Trial/Block/Subject/Experiment/Session; default: Each Trial]: The timing of network initialisation. When the value is Each Trial, the network automatically initialises itself at the beginning of each trial. Similarly for other values. Thus, if the value is Each Subject, activations will be preserved within subjects but re-initialised with each new subject.

Initial Acts [uniform/normal: default: uniform]: The shape of the initial activation distribution function. If uniform, then on initialisation activations are randomly selected from a uniform (i.e., flat) distribution. The parameters that govern that distribution are determined by Act Parameter A and Act Parameter B. If Initial Acts is normal, then on initialisation activations are randomly selected from a normal distribution. Again, the parameters that govern that distribution are determined by Act Parameter A and Act Parameter B.

Act Parameter A [any real number; default: -1.00]: If Initial Acts is set to uniform, this specifies the lower limit of the activation distribution. It Initial Acts is set to normal, this specifies the mean of the activation distribution.

Act Parameter B [any real number; default: 1.00]: If Initial Acts is set to uniform, this specifies the upper limit of the activation distribution. It Initial Acts is set to normal, this specifies the standard deviation of the activation distribution.

C.16.3 Self Activation Properties

Self Influence [True/False; default: False]: If Self Influence is True, nodes will be subject to self activation, which is calculated using the Self Parameter and Self Baseline properties as described below.

Self Parameter [any real number; default: 0.50]: A scaling factor which is multiplied with the self input (see Self Baseline) to give the self activation.

Self Baseline [Min Act/Rest Act; default: Rest Act]: The baseline with respect to which self activation is calculated. If Rest Act is selected, the value of Rest Act is subtracted from the node's current activation to give the self input. In this case self input can be positive or negative, depending on whether the node's activation is greater or less than Rest Act. If Min Act is selected, the value of Min Act is subtracted from the node's current activation to give the self input, so self influence is always positive.

C.16.4 Lateral Inhibition Properties

Lateral Influence [None/Whole Net/Sub Net; default: None]: If None is selected, lateral influence is disabled. If Whole Net is selected, all nodes in the box contribute to each node's inhibition. If Sub Net is selected, each node is inhibited only by other members of its sub-network.

Lateral Parameter [any real number; default: 0.50]: A scaling factor that is multiplied with the output of Lateral Function to give the lateral inhibition on each node.

Lateral Function [Sum/Mean/Max; default: Sum]: A function that determines how lateral influence is calculated from the influences of individual competitor nodes. If Sum is selected, influences from competitors are summed. If Max is selected, the maximum individual influence is used. If Mean is selected, influences are averaged.

Lateral Baseline [Min Act/Rest Act; default: Rest Act]: As with Self Baseline, Lateral Baseline switches between using alternative baseline values for the calculation of lateral influence. To calculate the lateral influence, current activations of competitors are subtracted from the baseline (either Rest Act or Min Act). If the baseline is Rest Act, then lateral influences can be either inhibitory or excitatory. If the baseline is Min Act, then lateral influences will always be inhibitory.

D Representation and Variable Binding

D.1 Representing Information

Any information processing model requires that the information to be processed is represented in a consistent way so that the model can apply well-defined mechanisms to process or manipulate the representa-

Table 2: Types of term within COGENT's representation language

Type	Description	Examples
Number	Represents numeric information	`3` 12.00 `−7.24`
Atom	Represents information with no internal structure	`column` `x23` `cat`
Variable	Represents information that is unknown or that varies	`X` `Time` `Height`
List	Represents sequences of information	`[mouse, cat, dog, horse]` `[london]`
Compound	Represents complex information with arbitrary internal structure	`goal(subtract(c2, c1))` `features(X, [legs(4)])`

tions. Within COGENT it is necessary to be able to represent, for example, the information contained within buffers and the content of messages that pass between ¡components. COGENT's representation language is borrowed from Prolog, a programming language originally developed for artificial intelligence applications. The following sections give an overview of the language. More information may be obtained from any standard Prolog text (e.g., Bratko, 1986; Clocksin & Mellish, 1987).

The principal representational unit of COGENT (and Prolog) is the *term*. All information that is to be represented must be represented as a term. There are a number of different types of term, allowing the representation of a number of different types of information. Table 2 gives a brief description and examples of each type. Any or all of these terms could appear within COGENT as, for example, an element within a buffer or the content of a message.

D.1.1 Numbers

Numbers are represented within COGENT using the standard notation consisting of digits and an optional decimal point. Following Prolog, COGENT treats integers (e.g., `9`) and real numbers (e.g., `3.14159`) slightly differently. Be aware that, for example, `6` and `6.0` are *not* identical.

D.1.2 Atoms

Atoms are generally used to represent atomic things. That is, to represent symbols that have no internal structure (or whose internal structure is not relevant to the current task). Any unbroken sequence of letters or other characters that begins with a lower-case letter (such as `cat`) is interpreted by COGENT to be an atom, provided that the characters following the first letter are upper-case letters, lower-case letters, digits, or the underscore character ("_"). Thus `dog`, `four_legs`, and `a_X3bu` are all atoms. Other combinations of characters can also be made into atoms by enclosing them in single quotation marks. The following are thus also atoms: `'CAT'`, `'four-legs'`, `'A&B'`.

D.1.3 Variables

Variables allow the representation of information that is either unknown or that may vary. Like atoms, variables are denoted by sequences of letter, digits, and underscore characters, but variables must begin with an upper-case letter or the underscore character. Thus, `CAT`, `Rat_4`, and `_myvar` are all variables. Note that variables must not have quotation marks around them: a character sequence beginning with an upper-case letter that is surrounding by single quotes is understood by COGENT (and Prolog) to be an atom.

D.1.4 Lists

Much of the power of the representation language comes from the possibility of constructing new terms from other elements of the language. Lists are one type of term that employs this construction. Lists are generally used to represent sequences of information. Thus, a list might be an appropriate representation to use when modelling a task involving a sequence of activities, where order in the sequence is important.

Syntactically, a list consists of a left square bracket followed by a comma-separated sequence of terms followed by a right square bracket. Thus, $[\mathtt{cat}, \mathtt{elephant}, \mathtt{fish}, \mathtt{lion}, \mathtt{dog}, \mathtt{fish}]$ is a list with six atomic elements. Lists can have any number of elements, but the list with zero elements is special. It is known as the empty list, and represented as $[\,]$.

A second common use of lists is to represent sets of things (or even multi-sets: sets whose elements may occur more than once). This can be done by simply ignoring the sequential ordering information contained in the list representation. Thus, the list $[\mathtt{cat}, \mathtt{elephant}, \mathtt{fish}, \mathtt{lion}, \mathtt{dog}]$, may be used to represent a set, rather than a sequence, of animal names, provided that the functions and processes that operate on the representation do not make use of order information contained within the list.

The elements of a list need not be atoms — they may be terms of any type. Hence, variables and lists may occur as elements of a list. Thus, $[\mathtt{cat}, \mathtt{ANIMAL}, \mathtt{fox}, [\mathtt{rabbit}, \mathtt{rat}, \mathtt{mouse}]]$ is a list whose second element is a variable, whose first and third elements are atoms, and whose fourth element is itself a list.

D.1.5 Compound Terms

Compounds terms are, like lists, terms built from other terms. They are frequently used to represent structured information in which the structure is more complex than that which occurs in lists. Compound terms allow, for example, the representation of the meaning of sentences in terms of representations of the sentence parts. Thus, the meaning of "Tigger is miaowing" might be represented by the compound term $\mathtt{miaows}(\mathtt{tigger})$.

In general, a compound term consists of an atom (in the above case $\mathtt{miaows}$) immediately followed by a left round bracket followed by a comma-separated sequence of other terms, followed by a right round bracket. The initial atom is referred to as the compound term's *functor*. The sub-terms between a compound term's brackets are known as its arguments, and the number of arguments is the term's *arity*. Note that the comma-separated sequence of terms cannot be empty (i.e., the arity of a compound term cannot be 0), and there must not be any space between the compound term's functor and the opening round bracket. Space may be inserted freely between a compound term's arguments (or between those arguments and the commas that separate them), and should be used consistently to improve the readability of the representation.

The example compound terms given in Table 2 illustrate that compound terms may be embedded (i.e., an argument of a compound term may itself be a compound term), or contain lists and variables within their arguments. Highly complex representations may be built by using this structuring of terms.

D.1.6 Operators

In the language as described so far, a compound term representing a simple arithmetic expression (e.g., 3 + 4) must be written using a very clumsy notation: $'+'(3, 4)$. This is a compound term whose functor is $'+'$ and whose arity is 2. The representation language allows some compound terms (especially arithmetic expressions) to be written in a more readable way through the use of operators.

Certain pre-defined functors are understood by COGENT/Prolog to be operators. If a functor is a *binary operator*, then a term of the form $'+'(3, 4)$ can be written in the conventional way, as $3 + 4$. Note that the brackets and the single quotes are not required when the alternate notation is used. Operators may be used in all kinds of terms (not just arithmetic expressions). Thus, $\mathtt{has} - \mathtt{fur}$ is the same as $'-'(\mathtt{has}, \mathtt{fur})$, and $\mathtt{a}/\mathtt{b}$ is the same as $'/'(\mathtt{a}, \mathtt{b})$.

The set of pre-defined operators includes all of the standard arithmetic operators ($+$, $-$, $*$, $/$, $>$ and $<$). These operators can be used in complex expressions, and when used in such expressions they have the usual precedences. Thus, $3 + 4 * 5$ is is a compound term with arity 2 and functor "$+$". The second argument of this term is $4 * 5$, itself a compound term. Precedence can be over-ridden by using round brackets. Thus $(3 + 4) * 5$ is is a compound term with arity 2 and functor "$*$". The first argument of this term is $3 + 4$, again a compound term.

A second common use of operators within COGENT is in specifying positional information for objects within analogue buffers. As described elsewhere, analogue buffers contain terms that represent objects located in one-dimensional or two-dimensional space. A graphical object may be centred (or aligned vertically or horizontally) by using the `aligned` operator in the object's position specification:

```
text("Centred text", (300, 200) aligned (c, c), [colour(green)])
```

D.2 Unification: Matching and Variable Binding

The combination of compound terms and variables provides a general, expressive representational system. The representational system also provides a mechanism for matching terms (e.g., in the triggering patterns and conditions of rules) and binding variables. Consider the Rule 1 of Figure 27, which comes from a production system interpreter and which operates on two buffers, *Productions* and *Matches*. Suppose that, at some point in processing, *Productions* contains the term:

```
prod([subtrahend(0)], [do(copy_minuend)]
```

The first condition of Rule 1 may be satisfied if the compound term `prod(C, A)` may be matched with the element. This in turn involves mapping or binding the variable `C` to `[subtrahend(0)]` and the variable `A` to `[do(copy_minuend)]`. The operation in which `prod(C, A)` is matched to a buffer element to produce this binding is known as unification. Table 3 shows several examples of unification between terms and the resultant variable bindings. Notice that, in the case of compound terms and lists, unification is a recursive process: two compound terms unify if they have the same functor and arity and each of their arguments unify; two lists unify if they have the same length and each of their arguments unify.

Rule 1 (unrefracted): *Add matching production instances to match memory*
IF: `prod(C, A)` is in *Productions*
`preconditions_hold(C, M)`
THEN: add `prod(C, A, M)` to *Matches*

Rule 1 with variables partially instantiated:
IF: `prod([subtrahend(0)], [do(copy_minuend)]` is in *Productions*
`preconditions_hold([subtrahend(0)], M)`
THEN: add `prod([subtrahend(0)], [do(copy_minuend)], M)` to *Matches*

Figure 27: A sample rule from a production system interpreter, and a partially instantiated instance of the rule

Returning to Rule 1 of Figure 27, unification of the rule's first condition with the element in *Productions* effectively yields a new instance of the rule, as shown in the lower half of the figure. In this instance the variables `C` and `A` have been instantiated with the terms resulting from the successful match against *Productions*. The rule's second condition inherits the instantiation of the variables, and so is more specific than in the abstract rule. The rule's action also inherits the instantiation of variables, and is similarly more specific. In order for this instance of the rule to fire, however, the second (more specific) condition must also be satisfied. This condition may lead to the variable `M` becoming instantiated before the rule fires. Alternatively, the second condition may fail (i.e., there may be no way in which the variables in the condition may be successfully instantiated, given the instantiation of `C`). In this case the rule will not fire.

The buffer that is being matched (*Productions*) may also contain multiple elements that match the rule's first condition. In such cases each matching element leads to a separate instance of the rule, differing only in the terms to which the variables `C` and `A` are instantiated. All instances of all rules are normally considered on each COGENT processing cycle (unless the rule is explicitly marked to fire at most once on any cycle). Thus, a single rule may fire multiple times on the same cycle, each time with a different instantiation. (COGENT rules may therefore be understood as statements of predicate logic, in which the variables are implicitly universally quantified.)

Table 3: Examples of terms, their unification, and the resultant variable bindings

Terms	Their Unification	Variable Bindings

`X`
`word(green)`
	`word(green)`	`X` ↦ `word(green)`

`f(alpha, B, gamma)`
`f(A, beta, G)`
	`f(alpha, beta, gamma)`	`A` ↦ `alpha`
		`B` ↦ `beta`
		`G` ↦ `gamma`

`f(X)`
`g(X)`
	Unification fails	

`f([a, B], [])`
`f([B, a], M)`
	`f([a, a], [])`	`B` ↦ `a`
		`M` ↦ `[]`

`f([a, B], [])`
`f(L, B)`
	`f([a, []], [])`	`B` ↦ `[]`
		`L` ↦ `[a, []]`

`f([a, B], [])`
`f([B, B], B)`
	Unification fails	

E The Rule Language

There are two kinds of rules that may be used for manipulating information and generating messages: triggered rules (consisting of a triggering patter, a list of conditions and a list of actions), and autonomous rules (consisting of a list of conditions and a list of actions).

A triggered rule is activated whenever the process that contains the rule receives a message that unifies with the rule's triggering pattern. If, once activated, the rule's conditions are satisfied, then the rule fires, with all of its actions being performed in pseudo-parallel. Autonomous rules do not have a triggering pattern. They fire whenever their conditions are satisfied. Triggered and autonomous rules may be mixed freely within a single process.

Triggered rules are normally triggered by messages generated by the firing of other rules. There is one special trigger, however, that is automatically generated by COGENT. At the end of each trial, block, subject and experiment, COGENT automatically generates a message of the form `system_end(Level)` (for `Level` equal to `trial`, `block`, `subject` or `experiment`, respectively). These triggers are intended for use by task environment boxes, for example to trigger statistical analysis at the end of a trial, block, subject or experiment.

Each condition in a rule's list of conditions is a logical test that may include unbound variables. COGENT provides a wide range of built-in conditions for operating on list representations, evaluating arithmetic expressions, etc. (see Section F), but additional conditions may be defined within a process (using the condition editor) and then accessed within rules.

A rule's conditions are satisfied with a particular binding of variables if each each condition in its condition list is true for the given variable binding. Thus, the single condition:

> `X` is a member of `[cat, dog, mouse]`

may be satisfied in three ways (with `X` bound to either `cat`, `dog` or `mouse`), but the pair of conditions

> `X` is a member of `[cat, dog, mouse]`
> `X` is a member of `[rat, mouse, gerbil]`

may only be satisfied with `X` bound to `mouse`.

By default, COGENT will consider all possible variable bindings of each condition. However, conditions may also be qualified in several ways to limit the firing of rules:

not *condition*: Succeeds if and only if *condition* is false.

exists *condition*: Succeeds if and only if there is at least one way of binding variables such that *condition* succeeds. Uninstantiated variables within *condition* will not be bound even if the qualified condition succeeds.

once *condition*: Succeeds if and only if there is at least one way of binding variables such that *condition* succeeds. Unlike exists, uninstantiated variables within *condition* will be bound.

unique *condition*: Succeeds if and only if *condition* has a unique solution, i.e., if and only if there is a unique way of successfully binding any uninstantiated variables within *condition*. Variables will be bound only if the qualified condition succeeds.

trace *condition*: Succeeds if and only if *condition* succeeds, with the side-effect of printing the condition and its variable bindings to the process box's Messages window. This qualifier is for debugging purposes only. It has no effect on model execution.

Qualifiers may also be applied to sequences of conditions. Thus, not *condition1 condition2* will succeed if and only if there is no variable binding that simultaneously satisfies both *condition1* and *condition2*.

If a rule's condition list is satisfied by a particular binding of variables (and two other criteria described in the next paragraph are met), the rule will fire. This involves binding any variables in the actions and generating messages corresponding to each action in the rule's action list. These messages will then be sent to the boxes listed in the actions. If any message sent to a box consists of the single term `stop`, then when the target box receives that message its processing will be completely halted until the next initialisation. If the box is a compound then all boxes within the compound will be stopped. The `stop` message takes effect immediately, overriding any other messages sent to a box on the same cycle.

In addition to the above, rules may be marked as refracted and/or as firing once per cycle. Refracted rules fire only once with each pattern of instantiations of their variables. (The refractory test is reset each time the process containing the rule is initialised.) A rule that fires once per cycle will only succeed for one binding of its variables on any processing cycle. If a rule is marked as refracted and as firing once per cycle, the rule will fire for different instantiations on different cycles. This may be used to generate a series of sequential behaviours.

F Built-In Conditions

COGENT supports several classes of built-in conditions that may be included in rules or condition definitions within process boxes.

Type checking: A series of conditions that test the type of term to which their single argument is bound:

atom: Succeeds if and only if the argument is bound to an atom.
integer: Succeeds if and only if the argument is bound to an integer.
number: Succeeds if and only if the argument is bound to a number.
variable: Succeeds if and only if the argument is uninstantiated.

Term comparison: A series of conditions that compare two arguments:

is identical to: Succeeds if and only if both terms are identical. This requires that uninstantiated variables occur in the same places in both terms, and that all bound elements are equal.

is distinct from: Succeeds if and only if is identical to fails.

unifies with: Succeeds if and only if the two terms unify. This may result in the instantiation of uninstantiated variables in either term.

arithmetic: is equal to: Succeeds if and only if the two arguments, when evaluated, are numerically equal. This should be used in preference to is identical to for equality checking, as it correctly compares integer and real numbers.

arithmetic: is not equal to: Succeeds if and only if the two arguments, when evaluated, are numerically unequal.

arithmetic: is less than: Succeeds if and only if the first argument is less than the second.

arithmetic: is not less than: Succeeds if and only if the first argument is greater than or equal to the second.

arithmetic: is greater than**:** Succeeds if and only if the first argument is greater than the second.

arithmetic: is not greater than**:** Succeeds if and only if the first argument is less than or equal to the second.

alphanumeric: is before**:** Succeeds if and only if the first argument is before the second in the standard alphanumeric order.

alphanumeric: is not before**:** Succeeds if and only if the first argument is equal or after the second in the standard alphanumeric order.

alphanumeric: is after**:** Succeeds if and only if the first argument is after the second in the standard alphanumeric order.

alphanumeric: is not after**:** Succeeds if and only if the first argument is equal or before the second in the standard alphanumeric order.

List processing: A range of conditions that operate on lists:

member**:** `Term` is a member of `List`
: Succeeds if `Term` can be unified with a member of `List`.

select**:** select `Term` from `List1` leaving `List2`
: Succeeds if `Term` can be unified with a member of `List1` and `List2` is the list of all other elements of `List`.

length**:** `Integer` is the length of `List`
: Succeeds if `Integer` can be unified with the length of `List`. `List` should be instantiated.

reverse**:** `List2` results from reversing `List1`
: Succeeds if `List2` can be unified with the reverse of `List1` (i.e., `List1` with its elements in reverse order.) `List1` should be instantiated.

replace**:** replace the first `Term1` in `List1` with `Term2` to give `List2`
: Succeeds if `List2` results from replacing the first instance of `Term1` in `List1` with `Term2`. `Term1`, `Term2` and `List1` should be instantiated.

replace all**:** replace each `Term1` in `List1` with `Term2` to give `List2`
: Succeeds if `List2` results from replacing all instances of `Term1` in `List1` with `Term2`. `Term1`, `Term2` and `List1` should be instantiated.

sort**:** `List1` results from sorting `List2`
: Succeeds if `List1` is the result of sorting `List2`. Sorting uses the standard order of terms, in which numbers come before terms beginning with upper-case letters which in turn come before terms beginning with lower-case letters. `List2` should be instantiated.

first element**:** `Term` is the first element of `List`
: Succeeds if `Term` can be unified with the first element of `List`. `List` should normally be instantiated.

last element**:** `Term` is the last element of `List`
: Succeeds if `Term` can be unified with the last element of `List`. `List` should normally be instantiated.

append**:** `List1` results from appending `List2` to `List3`
: Succeeds if `List1` is the result of appending `List2` to `List3`. With `List2` and `List3` instantiated, and `List1` uninstantiated, append will create a new `List1`. However, with only `List1` instantiated, append can be used to split the list into two parts, generating all different ways of splitting the list.

delete**:** `List1` results from deleting `Term` from `List2`
: Succeeds if `List1` is the result of deleting a term that unifies with `Term` from `List2`. If `Term` unifies with more than one element of `List2` (e.g. if `Term` is uninstantiated), delete will be satisfiable in multiple ways for each element of `List2`.

rank list**:** `Ranks` are the ranks of `List`
: `List` must be a list. `Ranks` is a list of exactly the same length, with elements corresponding to the ranks of elements in `List`. For example: the ranks of the list $[3, 8, 7, 8]$ are $[1, 3.5, 2, 3.5]$. The elements of `List` do not need to be numbers. If they are not, then the standard order of terms is used for ranking. In this order, numbers come before terms beginning with upper-case letters which in turn come before terms beginning with lower-case letters. Note that if the elements of `List` are expressions that may evaluate to numbers, then the expressions are ranked, *not* the numbers that they evaluate to.

sublist**:** `List1` is a sublist of `List2`

> Succeeds if `List1` is a sublist of `List2`. `List2` should normally be instantiated. If `List1` is uninstantiated, sublist will generate all possible sublists.

find all**:** `List` is the list of all `Term` such that *condition*

> Succeeds if `List` matches the list of all `Term` such that *condition* holds. *condition* may be any complex condition (and is selected as a subcondition using COGENT's usual pull-down condition editor menus). Normally `Term` will contain at least one uninstantiated variable which also occurs in *condition*, so that `List` is a list of solutions to *condition*.

Arithmetic: Arithmetic conditions consist of an expression and a term, and involve equating the term to the expression. In all cases, any variables within the expression must be bound prior to evaluating the arithmetic condition, and the term must either be an unbound variable or a number. If the term is unbound, the condition will succeed and the term will be bound to the result of evaluating the expression. If the term is bound, the condition will only succeed if it is bound to the result of evaluating the expression.

is**:** `Number` is `Expression`
plus**:** `Number1` is `Number2` + `Number3`
minus**:** `Number1` is `Number2` − `Number3`
times**:** `Number1` is `Number2` × `Number3`
divide**:** `Number1` is `Number2` / `Number3`
sqrt**:** `Number1` is the square root of `Number2`
abs**:** `Number1` is the absolute value of `Number2`
log**:** `Number1` is the natural log of `Number2`
exp**:** `Number1` is e to the power `Number2`
power**:** `Number1` is `Number2` to the power `Integer`
sigmoid**:** `Number1` is sigmoid(`Number2`)
factorial**:** `Integer1` is `Integer2` factorial
random number: uniform**:** `Number` is randomly drawn from U(`Lower`,`Upper`)

> Succeeds if `Number` matches a number drawn at random from the interval [`Lower`,`Upper`), with all points being equally likely. This condition is very unlikely to succeed if `Number` is instantiated. Instead, it should be used to generate a random number within a specified interval.

random number: normal**:** `Number` is randomly drawn from N(`Mean`,`Variance`)

> Succeeds if `Number` matches a random number drawn from the normal distribution with the specified `Mean` and `Variance` (i.e., the square of the standard deviation). This condition is very unlikely to succeed if `Number` is instantiated.

random number: integer**:** `Number` is a random integer drawn from `L` to `U` inclusive

> Succeeds if `Number` matches a random integer drawn from the interval `L` to `U` inclusive.

trig function: sin**:** `Number1` is sin(`Number2`)
trig function: cos**:** `Number1` is cos(`Number2`)
trig function: tan**:** `Number1` is tan(`Number2`)
trig function: asin**:** `Number1` is asin(`Number2`)
trig function: acos**:** `Number1` is acos(`Number2`)
trig function: atan**:** `Number1` is atan(`Number2`)
trig function: cartesian / polar**:** cartesian `CC` is equivalent to `PC` with origin `O`

> `CC` represents a point in cartesian (X, Y) coordinates. `PC` represents the same point in polar [Radius, Angle] coordinates, relative to the cartesian origin `O`.

list arithmetic: minimum value**:** `Number` is the smallest numeric element in `List`
list arithmetic: maximum value**:** `Number` is the largest numeric element in `List`
list arithmetic: sum**:** `Number` is the sum of the elements of `List`
list arithmetic: product**:** `Number` is the product of the elements of `List`
list arithmetic: arithmetic mean**:** `Number` is the average of the elements of `List`
list arithmetic: geometric mean**:** `Number` is the geometric mean of the elements of `List`
list arithmetic: standard deviation**:** `Number` is the s.d. of the elements of `List`
list arithmetic: dot product**:** `Number` is the dot product of `List1` and `List2`

Statistics: A set of built-in statistical functions and tests:

statistics: probability**:** `Alpha` is the probability of being greater than `X` in `D`

Calculate `Alpha`, the area under the curve defined by the distribution `D` between `X` and $+\mathtt{inf}$. `X` should be numeric or evaluate to a number. `D` must be one of:

- `z`: the standard normal distribution;
- `t(DF)`, where `DF` is a positive integer: the student's t distribution with `DF` degrees of freedom;
- `r(DF)`, where `DF` is a positive integer and $(-1 \leq \mathtt{X} \leq 1)$: Pearson's r distribution;
- `chi_sq(DF)`, where `DF` is a positive integer: the χ^2 distribution with `DF` degrees of freedom;
- `f(N,D)`, where `N` and `D` are positive integers: the F distribution with `N` and `D` degrees of freedom;
- `beta(A,B)`, where `A` and `B` are multiples of exactly 0.5: the beta distribution with parameters `A` and `B`; or
- `gamma(A,B)`, where `A` and `B` are multiples of exactly 0.5: the gamma distribution with parameters `A` and `B`.

Note: if you want to calculate significance levels for `z`, `t` or r, be aware that `Alpha` is the probability associated with the positive tail of the distribution. If you predict negative values, use `-X` instead of `X` as an argument. For two-tailed significance values, use `abs(X)` and multiply `Alpha` by 2.

statistics: correlation**:** `C` is the `Type` correlation coefficient of `List`

`List` must be a list of pairs of numbers (e.g., `[(2.3,4.2),(2.1,5.1)]`). `Type` must be `pearson`, `spearman` or `kendall`. The condition returns `C`, the Pearson's, Spearman's or Kendall's correlation co-efficient of the list of numbers.

statistics: related t**:** `T` is the t statistic of `List`

`List` must be a list of pairs of numbers (e.g., `[(2.3,4.2),(2.1,5.1)]`). The condition returns `T`, the `t` statistic calculated from the pairs of numbers.

statistics: unrelated t**:** `T` is the t statistic of `List1` and `List2`

`List1` and `List2` must be lists of numbers (e.g., `[2.3,4.2,2.1,5.1]`). Both lists should be of equal length. The condition returns `T`, the `t` statistic calculated between the two lists (which are assumed to be independent samples).

statistics: chi squared**:** `CS` is chi^2 of `Table1` (with expected values `Table2`)

`Table1` represents a contingency table. It must either be a list of numbers (normally integers), representing a one-dimensional contingency table, or a list of lists of numbers, with each of the inner lists having the same length, representing a two-dimensional contingency table. In both cases, `CS` is the value of the χ^2 statistic calculated on the table. If `Table2` is not specified (i.e., not a variable), it must be specify a table of values of the same size as `Table1`. These values are taken as expected values for each of the cells in `Table1`, and the χ^2 statistic calculated according to these expected values rather than according to expected values calculated by assuming a random distribution of cell values.

Miscellaneous: Additional conditions include:

match**:** `Term` is in *buffer*

Succeeds if `Term` can be unified with an item in the buffer named *buffer*.

call in module**:** call `condition` in *buffer*

Calls the user-defined condition `condition` from the buffer *buffer*. Succeeds if the call succeeds.

get value of property**:** the value of the *property* property is `Term`

Succeeds if *property* is a property of the current box, and the value of that property within the current box unifies with `Term`. This is used for retrieving the values of box properties.

current cycle**:** the current cycle is `I`

Succeeds if `I` matches an integer specifying the current COGENT cycle number.

current trial**:** the current trial is `I`

Succeeds if `I` matches an integer specifying the current COGENT trial number.

current block**:** the current block is `I`

Succeeds if `I` matches an integer specifying the current COGENT block number.

current subject**:** the current subject is `I`

Succeeds if `I` matches an integer specifying the current COGENT subject number.

current experiment**:** the current experiment is `I`

Succeeds if `I` matches an integer specifying the current COGENT experiment number.

is composed of**:** `Term` is composed of `List`

Succeeds if `Term` is a term whose components are the elements of `List`. For example, `f(a, b, c(d))` is composed of `[f, a, b, c(d)]`. This condition may be used to create or decompose terms. Similar to Prolog's `univ` function.

generate symbol**:** `S` is a new symbol with base `B`

Generates a symbol by appending an integer to `B`, such that each call yields a new symbol. Similar to the `gensym` function available in some programming languages.

call Prolog**:** call `Term`

Succeeds if `Term` succeeds as a Prolog goal. This is used for calling built-in Prolog predicates that are not listed elsewhere in this appendix.

cut!**:** !

The Prolog cut. This condition always succeeds, but fixes any variables bound in conditions prior to it. (See Bratko (1986) or Clocksin & Mellish (1987).) The condition can only be used within condition definitions. It is not available in the condition list of rules.

G User-Defined Properties

The set of properties associated with a box may be augmented on a box-by-box basis through the addition of user-defined properties. This facility allows the user to define additional properties associated with a box, and then relate aspects of the box's behaviour to those properties. Six types of user-defined properties are available:

Enumerated: The value may be any one of a number of explicitly listed values.
Positive Integer: The value may be any positive integer.
Boolean: The value may be true or false.
Real Number: The value may be any real number.
Bounded Real: The value may be any real number between two specified limits.
String: The value may be any character string.

User-defined properties are of most use within process boxes. Rules within such boxes may test property values, and thereby ensure that the process' behaviour is appropriately dependent on a user-defined property. For example, an enumerated property with name Bias and values positive and negative may be defined within a process. The following rule within that process will then apply only when Bias is positive.

```
IF:     the value of the “Bias” property is positive
        ...
THEN: ...
```

User-defined properties provide a convenient way to parameterise processes (and thereby models). Once a model is parameterised, COGENT's scripting language (see Section I) may be used to conduct computational experiments in which parameters are systematically varied in order to determine their effects upon relevant dependent variables.

H The Execution Model

From a computational perspective, a COGENT model may be considered to consist of a data bus or blackboard (to which messages that pass between boxes are posted), and a set of modules (corresponding to the boxes in the box and arrow diagram), with each module comprising a state, a state transition function, and an output function. A buffer, for example, corresponds to a module whose state at any time is the list of elements in the buffer, whose state transition function is determined in part by the buffer's properties concerning capacity limitations and decay, and whose output function is null (because buffers do not generate output). In contrast, the state of a process is the list of rules and condition definitions contained in the process, the state transition function is null (because the state doesn't change during processing), and the output function is determined primarily by the rules and condition definitions contained within the module (with properties of the process also playing a part). Other classes of box are treated in an analogous way.

Processing within the system consists of two phases. In the first phase all modules operate in parallel to produce output from the messages on the bus (by applying their output functions). In the second phase all modules change state in response to their state transition functions and the contents of the data bus. The two phase cycle then repeats until no further processing occurs. This point marks the end of the trial. (If that trial was the last trial in the current block, it also marks the end of the block. Similarly if the current block is the last block for the current subject, it marks the end of the subject, and if the current subject is the last subject for the current experiment, it marks the end of the experiment.) As described elsewhere, each of these events may lead to the generation of `system_end(Level)` triggers, which are processed within the same two-phase processing cycle.

Cooper (1995) provides a more formal description of an earlier version of the execution model. The principal difference between that description and the execution model now in use is that processing is shifted one half phase. That is, the previous "update then generate" cycle has been replaced by a "generate then update" cycle.

I Experiment Scripts

An experiment script specifies a sequence of steps comprising an experiment. It may, for example, specify twenty subjects each in three conditions performing five blocks of trials, with each block consisting of twenty separate trials. Once an experiment script has been created, it may be used to control the execution of a model over an extended period of time, allowing the model to effectively replicate the standard process of laboratory-based experimentation.

Execution of experiment scripts is controlled by the four run buttons on the top right of each box's window. These buttons have the following functions:

Initialise: Initialise the model. This is a session initialisation, which subsumes trial, block, subject and experiment initialisation.
Step: Step through one cycle of model execution.
Run: Execute the current experiment script.
Stop: Interrupt execution of the current experiment script.

When a model is created, two default experiment scripts are also created. One, called *Trial*, consists of three commands:

```
initialise trial
run to end of trial
end trial
```

When run, this script initialises a trial (thereby initialising any boxes whose Initialise property is set to Each Trial), runs the execution model to the end of the trial, and then does any end of trial processing (by issuing the system-generated `system_end(trial)` trigger). This script specifies a simple one-trial experiment.

The second script that is created is called *Default*. It is the one that, by default, is initially associated with the Run button. It consists of the single command:

repeat *Trial* 1 times

This script instructs the system to run the above *Trial* script once. Thus, each time the Run button is pressed, one complete trial will be performed. If the number of repetitions is altered and the script saved, pressing the Run button will instead run the new number of trials.

The script language allows more complex scripts to be written. It includes the following commands:

initialise *Level*: Initialise the model at the specified initialisation level. *Level*, which is selected from a pull-down menu, may be trial, block subject or experiment.

end *Level*: Perform appropriate end of the level processing. (A corresponding initialise *Level* command should occur sometime before this command.) If the model uses the `system_end(Level)` special trigger for the specified level, it will be sent at this point, and the model will again execute until no more processing occurs.

run to end of trial: Execute the model until it reaches the end of a trial.

repeat *SubScript* `N` times: Execute the specified subscript `N` times, where `N` may be any positive integer. *SubScript* is selected from a pull-down menu listing the scripts already defined for this model.

the value of *Property* in *Box* is `Value`: Set the specified property in the specified box to the specified value. Properties set in this way have effect within the current script and any subscripts until the end of the current script, when they are restored to their previous values.

These commands may be assembled into complex scripts (possibly calling embedded scripts and setting properties to various values) in order to conduct extended computational experiments.

Bibliography

Abney, S. (1989). A computational model of human parsing. *Journal of Psycholinguistic Research, 18*, 129–144.

Altmann, G., & Steedman, M. J. (1988). Interaction with context during human sentence processing. *Cognition*, *30*(3), 191–238.

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.

Bever, T. G. (1970). The cognitive basis for linguistic structures. In Hayes, J. R. (Ed.), *Cognition and the Development of Language*. Wiley, New York, NY.

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

Burton-Roberts, N. (1986). *Analysing Sentences: An Introduction to English Syntax*. Longman, London, UK.

Chomsky, N. (1959). Review of Skinner's "Verbal Behaviour". *Language*, *35*, 26–58.

Chomsky, N. (1965). *Aspects of the Theory of Syntax*. MIT Press, Cambridge, MA.

Clocksin, W. F., & Mellish, C. S. (1987). *Programming in Prolog* (3rd edition). Springer Verlag, Berlin.

Coltheart, M., Sartori, G., & Job, R. (Eds.). (1987). *The Cognitive Neuropsychology of Language*. Lawrence Erlbaum Associates, hove, UK.

Cooper, R. (1995). Towards an object-oriented language for cognitive modeling. In Moore, J. D., & Lehman, J. F. (Eds.), *Proceedings of the 17th Annual Conference of the Cognitive Science Society*, pp. 556–561. Pittsburgh, PA.

Cooper, R., & Shallice, T. (2000). Contention Scheduling and the control of routine activities. *Cognitive Neuropsychology*, *17*, 297–338.

Crain, S., & Steedman, M. J. (1985). On not being led up the garden path: the use of context by the psychological syntax processor. In Dowty, D. R., Kartunnen, L., & Zwicky, A. (Eds.), *Natural Language Parsing: Psychological, Computational and Theoretical Perspectives*, pp. 320–358. Cambridge University Press, Cambridge, UK.

Crocker, M. (1999). Mechanisms for sentence processing. In Garrod, S., & Pickering, M. (Eds.), *Language Processing*, chap. 7, pp. 191–232. Psychology Press, Hove, UK.

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

Frazier, L., & Fodor, J. D. (1978). The sausage machine: A new two-stage parsing model. *Cognition*, *6*(4), 291–325.

Frazier, L., & Rayner, K. (1982). Making and correcting errors during sentence comprehension: Eye movements in the analysis of structurally ambiguous sentences. *Cognitive Psychology*, *14*, 178–210.

Houghton, G. (1990). The problem of serial order: A neural network model of sequence learning and recall. In Dale, R., Mellish, C., & Zock, M. (Eds.), *Current Research in Natural Language Generation*, chap. 11, pp. 287–319. Academic Press, London, UK.

Kimball, J. (1973). Seven principles of surface structure parsing in natural language. *Cognition*, *2*, 15–47.

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.

Lewis, R. (1993). *An architecturally-based theory of human sentence comprehension*. Ph.D. thesis, Carnegie Mellon University, Pittsburg, PA.

MacDonald, M. C., Pearlmutter, N. J., & Seidenberg, M. S. (1994). Lexical nature of syntactic ambiguity resolution. *Psychological Review*, *101*, 109–134.

McClelland, J. L., & Rumelhart, D. E. (1981). An interactive activation model of context effects in letter perception: part 1. An account of basic findings. *Psychological Review*, *88*(5), 375–407.

Minsky, M., & Papert, S. (1988). *Perceptrons: An Introduction to Computational Geometry* (Second edition). MIT Press, Cambridge, MA.

Mitchell, D. C. (1994). Sentence Parsing. In Gernsbacher, M. A. (Ed.), *Handbook of Psycholinguistics*, chap. 11, pp. 375–409. Academic Press, San Diego, CA.

Newell, A. (1990). *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA.

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.

Tanenhaus, M. K., Spivey-Knowlton, M. J., Eberhard, K. M., & Sedivy, J. C. (1995). Integration of visual and linguistic information in spoken language comprehension. *Science*, *268*, 1632–1634.

Thorndike, E. L. (1911). *Animal Intelligence*. Macmillan, New York, NY.

Trueswell, J. C., & Tanenhaus, M. K. (1994). Towards a lexicalist framework of constraint-based syntactic ambiguity resolution. In Clifton, C. C., Frazier, L., & Rayner, K. (Eds.), *Perspectives on Sentence Processing*, pp. 115–180. Lawrence Erlbaum Associates, Hillsdale, NJ.

Vosse, T., & Kempen, G. (2000). Syntactic structure assembly in human parsing: A computational model based on competitive inhibition and a lexicalist grammar. *Cognition*, *75*, 105–143.

Wanner, E. (1980). The ATN and the Sausage Machine: Which One is Baloney? *Cognition*, *8*, 209–225.

Wanner, E. (1988). The parser's architecture. In Kessel, F. S. (Ed.), *The Development of Language and Language Researchers*. Lawrence Erlbaum Associates, Hove, UK.