

## Perseverative Subgoaling and Production System Models of Problem Solving

**Richard Cooper**

Department of Psychology  
Birkbeck College, University of London  
Malet St., London, WC1E 7HX  
r.cooper@psyc.bbk.ac.uk

### Abstract

Perseverative subgoaling, the repeated successful solution of subgoals, is a common feature of much problem solving, and its pervasive nature suggests that it is an emergent property of a problem solving architecture. This paper presents a set of minimal requirements on a production system architecture for problem solving which will allow perseverative subgoaling whilst guaranteeing the possibility of recovery from such situations. The fundamental claim is that perseverative subgoaling arises during problem solving when the results of subgoals are forgotten before they can be used. This prompts further attempts at the offending subgoals. In order for such attempts to be effective, however, the production system must satisfy three requirements concerning working memory structure, production structure, and memory decay. The minimal requirements are embodied in a model (developed within the COGENT modelling software) which is explored with respect to the task of multicolumn addition. The inter-relationship between memory decay and task difficulty within this task (measured in terms of the number of columns) is discussed.

### Introduction

Since the work of Newell & Simon (1972), production system models have been successfully used to account for a number of aspects of human problem solving. Much of this work has been brought together within Soar (Laird, Newell & Rosenbloom, 1987; Newell, 1990), a research programme aimed at modelling all aspects of cognition within a single production system architecture. Central to the Soar production system approach is the *problem space computational model* (Newell, 1990), a model of cognitive processing which embodies a number of findings concerning human problem solving. According to this model, problem solving (and cognition in general) involves the sequential application of operators to a state, successively modifying that state until a goal state is reached or some goal condition is satisfied.

Within the problem space computational model it is assumed that, due to capacity limitations, only a single state can be represented within working memory at a time. This single state principle is held to account for one strategy common in search intensive problem solving: progressive deepening (see, e.g., Newell, 1990). Progressive deepening is characterised by the initial shallow investigation of alternative solution paths, followed by the elimination of some solution paths and the further investigation of those remaining. Critically to the current work, progressive deepening typically involves the repeated solution of the same subgoals as the search space is explored. We refer to the process of solving a subgoal re-

peatedly as *perseverative subgoaling*. Note that, as defined here, perseverative subgoaling is *not* the same as progressive deepening. Perseverative subgoaling is a phenomenon that occurs as a consequence of progressive deepening, but there is more to progressive deepening than perseverative subgoaling, and, crucially, there is no *a priori* reason why perseverative subgoaling should be restricted to situations involving progressive deepening. With this in mind, it is clear that, despite the well established nature of progressive deepening as a cognitive search strategy, it can only account for perseverative subgoaling under conditions which involve some degree of search or look-ahead. Though empirical evidence is scarce, the research reported, and model developed, here are founded on the claim that perseverative subgoaling occurs in situations which do not involve search (but which are nevertheless working memory intensive), and as such, in at least some cases, perseverative subgoaling cannot be explained by an appeal to the single state principle. When perseverative subgoaling occurs in search-free tasks, an alternate explanation is required.

The fact that the Soar research community appeals to the single state principle in order to account for progressive deepening indicates that they are sensitive to the issue of working memory capacity. However, the single state principle operates at the level of the problem space computational model. It does not operate at the level of the production system architecture which is held to implement the problem space computational model. The underlying production system architecture is assumed to have a working memory which is not capacity bound. This assumption of an unlimited working memory within Soar's production system base is one architectural assumption which has been challenged on the grounds that it lacks psychological plausibility (cf. Cooper & Shallice, 1995).

In order to explore the extent to which unlimited working memory is critical to Soar's performance, earlier work (Cooper, Fox, Farrington & Shallice, in press) explored a number of potential mechanisms for reducing the working memory requirements of the underlying production system. Several soft constraints on working memory capacity (generally forms of memory decay) were implemented and the resulting performance compared within an empirical/computational methodology (cf. Cohen, 1995). It was observed that one such mechanism (probabilistic decay of refractory memory elements) lead to a system which, on occasions, exhibited perseverative subgoaling (independently of progressive deepening). However, unlike human problem solvers, only rarely was the modified Soar system able to

recover from this subgoal and solve the original task. Instead, problem solving would often breakdown irrecoverably, with the system resorting to its default behaviour.

The work reported here further investigates perseverative subgoal in production system architectures. However, rather than adopting the Soar research strategy of developing, exploring, and possibly extending, a complex production system in order to generalise the domains that it may account for (a strategy which has received substantial criticism: see Cooper & Shallice (1995) and references cited therein), an alternate strategy — that of attempting to determine minimal requirements for the behaviour in question — has been adopted. This strategy is founded on the fact that, given the level at which psychological theorising takes place, any computational model will necessarily include aspects which are not theoretically motivated (i.e., implementation details). It is therefore not sufficient to simply demonstrate that a particular model may simulate interesting behaviour. Rather, it is necessary to know which aspects of that model lead to the interesting behaviour, and which aspects are necessary only for the purposes of implementation. (For more details and justified of this strategy, see Cooper *et al.* (in press).) The aim of this work, then, is to determine minimal requirements on a production system architecture (irrespective of their manifestation in any particular production system) which will allow recoverable perseverative errors without substantially increasing the likelihood of problem solving failure.

The assumption of a production system architecture as an appropriate starting point is justified by the success of such architectures in modelling other aspects of problem solving behaviour (as noted above). Two obvious competitors to this starting point are analogical approaches and connectionist approaches. Connectionist approaches, highly popular in other domains of cognitive science, have offered few insights into problem solving, being limited mainly to the implementation of production systems (e.g., Touretzky & Hinton, 1988) and analogical models (e.g., Holyoak & Thagard, 1989). Analogical approaches to problem solving (e.g., Gentner, 1983; Keane, 1988) offer a more serious alternative. Though they have generally only been applied within problem solving situations where some variant of case-based reasoning is sufficient (i.e., where a sequence of problem solving decisions is not required), perseverative subgoaling should fall within their remit, and future work may explore how perseverative subgoaling might arise in such systems.

### Requirements for Perseverative Subgoaling

In order to develop a minimal model of perseverative subgoaling it is necessary to understand why this behaviour occurred when Soar was modified as mentioned above. It is also necessary to understand why perseverative subgoaling was only observed relatively rarely in the modified system. Each of these issues can be understood in the context of a simpler, more standard, production system.

A typical production system (see, for example, Charniak & McDermott, 1985) employs two primary memory components: a production memory (which contains schematic condition/action rules corresponding to problem solving knowledge) and a working memory (which contains a representation of the current problem solving state and which can trigger in-

stances of rules in production memory). Processing is cyclic and consists of repeatedly selecting a rule from production memory whose conditions are satisfied by the contents of working memory, and executing that rule. This typically results in the addition or deletion of elements from working memory, and hence leads to further production rules becoming applicable. Most production systems also employ a third memory component, a refractory memory. This memory contains the set of instances of rule which have already been employed in the current episode of problem solving, and is necessary because it prevents a single set of working memory elements from causing a single production rule to be executed more than once. To accommodate a refractory memory the processing cycle must be modified slightly: selection of an applicable rule from production memory must be limited to instances of rules which have not previously been executed, and when an instance of a production rule is executed, elements should be added to and deleted from working memory as usual, but the instance of the rule must also added to refractory memory.

Within Soar, this picture is somewhat more complicated. Various processes exist which modulate the effects of production firing on working memory (most notably the decision phase). These complications, however, are not germane to the issue of perseverative subgoaling, and are avoided here in the interests of clarity.

Returning then to the issue of perseverative subgoaling, the form of memory decay in Soar under which lead to this phenomenon consisted of a probabilistic decay of refractory memory elements, together with the removal from working memory of any elements that were added by the execution of the decayed refractory memory elements. Thus, on each problem solving cycle, there was a small change (typically of the order of 1 chance in 500) that each refractory memory element would decay (i.e., vanish from refractory memory). If a refractory memory element *r* did decay, then all those elements which were in working memory at the time of the decay and which had been added by the rule firing corresponding to *r* were also removed from working memory. This mechanism was designed to enforce some kind of decay on working memory elements (as is often argued for on psychological grounds), but at the same time to allow decayed elements to be replaced if necessary by the repeated firing of the instantiation of the production rule which originally lead to their creation. The constraint is a soft constraint on working memory size. It does not strictly limit the capacity of working memory, but should effectively reduce the cardinality of working memory by implementing the psychologically plausible assumption that working memory is an imperfect storage device.

It should be clear that perseverative subgoaling is a conceivable consequence of refractory memory decay. If, after a subgoal has been solved, the refractory memory element which lead to the subgoal's answer were to decay, then the subgoal's answer would disappear from working memory and it would be necessary to repeat the problem solving necessary to determine that answer. This would be possible because the refractory memory elements which would normally prevent repetition will also be absent. Perseverative subgoaling was not, however, the original intention (or even a prediction) of the introduction of this form of decay. It was observed *post*

*hoc* to arise on a small number of trails. More frequently, refractory memory decay was observed to lead to complete problem solving breakdown. Detailed analysis of the causes of such breakdowns indicates that two additional properties are required of a production system if it is to be subject to recoverable perseverative subgoaling: firstly, production rules must be “fine-grained” (in a sense to be elaborated below), and secondly, memory decay must not result in disconnected subgoals.

The requirement on the granularity of production rules may be stated more formally as the requirement that no production may add more than one element to working memory, and that all productions must explicitly list in their conditions each working memory element which must be present for their application. These restrictions may be rationalised informally by noting that the conditions of all rules must be sensitive to the possibility that working memory may not be complete. Rules can therefore not assume that the presence of one working memory element will guarantee that other working memory elements will necessarily be present. Indeed, the frequent breakdown of problem solving within Soar with refractory memory decay can be traced to such dependencies in Soar’s default productions (the productions which control standard problem solving heuristics). This is not intended as a criticism of Soar: the assumptions embodied in the default rules are valid in the normally functioning system, although leaving them implicit does mean that each rule cannot be understood in purely declarative terms.

The prohibition on disconnected subgoals is necessary to prevent problem solving from stalling after successful completion of a disconnected subgoal. Effectively the constraint states that the system cannot forget why it is attempting to solve a subgoal.<sup>1</sup> Without this constraint (and assuming that problem solving is under the service of a goal stack), situations could arise in which no goal would be within the current focus. The system could successfully solve a subgoal, but then be left not knowing what to do next. In Soar, with refractory memory decay, this constraint is satisfied because goal/subgoal relations are not created by production firing. As such, those relations cannot be deleted when refractory memory elements decay. However, in more standard production systems (in which productions can explicitly set subgoals) this constraint could be violated.

## A Model of Perseverative Subgoaling

### COGENT: The Modelling Environment

The model presented here was developed using the COGENT modelling software. COGENT (previously known as GOOSE: cf. Cooper, 1995) was developed from the Sceptic executable specification language (Hajnal, Fox & Krause, 1989; Cooper & Farrington, 1993) in work aimed at improving the methodology of computational modelling (cf. Cooper, *et al.*, in press). It provides a set of cognitive “objects” (in the object-oriented sense) and a graphical editor. Together, these allow the specification of executable models in the box/arrow style. This style, it is argued, is more akin to that used by experimental psychologists in outlining their theories of processing than traditional programming languages, and as such allows a more

<sup>1</sup>I am indebted to an anonymous reviewer for this locution.

direct and perspicuous mapping between theory and implementation. In addition, COGENT is a step towards reducing the computer skills required of psychologists who wish to develop their own computational models.

The specification of a model in COGENT involves firstly drawing the appropriate box/arrow diagram (with the aid of the graphical editor) and then specifying properties and other details for the individual boxes employed in the diagram. The provision of a set of basic box classes (represented by different shaped boxes), together with the use of techniques from object-oriented programming, minimises the effort in specifying properties and standardises the definitions of the fundamental box classes employed in a model (cf. Cooper, 1995). Apart from the work reported here, the environment has been employed in the development of models of concept combination (Cooper & Franks, 1996), prospective memory (Ellis, Shallice and Cooper, in submission), the interaction of memory recall and decision making (cf. Fox, 1980), and the creation of long term memory records (cf. Morton, Hammer-sley & Bekerian, 1985). Details of COGENT availability and system requirements are available from the author.

### The Functional Modules

The basic unit of COGENT is the *cognitive object*. This is the software’s equivalent of a psychologist’s functional module. In the production system model reported here, four major functional modules were employed: *Task Control*, *Working Memory*, *Perceptual Buffer*, and *Memory Decay*. A fifth object, *Transcript*, was used to record the model’s behaviour. Figure 1 shows the connectivity of the modules.

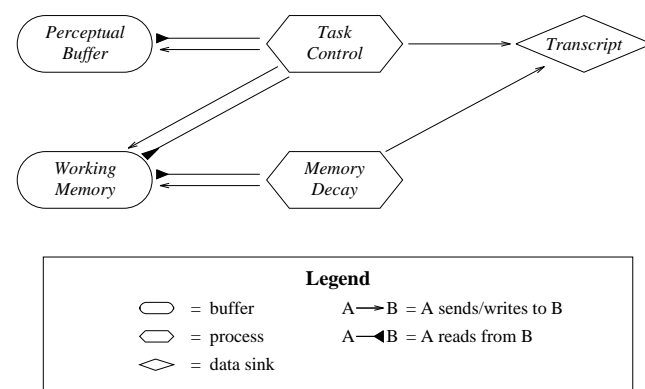


Figure 1: The simplified production system in COGENT

Refractory memory and refractory memory decay were not explicitly modelled. Instead, a simpler alternative was employed in which production rules were not refracted (and so any instantiation of a production rule could fire any number of times), but were modified so as to include in their conditions the logical negation of their actions. Thus a rule which added an element to working memory was modified to include an explicit check that the element in question was not already in working memory. Decay was then imposed on working memory. This scheme, which with the production rules employed in the current model is equivalent to refractory memory decay, leads to a number of simplifications in the basic production system architecture (such as removing the requirement for a refractory memory and an associated test against this memory

during the selection of productions to execute), and is consistent with the goal of demonstrating minimal requirements for perseverative subgoaling.

**Task Control** Most of the standard production system processing is performed by a single COGENT object: a rule-based process. This object is an instance of a standard object class provided by COGENT, and its behaviour is fully determined by a set of production-like rules which read from and write to other objects in the model (in particular, *Working Memory*). The process operates in a cyclic manner but without any form of conflict resolution. Thus several rules may, in principle, fire on a single cycle.

*Task Control* contains five declarative rules, each of which is task independent. Together they embody a general approach to tasks with a fixed (and known) goal/subgoal structure. In summary, the functions of the rules are:

1. focus on the first unresolved child of the lowest unresolved goal;
2. directly execute a childless goal (i.e., a primitive operator);
3. merge the results of a goal's subgoals to give the results of the goal;
4. delete intermediate subgoal results if a goal's result is known; and
5. delete intermediate subgoals if a goal's result is known;

All rules modify *Working Memory* by adding or deleting information about goal/subgoal relationships or goal/result relationships. The rules are not refracted, but are fully declarative, so fire only under appropriate conditions (including when their consequent elements decay from working memory).

Three additional pieces of information are required in order to execute a specific task: the task's goal/subgoal structure, the means of effecting all lowest level goals, and the means of assembling the resolution of a goal's subgoals into the resolution of that goal. This additional information is also associated with *Task Control* in the form of Prolog conditions.

**Working Memory** Working memory is modelled by an unlimited capacity (symbolic) buffer — another standard class of object provided by COGENT. Elements are added to, and deleted from, this buffer by *Task Control*, which also matches information in the buffer when determining which rules to fire. Note that no explicit constraint is placed on the buffer capacity. A fixed capacity working memory would be no more psychologically plausible than an unlimited capacity working memory. The current work therefore employs a soft constraint derived from working memory decay (explicitly modelled by another process). Assuming that elements can only be added to *Working Memory* at a finite rate, a high decay rate will naturally lead to a low (but variable) capacity.

**Memory Decay** Each COGENT object has a set of properties which determine its precise behaviour. Symbolic buffers, such as *Working Memory*, have two properties which govern decay (one specifying a decay function, and one specifying a decay rate). However, this built-in functionality does not take account of structure on the contents of the buffer in question, and so specifying decay from *Working Memory* in this way in the current model could lead to disconnected subgoals. There-

fore, in the model reported here decay of *Working Memory* is explicitly effected by a separate rule-based process, *Memory Decay*. On each production cycle, this process may randomly delete some terminal working memory elements. This behaviour is programmed by a single rule which generates a random number (uniformly distributed between 0 and 1) for each terminal working memory element, and, if that number is less than a preset threshold, deletes the corresponding element from *Working Memory*. The probability of deletion (i.e., the preset threshold) is independent of the working memory element (provided it is a terminal element) and given by a separate parameter of the process.

The use of a separate process governing the decay of working memory is an unsatisfactory aspect of the model, and reflects an inadequacy in the COGENT modelling environment. Ideally, memory decay should be an intrinsic property of the buffer concerned, not the result of some explicit decay process acting on the buffer. Thus, it should be possible to state the appropriate form of decay as a property of the buffer (as discussed in the preceding paragraph). This could be done with COGENT in its current form if 1) refractory memory was explicitly modelled (as in Soar), 2) subgoals were created by some mechanism other than production system firing (as in Soar), and 3) decay was imposed on refractory memory (contrary to Soar).

**Perceptual Buffer** In most problem solving situations it is possible to distinguish between two sorts of information: information generated and maintained by internal processes (in this case the contents of *Working Memory*), and perceptual information. Perceptual information is always immediately available to the problem solver. As such there is no clear sense in which it decays. The current model therefore makes use of one further unlimited capacity symbolic buffer, *Perceptual Buffer*, which contains a representation of information available in the external world. The precise contents of this buffer will depend on the external environment of the problem solving agent, but the buffer's existence is crucial to the agent's problem solving behaviour.

**Transcript** The last functional module employed in the model is *Transcript*, a data sink which is used to record the model's behaviour, including the elements which decay from working memory and the model's final solution to its problem solving task. This module has no theoretical import.

## The Task: Multicolumn Addition

The original work on refractory memory decay in Soar was based on two versions of a standard AI task, the monkey and bananas task. This task has many interesting features, but is more complex than necessary to show recoverable perseverative subgoaling. A different task is therefore considered here. The multicolumn addition task (see, e.g., Anderson, 1993, pp. 4–6) involves adding two “large” integers using the standard algorithm taught in most Western schools: first sum the units column, recording the units digit of the sum and carrying the tens digit (if any) to the next column, then sum the tens column, recording the units digit of the sum and carrying the tens digit (if any) to the next column, and so on. The task has the advantage of having a clear goal/subgoal structure, viz:

Goal	Subgoals
<i>multicolumn-add</i>	<i>process-column(1)</i> <i>process-column(2)</i> ⋮ <i>process-remaining-carry</i>
<i>process-column(N)</i>	<i>get-digits(N)</i> <i>add-digits</i> <i>add-previous-carry</i> <i>split-answer-units-and-carry</i>

Furthermore, the task can be made to place heavy demands on working memory by requiring that intermediate results (i.e., the sums of individual columns) be remembered during the computation (rather than being written down as the computation progresses), and those demands will increase as the number of columns in the sum increases. As discussed below, this makes the task ideal for empirical investigation.

Multicolumn addition was therefore modelled in COGENT by the inclusion of task specific goal/subgoal information in the *Task Control* object. In addition, the *Perceptual Buffer* was initialised with the stimulus information (a representation of the integers to be summed). As noted above, no decay was specified for this buffer, modelling the fact that this information would always be available throughout the task. Two experimental conditions were examined in the simulations. In the first, intermediate results were not added to the *Perceptual Buffer*, but were instead stored in *Working Memory* for the duration of the task, thus modelling the situation where the subject is required to perform the entire calculation without the aid of an external memory. In this condition, intermediate results are subject to potential decay. In the second condition, the results of summing individual columns were added to the *Perceptual Buffer*, and *Task Control* was able to freely consult this buffer. This condition models the task were the subject is able to record his/her results as problem solving progresses.

An additional rule was added to *Task Control* to detect when the task was complete and print the answer contained in *Working Memory*.

### Simulation Results

We consider first the results of the “difficult” condition in which intermediate results are stored in *Working Memory*. Figure 2 shows a trace of the primitive operators invoked by the model on one particular trial when working with a decay rate of 0.03. The model was attempting the sum:

$$\begin{array}{r} 895 + \\ 267 \end{array}$$

Of principal concern is the perseverative subgoaling occurring between cycle 25 and cycle 31. During this period, the model is solving the four subgoals which comprise *process-column(1)*. These subgoals were originally solved during cycles 4–10, after which their results were assembled to produce a result for *process-column(1)*. However, by cycle 25, this result was no longer in *Working Memory* (due to *Working Memory* decay), prompting the model to automatically repeat its prior problem solving. The model recovers after this bout of perseverative subgoaling, and produces the correct answer, although its solution time (measured in number of cycles to completion) is sub-optimal.

Cycle 4	<i>get-digits(column(1))</i>
Cycle 6	<i>add-digits(column(1))</i>
Cycle 8	<i>add-previous-carry(column(1))</i>
Cycle 10	<i>split-answer(column(1))</i>
Cycle 15	<i>get-digits(column(2))</i>
Cycle 17	<i>add-digits(column(2))</i>
Cycle 19	<i>add-previous-carry(column(2))</i>
Cycle 22	<i>split-answer(column(2))</i>
Cycle 25	<i>get-digits(column(1))</i>
Cycle 27	<i>add-digits(column(1))</i>
Cycle 29	<i>add-previous-carry(column(1))</i>
Cycle 31	<i>split-answer(column(1))</i>
Cycle 36	<i>get-digits(column(3))</i>
Cycle 38	<i>add-digits(column(3))</i>
Cycle 40	<i>add-previous-carry(column(3))</i>
Cycle 42	<i>split-answer(column(3))</i>
Cycle 47	<i>process-carry(column(4))</i>
Cycle 49	STOP: answer = 1162

Figure 2: Perseverative subgoaling in 3-column addition

Figure 2 shows, then, that the model is indeed subject to perseverative subgoaling. Perhaps surprisingly, it is robust against substantially greater decay rates than Soar was in the original experiments. A decay rate of 0.03 (which led to the transcript in Figure 2) is three times greater than the highest rate for which Soar’s behaviour was examined (and found wanting), and yet performance is only moderately impaired, showing slowing but not error.

Extensive experimentation has shown that the model is able to recover from any failure except premature decay of the root of the goal stack. Provided that decay does not occur over the first cycle of problem solving, the model will always solve the task, though the number of cycles required to solve the problem will vary depending on the decay rate and the complexity of the task (which is here determined by the number of columns). However, and in spite of this apparent robustness, the system is prone to error. In particular, the operator *add-previous-carry* assumes that if a carry should be added, it will be in *Working Memory*. Decay of carry information can therefore lead to errors.<sup>2</sup>

Also worthy of note is that the effect of decay on solution time increases super-linearly with the number of columns in the sum. A decay rate of 0.10 working memory elements per cycle led on average to a 1.6 times increase in solution time for two column sums. This grew to 3.2 for three column sums and 13.5 for four column sums. The number of incorrect solutions (due to failure to carry) increases in a similar way with the number of columns in the sum.

With regard to the second experimental condition, where the sums of columns were stored in the *Perceptual Buffer* (and hence not subject to decay), solution times as measured in number of cycles were, as expected, generally shorter than in the first condition, and there was substantially less variance.

<sup>2</sup>This source of error, decay of carry information, is not a necessary aspect of the model. The rules could be designed to treat a missing carry as a decayed subgoal result. However, this error appears to be consistent with human performance on the task, and so no attempt was made to avoid it.

Over ten trials of a 4-column task with a decay rate of 0.03, the average number of cycles to solution was 47.1 (s.d. = 2.5). This compares with 59.1 (s.d. = 12.5) for the equivalent task in the difficult condition.

### Future Developments

The simulation results suggest that perseverative subgoalting will occur most frequently when working memory capacity is stretched (e.g., in multicolumn addition involving many columns). Future work will empirically explore the nature of perseveration in problem solving with different working memory requirements. Multicolumn addition is an ideal task for this work as the demands on working memory can be easily varied by varying the number of columns in the sum. Alternately, the task can be coupled with some other concurrent, working memory intensive, task in order to investigate the effects of working memory load.

One hypothesis to be addressed by this work is that perseverative subgoalting will be greater when memory resources are in high demand. This hypothesis is motivated by a strengthening of the putative relationship between working memory load and working memory decay — that the rate of decay is dependent on the load, being greater when the load is greater. Should empirical work confirm this hypothesis, the COGENT modelling software will facilitate the exploration of capacity sensitive alternatives to the rules governing working memory decay presented here.

A second area of future work concerns uses of the underlying mechanism which leads to recoverable perseverative subgoalting, i.e., non-refracted fine-grained productions along with refractory memory decay. Together, these produce a mechanism which effectively refreshes working memory when critical elements decay. One clear theoretical domain where such a refreshing mechanism may be appropriate is the articulatory loop (see, e.g., Baddeley, 1986), an area in which network models (notably that of Burgess & Hitch, 1992) have shown some success. The mechanism presented here offers the promise of a symbolic alternative to such network approaches. More generally, the introduction of stochastic elements into symbolic models may allow the development of semantically perspicuous models with sufficient plasticity to rival their connectionist counterparts.

### Conclusions

We have isolated three requirements which will lead a production system to exhibit recoverable perseverative subgoalting: refractory memory decay, fine-grained productions, and a decay-free goal stack. None of the three requirements alone is sufficient if the resultant system is to exhibit the effect. The conjunction of the three, however, cannot be claimed to be necessary conditions. With sufficient ingenuity, alternate systems could undoubtedly achieve the same effect. The claim here is that no simpler system could do so.

### Acknowledgements

I am grateful to Nick Braisby, John Fox, Bradley Franks, and Tim Shallice for advice and discussion on issues related to the work presented here. Part of this work was supported by the Joint Council Initiative in Cognitive Science and Human-Computer Interaction, project grant #G9212530.

### References

- Anderson, J. R. (1993). *Rules of the Mind*. Lawrence Erlbaum Associates, Hove, UK.
- Baddeley, A. D. (1986). *Working Memory*. Oxford University Press, Oxford, UK.
- Burgess, N., & Hitch, G. J. (1992). Toward a network model of the articulatory loop. *Journal of Memory and Language*, 31, 429–460.
- Charniak, E., & McDermott, D. (1985). *Introduction to Artificial Intelligence*. Addison–Wesley, Reading, MA.
- Cohen, P. R. (1995). *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, MA.
- Cooper, R. (1995). Towards an object-oriented language for cognitive modeling. In *Proceedings of the 17<sup>th</sup> Annual Conference of the Cognitive Science Society*, pp. 556–561. Pittsburgh, PA.
- Cooper, R., & Farrington, J. (1993). Sceptic Version 4 User Manual. Tech. rep. UCL-PSY-ADREM-TR6, Department of Psychology, University College London, UK.
- Cooper, R., Fox, J., Farrington, J., & Shallice, T. (In press). A systematic methodology for cognitive modelling. *Artificial Intelligence*, 85.
- Cooper, R., & Franks, B. (1996). The iteration of concept combination in Sense Generation. In *Proceedings of the 18<sup>th</sup> Annual Conference of the Cognitive Science Society*. This volume. San Diego, CA.
- Cooper, R., & Shallice, T. (1995). Soar and the case for Unified Theories of Cognition. *Cognition*, 55(2), 115–149.
- Ellis, J., Shallice, T., & Cooper, R. (1996). Memory for, and the organization of, future intentions. In submission.
- Fox, J. (1980). Making decisions under the influence of memory. *Psychological Review*, 87, 190–211.
- Genter, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7, 155–170.
- Hajnal, S., Fox, J., & Krause, P. (1989). *Sceptic User Manual: Version 3.0*. Advanced Computation Laboratory, Imperial Cancer Research Fund, London, UK.
- Holyoak, K. J., & Thagard, P. (1989). Analogical mapping by constraint satisfaction. *Cognitive Science*, 13, 295–355.
- Keane, M. T. (1988). *Analogical Problem Solving*. Ellis Horwood, Chichester.
- Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33, 1–64.
- Morton, J., Hammersley, R. H., & Bekerian, D. A. (1985). Headed records: A model for memory and its failures. *Cognition*, 20, 1–23.
- Newell, A. (1990). *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA.
- Newell, A., & Simon, H. (1972). *Human Problem Solving*. Prentice–Hall, Englewood Cliffs, NJ.
- Touretzky, D. S., & Hinton, G. E. (1988). A distributed connectionist production system. *Cognitive Science*, 12, 423–466.