The Role of Cognitive Architecture in Modelling the User: Soar's Learning Mechanism

Andrew Howes
School of Psychology, University of Cardiff
PO Box 901, Cardiff CF1 3YG, UK
HowesA@cardiff.ac.uk

Richard M Young
MRC Applied Psychology Unit, 15 Chaucer Road
Cambridge CB2 2EF, UK
Richard.Young@mrc-apu.cam.ac.uk

August 1996

Revised version for *Human-Computer Interaction*
special issue on "Cognitive Architectures and HCI"

Contact: Andrew Howes
School of Psychology
Cardiff University of Wales
P O Box 901
Cardiff CF1 3 YG
U. K.

tel: (01222) 874 007
fax: (01222) 874 858
email: HowesA@cardiff.ac.uk

The Role of Cognitive Architecture in Modelling the User: Soar's Learning Mechanism

Abstract

What is the role of a cognitive architecture in shaping a model built within it? Compared with a model written in a programming language, the cognitive architecture offers theoretical constraints. These constraints can be "soft", in that some ways of constructing a model are facilitated and others made more difficult, or they can be "hard", in that certain aspects of a model are enforced and others are ruled out. We illustrate various of these possibilities. In the case of Soar, its learning mechanism is sufficiently constraining that it imposes hard constraints on models constructed within it. We describe how one of these hard constraints deriving from Soar's learning mechanism ensures that models constructed within Soar must learn a display-based skill and, other things being equal, must find display-based devices easier to learn than keyboard-based devices. We discuss the relation between architecture and model in terms of the degree to which a model is "compliant" with the constraints set by the architecture. Although doubts are sometimes expressed as to whether cognitive architectures have any empirical consequences for user modelling, our analysis shows that they do. Architectures play their part by imposing theoretical constraints on the models constructed within them, and the extent to which the influence of the architecture shows through in the model's behaviour depends on the compliancy of the model.

1. Introduction: Cognitive architectures and user modelling

A cognitive architecture embodies a scientific hypothesis about those aspects of human cognition which are relatively constant over time and relatively independent of task. Examples range from architectures claiming broad scope, such as Soar (Newell, 1990) and ACT-R (Anderson, 1993), through ICS (Interacting Cognitive Subsystems: Barnard, 1987), to more specialised ones such as C-I (Construction-Integration: Kintsch, 1988). The different architectures in principle offer alternative, competing theories of cognition, but the considerable overlap in their structures and assumptions, and the fact that they tend to exhibit different areas of strength and weakness, makes any evaluative comparison between them a complex and difficult undertaking.

As reflected in this issue of the journal *Human-Computer Interaction* and in the workshop from which it stems (Kirschenbaum, Gray & Young, 1996), a recent trend in work extending the state of the art in constructing psychological models of the computer user, has been to cast the models within some chosen cognitive architecture (e.g., Howes & Young, 1996; Anderson, Douglass, Lebiere & Matessa, this issue; Barnard & May, 1993; Kitajima & Polson, 1995). In such models, part of the content of the model is supplied by the cognitive architecture itself, while the rest is supplied by what the analyst has to add to the (generic) architecture in order to

construct a (specific) model. Below, we define that additional information as constituting the *model increment*. Thus the theoretical content of the model is distributed between the cognitive architecture and the model increment. The contrast to working with an architecture is for the analyst to implement the user model directly in some programming language, such as Lisp or C, chosen for its ease of programming and its convenience for expressing the model, but not for its theoretical contribution.

Building user models within a cognitive architecture in this fashion raises novel problems of practical and scientific evaluation. Given a model and a pattern of empirical evaluation against data and applications, how much of the credit for the successes, and blame for the failures, should be allocated to the architecture itself, and how much to the model increment? More generally, the approach raises the question of how we should conceptualise, discuss, and assess the contribution of the cognitive architecture to the properties of the specific model.

In this paper we offer an approach to answering the question, by exploring the notion of theoretical constraint. The core idea is that the cognitive architecture influences, or biases, the kinds of models that can be constructed within it, by placing constraints on what can be done by the possible model increments. These constraints can be "soft", in that the construction of some kinds of model is facilitated by aspects of the architecture, while other kinds are made more difficult. Or the constraints can be "hard", in that certain features of the resulting models are enforced, while others are ruled out. To determine how much of the quality of a specific model and of its explanatory force should be attributed to the architecture and how much to the model increment we need to determine the situations where the architecture imposes hard constraints on models built within it.

In the next Section of the paper, as a necessary preliminary to considering cases of hard and soft constraints, we illustrate various ideas about model increments and the contents of models. Then in Section 3 we present a case study, where we examine the degree to which, and the ways in which, a particular feature (the learning mechanism) of a particular architecture (Soar) places hard constraints on all models that make use of the feature. Section 4 discusses the general lessons suggested by this study with regard to the use of cognitive architectures in user modelling, and Section 5 concludes.

2. The role of the architecture

To gain a better understanding of the issues raised by architecturally-based models, we first (in Section 2.1) propose a framework for the information required to specify a particular model within a given cognitive architecture. We then (in Section 2.2) apply the framework to examine more closely the role of the model increment within a single architecture.

2.1. Specifying a model within a cognitive architecture

All models constructed within a given cognitive architecture share the features of that architecture. What distinguishes one model from another is the extra information that has to be provided in order to define a particular model within the architecture. As we have already seen, we refer to this further specification as the *model increment*. (The rationale for the term will become clearer below.) In this section, we look at the nature of the model increment in the case

of four illustrative cognitive architectures. The purpose of this section is not to compare the architectures or make comparative judgements about their value, but to present a framework for understanding architecture-based models and to demonstrate that the framework applies across a range of different architectures.

The form and content of the model increment varies from one architecture to another. In Soar (Newell, 1990), the model increment takes the form of a set of production rules, that encode the relevant knowledge the user is postulated to have. Their content can be about the task to be performed, about the device and its interface, about relevant background knowledge (such as about the meaning of words, or the order of days of the week), about control information for methods of performing tasks, and indeed about anything that may pertain to the user's behaviour. The view adopted is that the Soar architecture, together with the knowledge expressed as the model increment, operating in a (usually simulated) task environment, yields a prediction of the user's behaviour. Summarised schematically, we have

Soar + knowledge (of states, of operators, …) + task environment → behaviour,

where the model increment consists of just the knowledge component.

The model increment for ACT-R (Anderson, 1993) is in many ways similar to that for Soar, though there is more diversity in the way information is expressed. Firstly, knowledge is encoded in two different memories: a procedural memory, expressed as production rules, and a declarative memory, expressed as a network of linked nodes. Usually, simple factual knowledge is encoded in the declarative memory, while enactive knowledge about methods and procedures is encoded in the procedural memory. However, the analyst has a degree of discretion, so that declarative information about methods and procedures can be held in the declarative memory, while it is also possible for factual knowledge to be encoded in production rules that reflect the usage of that knowledge.[1] Secondly, a range of numerical parameters is open to specification, in order to determine link strengths, initial activation levels, time constants, prior probabilities and estimates of costs and rewards, and so on. Schematically, we have

ACT-R + declarative knowledge + procedural knowledge + numeric parameters + task environment → behaviour,

where the model increment consists of the two kinds of knowledge together with the parameter values.

The C-I architecture derives from a theory of sentence processing (Kintsch, 1988), so not surprisingly the information it requires to specify a model is primarily propositional in nature. Kitajima & Polson (1995) describe the relevant knowledge under six headings: task-goal, device-goal, display, long-term memory, candidate objects, and object-action pairs. In addition, there are six numerical parameters for the model, describing aspects such as the relative weight to be given to the link between two propositions that share a common term, the multiplier for making links between the goals and the rest of the network stronger than other links, and the number of memory samples to be taken. Schematically, this yields

C-I + six kinds of propositions + numeric parameters → action choice.

We have omitted the task environment from this formula, because C-I models are used to simulate extended sequences of interaction between user and device in a step-by-step fashion.

---

[1] Anderson (1993) offers various arguments in favour of more varied architectural

A C-I model is run to examine a single action decision at a particular point in an interaction sequence. To get the prediction for the next decision, the goals and display part of the model increment are re-set to describe the updated situation, and the model run again to make a new prediction.

The ICS model (Barnard, 1987) presents a still different picture. Unlike the other architectures considered here, ICS is not a simulation architecture. Instead, it provides a structure and a set of concepts concerning the user's cognition, in terms of which the analyst can describe an interactive situation and argue to a prediction of its consequences for the user. In certain cases, the role of the analyst can be partly taken over by an expert system (Barnard, Wilson & MacLean, 1988; Barnard & May, 1993) which receives help in mapping the situation into ICS terminology, then derives the consequences and reports the predictions. In these cases, the expert system is playing (part of) the role of analyst, not simulating the user. In order to apply ICS to a particular question, information has to be specified about the dynamic configuration of the internal components of ICS, about the degree of overlearning ("proceduralisation") of aspects of relevant skills in a task and perceptual setting, about the content of the memories, and about the representation of perceptual encodings (May, Barnard & Blandford, 1993). In terms of our schematic formula, we have

ICS + process configuration + proceduralisation + memory contents
+ perceptual representations + dynamic control
→ prediction of attributes of user behaviour.

Once again, there is no separate task environment, since an ICS model is not a simulation, and information about the task environment is distributed among other parts of the specification.

These four examples do not exhaust the possibilities for the forms the model increment can take. Their commonalities, however, serve to illustrate that in one way or another, and although the different influences can be distributed among the components in different ways, the resulting behaviour of a model built within a cognitive architecture depends upon

- the architecture itself,
- knowledge and other forms of specification added to identify a particular model, and
- the task and the environment within which the model operates.

For any one of these aspects of the model — such as the architecture — to be able to affect the behaviour, it must in some way constrain the influence of the other aspects, say, the model increment and the task environment. (If it exerts no constraint, it is hard to see how it can be playing a role in determining the behaviour.) In the next section of the paper we consider the case of a particular cognitive architecture, Soar, and examine how it constrains the properties of individual Soar models.

2.2 Role of the architecture

The decomposition of an architecturally based model, into architecture plus model increment plus task environment, raises problems of credit assignment. For a particular feature of the resulting behaviour, how do we tell which component it is due to? Such problems are especially relevant in the context of evaluating the model. If there is a discrepancy between the behaviour of the model and empirical data, where do we place the "blame"? Does it mean that the architecture is wrong, or that the model increment is wrong, or is there some error in the way we

have simulated the task environment? And correspondingly when the agreement between model and data is good, how do we allocate the "credit"? Should the agreement strengthen our confidence in the architecture, or does it tell us only that the added knowledge, so far as it goes, seems right?

In this section we explore further the way in which the cognitive architecture and a model increment work together to determine a model's behaviour. The key idea introduced here is that the architecture itself can propose actions to be taken, which the model increment can concur with, or can modify, or can simply override. To prepare the groundwork for that discussion, we begin by reviewing in some detail Laird's (1986) work on "universal weak method".

2.2.1 Universal weak method

One of the (many) ways in which a cognitive architecture differs from being just a programming environment, is that a cognitive architecture may be able to proceed with an "incomplete" model increment in a way that has no counterpart for ordinary programs. In other words, a programming environment or language, such as C, has no "ideas of its own" about what to do. It does nothing at all until it is told what to do, by being given a program. And that program has to be "complete" — at least in the parts that get executed — otherwise, if the execution reaches a part of the program which is undefined, it will stop with an error message. That need not be so for a cognitive architecture. Soar, in particular, will "run" even with a missing or highly incomplete model increment, by relying heavily on the notion of default behaviour. Given the description of a problem or task, Soar will set about trying to perform it, even without any information about how to do so. Of course, its behaviour under these conditions is not very "intelligent", but it is far from trivial. For internal problem solving, Soar will by default engage in a form of depth-first search.

Laird (1986) extended the basic Soar architecture with a set of default production rules, to yield an architecture exhibiting what he termed "universal weak method" (UWM). In this context, a weak method (Newell, 1969) is a problem solving method that is very general and requires little knowledge about the task. Examples are means-ends analysis, and generate-and-test. Because a weak method demands little domain-dependent knowledge, it can be used in many domains, especially when "stronger", more domain-specific knowledge is unavailable. Laird showed that by adding to UWM what he called *method increments* — small collections of production rules that encode information about the domain and task — he could generate most of the known weak methods.

To understand Laird's demonstration, we need first to consider Soar's normal behaviour. When Soar is performing a task for which the model increment directly supplies the necessary knowledge, it works in a problem space in which, from its current state, it repeatedly chooses an operator, and then applies that operator to the current state to get another state which it treats as its new current state. When there is uncertainty about which operator to apply, Soar drops into a subgoal whose purpose is to resolve the uncertainty by selecting one of the operators. Without any further information, i.e. with a null knowledge increment, UWM's default behaviour is to resolve uncertainty between operators by evaluating each of the candidates, which it does by applying the operator to the current state in a separate context, and seeing where it leads. This process generates, as we have noted, a form of depth-first search, which is

Consider now what happens if we add to the UWM a method increment consisting of (1) information about the task domain, sufficient to evauate states, say on a simple numerical scale, and (2) an item of general control information, telling Soar that an operator leading to a state with a higher numerical evaluation than another, is preferable to the other. The effect of that method increment is that when Soar is uncertain about the choice of operators and is considering the options, it will select as best the operator which leads from the current state to the adjacent state with the highest evaluation. What emerges is behaviour described by the classical weak method of *steepest-ascent hill climbing*. It should be noted that the basic "engine" driving behaviour remains that of Soar/UWM, while what makes the behaviour take the form of hill climbing is the method increment.

Of interest for our present purpose is Laird's observation that not all the weak methods can be generated in this fashion. There exists a group of weak methods, clustered around breadth-first search and including best-first search, which do not lend themselves to description as incremental variations on UWM. The reason is not hard to see. UWM's "own" behaviour, of depth-first search, is not only economical of memory demands, but the storage it does require exhibits a strong property of "locality" in both time and space. The information that has to be stored in conjunction with a given state is generated at about the time that the state is the current state, concerns some other state adjacent in the search space, and can be discarded at around the same time as the state itself is abandoned. None of this is true for the "difficult" weak methods. Their "breadth-ness" means that they have heavy storage demands, scattered across the search space, which has to persist over time. Consequently, they require a processing regime fundamentally at variance with the UWM, in which all existing states must be searched in order to find the next one to treat as current.

Now, Soar is a computationally universal architecture, so it is not actually impossible to get it to perform breadth-first search. But to do so, the analyst has to write a model increment which essentially consists of a self-contained program for breadth-first search, or whatever. Unlike the case with hill climbing, where the behaviour emerges jointly from the architecture and the method increment with a distinct contribution being made by each, for breadth-first search the behaviour is determined almost entirely by the model increment (i.e. the production rules given to Soar) with almost no contribution from the architecture. The model cannot be decomposed in a meaningful or useful way into UWM together with a method increment. The breadth-first program achieves its ends by brute force, as it were, neither taking advantage of nor making concessions to the properties of the architecture. It might almost as well be written in a standard programming language.

Before we leave this topic, we should note that while the constraints of UWM are, in the terminology introduced earlier, soft rather than hard — in the sense that they do not make it literally impossible for Soar/UWM to exhibit, say, breadth-first search — they do nevertheless, in some informal but important sense, clearly delineate model increments that co-operate with the architecture from those that fight against it. A soft constraint is not necessarily a weak constraint. The contrast between the two classes of methods is clearly marked.

2.2.2 Compliant models

What we take from this discussion of Laird's analysis is a subtly different conception of the

model increment to architecture is not at all like that of program to programming language. It is not the case (at least for Soar) that the architecture simply provides a framework within which the model is "programmed". Instead, the architectural mechanisms can propose a course of action. The model increment can go along with that proposed action, or it can add to the action (which is what is done by the method increment for most of the weak methods), or it can oppose and override the architecture's proposals (which is what has to happen to yield breadth-first search), or anything in between. In this way, we can see the role of the model increment as being not so much to generate behaviour, because the architecture will behave anyway, but rather to modify or *modulate* the behaviour that would otherwise occur.

Models built within a cognitive architecture clearly vary in the extent to which they follow the direction set by the architecture. We refer to this property of the model as its compliancy. A compliant model, like that for hill-climbing, allows the architecture to play a significant role in determining behaviour. A non-compliant model, like that for breadth-first search, consistently overrules the architecture. (Young [1982] provides an earlier attempt to define compliancy, but the version given here is considerably more precise. The idea goes back at least to the notion of "natural methods" introduced by Moore & Newell [1974].)

For Soar, at least, this idea of compliancy can be translated into concrete terms. Because Soar, no matter how little or how much knowledge it is given about a task, is usually in a position to behave, it always has proposals for the selection and sequencing of actions — what steps should be taken, and in what order. A model is compliant to the extent that it follows those proposals. In the extreme case, Soar offers its "default" behaviour, as described above. If the model increment defines a set of task-relevant possible moves, but provides no control information — i.e., adds no knowledge about the selection and sequencing of the moves — then the default behaviour will be exhibited. A highly compliant model increment will — like the method increment for hill-climbing — add just a little control information, that occasionally directs the processing in a direction determined by the model, but most of the time the decisions about selection and sequencing can still be left to the architecture. Models that are less compliant will themselves specify more and more about the processing, leaving less and less to the architecture, while a highly non-compliant model provides essentially a complete algorithm for the desired behaviour, permitting no influence to the architecture.

3. Soar's learning mechanism

In this section we continue our investigation of the relationship between cognitive architecture and model increment, by examining cases where the architecture imposes hard constraints on the possible models. We consider a number of empirical regularities describing how people interact with computers, and a number of models constructed in the Soar cognitive architecture, and attempt to determine whether the behaviour of the model is a consequence of the architecture or of the encoded knowledge, i.e., of the particular model increments. In particular, this section examines the role of an aspect of Soar's learning mechanism called backtracing. Due to this mechanism, Soar models necessarily exhibit certain behaviours seen in HCI, but must be configured with particular model increments in order to capture others. We draw on a number of examples taken from a model of the acquisition of task-action mappings (Howes and Young, 1996).

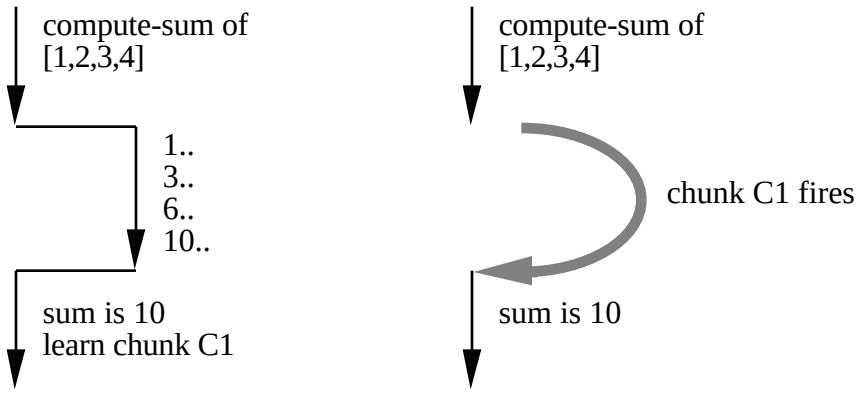

Figure 1a: Before chunking. Figure 1b: After chunking.

Figure 1: Chunking encodes an episodic memory of problem solving activity. (Reproduced from Howes & Young, 1996)

Soar's learning mechanism is known as chunking. It is an explanation-based mechanism in the sense of Mitchell (1986; see Rosenbloom & Laird, 1986). It learns by finding the conditions for why a particular conclusion was drawn and then creating new rules, or chunks, that propose the conclusion given the conditions.

Chunking can be viewed as a mechanism that caches the result of problem solving so that on future occasions the result can be retrieved without further processing. As an example consider the task of computing the sum of a list of numbers: Compute-sum of [1,2,3,4]. Imagine that the algorithm used consists of starting with a sum of S = 0 and then adding the value of each number in the list to S. S would first take the value 1, then 3, then 6, and lastly 10, which would be returned as the result. This process can be visualised with Figure 1a. It results in acquisition of the chunk:

C1: **IF** the task is to compute-sum of [1,2,3,4]
THEN respond "sum is 10."

The action or right-hand-side of chunk C1 is to return the result "sum is 10." The chunk's conditions, or left-hand-side, consist of all of the items in the initial state that were required to find the result, which in this case means the function name and the numbers that were summed.

The chunking mechanism models a person who encodes an episodic memory of their experience computing sums. The acquisition of chunk C1 buys the problem solver an efficiency advantage. The chunk is stored in a parallel recognition memory and on future occasions will propose its answer as soon as its conditions are met, thereby obviating the need to recompute a result by summing the individual numbers in the list. This process is illustrated with Figure 1b: the calculation needed in Figure 1a has been replaced by the firing of chunk C1. In the terminology of Explanation-based Learning (Mitchell, 1986) the chunk is an *operationalised* form of the knowledge of how to compute sums: it improves the efficiency of the problem solver.

The conditions of chunk C1 illustrate the consequences of Soar's *backtracing* mechanism. In the next section, we will see that backtracing has what may at first appear to be surprising consequences. Backtracing ensures that the conditions of a chunk that returns a result R consist of all of the items used in order to form R. In this paper we consider a 'strict' version of Soar in which backtracing cannot be avoided.[2] Backtracing can be avoided in the current version of Soar by the use of a 'best' preference. We believe that the arguments advocated by Vera, Lewis & Lerch (1993) and expanded in this paper support the view that 'strict' backtracing should not be avoided in the Soar architecture.

3.1 Users acquire display-based skills

Larkin (1989) coined the term *display-based* to refer to skills that are inherently dependent on cues from the environment. She describes how, when making coffee, people do not have to remember the sequence of goals. Instead, the state of the external world cues which tasks can be done when, e.g., water cannot be poured into a mug until the mug has been retrieved from the cupboard.

There has been much interest in studying display-based skill within HCI (Draper, 1986; Mayes, Draper, McGregor & Oatley, 1988; Kitajima, 1989; Howes & Payne, 1990; Briggs, 1990; Payne, 1991; Vera, Lewis & Lerch, 1993; John, Vera & Newell, 1994; Howes & Young, 1996; Howes, 1994; Kitajima & Polson, 1995; Bauer & John, 1995; Franzke, 1995; Altmann, Larkin & John, 1995). An important result in this literature is that under certain circumstances, expert users of computer systems may be incapable of recalling how to perform a task (when away from the device) even though they do it regularly in their everyday work (Mayes et al, 1988; Payne, 1991).

Vera et al.'s model of display-based skill offers a Soar account of these results. It is a model of instruction taking that learns how to get cash from an Automatic Teller Machine (ATM) and it shows how the backtracing mechanism necessarily leads Soar to acquire display-based chunks for interactive tasks, i.e. tasks for which actions are directly cued by the device. That is, the chunk conditions refer to the state of the display. Applied to the task of opening a file in Microsoft Word, Vera *et al*'s analysis yields the chunk C2:

C2: **IF** the task involves using a file
& the item "file" is on the display
THEN move the mouse pointer to the item "file."

The chunk is display-based, the "file" item must be present on the display for it to fire. Why do we claim that Soar must learn chunks, like C2, with display-based conditions? We will consider **three examples**, (1) learning from instruction; (2) learning from exploration, and **(3) comprehending a computer program**. We show that the claim follows in each case.

3.1.1 Learning from instruction

If Soar is given the instruction "move the mouse to the file option" then it can determine that the action required is to move the mouse pointer to the "file" item on the display, and it can

[2] In the version of Soar current when this paper was written backtracing could be avoided by

learn a rule that has this action as its right-hand-side.[3] But what will be the conditions of the learnt rule? The backtracing mechanism of Soar determines rule conditions by collecting together the information that contributed to the proposal of the action. In the compute-sum example, these conditions were the function name and the numbers that were summed. They can be thought of as the *reasons* that an answer of 10 was proposed. However, in the "move the mouse to the file option" example, the reasons that Soar proposes to move the mouse to the "file" item are because the model was instructed that this was the right thing to do, and because the model has successfully provided an interpretation, or rationale, for the instruction in terms of the task. Therefore, rather than chunk C2, Soar actually learns:

C3: **IF** the task involves using a file
& there is an instruction "move the mouse to the file option"
THEN move the mouse pointer to the item "file."

This chunk is not what is required, and it raises a problem. The chunk in fact appears to be useless, as it will not recall the required action — "move the mouse pointer to the file option" — given just the task description and the display. Instead, it will fire only when an instruction is present. The conditions seem to imply that despite the fact that Soar has been instructed how to do the task, it will continue to require instruction on subsequent trials.

C3 is an *instruction-based* chunk. Chunks like C3 form whenever instruction is used to determine the right-hand-side of a rule. The apparent uselessness of instruction-based chunks seems to suggest that whilst chunking is good at learning the results of computations that the problem solver already knows how to perform (e.g. sums), it is unable to learn new rules that do not derive from knowledge the system already has. This problem, termed the *data-chunking* problem, was first observed by Rosenbloom, Laird & Newell (1987).

Fortunately, Rosenbloom et al. (1987, 1988), Rosenbloom & Aasman (1990), and Vera *et al* (1993) offer a way to use instruction-based chunks when there is no instruction available. The first part of the solution is to realise that these chunks can be used to *recognise* instructions that have been given on a previous occasion. When an instruction such as "move-to file" is given for the first time, a Soar model must problem solve in order to understand the instruction and the instruction-based chunk C3 is formed as consequence. But when the model is given the same instruction a second time, chunk C3 will fire, obviating the need to redo the interpretation. The fact that the chunk has fired can be used by Soar to infer that the "move-to file" instruction had already been given on some previous occasion, i.e. to recognise the instruction. The second part of the solution is to propose that in the absence of external instruction, the way to recall what was instructed is to *reconstruct* the instruction internally, i.e. guess what the instruction might have been, and then hope that one of the instruction-based chunks fires, thereby "recognising" which instruction had previously been given. This *generate and recognise* model of memory retrieval was proposed many years ago in the psychology literature (Anderson & Bower, 1972; Kintsch, 1970).

The question then is: How do computer users construct appropriate guesses? A number of authors have reported models that use the external display as a generator of possible actions (Howes & Payne, 1990; Kitajima & Polson, 1995; Howes & Young, 1996; Bauer & John, 1995; Altmann, Larkin & John, 1995). Vera *et al* (1993) tie this idea into Soar as a solution to

[3] Here, we assume that the task includes the specific lexeme "File" to be selected. In general a

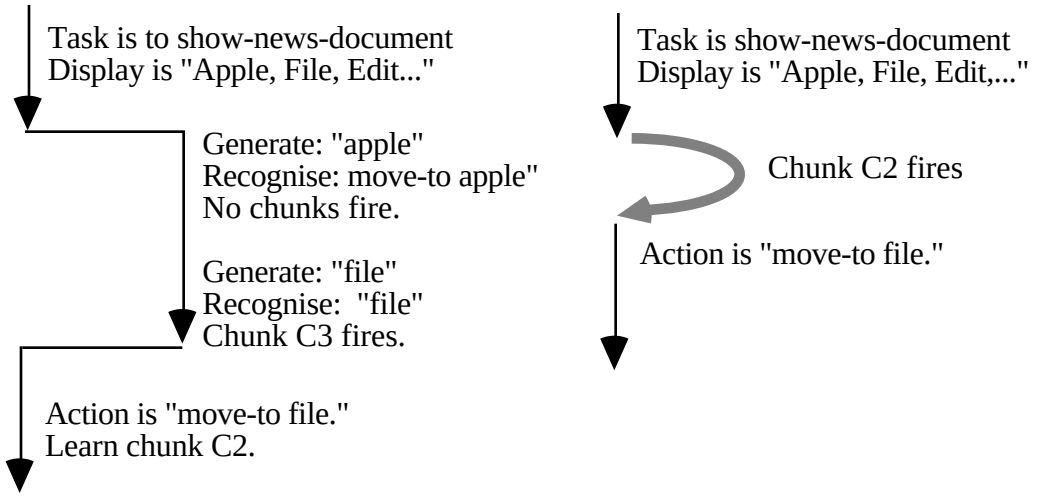

Figure 2a: C3 recognises the action. Figure 2b: C2 recalls the action.

Figure 2: Learning to recall an instructed action by generating and recognising candidate actions. (Reproduced from Howes & Young, 1996.)

the reconstruction of instructions. They suggest that display items are examined until an instruction-based chunk fires. Soar must examine each display item in turn (e.g. " Apple, File, Edit...") and internally "imagine" that an instruction has been given to select that item. Display items continue to be examined until the conditions of some instruction-based chunk are met, whereupon the chunk fires, recognising the display item. In this way the device display acts as a "reminder" of the content of the instructions.

For example, after Soar has learnt a set of instruction-based chunks for a task, when it starts the same task for a second time there are two possible courses of action: Either ask for instruction again, or attempt to remember the instruction by generating plausible possibilities and recognising one. Suppose the model chooses to generate and recognise, and that it selects as generator the labels on the display. First, it examines the "apple" menu item and constructs the instruction, "move-to apple," but no instruction-based chunk fires. Next it examines the word "document" and constructs the instruction, "move-to document," but again no instruction-based chunk fires. Then it examines the word "file" and constructs the instruction, "move-to file." For this instruction the conditions of chunk C3 are met, so the chunk fires, and proposes the action of moving the mouse to the "file" item. The required action has thereby been recalled by the use of a generate and recognise algorithm (Figure 2a). In this process a new chunk is learned that operationalises the problem solving that found the action. On this occasion there was no external instruction, only an internally imagined one. What did contribute to the formation of the chunk was the display and the task. Hence, the new rule (C2) does not require the instruction to be present in order for it to fire (Figure 2b), but it is display-based.

To summarise this first example, we see that the way in which Soar's backtracing mechanism determines chunk conditions means that it must use a generate and recognise method in order to

recall instructions. In cases where the external display is used as a generator, this process necessarily leads to display-based chunks, and the architecture therefore determines that Soar models capture the data of Mayes et al (1988) and of Payne (1991).

3.1.2 Learning from exploration

Soar's backtracing mechanism also constrains the exploratory acquisition of knowledge about the sequence of actions to achieve a goal. Ayn (Howes, 1994) is a model (implemented in Prolog and Soar)[4] that learns menu structures by acquiring recognition chunks that encode which options the model has selected and which lead to the goal. For example, when Ayn finds a goal state after some exploration, it is then in a position to learn that the previous action leads to success. Once again, all the conditions of the chunk are external to the problem solving, and the chunk will be recognitional:

C4: **IF** the previous action was to select "open"
& the current state is a goal state
THEN the action "open" is correct.

In order to use that "outcome-based" chunk to guide its behaviour towards the goal state, Ayn has to employ a generate-and-recognise technique analogous to that for instruction learning. Using the options available in the previous state as a source, it has to generate each action and "imagine" it leading to the goal state. For the correct choice, a recognition chunk will fire. Thus, does the option "New" lead to success? No recognition chunk fires. Does "Close" lead to success? No chunk fires. Does "Open" lead to success? Chunk C4 fires, so "Open" must be the correct choice. The result of that problem solving is a new chunk recalling that "Open" leads to the goal state, but it is still display-based in so far as one of its conditions is that the "Open" option be available on the screen.

To summarise the first two examples, if the display is the only means for a Soar program to generate possible actions then the acquired skill will be display-based. Further, this appears to be true for both learning from instruction and learning from exploration. Thus, through its chunking mechanism, the Soar architecture captures the HCI data indicating that expert performance is display-based (Payne, 1991; Mayes et al., 1988). The architecture achieves this by virtue of its backtracing mechanism for determining chunk conditions, which requires the use of generate and recognise for the acquisition of recall knowledge.

3.1.3 Comprehending a computer program

Altmann and colleagues (Altmann, Larkin & John, 1995; Altmann, 1996) describe a Soar model of a programmer's scrolling behaviour during the comprehension of a computer program. The model captures circumstances in which the programmer uses scrolling in order to display hidden information. Consider, for example, a situation in which, whilst comprehending a Pascal program, the model comes across a definition for a function called 'calculate_cost' (footnote).[5] The model will attempt to comprehend what this function achieves and in so doing will encode recognition chunks in exactly the same way as the models in the previous examples.

---

[4] The Soar implementation of Ayn was done in collaboration with John Rieman.

[5] For clarity, we have assumed a Pascal program where Altmann's original model dealt with a

Now imagine that at a later time, after the definition of 'calculate_cost' is no longer on the display, the model encounters a call to the function. In this situation, the model again needs to know what calculate_cost does, and it again sets the goal of comprehending the function. In order to do so, it needs to use the definition of calculate_cost, which because of the backtracing constraint it has not encoded in a recallable form. The model does not know where in the program file the definition of calculate_cost is located. However, if it recognises having previously comprehended the function then it can use this knowledge to direct scrolling to parts of the program file that have already been seen. For this reason the model invokes a scroll action until the required function is displayed.

In this way Altmann et al.'s model captures the fact that when trying to understand a program, programmers do not simply read through the file line by line, rather they jump around examining and re-examining parts of the code, as they are needed, in a non-linear fashion. Because of Soar's backtracing mechanism, this behaviour is exactly what emerges in Altmann et al.'s model.

3.2 Recapitulation: Recognition, reconstruction, and recall in Soar

It may at this point be helpful to summarise what we have discovered from the analysis of the relation of Soar's chunking mechanism to display-based skills, because the discussion of the other empirical regularities we shall examine will make repeated reference back to these basic results. We have worked through the following argument:

- A Soar model cannot directly acquire rules that recall information originating from the external world. The desired information can be captured on the right-hand-side of a chunk, but because the rationale for the chunk is derived from the environment, the information will be included on the left-hand-side too. The chunk is therefore "circular", for instance an instruction chunk that depends upon the presence of the given instruction. These chunks arise automatically from problem solving, e.g., the process of interpreting an instruction.
- Such circular chunks encode recognition knowledge. Given the instruction (and assuming the other conditions are met), the instruction-based chunk will fire, in effect telling the Soar model "Yes, I have seen that instruction before".
- In order to acquire a recall chunk that gives the Soar model information without that information present, the model must first reconstruct the information. It does so by generating suitable candidates, and using the circular recognition chunks to identify which candidate is correct.
- In many HCI settings, the most easily available source for the generator — or the only source — is the external display. Recall chunks formed from such a generator encode a display-based skill, because the chunks depend upon the external presence of the cue for action (such as an item to be selected).

We shall see that some HCI situations demand the use of an internal generator, because no external source is available. In the analysis of the remaining HCI phenomena, much of the

discussion will focus on this question of the ease or difficulty of generating the plausible candidates.

3.3 Keyboard users need meaningful and mnemonic command names

The role of meaningfulness and mnemonics in the design of keyboard-based devices has been studied over many years (e.g., Barnard & Grudin, 1988; Furnas, Landauer, Gomez, Dumais, 1987). It is a truism that a good mnemonic helps recall. Consider the user of a keyboard-based device who, having previously been told, needs to recall the name of the command that gets rid of a file. What was the command? Was it some arbitrary letter sequence, CTRL-X perhaps? Or was it some arbitrary word, perhaps "rabbit"? Alternatively, the command name may reflect the semantics of the task. It may be something derived from "get rid of", e.g. "delete", or "remove", or some other word with a meaning related to the task. It may also be an abbreviation of one of these words. Data support the view that meaningful and mnemonic command names help recall. Can Soar's backtracing mechanism capture this effect?

In Unix, the command that gets rid of files is "rm", derived from the word "remove". As in Section 3.1, in order to learn this command from instruction, a Soar model must first acquire instruction-based chunks:

C5: **IF** the task involves getting rid of something
& there is an instruction to use the word "remove"
THEN use the word "remove"

C6: **IF** the task involves using the word "remove"
& there is an instruction to use the first two consonants
THEN use the first two consonants

As we have seen, when given the "get rid of" task again the only way that a Soar model can recall the instructions it was given is by a process of generate and recognise. If we assume that the model has the semantic knowledge to generate appropriate words from the "get rid of" task description, then an internal generator based upon processes of semantic and lexical search can generate plausible words such as "delete", "cut", "throw out", and eventually " remove". (The semantic knowledge may be derived from a technique such as that suggested in Miller & Charles [1991], and programmed into Soar's production memory.) Once "remove" is generated, chunk C5 will fire, indicating that "remove" was the instructed word. Subsequently a second process of generate and recognise, this time generating types of abbreviation (first two letters, first letter, …, first-two-consonants) should lead chunk C6 to fire.

Now contrast the case of "rm" to another Unix command, this time for finding files with particular contents. The required command is "grep", which stands for "get regular expression". Given the instructions for this command a Soar model would learn the instruction-based chunks:

C7: **IF** the task involves finding a file by contents
& there is an instruction to use the word "get regular expression"
THEN use the word "get regular expression"

C8: **IF** the task involves using the word "get regular expression"
& there is an instruction to take the first letter
of the first word, the first two letters of the second ...
THEN take the first letter of the first word, the first two letters of the second ...

Chunks C7 and C8 encode the instructions, but they can only be used for recognition. In order to extract the information implicit within them, a generate and recognise process must be used. But on what basis can plausible candidates be generated? It would be hard to justify a Soar model that, given the task to find a file by contents, just happened to generate the phrase "get regular expression". Although this may be an accurate model for some, most users who do not already know the command are unlikely to generate it spontaneously. Instead the model must include back-up generators that operate when semantic generators fail. What the back-up generators are and how they work is less clear. It may even be the case that an elaborated set of recognition rules would have to be acquired to guide the generator. Perhaps separate recognition rules are learnt for each word, others that recognise the first letters of the word, and others that recognise syntactic types. The details are beyond the scope of this paper, but it is clear that the generate and recognise process would be less constrained for "grep" than for "rm", and — because the search space is potentially much larger — more time consuming and prone to failure.

In summary, because a Soar model capable of learning keyboard commands from instruction requires the use of semantic generators, computer interfaces designed with meaningful and mnemonic command names will be easier to learn than those employing arbitrary command names. This observation indicates that Soar is aligned with broad aspects of the psychological data on command naming.

3.4 Display-based devices are easier to learn than keyboard-based devices

Display-based devices constrain the next action that the user selects by forcing a choice from a limited set of externally presented options. In contrast, keyboard-based devices require the user to recall sequences of command words and type them in. This constraint gives display-based devices a considerable advantage over keyboard-based devices for exploratory learning. This advantage has been demonstrated by Charney, Reder & Kusbit (1990) and is illustrated by the fact that the overwhelming majority of studies of exploratory learning in HCI are carried out using display-based devices (Franzke, 1995; Rieman, 1994; Polson & Lewis, 1990; Carroll & Rosson, 1987; Lewis, 1988; Robert, 1987). The difference in learnability between display-based and keyboard-based devices under exploratory learning conditions is easy to understand without recourse to the architectural learning mechanism. In one case actions are generated from the external world and in the other they must be generated from the user's knowledge base. If users have had no prior exposure to a keyboard-based device, there is no reason to suppose that they can generate the right action without help (see Section 3.3). In fact Furnas et al. (1987) have shown that even the most careful design can expect to achieve only low guessability of command names without external support.

But is there a difference in learnability between keyboard-based and display-based even when users are instructed? Under instructional learning conditions, users can be provided with sufficient and informationally equivalent advice (see Larkin & Simon, 1987 for a definition) on how to use either kind of device. However, the fact that users can be given instructions does not necessarily mean that they can learn them. As we have seen, users cannot learn to retrieve arbitrarily complex instructions at will. One of the factors affecting the likelihood that instructions will be retrieved is the cues provided by the display of the device being learned. Does the backtracing mechanism of Soar predict a difference in learnability, or can an analyst

program Soar with model increments that find either kind of interface equally easy or hard to learn?

Once again, in order to acquire rules that directly recall instructions, a Soar model must be programmed with a generate and recognise algorithm. Howes & Young (1996) report a Soar model, called Task-Action Learner (TAL), that uses generate and recognise to learn both Microsoft Word tasks and Unix tasks. TAL uses the display as a source of candidate instructions for the display-based device (see Section 3.1). But to use a keyboard-based device, TAL has to employ four internal generators in a hierarchical fashion. The model works downward from the task description, decomposing it into finer grain subtasks until an action that can be performed on the device is found. It first determines which feature of the task is to be communicated to the device (e.g. Effect=get-rid-of); then determines which word is to be used in order to communicate the feature (e.g. "remove"); then determines whether and how to abbreviate the word (e.g. take first two consonants); and lastly, computes the abbreviation ("rm") and types the characters on the keyboard. The four generators use long-term memory as their source of candidate actions, each attempting to reconstruct one of the steps.

The internal generators define a search space in just the same way as do the menu labels and icons on a display-based device. In both cases, the instruction-based chunks can be used as recognitional control knowledge to help the problem solver find the instructed path through this space. For Human-Computer Interaction, the important differences are: (1) the internal search space required for keyboard-based devices is almost invariably larger than the external search space required for display-based devices; (2) the internal search space is subject to differences in individuals' knowledge, in a way that the external search space is not (see Furnas et al., 1987); (3) with keyboard-based devices, Soar — like people — must rely on mnemonic encoding of commands in order to make the process of reconstructing instructions reasonably efficient (see Section 3.3).

In consequence, we can say that the backtracing mechanism forces Soar models to exhibit a learnability difference between display-based and keyboard-based devices. The difference stems from the architecture, not from what the analyst chooses to put in the model increments.

3.5 Consistent interfaces are easier to learn and use

There are many ways in which a computer interface can be consistent. Our primary interest here is in the *internal consistency* of device methods, that is the consistency of methods, or task-action mappings (Payne & Green, 1986), with each other for a single device. The internal consistency of a device constrains learning when users assume that task-action mappings learned for one task will apply to all semantically similar tasks. For example, if a method for opening a file has a certain syntax, then people will assume that the mapping for closing a file will have the same syntax.

Consider *syntactic* consistency and *mnemonic* consistency. An interface is syntactically consistent if the order in which types of items occur is the same across a broad set of tasks. For example, if in the command to copy a file the word "copy" must be typed before the description of the file, then the user may expect that in the command to delete a file the word "delete" must also be typed first. The mnemonics of a language are consistent if the way in which abbreviations are derived from command names is the same across many tasks. For

example, if the command to "remove" a file is executed by typing the first two consonants of the command name, then it will help learnability if other commands are executed also by taking the first two consonants of their names.

Payne & Green (1986, 1989) report a number of experiments to show that consistent interfaces are easier for people to learn and use. For example, Payne and Green (1989) found that the number of syntax errors made by subjects was greater for devices with inconsistent grammars. Other studies, for example, Barnard, Hammond, Morton, Long & Clark (1981), Payne (1985), Kellog (1987) and Lee, Foltz & Polson (1994), support these findings. Can the backtracing mechanism of the Soar architecture predict these results? Or is it the case that a model constructed in Soar can be programmed to find either a consistent interface or an inconsistent interface easier to learn?

Howes & Young's (1996) TAL (see Section 3.4) uses assumptions from the Task-Action Grammar theory of command-language consistency (TAG: Payne & Green, 1986, 1989). Tasks in TAG are represented as feature/value pairs. For example, a task to open a file might be described by [Effect=open, Object=file, Name=newsletter]. This representation provides TAL with the flexibility to learn chunks that are sensitive only to the feature name and not to the feature value. For example, the following chunk encodes the knowledge that for all tasks that contain an Effect feature, the first subtask should be to communicate the value of this feature to the device:

C9: **IF** task contains the feature Effect
& there was no previous subtask
THEN make the subtask to communicate Effect.

These generalised chunks align TAL with Payne and Green's data. TAL learns consistent interfaces faster than inconsistent ones because the generalised chunks mean that less instruction is required, and it makes errors on inconsistent interfaces because of the now over-general chunks.

However, the generalised chunks are acquired only because of the way that Howes & Young programmed TAL to generalise across feature values. There appears to be no architectural constraint forcing models to be that way. A Soar model could just as easily have been programmed to learn rules of the form:

C10: **IF** task contains the feature Effect=open
& there was no previous subtask
THEN make the subtask to communicate Effect=open

A model that used these specific chunks would learn a consistent interface as slowly as an inconsistent one, since in either case it would have to acquire an independent rule for each value of the feature, and it would fail to make errors due to over-generalisation.

A tentative conclusion[6] must be that Soar's backtracing mechanism does not force the learning of consistent interfaces to be easier than that of inconsistent ones. Unlike for the other HCI

---

6 We say "tentative conclusion" because the analysis here is only shallow. It takes no account, for example, of how the knowledge about what aspects of the task to attend to is itself acquired. A deeper analysis might perhaps reveal ways in which Soar is biased towards learning the generalised rules. Or it might not. The topic lies beyond the scope both of this paper and of our

phenomena we have examined so far, a Soar model that makes predictions about the effects of interface consistency in line with the empirical data does so, not because of anything inherent to Soar, but by virtue of the way the model increment has been programmed.

3.6 Learning locational knowledge

An important aspect of skill for display-based devices is knowledge about where items are located on the display (e.g. see Lansdale, 1991). A user who knows that the 'wastebasket' is located in the bottom right of the display will not have to spend time scanning the display looking for it. Cognitive psychology has seen a long running debate about whether locational knowledge is acquired automatically. For example: Andrade & Meudell (1993) claim their data shows automatic memory for locations; Naveh-Benjamin (1987) claims that spatial tasks fall upon a continuum of ease-of-learning; while Lansdale (1995) claims there is little evidence either way.

Does the Soar architecture force Soar models to be aligned with either side of the debate? Operationalising the notion of automaticity for a cognitive architecture such as Soar is difficult. However, here we argue that Soar will not automatically learn to recall locational information. Instead, a Soar model would have to make a deliberate effort to learn locations.

In an unreported Soar model programmed by one of the authors (AH), the task involves "visually" finding the right menu item by its label (e.g. Format) on a simulated menu and then moving the mouse to the location of that label. Chunks such as C11 are formed:

C11: **IF** the task is to move the mouse to "Format"
& "Format" is in the middle
THEN move the mouse to the middle

This chunk adds knowledge that recognises the location of Format. It can be learned only after the location has been determined externally. As before, the acquisition of these recognition chunks is automatic and a direct consequence of the application of the backtracing mechanism to the current problem solving task.

As before, because locational knowledge is acquired initially within recognition chunks, a generate and recognise process is needed for the knowledge to be captured in a form in which it can be retrieved and used to guide behaviour. To implement a generate and recognise for the target "Format", the problem solver might pose the question, "was Format to the right?", "was it to the left?", and continue doing this until the options are exhausted or until a chunk such as C11 fires. However, whether a model engages in this process of generate and recognise is a matter of strategic choice. Because the item to be found (e.g. "Format") is available on the device display, the location does not have to be retrieved from memory for the task to be achieved. It may be the case that scanning for the item is a more efficient way of achieving the task on this trial than taking the time to generate and recognise. Thus, although learning to recall a location would have a long term benefit (i.e., on trials subsequent to its acquisition it will decrease performance time), on the trial in which it is acquired, the process of acquiring the chunk may increase performance time for the overall task.

In the case described so far, knowledge used to guide the strategic choice is part of the model increment, which of course may itself be dependent on knowledge of the task environment. In other cases the task environment *dictates* that locational knowledge be acquired. Bauer & John

(1995) describe a Soar model of a player of a Nintendo video game. In order to do well at the game, the player must learn to avoid enemies by jumping over them. To jump successfully the distance between the player and the enemy must be exactly right. If the distance is too short, the player will not have time to get off of the ground; too long and the player will land before the enemy has passed underneath. The only way for the Soar model to learn the optimal distance is, as before, by first learning recognition chunks and then using the generate and recognise algorithm to acquire recall chunks. The recognition chunks encode the distance of the enemy when the jump was launched and the result of the jump (success or failure). Generation involves "imagining" various distances and outcomes and choosing behaviour in the current context according to what is recognised. As in the case of learning to recall menu locations, the architecture determines that to learn to recall jump locations, generate and recognise is required. However, the two examples differ in that the task environment determines that if the Nintendo locations are not learnt then the player will be 'killed' and the task will not be completed. Another Soar model that learns to operate in a task environment where locational knowledge is essential is described by Miller, Lehman & Koedinger (submitted).

To summarise this discussion of locational knowledge, we can say that as a consequence of the backtracing mechanism the Soar architecture may acquire recognition chunks of locations automatically, but whether recall chunks are subsequently learnt from these recognition chunks depends on whether the problem solver decides to take time to go through the generate and recognise process. This choice is determined not by the architecture but by the task environment or by the way in which the model trades off immediate performance time against benefit of learning.

4. Discussion

4.1 Constraints from Soar's learning mechanism

Section 3 examined in some detail the ways in which Soar's chunking mechanism determines aspects of the behaviour of Soar models that learn. We have seen examples of "hard" constraints, where the cognitive architecture necessarily enforces certain properties on the model. We saw, for example, how Soar models that learn to operate a certain class of interactive device will necessarily acquire a display-based skill, whether they learn by instruction or through exploration. The backtracing mechanism of the Soar architecture leads automatically to the acquisition of recognition chunks, which in turn leads to the need for a generate and recognise mechanism in order to learn recall chunks by reconstructing the context of the recognition chunks. This mechanism in turn determines that a Soar model of a user must *necessarily* acquire display-based skills. Similarly, we have seen how, given certain plausible assumptions about a device, a display-based interface is necessarily easier for a Soar model to learn than one that is keyboard-based. And again, the nature of Soar's chunking mechanism forces the result that learning by instruction will be easier for meaningful or mnemonic command names, which allow users to reconstruct instructions for keyboard-based devices.

The generate and recognise model of retrieval has been established for many years in cognitive psychology (e.g. Bahrick, 1970). The similarity between Soar's emergent technique and other generate and recognise models (e.g. Anderson & Bower, 1972; Kintsch, 1970) has been

observed by Rosenbloom et al. (1987). The arguments made in the current paper add a number of HCI phenomena to the list of those explainable in terms of a generate and recognise model. Further, the fact that generate and recognise is a consequence of the Soar architecture adds weight to the claim that Soar's learning mechanism has psychological plausibility.

In contrast to the strong role attributable to Soar's backtracing mechanism in accounting for the phenomena just mentioned, other empirically observed regularities seem to depend upon the assumptions built into particular Soar models. For example, the TAL model requires less instruction and makes fewer errors when learning consistent as against inconsistent interfaces, because it is programmed with an algorithm that learns recognition rules generalised according to the semantics of the task, and the evidence is that people do the same. But Soar could just as easily be programmed with an algorithm that acquires specific recognition rules that do not generalise to whole categories of task. We could, say, construct a Soar model in which the recognition rule for an instruction to use the "cut" menu option to delete the word "rabbit" contains a reference to the word "rabbit". The model would then not make the generalisation that it should use "cut" for deleting other words too.

This topic of the initial acquisition of recognition rules does seem to be an area where the Soar architecture is currently under-constrained. As it stands, Soar can acquire in one learning event a recognitional chunk of arbitrary complexity, and will never confuse it with another chunk no matter how similar their conditions. We believe that a key research objective for people working with the Soar architecture should be the development of constraints on the acquisition of recognition chunks, to reflect the fact that people's ability to discriminate between objects is dependent on aspects of distinctiveness, familiarity, and frequency. Some Soar techniques (e.g. Miller & Laird, 1992) may offer solutions to this problem, as may perhaps other approaches based on ideas of discrimination networks (Richman, Staszewski & Simon, 1995), but at present they are not an integral part of Soar.

Lastly, while it has not been a major theme in this paper, we have also seen how some aspects of a model's behaviour are determined by the task environment. This point is by now so obvious that we will not labour it (see, e.g., Vera *et al*, 1993). It is, however, worth restating the role of the environment in determining strategic choices such as whether or not to make the effort to learn locational knowledge: In some environments locational knowledge is essential to the task, while in others it merely provides a gain of efficiency.

4.2 Role of the architecture in Soar models

We have seen that in the case of "hard" constraints, the Soar architecture enforces certain necessary properties on a model. For the properties discussed above, the backtracing mechanism of learning enforces these results: the architecture cannot be programmed with algorithms that avoid these behavioural outcomes. In such cases, the existence of the hard constraint makes it straightforward to assign responsibility for the phenomenon to the cognitive architecture. It becomes meaningful, and correct, to claim for example that *Soar predicts that learning to use an interactive device through exploration will result in a display-dependent skill.* We are able to go beyond an assertion such as "Within Soar, the modeller has the possibility of constructing models of learning that result in display-dependent skill", which implies that the modeller has a choice and that Soar would allow the construction of models

Conversely, we have seen cases where a feature of the model's behaviour is not dictated by the architecture. It appears, for example, that taking advantage of consistency in the interface to learn generalised rules is something that depends upon the strategic knowledge in the model increment, and about which the architecture itself is neutral.

The responsibility of the architecture in shaping the behaviour of models is not, however, always the black-and-white issue that these examples might suggest. It is not always the case that the architecture either enforces a property, or else is totally neutral about it. In Section 2.2 we examined soft constraints, where the architecture "favours" a property without actually enforcing it. We were led to the notion of compliancy to express the degree to which the model increment allows the architecture its say in determining behaviour.

This issue of compliancy is central to the questions we are addressing in this paper, about the consequences of constructing user models within a cognitive architecture and about how to allocate responsibility for the predictions made about users' behaviour between the architecture itself and the particular model increment. With models of a moderate to high degree of compliancy, a good part of the credit or blame lies with the architecture, which is playing a moderate-to-high profile role in determining the predictions of behaviour. In the extreme case, as we have seen with some of the models that exploit Soar's learning mechanism, the architecture may be imposing a "hard" constraint on the model, so that certain aspects of the predictions are the clear responsibility of the architecture. With low compliancy or non-compliant models, any guidance offered by the architecture is being disregarded or overruled by the model increment, with the result that the predictions of behaviour are due almost entirely to the model increment, i.e., to the way the model has been programmed.

5. Conclusions

The paper began by noting that a style of user modelling, in which models are constructed within a fixed cognitive architecture, raises novel problems of evaluation for both practical and scientific purposes. We asked about how the responsibility for any empirical successes or failures of such user models can be apportioned to the architecture within which the model is built, or to the model increment which specifies the particular model.

We saw that in some cases, it may be possible to give a clear-cut answer to the question. If the architecture imposes a "hard" constraint on the model, so that any model built within the architecture exhibits a certain feature (such as display-based skill), then we can clearly point to the architecture as being responsible for that feature. Conversely, if the architecture is neutral on some point, allowing models to be built that equally well display one feature or another (e.g., a learning advantage for consistent interfaces), then we can unambiguously attribute the feature to the model increment, i.e., to the way the model has been "programmed". We also saw that the question does not always permit such black-and-white answers. The behaviour of the model, after all, depends in general upon both the cognitive architecture and the particular model specification (as well as on the task environment), so the responsibility for the behaviour of the model must in some sense belong jointly to both.

We can approach that issue of joint responsibility from either side. Coming from the architecture side, as it were, we meet the notions of hard and soft constraints, where a hard

constraint enforces a property on the model, while a soft constraint merely makes it easier to build the model with a particular property than not. The notion of soft constraints is obviously graded, since there can be a greater or lesser difference in the ease of building the model one way or the other. Coming from the other side, i.e., from the side of the model increment, we encounter the idea of compliancy. A compliant model increment is one that allows the architecture to play a role where it can in guiding the model's behaviour. A non-compliant model is one that blocks the architecture's role, either by disregarding it or by opposing it. Again, compliancy is obviously a graded concept. Models can be more or less compliant with their architecture. (We note too that a model can be more or less compliant with the guidance being offered by the environment.)

We have seen also that, at least for Soar, we can concretise these ideas about compliancy. Soar has built in mechanisms for the selection and ordering of processing steps, though these can of course be overridden by additional control knowledge. A Soar model is compliant to the extent that it allows processing to follow the course proposed by these mechanisms. It is non-compliant to the extent that it overrides them.

Although doubts are sometimes expressed as to whether cognitive architectures have any empirical content, and therefore any scientific contribution to make to user modelling, our analysis shows that indeed they do. Architectures play their part by imposing theoretical constraints on the models constructed within them. In some cases, such constraints can be strong enough to *dictate* that models exhibit certain characteristic properties. In others, the architecture can influence the model more subtly: The extent to which it is allowed to do so depends on an aspect of the way a particular model is specified, which we have termed its compliancy.

Acknowledgements

Thanks to Bonnie John, Wayne Gray, Erik Altmann, and two anonymous reviewers for helpful comments on an earlier draft. The work reported in this paper was done as part of a project funded by the UK Joint Councils Initiative in Cognitive Science and HCI.

References

Altmann, E. M., Larkin, J. H., & John, B. E. (1995) Display navigation by an expert programmer: A preliminary model of memory. *Proceedings of CHI, 1995* (Denver, Colorado, May 7-11, 1995) ACM, New York. pp. 3-10.

Altmann, E.M., (1996) Episodic memory for external information. Doctoral Thesis, School of Computer Science, Carnegie Mellon University. CMU-CS-96-167.

Anderson, J. R. (1993) *Rules of the Mind*. Hillsdale, NJ: Lawrence Erlbaum.

Anderson, J. R. & Bower, G. H. (1972) Recognition and retrieval processes in free recall. *Psychological Review, 79,* 97-123.

Anderson, J. R., Douglass, S., Lebiere, C. & Matessa, M. (19**) The ACT theory and the visual interface. *Human-Computer Interaction*, THIS ISSUE.

Andrade, J. & Meudell, P. (1993) Is spatial information encoded automatically in memory. *Quarterly Journal of Experimental Psychology: Section A - Human Experimental Psychology, 46*, 365-375.

Bahrick, H. P. (1970) Two-phase model for prompted recall. *Psychological Review, 77*, 215-222.

Barnard, P. J. (1987) Cognitive resources and the learning of human-computer dialogues. In J. M. Carroll (Ed.) Interfacing Thought: Cognitive Aspects of Human-Computer Interaction. Cambridge, Mass: MIT Press,112-158.

Barnard, P. J. & Grudin, J. (1988) Command names. In M. Helander (Ed.), *Handbook of Human-Computer Interaction*, 237-255. Elsevier Science Puhlishers.

Barnard, P. J., Hammond, N., Morton, J., Long, J. B. & Clark, I. A. (1981) Consistency and compatibility in human-computer dialog. *International Journal of Man-Machine Studies, 15*, 87-134.

Barnard, P. J. & May, J. (1993) Cognitive modelling for user requirements. In P. Byerley, P. J. Barnard & J. May (Eds), *Computers, Communication and Usability: Design Issues, Research and Methods for Integrated Services*. Chapter 2.1, 101-146. Studies in Telecommunications. Amsterdam: North Holland.

Barnard, P. J., Wilson, M. & MacLean, A. (1988). Approximate modelling of cognitive activity with an expert system: A theory based strategy for developing an interactive design tool. *The Computer Journal, 31*, 445-456.

Bauer, M. I. & John, B. E. (1995) Modeling time-constrained learning in a highly-interactive task. *Proceedings of CHI, 1995* (Denver, Colorado, May 7-11, 1995) ACM, New York. pp. 19-26.

Briggs, P. (1990) Do they know what they're doing? An evaluation of word-processor users' implicit and explicit task-relevant knowledge, and its role in self-directed learning. *International Journal of Man-Machine Studies 32*, 385-398.

Carroll, J. M. & Rosson, M. B. (1987) Paradox of the active user. In J.M. Carroll (Ed.) *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*. Bradford Books/MIT Press.

Charney, D., Reder, L., & Kusbit, G. (1990) Goal setting and procedure selectoin in acquiring computer skills. A comparison of problem solving and learner exploration. *Cognition and Instruction, 7*, 323-342.

Draper, S. W. (1986) Display managers as the basis for user-machine communication. In D. A. Norman & S. W. Draper (Eds) *User Centered System Design*, 339-352. Erlbaum.

Franzke, M. (1995) Turning research into practice: Characteristics of display-based interaction. In I.R.Katz, R. Mack, L.Marks, M.B.Rosson, J.Nielsen (Eds) *Human Factors in Computing Systems: CHI'95 Conference Proceedings*. ACM Press. 421-427.

Furnas, G. W., Landauer, T. K., Gomez, L. W., & Dumais, S. T. (1987) The vocabulary problem in human-system communication. *Communications of the ACM, 30*, 11. 964-971.

Howes, A. (1994) A model of the acquisition of menu knowledge by exploration. In B.Adeslon, S.Dumais, J.Olson (Eds.) *Proceedings of Human Factors in Computing Systems CHI'94*. Boston, MA: ACM Press. 445-451.

Howes, A. & Payne, S. J. (1990) Display-based competence: Towards user models for menu-driven interfaces. *International Journal of Man Machine Studies, 33*, 637-655.

Howes, A. & Young, R. M. (1996) Learning consistent, interactive and meaningful device methods: a computational model. *Cognitive Science, 20, 301-356*.

John, B. E., Vera, A. H., & Newell, A. (1994). Toward real-time GOMS: A model of expert behavior in a highly interactive task. *Behavior and Information Technology, 13*(4),

Kellogg, W. A. (1987) Conceptual consistency in the user interface: effects on user performance. In H.J.Bullinger & B.Shackel (Eds) *Proceedings of Human-Computer Interaction - INTERACT'87*. The Netherlands: Elsevier.

Kintsch, W. (1970) Models for free recall and recognition. In D.A. Norman (Ed.), *Models of human memory*. New York: Academic Press.

Kintsch, W. (1988) The role of knowledge in discourse comprehension: A construction-integration model. *Psychological Review, 95,* 163-182.

Kirschenbaum, S. S., Gray, W. D. & Young, R. M. (1996) Cognitive architectures and HCI (workshop report). *SIGCHI Bulletin, 28* (2), 18-21.

Kitajima, M. (1989) A formal representation system for the human-computer interaction process. *International Journal of Man-Machine Studies, 30,* 669-696.

Kitajima, M. & Polson, P. G. (1995) A comprehension-based model of correct performance and errors in skilled, display-based, human-computer interaction. *Internation Journal of Human-Computer Studies, 43,* 65-99.

Laird, J. E. (1986) Universal subgoaling. In J. E. Laird, P. S. Rosenbloom & A. Newell (Eds), *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies,* 1-131. Kluwer Academic Publishers.

Lansdale, M. W. (1991) Remembering about documents: memory for appearance, format, and location. *Ergonomics, 34* (8), 1161-1178.

Lansdale, M. W. (1995) *Modelling errors in the recall of spatial location*. Technical report: Cognitive Ergonomics Research Group Loughborough University of Technology.

Larkin, J. H. (1989) Display-based problem solving. In D. Klahr, & K. Kotovsky (Eds) *Complex Information Processing: The Impact of Herbert A. Simon*. Hillsdale, NJ: Erlbaum. 319-342.

Larkin, J. H. & Simon, H. A. (1987) Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science, 11*, 65-99.

Lee, A. Y., Foltz, P. W. & Polson, P. G. (1994) Memory for task-action mappings - mnemonics, regularity and consistency. *International Journal of Human-Computer Studies, 40,* 771-794.

Lewis, C. H. (1988) Why and how to learn why: Analysis-based generalization of procedures. *Cognitive Science 12,* 211-256.

May, J., Barnard, P. J. & Blandford, A. E. (1993) Using structural descriptions of interfaces to automate the modelling of user cognition. *User Modelling and User-Adapted Interaction, 3,* 27-64.

Mayes, J. T., Draper, S. W., McGregor, M. A. & Oatley, K. (1988) Information flow in a user interface: the effect of experience and context on the recall of MacWrite screens. In D.M. Jones & R. Winder *People and Computers IV.* C.U.P.

Miller, C. S., Lehman, J. F., & Koedinger, K. R. (submitted) Goal-directed learning in microworld interaction. Submitted for publication.

Miller, C.S . & Laird, J. E. (1992) A simple symbolic algorithm for incremental concept acquisition. Artificial Intelligence Laboratory, University of Michigan, January 1992.

Miller, G. A. & Charles, W. G. (1991) Contextual correlates of semantic similarity. *Language and cognitive processes, 6* (1) 1-28.

Mitchell, T. M. (1986) Explanation-based generalisation: A unifying view. *Machine Learning 1,* 47-80.

Moore, J. & Newell, A. (1974) How can Merlin understand? In L. W. Gregg (Ed.), *Knowledge and Cognition*, 201-252. Potomac, MD: Lawrence Erlbaum.

Naveh-Benjamin, M. (1987) Coding of spatial information - an automatic process. Journal of Experimental Psychology: *Learning Memory and Cognition*, *13*, 595-605.

Newell, A. (1969) Heuristic programming: Ill-structured problems. In J. Aronofsky (Ed.), *Progress in Operations Research III*. New York: Wiley.

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

Payne, S. J. (1985) *Task-action grammars. The mental representation of task languages in human-computer interaction*. Unpublished doctoral dissertation, University of Sheffield.

Payne, S. J. (1991) Display-based action at the user interface. *International Journal of Man-Machine Studies, 35,* 275-289.

Payne, S. J. & Green, T. R. G. (1986) Task-action grammars: A model of the mental representation of task languages. *Human-Computer Interaction, 2,* 93-133.

Payne, S. J. & Green, T. R. G. (1989) The structure of command languages: an experiment on task-action grammar. *International Journal of Man-machine Studies, 30,* 213-234.

Polson, P. G. & Lewis, C. H. (1990) Theory-based design for easily learned interfaces. *Human-Computer Interaction, 5,* 191-220.

Richman, H. B., Staszewski, J. J. & Simon, H. A. (1995). Simulation of expert memory using EPAM IV. *Psychological Review, 2,* 305-330.

Rieman, J. F. (1994) *Learning strategies and exploratoy behavior of interactive computer users*. Unpublished PhD Thesis, University of Colorado. CU-CS-723-94.

Robert, J. M. (1987) Learning a computer system by unassisted exploration. In H.J. Bullinger and B.Shackel (Eds) *Human Compute Interaction: Interact'87*. Elsevier: North Holland.

Rosenbloom, P. S. & Aasman, J. (1990) Knowledge level and inductive uses of chunking (EBL). *Proceedings of the Eighth National Conference on Artificial Intelligence,* AAAI, 821-827.

Rosenbloom, P. S. & Laird, J. E. (1986) Mapping explanation-based generalisation onto Soar. *Proceedings of AAAI-86*. Los Altos, CA: Morgan Kaufman.

Rosenbloom, P. S., Laird, J. E. & Newell, A. (1987) Knowledge level learning in Soar. *Proceedings of AAAI-87, Sixth National Conference on Artificial Intelligence,* 499-504. Morgan Kaufman.

Rosenbloom, P. S., Laird, J. E. & Newell, A. (1988) The chunking of skill and knowledge. In B.A.G. Elsendoorn & H. Bouma (Eds.), *Working Models of Human Perception*. London: Academic Press.

Vera, A. H., Lewis, R. L., Lerch, F. J. (1993) Situated decision-making and recognition-based learning: Applying symbolic theories to interactive tasks. *Proceedings of the 15th Annual Conference of the Cognitive Science Society:* Boulder, Colorado. 84-95.

Young, R. M. (1982) Architecture-directed processing. *Proceedings of the Fourth Annual Conference of the Cognitive Science Society,* 164-166. Ann Arbor, Michigan.