CMU-CS-82-135

# Learning by Chunking
# A Production-System Model of Practice

Paul S. Rosenbloom and Allen Newell
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

September 1982

To be published in D. Klahr, P. Langley, & R. Neches (Eds.),
*Self-Modifying Production System Models of Learning and Development*, In Preparation.

## Abstract

The *power law of practice* states that the time to perform a task decreases as a power-law function of the number of times the task has been performed. One possible explanation for this ubiquitous regularity is the *chunking theory of learning*. It proposes that the acquisition and use of *chunks* is the basis for these improvements. In this article we describe a first attempt at implementing a learning mechanism based on the chunking theory of learning. This work includes: (1) filling out the details of the chunking theory; (2) showing that it can form the basis of a production-system learning mechanism; (3) showing that the implemented mechanism produces power-law practice curves; and (4) investigating the implications of the theory for production system architectures in general. The approach we take is to implement and analyze a production-system model of the chunking theory in the context of a specific task — a 1023-choice reaction-time task. In the process, we develop a control structure for the task; describe the three components of the implemented model — the ***Xaps2*** production-system architecture, the performance model for the task, and the chunking mechanism; and analyze simulations to verify that the implemented model does generate power-law practice curves.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

## Table of Contents

## List of Figures

## List of Tables

# Learning by Chunking
# A Production-System Model of Practice[1]

## 1. Introduction

Performance improves with practice. More precisely, the time to perform a task decreases as a power-law function of the number of times the task has been performed. This basic law — known as the *power law of practice* or the *log-log linear learning law*[2] — has been known since Snoddy (1926). While this law was originally recognized in the domain of motor skills, it has recently become clear that it holds over the full range of human tasks (Newell & Rosenbloom, 1981). This includes both purely perceptual tasks such as target detection (Neisser, Novick, & Lazar, 1963), and purely cognitive tasks such as supplying justifications for geometric proofs (Neves & Anderson, 1981) or playing a game of solitaire (Newell & Rosenbloom, 1981).

The ubiquity of the power law of practice argues for the presence of a single common underlying mechanism. The *chunking theory of learning* (Newell & Rosenbloom, 1981) proposes that *chunking* (Miller, 1956) is this common mechanism — a concept already implicated in many aspects of human behavior (Bower & Winzenz, 1969; Johnson, 1972; DeGroot, 1965; Chase & Simon, 1973). Newell and Rosenbloom (1981) established the plausibility of the theory by showing that a model based on chunking is capable of producing log-log linear practice curves[3]. In its present form, the chunking theory of learning is a macro theory; it postulates the outline of a learning mechanism, and predicts the global improvements in task performance.

This paper reports on recent work on the chunking theory and its interaction with production-system architectures (Newell, 1973)[4]. Our goals are fourfold: (1) fill out the details of the chunking theory; (2) show that it can form the basis of a production-system learning mechanism; (3) show that the full model produces power-law practice curves; and (4) understand the implications of the theory for production-system architectures. The approach we take is to implement and analyze a production-system model of the chunking theory in the context of a specific task — a 1023-choice reaction-time task (Seibel, 1963). The choice of task

---

[1]We would like to thank John Anderson, Pat Langley, Arnold Rosenbloom, and Richard Young for their helpful comments on drafts of this paper.

[2]Power-law curves plot as straight lines on log-log paper.

[3]For a summary of alternative models of the power law of practice, see Newell and Rosenbloom (1981). Additional proposals can be found in Anderson (1982) and Welford (1981).

[4]A brief summary of this work can be found in Rosenbloom & Newell (1982).

should not be critical because the chunking theory claims that the *same* mechanism underlies improvements on *all* tasks. Thus, the model, as implemented for this task, carries with it an implicit claim to generality, although the issue of generality will not be addressed.

In the remainder of this paper we describe and analyze the task and the model. In Section 2 we lay the groundwork by briefly reviewing the highlights of the power law of practice and the chunking theory of learning. In Section 3 the task is described. We concentrate our efforts on investigating the control structure of task performance through the analysis of an informal experiment. In Section 4 we derive some constraints on the form of the model. Sections 5, 6, and 7 describe the three components of the model: (1) the ***Xaps2*** production-system architecture; (2) the initial performance model for the task; and (3) the chunking mechanism. Section 8 gives some results generated by running the complete model on a sample sequence of experimental trials. The model is too costly to run on long sequences of trials, so in addition to the simulation model, we present results from an extensive simulation of the simulation (a meta-simulation). We pay particular attention to establishing that the model does produce power-law practice curves. Finally, Section 9 contains some concluding remarks.

## 2. Previous Work

The groundwork for this research was laid in Newell and Rosenbloom (1981). That paper primarily contains an analysis and evaluation of the empirical power law of practice, analyses of existing models of practice, and a presentation of the chunking theory of learning. Three components of that work are crucial for the remainder of this paper, and are summarized in this section. Included in this summary are some recent minor elaborations on that work.

### 2.1. The structure of task environments

In experiments on practice, subjects are monitored as they progress through a (long) sequence of trials. On each trial the subject is presented with a single *task* to be performed. In some experiments the task is ostensibly identical on all trials; for example, Moran (1980) had subjects repeatedly perform the same set of edits on a single sentence with a computer text editor. In other experiments the task varies across trials; for example, Seibel (1963) had subjects respond to different combinations of lights on different trials. In either case, the *task environment* is defined to be the ensemble of tasks with which the subject must deal.

Typical task environments have a *combinatorial* structure (though other task structures are possible); they can be thought of as being composed from a set of elements which can vary with respect to attributes, locations, relations to other elements, etc. Each distinct task corresponds to one possible assignment of values to the elements. This structure plays an important role in determining the nature of the practice curves produced by the chunking theory.

### 2.2. The power law of practice

Practice curves are generated by plotting task performance against trial number. This cannot be done without assuming some specific *measure* of performance. There are many possibilities for such a measure, including such things as quantity produced per unit time and number of errors per trial. The power law of practice is defined in terms of the time to perform the task on a trial. It states that the time to perform the task ($T$) is a power-law function of the trial number ($N$):

$$T = BN^{-\alpha} \tag{1}$$

If this equation is transformed by taking the logarithm of both sides, it becomes clear why power-law functions plot as straight lines on log-log paper:

$$\log(T) = \log(B) + (-\alpha)\ \log(N) \tag{2}$$

Figure 2-1 shows a practice curve from a 1023-choice reaction-time task (Seibel, 1963), plotted on log-log paper. Each data point represents the mean reaction time over a block of 1023 trials. The curve is linear over

much of its range, but has deviations at its two ends. These deviations can be removed by using a four-parameter *generalized* power-law function. One of the two new parameters ($A$) takes into account that the asymptote of learning is unlikely to be zero. In general, there is a non-zero minimum bound on performance time, determined by basic physiological and/or device limitations — if, for example, the subject must operate a machine. The other added parameter ($E$) is required because power laws are not translation invariant. Practice occurring before the official beginning of the experiment — even if it consists only of transfer of training from everyday experience — will alter the shape of the curve, unless the effect is explicitly allowed for by the inclusion of this parameter. Augmenting the power-law function by these two parameters yields the following generalized function:

$$T = A + B(N+E)^{-\alpha} \tag{3}$$

A generalized power law plots as a straight line on log-log paper once the effects of the asymptote ($A$) are removed from the time ($T$), and the effective number of trials prior to the experiment ($E$) are added to those performed during the experiment ($N$):

$$\log(T-A) = \log(B) + (-\alpha)\log(N+E) \tag{4}$$

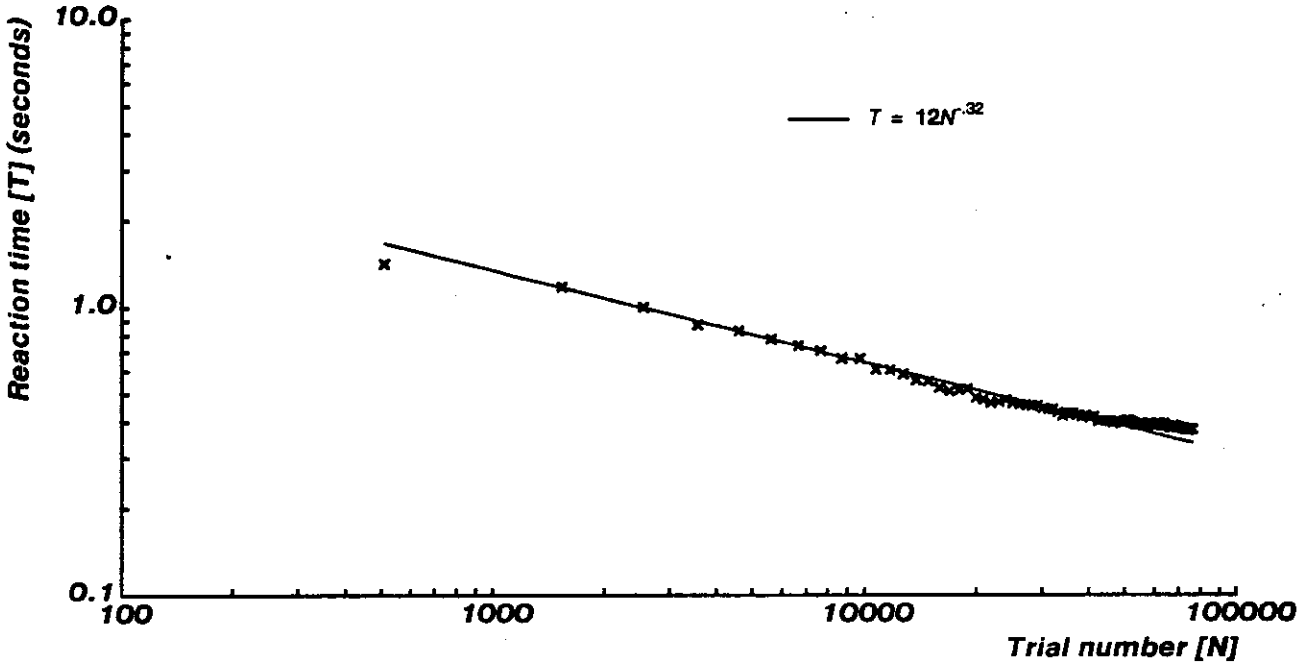



**Figure 2-1:** Learning in a Ten Finger, 1023 Choice Task (Log-Log coordinates).
Plotted from the original data for Subject JK (Seibel, 1963).

Figure 2-2 shows the Seibel data as it is fit by a generalized power-law function. It is now linear over the

whole range of trials. Similar fits are found across all dimensions of human performance; whether the task involves perception, motor behavior, perceptual-motor skills, elementary decisions, memory, complex routines, or problem solving. Though these fits are impressive, it must be stressed that the power law of practice is only an *empirical* law. The true underlying law must resemble a power law, but it may have a different analytical form.

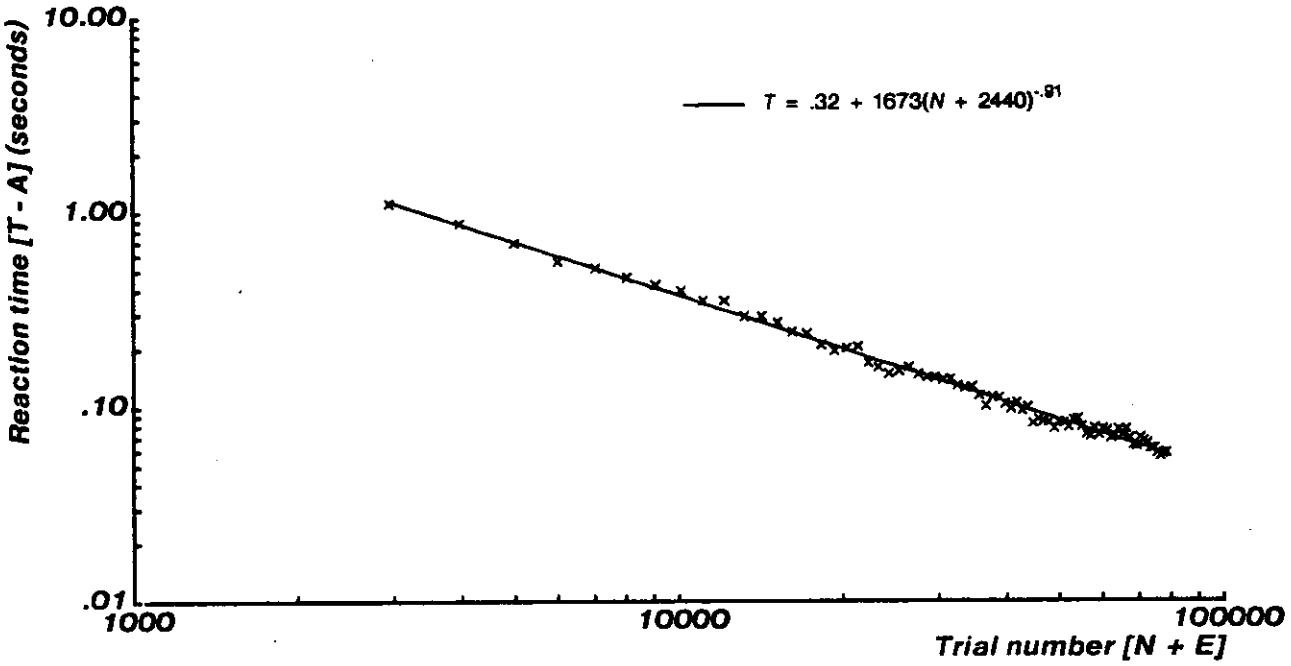



Figure 2-2: Optimal general power law fit to the Seibel data (Log-Log coordinates).

## 2.3. The chunking theory of learning

The chunking theory of learning proposes that practice improves performance via the acquisition of knowledge about patterns in the task environment. Implicit in this theory is a model of task performance based on this pattern knowledge. These patterns are called *chunks* (Miller, 1956). The theory thus starts from the *chunking hypothesis*:

- *The Chunking Hypothesis*: A human acquires and organizes knowledge of the environment by forming and storing expressions, called *chunks*, which are structured collections of the chunks existing at the time of learning.

The existence of chunks implies that memory is hierarchically structured as a lattice (tangled hierarchy, acyclic directed graph, etc.), rooted in a set of *primitives*. A given chunk can be accessed in a top-down fashion, by *decoding* a chunk of which it is a part, or in a bottom-up fashion, by *encoding* from the parts of the chunk. Encoding is a recognition or parsing process.

This hypothesis is converted into a performance model by adding an assumption relating the presence of chunks to task performance.

- *Performance Assumption*: The performance program of the system is coded in terms of high-level chunks, with the time to process a chunk being less than the time to process its constituent chunks.

One possible instantiation of this assumption is that performance consists of serially processing a set of chunks, with the components of each chunk being processed in parallel. Performance is thus improved by the acquisition of higher-level chunks. A second assumption is needed to tie down this acquisition process:

- *Learning Assumption*: Chunks are learned at a constant rate on average from the relevant patterns of stimuli and responses that occur in the specific environments experienced.

This assumption tells us the rate at which chunks are acquired, but it says nothing about the effectiveness of the newly acquired chunks. Do all chunks improve task performance to the same extent, or does their effectiveness vary? The answer to this question can be found by examining the structure of the task environment. If patterns in the task environment vary in their frequency of occurrence, then the effectiveness of the chunks for those patterns will also vary. The more frequently a pattern occurs, the more the chunk gains. The final assumption made by the theory is that the task environment does vary in this fashion:

- *Task Structure Assumption*: The probability of recurrence of an environmental pattern decreases as the pattern size increases.

This assumption is trivially true for combinatorial task environments. As the pattern size grows (in terms of the number of elements specified), the number of possibilities grows exponentially. Any particular large pattern will therefore be experienced less often than any particular small pattern.

There is one notable case where the task structure assumption is violated — when the task environment consists of just one task. This lack of variability means that the subject is presented with the identical experimental situation on every trial. One way to look at this situation is as a combinatorial task environment in which each element can take exactly one value. Now, as the size of the pattern grows, the number of patterns of that size *decreases* rather than increases. As we shall see shortly, this violation does not result in a markedly different prediction for the form of the practice curve.

### 2.3.1. The chunking curve

Starting with the three assumptions, and with a little further specification, an approximate functional form can be derived for the practice curves predicted by the chunking theory.

- *Performance*: Assume that performance time is proportional to the number of chunks that are processed, with $P$ — the number of elements in the task environment — chunks required initially.

- *Learning*: Let $\lambda$ be the constant rate of acquisition of new chunks.

- *Task structure*: Let $C(s)$ be the number of chunks needed to cover all patterns of $s$ elements or less in the task environment.

Given these additional specifications, the chunking theory of learning predicts a learning curve of the form:

$$\frac{dT}{dN} = -\frac{\lambda}{P}\left(\frac{dC}{ds}\right)^{-1} T^3 \tag{5}$$

This equation depends on the structure of the task environment, as described by $C(s)$. It is a power law when $C(s)$ is a power law. For a combinatorial task environment, $\frac{dC}{ds}$ is given by $\frac{P}{s} b^s$, where $b$ is the number of values that each element can take. For $b > 1$ — a standard combinatorial environment — the chunking theory predicts the following learning curve (for arbitrary constants $A$, $B$, $D$, and $E$):

$$T = A + \frac{B}{D + \log(N+E)} \tag{6}$$

Figure 2-3 shows the Seibel data plotted in coordinates in which practice curves predicted by the combinatorial chunking model are straight lines. The linearity of this plot is as good as that for the general power law (Figure 2-2), and the $r^2$ values are comparable: 0.993 for the power law vs. 0.991 for the chunking model. This function and the power law can mimic each other to a remarkable extent. Figure 2-4 contains two curves; one of them is the combinatorial chunking law fit to the Seibel data (Figure 2-3), and the other is the best power law approximation to that curve. The curves are indistinguishable ($r^2$ of 1.000).

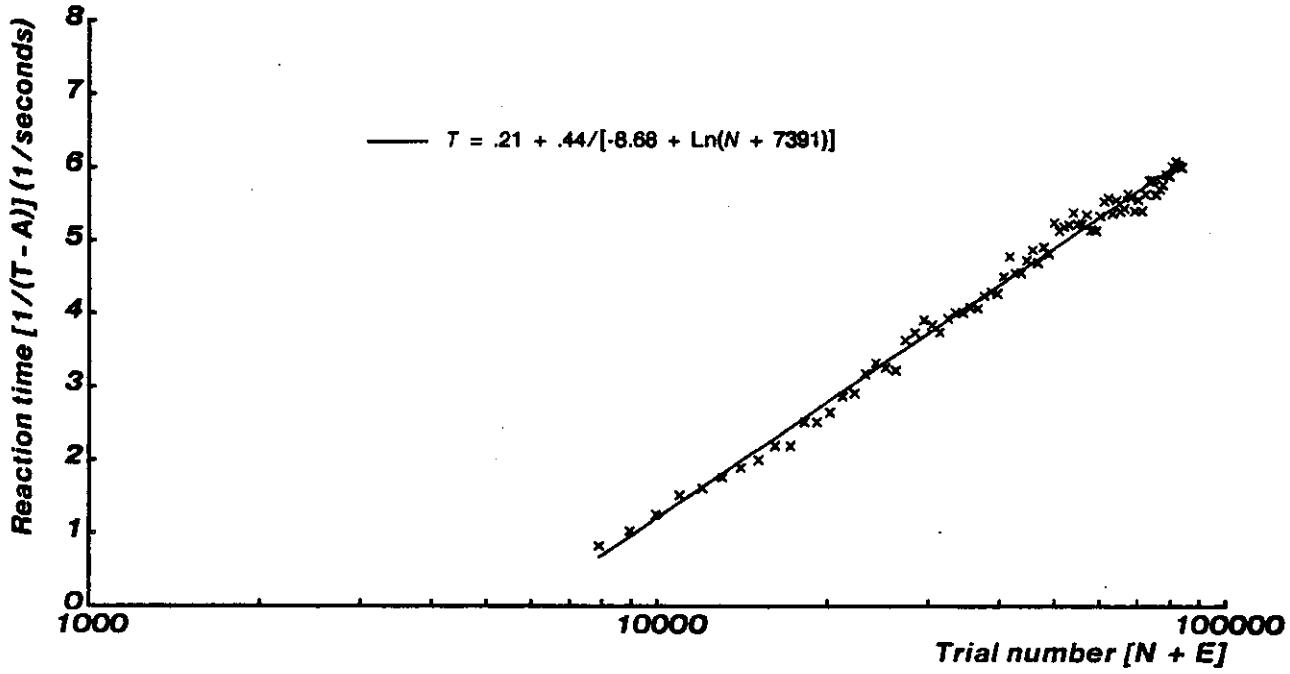



Figure 2-3: Best fit of the Seibel data by the combinatorial chunking function (X-axis log scaled).

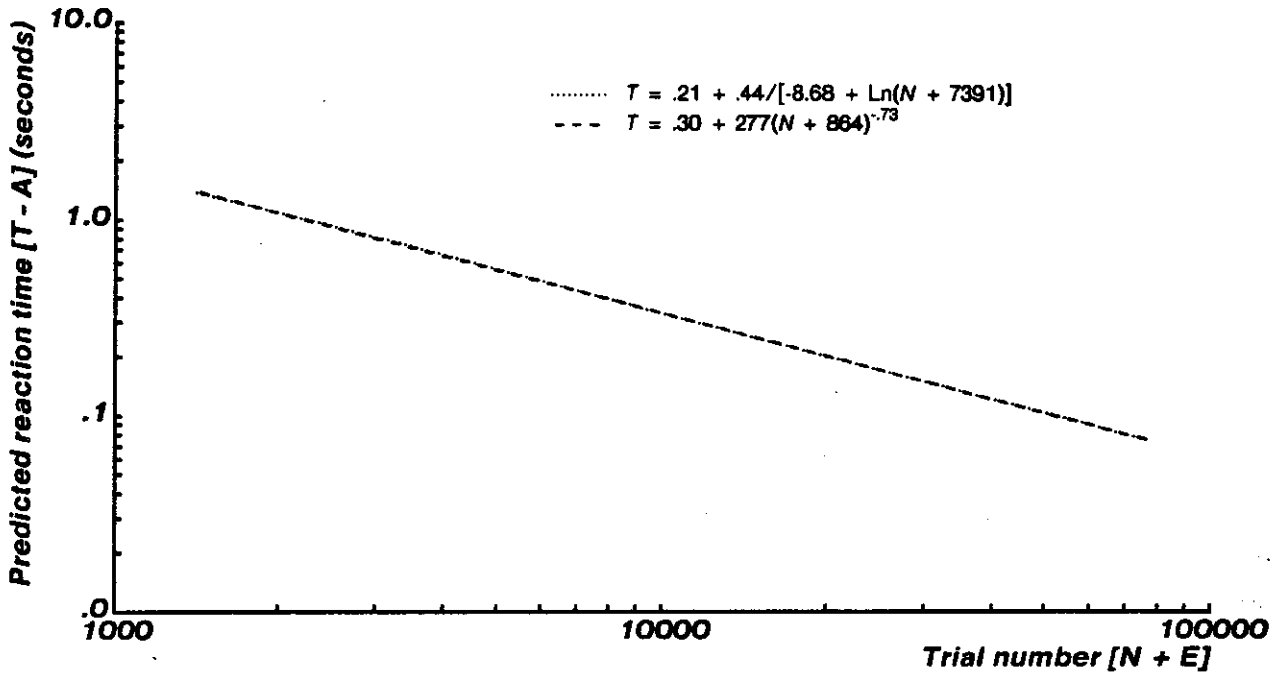



Figure 2-4: A practice curve generated by the chunking theory of learning (Log-Log coordinates). Its optimal power law fit is also shown.

For $b=1$ — only one task in the task environment — $\frac{dC}{ds}$ becomes simply $\frac{P}{S}$. Plugging this expression into Equation 5 yields a hyperbolic curve of the form (for arbitrary constants $A$, $B$, and $E$):

$$T = A + \frac{B}{N+E} \tag{7}$$

Since a hyperbolic function is just a special case of the power law (with $\alpha=1$), this function is trivially well fit by a power law.

## 3. The Task

It is difficult, if not impossible, to produce and evaluate a theory of learning without doing so in the context of some concrete task to be learned. The first characteristic required of such a task is that it be understood how the learning mechanism can be applied to it. For our purposes, this corresponds to understanding what aspects of the task can be chunked. The second characteristic required of the chosen task is that the control structure of initial task performance be understood. Discovering such a performance model is well within the domain of a learning system, but practice is the subclass of learning that deals only with improving performance on a task that can already be successfully performed. Thus, our models will always start with some method, no matter how inefficient, for performing the chosen task.

The task that we shall employ is a 1023-choice reaction-time task (Seibel, 1963). This is a perceptual-motor task in which the task environment consists of a stimulus array of ten lights, and a response array of ten buttons in a highly compatible one-one correspondence with the lights. On each trial, some of the lights are *On*, and some are *Off*. The subject's task is to respond by pressing the buttons corresponding to the lights that are *On*. Ten lights, with two possible states for each light, yields $2^{10}$ or 1024 possibilities. The configuration with no lights on is not used, leaving 1023 choices.

This task easily meets the first criteria; in fact, the macro-structure of chunking in this task has already been analyzed (Newell & Rosenbloom, 1981). The task has an easily recognizable combinatorial structure. The lights and buttons are the elements of the task environment. Each element has one attribute with two possible values — *On* and *Off* for the lights, *Press* and *NoPress* for the buttons. These lights and buttons will form the primitives for the chunking process.

One auxiliary benefit of selecting this task is that it fits in with a large body of experimental literature, allowing exploration of the theory's implications to a large range of nearby phenomena. The literature on perceptual-motor reaction time is a rich source of data on human performance. For this particular task, practice curves already exist out to more than seventy thousand trials (Seibel, 1963). Figures 2-1, 2-2, and 2-3 are plotted from the data for one subject in that experiment.

Unfortunately, the existing practice curves present highly aggregated data (by 1023-trial blocks), leaving us with little evidence from which to deduce the within-trial structure of performance (the second required characteristic). In order to gain some insight into this structure, an informal investigation into the performance of human subjects in this task was performed. A version of the experiment was programmed on an ***Alto*** personal computer (Thacker, McCreight, Lampson, Sproull, & Boggs, 1982). The bit-map screen (8.5" x 11" at 72 points to the inch) was used for the presentation of the stimulus lights. The screen was dark, allowing *On*-lights to be represented as solid white squares (.85 cm. on a side), while *Off*-lights were dark with

a white rim. The lights all fit into a rectangle 12 cm. wide by 2.8 cm. high (Figure 3-1). At an approximate distance of 43 cm., the display covers 16°of visual arc.

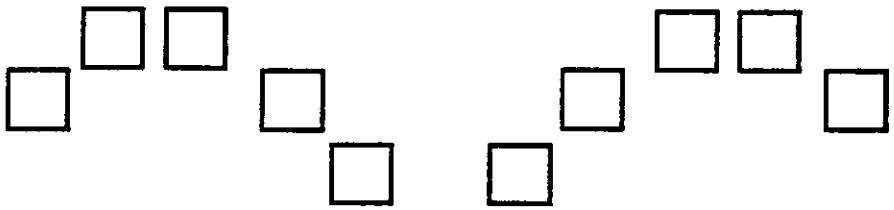


**Figure 3-1:** The stimulus array of ten lights.

The *Alto* keyboard allows the sensing of multiple simultaneous key-presses, so ten keys were selected and used as buttons. The keys (a, w, e, f, v, n, j, i, o, ;) were chosen so as to make the mapping between stimulus and response as compatible as possible. The keys all fit into a rectangle 18.3 cm. wide by 4.9 cm. high (Figure 3-2).

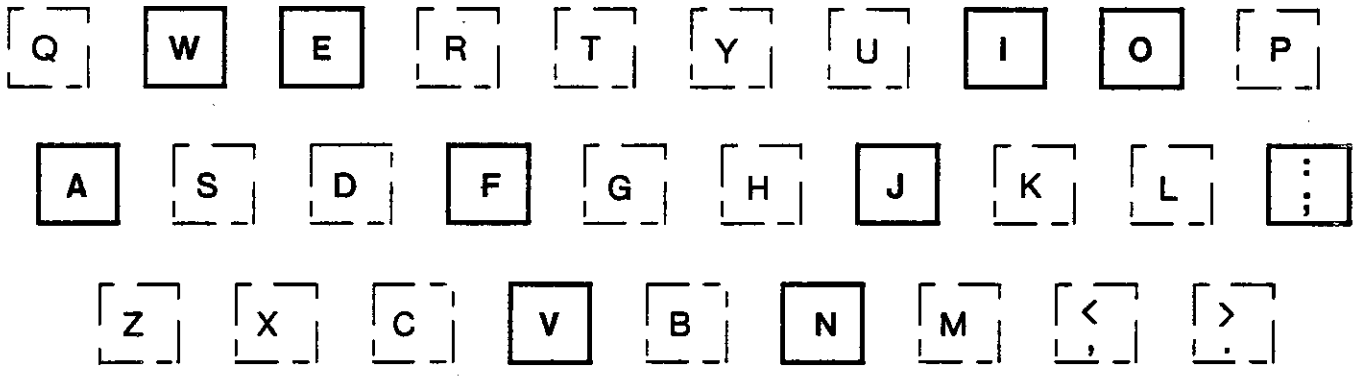



**Figure 3-2:** The response array (a portion of *Alto* the keyboard) with the ten keys highlighted.

Each trial began with the presentation of a fixation point between the center two lights. After 333 msecs, the fixation point was replaced by the ten lights. A clock was started at this point (accurate to within one jiffy, or about 17 msecs), and the time was recorded for each button that was pressed. The trial continued until the subject had pressed the button corresponding to each *On*-light, even if wrong buttons had been pressed along

the way. It was not necessary to hold down the buttons; once they had been pressed they could be released. Once all of the required buttons had been pressed, the screen was cleared. With this setup, the fastest way to complete a trial was to press all ten buttons as soon as the lights appeared. To rule out this possibility, the subject received, via the screen, feedback about the correctness of the response, and his cumulative percent correct over the previous 20 trials. Subjects were instructed to respond as quickly as possible while maintaining a low error percentage. Following error feedback, the screen went blank until the start of the next trial — approximately 2 seconds.

Four male graduate students in their twenties were run informally in a somewhat noisy environment (a terminal room). A trial sequence was generated from a random permutation of the 1023 possibilities. This fixed sequence was divided into 102-trial blocks (except for the last block) and used for all subjects. Subject 1 completed 2 blocks of trials (or 204 trials); subject 2 completed 1 block; subject 3 completed 4 blocks; and subject 4 completed 1 block. Any trial in which a bad key — one corresponding to an *Off*-light — was pressed was considered to be an error of *commission*, and any trial that took longer than 5 seconds was considered to be an error of *omission*. These error trials — 12% of the total trials — were removed from the main analysis.

All four subjects improved during the course of the experiment, but the data is somewhat ragged due to the small number of trials. All of the data for any of these subjects fits within one data point of the graphs of Seibel's subjects. Figure 3-3 shows the longest of these curves (Subject 3: 408 trials).

The learning curve verifies the power-law nature of learning in this task, but we are primarily interested in the micro-structure of within-trial performance. Figure 3-4 shows the within-trial structure of some typical correct trials. The ten short, wide bars represent the lights, separated into the two hands. A solid outline signifies an *On*-light, while a dotted outline marks an *Off*-light. The narrow bars record the amount of time between the start of the trial, and when the button corresponding to that light was pressed.

Most of the trials show a similar pattern: groups of buttons are pressed nearly simultaneously, with gaps between these compound responses. These compound responses are termed *response groups*. The numbers above the bars denote the response group to which each keypress belongs. The response groups primarily fall into four classes: (1) A single *On*-light; (2) A group of adjacent *On*-lights; (3) All of the *On*-lights in a single hand; and (4) All of the *On*-lights.

For subjects 1 and 2, response groups were computed by forcing any two responses within 100 msec of each other to be in the same response group. This algorithm works because the distribution of inter-button response times is bimodal, with a break at about 100 msec. The distributions for subjects 3 and 4 are

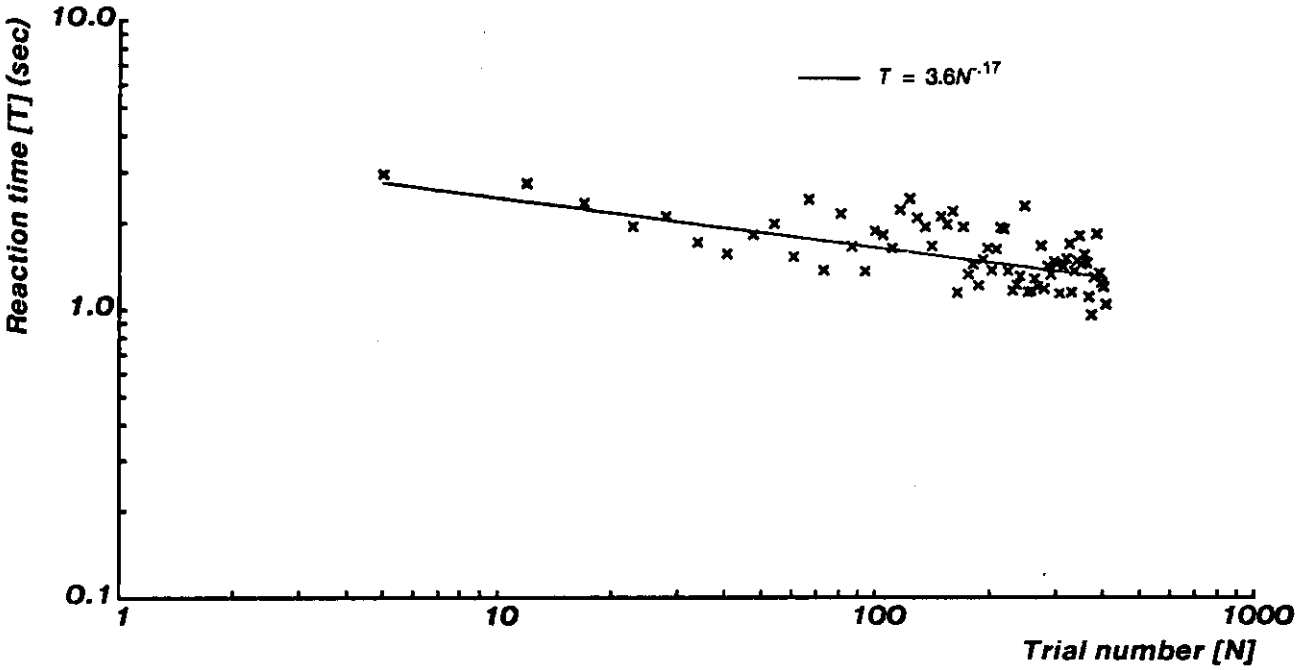


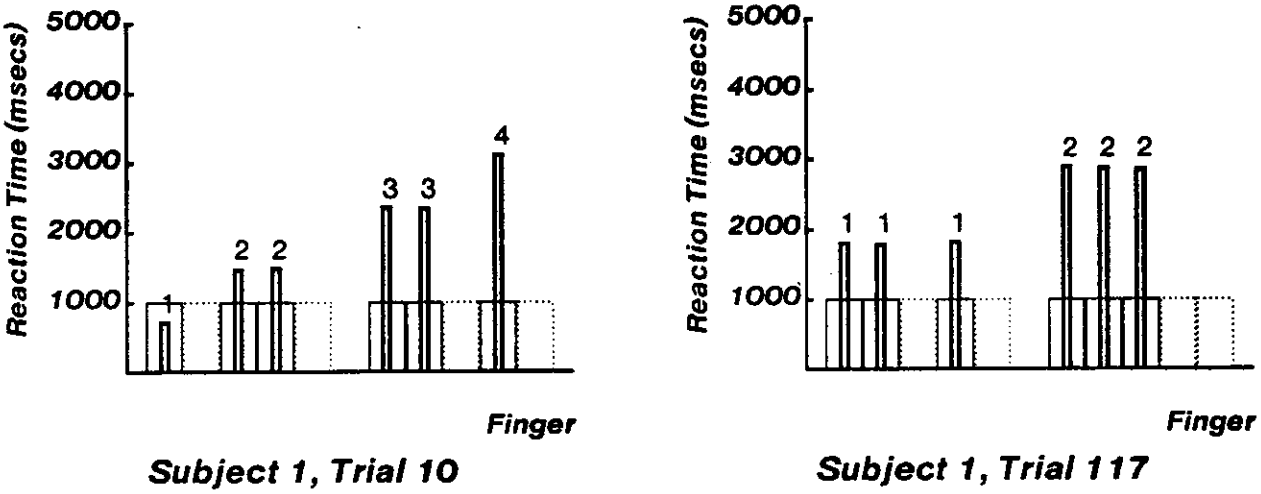

**Figure 3-3:** Subject 3's learning curve (log-log coordinates).
The data is aggregated by groups of five trials.

unimodal, so the concept of a response group is more difficult to define, and thus less useful for them. This difficulty is partly due to the high degree to which both of these subjects responded to all lights at once, resulting in a paucity of inter-group times.

For all four subjects, the response groups *increased* in size with practice. This is exactly what we would expect if chunking were operating. Unfortunately, it is not possible to conclude that either the presence or growth of response groups are evidence for chunking. The size of response groups can be artificially increased by sequentially preparing a number of fingers (poising them above the appropriate keys), and then pressing all of the prepared fingers simultaneously (and some subjects behaved this way). It is not possible to fully disentangle the effects of these consciously selected strategies, and changes in strategy, from the effects of chunking.

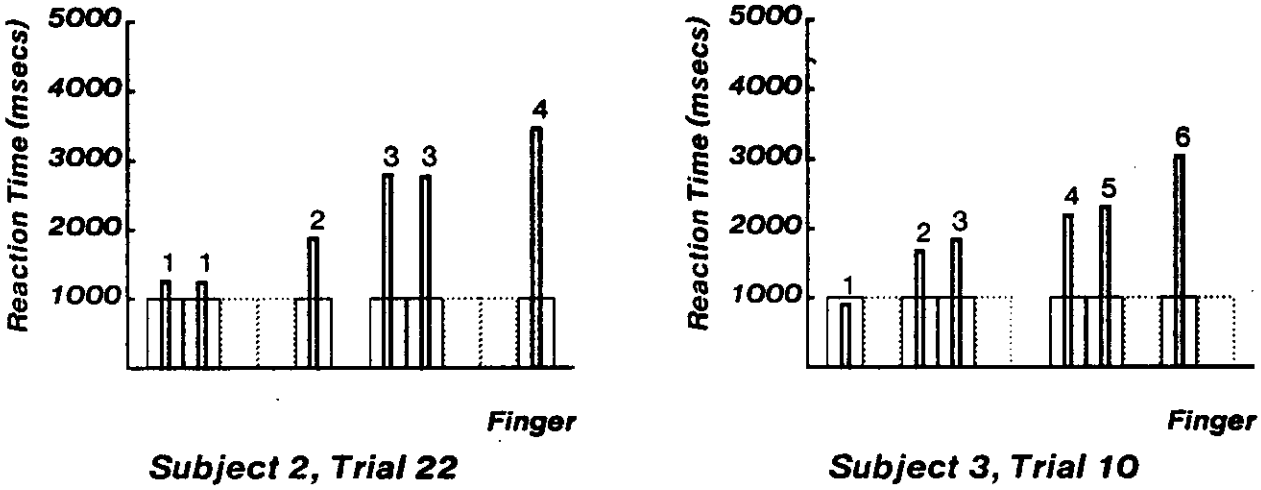
Though the experiment provides no clear evidence for chunking, it does provide some guidance for the construction of a model of the task. The following facts are true of the data:

- There is no single *right* control structure for the task. The subjects employed qualitatively different strategies.
- Each subject used predominantly one strategy, but not exclusively.













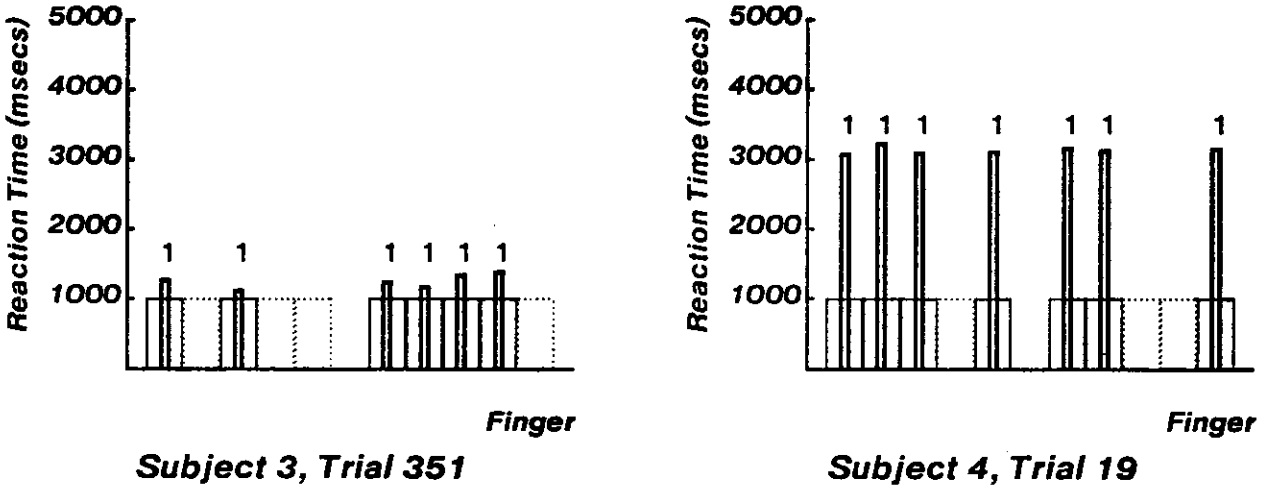






Figure 3-4: Six typical trials.

- The lights were processed in a predominantly left-to-right fashion.

- The number of *On*-lights is a significant component of the reaction time. Ignoring those trials on which all five lights on a hand were on[5], Figure 3-5 plots reaction time versus the number of *On*-lights. The figure reveals the strong linear trend of this data; all but the final point (one *Off*-light per hand) fall close to the regression line. Subjects may be employing a different strategy for those trials, such as preparing to press all of the keys, and then unpreparing the key corresponding to the *Off*-light.

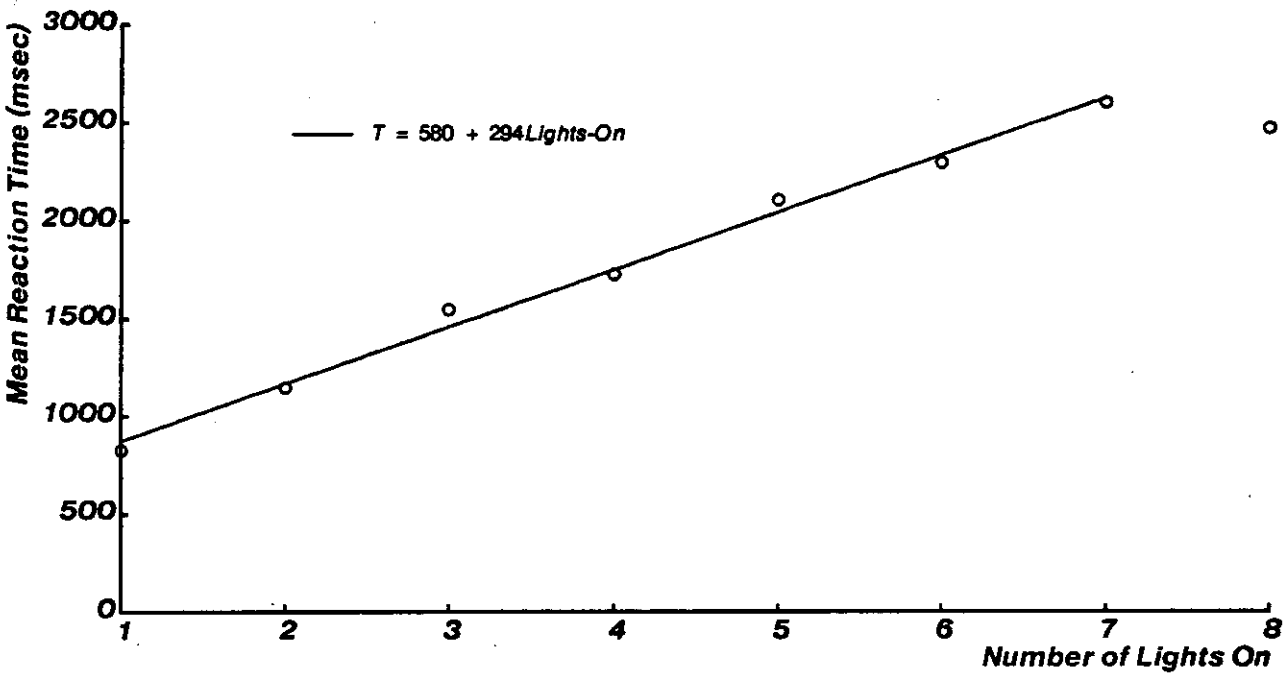



Figure 3-5: Reaction times versus number of *On*-lights. The data is averaged over all trials of the four subjects, except those with 5 *On*-lights on one hand (see text).

Taking these facts into consideration, a reasonable control structure for this task can be based on a simple iterative algorithm.

[5]When all of the lights on one hand are on, it can be treated as a single response of the whole hand, rather than as five individual responses. In fact the reaction times for these trials are much faster than would be expected if the five lights were being processed separately.

**Focus** a point to the left of the leftmost light.
**While** there is an *On*-light to the right of the focal point **Do**
  **Locate** the *On*-light.
  **Map** the light location into the location of the button under it.
  **Press** the button.
  **Focus** the right edge of the light.

The model of practice for this task starts with this simple model of initial task performance and a chunking mechanism. As experimental trials are processed, chunks are built out of the stimulus and response patterns that are experienced. These chunks reduce the response time on later trials by allowing the subject to process a group of lights on each iteration, rather than just a single light.

We will return to describe the implementation of the initial performance model in Section 6 after the architecture of the production system in which it is implemented has been laid-out.

## 4. Constraints on the Model

A useful initial step in the development of the model is the derivation of *constraints* — codifications of fundamental limitations — on the form of the model. Any particular implementation of the model is going to be wrong, at least along some dimensions. By delineating boundaries on the set of *all* legitimate models, constraints provide a means by which knowledge can be transferred to later more "correct" models of the task, and to models in other task domains. Constraints can be formulated in terms of either the *behavior* of the model, or its *structure*. Behavioral constraints are the weakest, and are provided in profusion by the results of psychological experiments. The power law of practice is an example, one we are using heavily. In essence, behavioral constraints circumscribe the input-output pairs of the model, leaving the model itself as a black box with any number of possible internal structures.

Structural constraints are much stronger; they limit the space of structures that may be inside the box. At their most powerful, structural constraints are capable of ruling out large classes of architectures. Though the rewards can be substantial, formulating and proving the correctness of such constraints is difficult. For each proposed constraint, proofs of four properties are required.

- *Universality*: The limitation must be a universal property of the performance of the system.
- *Architecturality*: The universality of the limitation must be a reflection of an architectural restriction, not just a regularity of the task environment.
- *Minimality*: The constraint must describe the minimum architectural restriction capable of producing the limitation.
- *Uniqueness*: The constraint must be the only valid implementation of the architectural restriction.

Ignoring these properties can lead to overly restrictive, or simply incorrect, constraints. However, these properties are difficult to establish, and we have not succeeded in formally proving them for the constraints we propose. Minimality is a particularly hard concept to work with because it requires an ordering on the space of constraints, an ordering that is not always clearly defined. For some of the constraints we will not even attempt to evaluate this property. For the remaining three properties, we have resorted to informal arguments in their favor. Though this is a serious shortcoming, we still feel that these constraints are reasonable and interesting.

Once we have outlined the constraints, we can proceed to a discussion of one model that sits within the design space bounded by the constraints. Wherever possible, design decisions are justified on the basis of these constraints. But, as long as the constraints leave room for variation, these justifications can extend only as far as showing that a mechanism is *sufficient* to meet a constraint. *Necessity* can only come from a more constrained design space.

Many sources of knowledge can be employed to generate constraints on the model. For example, a number of computational and psychological considerations have lead to the choice of a production-system architecture as the basis for the performance system (Newell & Simon, 1972; Newell, 1973). Other psychological constraints have lead to further refinement in the class of production-system architectures that is explored (see, for example, Anderson, 1976, Newell, 1980, and Thibadeau, Just, & Carpenter, In Press). In this section we focus on those constraints derivable from the chunking theory.

The five strongest constraints have significant impact on the nature of the production-system architecture used as the basis of the model. These five constraints are described here, leaving other weaker constraints to be discussed as they arise in the model. The theoretical constraints take the form:

> *If* the chunking theory is correct,
> *then* any architecture that implements it must meet some condition.

An example of this kind of argument in another domain is:

> *If* the theory of evolution is correct,
> *then* any model of reproduction must allow for the inheritability of mutations.

The form of argument is similar in spirit to that made by VanLehn (1981) concerning the implications of *repair theory*.

It is possible to make general arguments that all constraints derivable from the chunking theory must be universal and are almost certainly architectural. Universality follows directly from the assumed universality of the chunking theory of learning. If the model is universal, and some limitation is necessary wherever the model applies, then the limitation must also be universal. Architecturality is more difficult to establish conclusively, so we resort to a simpler plausibility argument. We start by noticing that if the limitations are universal, then either the restriction is architectural, or all task environments must reflect the limitation. Since the former choice is the simpler assumption of the two, we feel safe in assuming it until proven incorrect.

The first major constraint is a direct consequence of the performance assumption — chunks improve performance because the time to process a chunk is less than the time required to execute the constituent chunks. This assumption rules out one of the more obvious schemes for processing chunks: decode the chunk into a set of sub-chunks, and then serially process the sub-chunks. Such a scheme is reasonable if chunks are thought of as nothing more than a space-efficient code for information, but it results in the time to process a chunk being the sum of the times to process the constituent chunks, plus some overhead for decoding. This consideration leads us to the following constraint:

> *The parallel constraint*: The model must contain some form of parallel processing — it cannot have a control structure that is totally serial.

This constraint is definitely not unique. One alternative is that subchunks are processed by a second, faster,

serial process. It is difficult to distinguish a parallel process from an arbitrarily fast serial process, so the result must be left open to this extent.

The simplest control structure meeting the parallel constraint is one that processes chunks (and everything else) in parallel. However, such an architecture would also violate the performance assumption. If any set of chunks can be processed in parallel, packaging them into a higher-level chunk will not improve performance. At the extreme, all of the light-button pairs in Seibel's task could be processed simultaneously, achieving an optimal algorithm. A second constraint is required if the chunking process is to produce performance improvements:

> *The bottleneck constraint*: The parallel capability of the control structure must be restricted so that a bottleneck exists. This can be a serial bottleneck, or more generally, some form of capacity limitation.

Chunking thus improves performance by alleviating the flow through a bottleneck. As with the parallel constraint, the bottleneck constraint is not the unique response to the required limitation. One alternative is that the problem stems from a limitation on the learning system (another part of the the architecture), rather than on the performance system. That is, the performance system is capable of executing any set of productions that can be acquired by the subject, but the learning system is only capable of acquiring knowledge in terms of chunks. Without a principled reason for choosing this alternative, we believe that the bottleneck constraint should be preferred because of the existing evidence for capacity limitations in humans (e.g., Miller, 1956).

From the first two constraints we get the picture of a parallel system with at least one bottleneck. We can constrain this architecture even further by bounding the location of the bottleneck.

> *The encoding constraint*: The bottleneck occurs after the process of chunk encoding has completed.

> *The decoding constraint*: The bottleneck occurs before the final process of chunk decoding has begun.

If either of these constraints did not hold, then chunking would not fulfill its function of reducing the flow through the bottleneck. These constraints appear to be minimal and unique.

With the addition of these two constraints, the architecture has been limited to one that is parallel at its extremes, with a limited capacity in between. However, it is a mistake to think that this implies a serial cognitive system with parallel sensory and motor components. Practice improves performance on the full range of human tasks, specifically including purely cognitive tasks. The processing of cognitive chunks therefore requires parallel computation just as do sensory and motor chunks. The last theoretical constraint — a minimal and unique one — is thus:

*The cognitive-parallelism constraint*: The locus of parallelism in the system cannot be constrained to the sensory and motor portions of the system

## 5. The Xaps2 Production-System Architecture

The *Xaps* architecture (Rosenbloom, 1979) was designed to investigate merging the concepts from (1) symbolic production systems (specifically, the *Ops2* language (Forgy, 1977)), (2) activation models of short-term memory (e.g. Anderson, 1976; Shiffrin & Schneider, 1977), and (3) linear associative memories (see Anderson & Hinton (1981) for a good review of this work). From (1), *Xaps* got its production-system control structure and list-based representation; from (2) came the activation values attached to working memory elements (modeling recency, priming effects, importance, focus of attention, etc.); and from (3) came the parallel firing of productions. The resulting design bore many resemblances to *Hpsa77* (Newell, 1980) and *Act* (Anderson, 1976).

The main function of *Xaps* was as a testbed for assumptions about the decay of working-memory elements (such as item decay, time decay, and fixed total activation), the combination of activation values in the match, and the use of productions as conduits of activation. Its development was actively driven by work in the LNR group at UCSD on the modeling of motor skills (such as finger tapping and typing (Rumelhart & Norman, 1982)), recognition networks (McClelland & Rumelhart, 1981; Rumelhart & McClelland, 1982), and slips of the mind (Norman, 1981).

The *Xaps2* architecture[6] is a direct descendant of *Xaps*. Like *Xaps*, it is a parallel, symbolic, activation-based production system. *Xaps2* differs in having an object oriented representation (rather than a list one), and in its control and activation assumptions[7]. The changes have been driven by the needs and constraints of the current research. While the justifications for the design decisions in *Xaps2* will often be given in terms of the task at hand, we must stress that there are very few features in the language particular to this task. *Xaps2* is a general purpose production-system architecture.

The following sections describe the primary non-learning components of the *Xaps2* architecture: working memory, production memory, and the cycle of execution. The chunking mechanism is properly a part of this architecture (as it is implemented by *Lisp* code, not productions), but it is described later (Section 7) because its implementation is dependent on the form of the initial performance model (Section 6)[8].

---

[6] *Xaps2* is implemented in *MacLisp* running on a DecSystem 2060.

[7] The assumptions in *Xaps2* bear a strong family resemblance to those in the *Caps* architecture (Thibadeau, Just, & Carpenter, 1982).

[8] Our ultimate goal is to develop a task independent implementation of chunking, but until that is possible, we must live with this unsatisfactory but necessary dependence.

## 5.1. Working memory

Most modern production systems concur in defining working memory to be the active set of data to which productions attend. They diverge on the choice of a representation for the individual data elements that compose this working-memory set. In ***Xaps2***, the basic working memory data structure is the *object*. In the model of Seibel's task, individual objects are used to represent instances of goals, patterns of perceived stimuli, and patterns of responses to be executed.

Objects are tokens, in the sense of the classical type-token distinction; they are defined by the combination of a *type* and an *identifier*. This style of representation allows more than one instance of a type to be active and kept distinct simultaneously, a facility required both to represent the goal structure of Seibel's task, and for chunking stimulus and response patterns. As an example, the following object represents an instance of the goal to perform one trial of the task[9]:

(**OneTrial** `Object134`)

The type encodes the name of the goal, while the identifier is a unique symbol encoding the goal instance. Identifiers tend to be arbitrary symbols because they are created on the fly during execution of the model.

In addition to its two defining symbols, each object optionally has a set of *attributes*. These attributes provide a description of the object containing them. For example, the description of a goal includes a statement of its current STATUS — should we *Start* fulfilling the goal, or is it already *Done*. When the relevant attributes, and their values, are added to the above goal instance, the object looks like:

(**OneTrial** `Object134` [INITIAL-LOCATION *0.99*][STATUS *Done*])

The object has two attributes: INITIAL-LOCATION, which tells us to start looking for stimulus patterns at location *0.99*, and STATUS, which tells us that this particular goal instance is *Done*. The type and identifier are *definitional* characteristics of an object in that changing either of them creates a new object. In contrast, the values of attributes can be changed as needed — such as when the STATUS of a goal changes — without affecting the identity of the object.

An important (and unusual) aspect of the ***Xaps2*** working memory representation is that an attribute may in fact have more than one value represented for it at the same time. Each reflects a *proposed* value for the attribute. Parallel execution of productions often results in two productions simultaneously asserting different values for the same attribute of an object. This conflict is resolved by allowing both (or all) of the proposed values to be placed in working memory. The final selection of which value to employ is made on the basis of accumulated evidence about the choices.

---

[9] In this and following examples, the notation has been modified for clarity of presentation. Some of the names have been expanded or modified. In addition, types have been made **bold**, identifiers are in the normal roman font, attributes are in SMALL CAPITALS, and values are *italicized*.

Augmenting the previous example with the appropriate additional values yields the full symbolic representation of the object:

```
(OneTrial  Object134
           [INITIAL-LOCATION 0.99 None 0.0 0.36]
           [STATUS Done Initialize Object138 Object144 OnePattern Start])
```

Up to this point, working memory has been described as a totally symbolic structure consisting of a set of objects. What has been ignored is the active *competition* for attention that occurs among these symbols. Attention is modelled by associating an *activation* value with each type, identifier, and value. This is simply a real number in the range [-1, 1]. A positive value indicates active presence in working memory, a negative value indicates *inhibition* of the item, and a zero value is equivalent to absence from working memory. Activation values in the range [-.0001, .0001] are sub-threshold, and assumed to be zero.

While types, identifiers, and values all have activation levels, they do not directly compete with each other. Instead, each competes within a limited scope: the types compete with each other; the identifiers compete with each other within each type; and the values compete with each other within each attribute of each object. The rule is that competition occurs between two symbols if and only if a process of selection is possible between them.

Competition clearly occurs among the alternative values within an attribute. If we want to know the STATUS of a goal, we need to select the *best* value from the set of proposed ones. Competition also occurs between instances of the same type; for example, the model of Seibel's task must repeatedly choose one light to attend to out of all of the stimuli that make up the model's visual input. Competition between types generally involves a choice between different goals. The utility of this is currently unclear.

One place where competition is not appropriate is among attributes within an object. Attributes contain information about orthogonal dimensions of an object, so competition is not only not required, but it is actually a hindrance. An early version of ***Xaps2*** did include competition between attributes, but it was abandoned because of the ensuing difficulty of simultaneously maintaining the independent pieces of knowledge about an object.

Within the scope of each competition, the activation values are kept normalized by forcing the absolute values of the activations to sum to 1. The absolute values of the activations are used because of the presence of negative activations (inhibition). This concept of normalizing working memory is derived from work on associative memories (Anderson, 1977). It is employed in ***Xaps2*** to ensure that the levels of activation in the system remain bounded. The normalization process has a number of implications for the maintenance of activation in ***Xaps2***.

- It produces a set of independent capacity limitations — one for each scope of competition. These are not fixed limits on the number of items; instead they are limits on the amount of activation available to keep items in working memory. They impose variable limits on the number of active items.

- It implements a form of mutual inhibition between the competing items. Which items are inhibited, and by how much, is determined dynamically as a function of the competing items.

- It defines a saturation level (ceiling) of 1 for the activation of any item.

With activation values included (in angle brackets), the example working memory object becomes:

```
(OneTrial<0.1606>  Object134<1.0>
                   [INITIAL-LOCATION 0.99<0.75> None<0.1406>
                                     0.0<0.0156> 0.36<0.0938>]
                   [STATUS Done<0.5> Initialize<0.0078>
                           Object138<0.0313> Object144<0.125>
                           OnePattern<0.3281> Start<0.0078>])
```

For convenience we will refer to that value of an attribute that has the highest activation level as the *dominant* value of that attribute. In this object, the values *0.99* and *Done* are the dominant values for the INITIAL-LOCATION and STATUS attributes respectively.

## 5.2. Production memory

Productions in *Xaps2* contain the standard components: a set of conditions, which are partial specifications of objects in working memory; and a set of actions that result in changes to working memory. In addition, each production has an *execution-type*; either **Always**, **Once**, or **Decay**. These have implications for conflict resolution and action execution, and are therefore discussed in those sections. Here is the definition of a typical production along with a rough english-like paraphrase of what it does[10]:

```
(DefProd AfterOneStimulusPattern Always
    ((OnePattern =GoalIdentifier [STATUS =SubGoalIdentifier])
     (OneStimulusPattern =SubGoalIdentifier [STATUS Done]))
    →
    ((OnePattern =GoalIdentifier [STATUS MapOnePattern])))
```

*If* there is an instance of the **OnePattern goal**
  that is suspended while a sub-goal is being pursued
*and* that subgoal is an instance of the **OneStimulusPattern goal**
  that has completed (the STATUS is *Done*),
*then* establish a new sub-goal for the **OnePattern goal**
  that is an instance of the **MapOnePattern goal.**

[10] The syntax of this production has been modified slightly for presentation purposes. Symbols beginning with "=" are *variables* (see Section 5.3.1). This is the only example of the internal form of a production to appear; in the remainder of this paper we use the paraphrase form.

This production, called AfterOneStimulusPattern, is an Always production with two conditions and one action. Notice that conditions and actions do *not* specify activation values (or weights). The activation values in working memory are used in the production system cycle (see below), but productions themselves are totally symbolic. The details of the conditions and actions are discussed in the following section on the production system cycle.

## 5.3. The production system cycle

The recognize/act production system cycle is mapped onto a four stage cycle in ***Xaps2***. Recognition consists of (1) the match, and (2) conflict resolution. Actions are performed by (3) executing the production instantiations, and (4) updating working memory. The match starts with the productions and working memory, and returns a set of legal production *instantiations* (productions with all free parameters bound). Conflict resolution takes this set of instantiations and determines which ones should be executed on the current cycle. These production instantiations are then executed, with the results being accumulated into a *pre-update* structure. This structure is then merged with working memory in the final stage, to yield the working memory for the next cycle. The following sections treat these four topics in more detail.

### 5.3.1. The match

The match is responsible for computing a candidate set of executable productions and parameters for those productions. This is accomplished by matching production conditions to objects in working memory. The conditions of all of the productions are simultaneously matched against working memory; providing one source of the parallelism required by the parallel constraint. Each condition partially specifies an object as a pattern built out of constant symbols, variables, and real-number comparison functions. The components of an object (type, identifier, attributes, and values) differ in the kinds of patterns that can be created for them.

**Type** — Types are specified by constant symbols only — signifying that an exact match is required. In general, variables are required only when the correct value is not known a priori; that is, when a run-time selection must be made among competing alternatives. Since types compete for attention (Section 5.3.3), but dynamic selection among them is not required, variables are not needed.

Identifier — Identifiers are specified primarily by variables, but occasionally by constant symbols. A variable can match any identifier, as long as multiple instances of the same variable match the same thing, within a single production.

Identifiers are usually created dynamically by the system — as in the creation of a new goal instance — and are thus not available to existing productions. Run-time selection among identifiers is the rule. Specification of the type, and a description in terms of the object's attributes, yields the identifier of the object as the binding of a variable.

ATTRIBUTES — Every attribute that is specified in the condition must successfully match against the object

in working memory, but the reverse is not true. There may be attributes in the object that are not specified in the condition. This allows conditions to partially specify objects by describing only known attributes.

When an attribute is specified, it is always a constant symbol. Variables are not allowed because, as with types, attribute names are known when productions are created. Searching for a value when the attribute is unknown is thus disallowed in the match.

*Values*

Values are specified by constant symbols, variables, and real-number comparison functions (discussed below). The specification is matched only against the dominant value of the attribute. Many values can be proposed for each attribute, but the match can see an object in only one way at a time (according to its dominant values). This mechanism enables productions to perform a selection of the "best" value for each attribute.

If the pattern for a value is *negated*, the match succeeds only if *no* successful match is found for the pattern. Again, only the dominant values are involved in the determination.

The process of matching the condition side of a production to working memory produces *instantiations* of the production. An instantiation consists of all of the information required to execute the actions of a production: the production name, bindings for all of the variables in the production, and an activation value. More than one instantiation can be generated for a production through different combinations of bindings to the variables. For an instantiation to be generated, each condition must be successfully matched to some positively activated object in working memory. As with values of attributes, whole conditions can be negated, signifying that the match succeeds only if no successful match is found for the condition.

The function of the first two components of an instantiation — the production name and variable bindings — should be obvious; they determine which actions to execute, and what values will be passed as parameters to those actions. The functions of the activation value are less obvious: it establishes an ordering on the instantiations of a single production (used in conflict resolution, Section 5.3.2), and it provides activation values for the results of the production's actions (Section 5.3.3). The value is computed by first calculating an activation value for the match of each condition, and then combining them into a single value. The combination function has been tuned so that it gives preference to large condition sides with well matched conditions (highly activated condition matches). Favoring large condition sides is analogous to the special-case conflict resolution strategy in the ***Ops*** languages. The computation is the sum of the condition activations divided by the square root of the number of conditions. This computation is half way between summing the activations, emphasizing the number of conditions, and taking the average, emphasizing the activation of the condition matches. Negated conditions have no effect on this computation.

The activation of a condition match is computed from (1) the working-memory activation of the object matched by the condition, and (2) the *goodness-of-fit* of the object's description (in terms of its attributes) to

the partial description specified by the condition. The first component, the activation of the working-memory object, is defined to be the product of the activations of its type, and its identifier. The choice of multiplicative combination (as opposed to additive, or some other scheme) is somewhat ad hoc. It was chosen because it causes the sum of the activations of all of the objects in a type to equal the activation of the type. `Object134` in Section 5.1 has an activation of 0.1606 ( = 0.1606×1.0).

The goodness-of-fit of the match is determined by how well the values of the object's attributes are fit by the patterns in the condition. When the values of an attribute are defined on a *nominal* scale (i.e., they are symbolic), the match must be all-or-none, so a successful match is a perfect match (assigned a value of 1.0). When the values are defined on a *ratio* scale (i.e. a real number), a more sophisticated partial match is possible (this is the real-number match alluded to earlier). The pattern for such a value specifies the *expected* value, and an *interval* around the expected value in which the match is considered to have succeeded. The goodness-of-fit is perfect (1.0) for the expected value, and decreases monotonically to threshold (0.0001) at the edge of the interval.

$$Activation = e^{-9.21[(value-expected)/interval]^2} \tag{8}$$

This partial match can be symmetric, with the match interval defined on both sides of the expected value, or one sided (just greater-than or just less-than). Though this partial matching mechanism was added to ***Xaps2*** to model the topological space in which Seibel's task is defined, its utility is definitely not limited to just this task.

The total goodness-of-fit measure is the product of the measures for each of the attributes in the condition. The activation of the condition match can then be defined as the product of this measure and the activation of the working-memory object. No deep justification will be attempted for this scheme; it is enough that it is one possible mechanism and that it has proven more tractable in practice than the alternatives that have been tried.

To make this whole mechanism more concrete, consider as an example the match of the following condition to `Object134`:

*If* there is an instance of the **OneTrial goal**
that has an INITIAL-LOCATION > 0.7 (within 1.0)

In this condition, the type is specified by the constant symbol **OneTrial**, and the identifier is specified as a variable (signified by an equals sign before the name of the variable) named Identifier. These both match successfully, so the activation of the object is 0.1606 (as computed above). There is only one attribute specified, so its goodness-of-fit is simply multiplied by 0.1606 to get the activation of the condition. The value is specified by the condition as a one-sided (greater than) real-number comparison with the interval for a

successful match set to 1.0. The expected value (0.7) is compared with the dominant value of the INITIAL-LOCATION attribute (0.99), yielding a goodness of fit of 0.4609 (from Equation 8). The activation for this match is thus 0.0740 ( = 0.1606×0.4609).

### 5.3.2. Conflict resolution

Conflict resolution selects which of the instantiations generated by the match should be fired on a cycle. This is done by using rules to eliminate unwanted instantiations. The first rule performs a thresholding operation.

- Eliminate any instantiation with an activation value lower than .0001.

The second rule is based on the hypothesis that productions are a limited resource.

- Eliminate all but the most highly activated instantiation for each production.

This rule is similar to the special case and working-memory recency rules in the *Ops* languages. It allows the selection of the most focussed (activated) object from a set of alternatives. Following the execution of these conflict-resolution rules, there is at most one instantiation remaining for each production. While this eliminates within-production parallelism, between-production parallelism has not been restricted. It is possible for one instantiation of every production to be simultaneously executing. This provides a second locus of the parallelism required by the parallel constraint.

What happens next depends on the execution-types of the productions that generated the instantiations (see Section 5.2).

- Instantiations of **Always** productions are always fired.
- Instantiations of **Decay** productions are always fired, but the activation of the instantiation is cut in half each time the identical instantiation fires on successive cycles. A change in the instantiation occurs when one of the variable bindings is altered. This causes the activation to be immediately restored to its full value.
- Instantiations of **Once** productions are fired only on the first cycle in which the instantiation would otherwise be eligible. This is a form of refractory inhibition.

Standard *Xaps2* productions are of execution-type **Always**, and nearly all of the productions in the model are of this type. **Decay** productions have found limited use as a resettable clock (Section 7.1.1.2), while no productions of execution-type **Once** have been employed.

### 5.3.3. Production execution

Following conflict resolution, all of the instantiations still eligible are fired in parallel, resulting in the execution of the productions' actions. Each production may execute one or more actions, providing a third type of parallelism in the architecture. The actions look like conditions (see the example in Section 5.2); they are partial specifications of working-memory objects. Execution of an action results in the creation of a fully specified version of the object. Variables in the action are replaced by the values bound to those variables during the match, and new symbols are created as requested by the action.

Unlike the ***Ops*** languages, actions only cause modifications to working memory; they are *not* a means by which the model can communicate with the outside world. Communication is an explicit part of the task model (Section 6.1). Actions modify working memory in an indirect fashion. The effects of all of the actions on one cycle are accumulated into a single data structure representing the updates to be made to working memory. The actual updating occurs in the next (and final) stage of the production system cycle (described in the following section).

The first step in creating the pre-update structure is to assign activation levels to the components of the objects asserted by the production actions. The identifier of the objects, and all of the values asserted for attributes, are assigned an activation level equal to the activation of the production instantiation asserting them. This allows activation to flow through the system under the direction of productions (like ***Hpsa77*** (Newell, 1980) and ***Caps*** (Thibadeau, Just, & Carpenter, 1982)), as opposed to the undirected flow employed by spreading-activation models (Anderson, 1976). No good scheme has been developed for assigning activation to the type; currently, it is just given a fixed activation of 1.0. The activation levels can be made negative by inhibiting either the whole object, or a specific value. This is analogous to the negation of conditions (and values) during the match.

If the same object is asserted by more than one action, the effects are accumulated into a single representation of the object: the type activation is set to the same fixed constant of 1.0; the identifier activations are summed and assigned as the identifier activation; and all of the activations of the same value (of the same attribute) are summed and assigned as the activation of that value. This aggregation solves the problem of synchronizing simultaneous modifications of working memory. Activation and inhibition are commutative, allowing the actions to be executed in any order without changing the result. The same is not true of the operations of insertion and deletion, as used in the ***Ops*** languages.

After the actions have been aggregated, any external stimuli to the system are added into this structure. External stimuli are objects that come from outside of the production system, such as the lights in Seibel's task. These stimuli are inserted into the pre-update structure just as if they were the results of production

actions. An activation value of 0.01 is used for them. This level is high enough for the stimuli to affect the system, and low enough for internally generated objects to be able to dominate them.

Following the inclusion of stimuli, the pre-update structure is normalized (just as if it were the working memory) and used to update the current working-memory state (Section 5.3.4). Normalization of the pre-update structure allows for the control of the relative weights of the new information (the pre-update structure) and the old information (working memory).

### 5.3.4. Updating of working memory

The updates to working memory could be used simply as a replacement for the old working memory (as in Joshi, 1978), but that would result in working memory being peculiarly memoryless. By combining the pre-update structure with the current working memory, we get a system that is sensitive to new information, but remembers the past, for at least a short time. Many combination rules (such as adding the two structures together) are possible, and many were experimented with in ***Xaps***. In ***Xaps2***, the two are simply averaged together. This particular choice was made because it interacts most easily with the lack of refractoriness in production firing. The updates can be thought of as specifying some *desired* state to which the productions are trying to drive working memory. Repetitive firing of the same set of production instantiations results in working memory asymptotically approaching the desired state. Any weighted sum of the new and the old (with the weights summing to 1) would yield similar results, with change being either slower or faster. Averaging (equal weights of 0.5) was chosen because it is a reasonable null assumption.

Anything that is in one of the states being combined but not the other is assumed to be there with an activation value of 0.0. Thus, ignoring normalization, items not currently being asserted by productions (ie., not in the pre-update structure) exponentially decay to zero, while asserted items exponentially approach their asserted activation levels. This applies to inhibited as well as activated items — inhibition decays to zero if it is not continually reasserted.

Averaging the two working-memory states preserves the total activation, modulo small threshold effects, so the effects of normalization are minimal when it is employed with this combination rule. It has a noticeable effect only when no item within the scope of the normalization is being asserted by a production. Without normalization, all of the items would decay to zero. With normalization, this decay is reversed so that the activations of the items once again sum to 1. The result is that the activations of the items remain unchanged. Basically, items stick around as long as they have no active competition. Old items that have competition from new items will decay away.

One consequence of the gradual updating of working memory is that it often takes more than one cycle to

achieve a desired effect. This typically happens when the dominant value of an attribute is being changed. Addition of new attributes can always be accomplished in one cycle, but modifying old ones may take longer. It is essential that knowledge of the desired change remain available until the change has actually been made. In fact, some form of test production is frequently required to detect when the change has been completed, before allowing processing to continue.

## 6. The Initial Performance Model

The chunking theory has been applied to Seibel's task, yielding a model that improves its performance with practice. Not covered by this model is the initial learning of a correct method for the task. Our future plans include extending the chunking theory to the domain of method acquisition, but until then, the model must be initialized with a correct method for performing the task. We consider only a single method, based on the algorithm at the end of Section 3, though subjects exhibit a range of methods. This method is straightforward but slow — efficiency comes from chunking.

The algorithm is implemented as a hierarchy of five goals (Figure 6-1). Each goal is a working-memory type, and each goal instance is an object of the relevant type. In addition to the goal-types, there are two types representing the model's interfaces with the outside world, one at the stimulus end, and the other at the response end. We will start with a description of these interfaces, and then plunge into the details of the model's internal goal hierarchy.

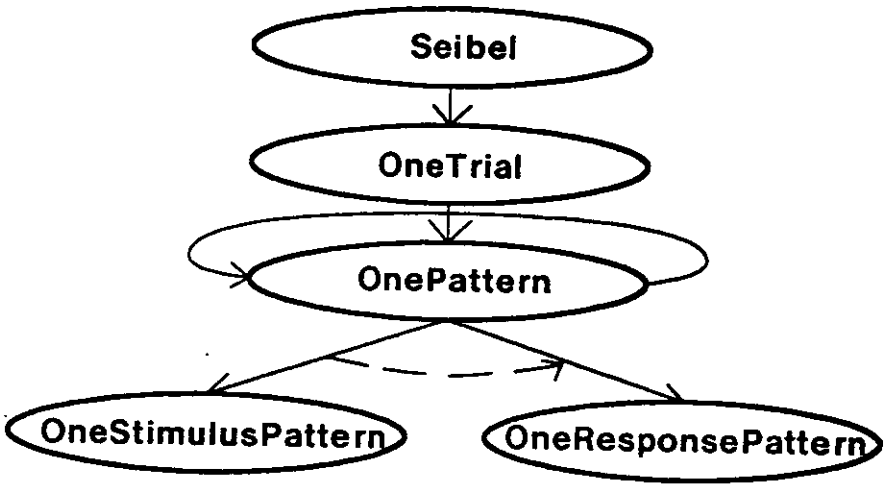



**Figure 6-1:** The model's goal hierarchy for Seibel's task.

### 6.1. Interfacing with the outside world

The model interacts with the outside world through two two-dimensional euclidean spaces. These spaces are defined in terms of object-centered coordinates. One space, the *stimulus* space, represents the information received by the model as to the location of the lights within the stimulus array. The other space, the *response* space, represents the information that the model transmits to the motor system.

The locations of objects within these spaces are specified by relative *x,y* coordinates. The exact coordinate system used is not critical, but this particular one has proven to be convenient. The use of rectangular coordinates allows left-to-right traversal across the lights to be accomplished by just increasing *x*. With relative coordinates, the left (top) edge of the space is 0.0, and the right (bottom) edge is 1.0. Since the buttons and lights in the task have been arranged so as to maximize the compatibility of their locations, using the same set of relative coordinates for the two spaces makes trivial the job of mapping stimulus locations into response locations.

### 6.1.1. The stimulus space

The stimulus space is a rectangle just bounding the total array of lights. To the model, this array appears as a set of objects representing the lights (both *On* and *Off*). A typical *On*-light looks like (ignoring activations):

(**External-Stimulus** `Object0012` [COMPONENT-PATTERN *On*] [SPATIAL-PATTERN *One*]
[MINIMUM-X *0.21*] [MAXIMUM-X *0.36*]
[MINIMUM-Y *0.00*] [MAXIMUM-Y *0.30*])

All stimuli have the same type (**External-Stimulus**), but the identifier is unique to this light on this trial. Productions must match the object by a description of it, rather than by its name.

A total of six attributes are used to describe the stimulus object. Two of the attributes (COMPONENT-PATTERN and SPATIAL-PATTERN) specify the pattern represented by the object. The particular stimulus object above represents just a single *On*-light, but stimulus objects can represent patterns of arbitrary complexity (such as an arrangement of multiple lights). The attribute COMPONENT-PATTERN specifies what kind of objects make up the pattern — limited to *On* and *Off* (lights) for this task. The other attribute, SPATIAL-PATTERN, specifies the spatial arrangement of those components. The value *One* given in object `Object0012` signifies that the object consists of one *On*-light, and nothing else. This single value suffices for the initial performance model, but others are created when new chunks are built.

The remaining four attributes (MINIMUM-X, MAXIMUM-X, MINIMUM-Y, and MAXIMUM-Y) define the *bounding box* of the stimulus pattern. The bounding box is a rectangle just large enough to enclose the stimulus. It is specified by its minimum and maximum x and y coordinates. For example, object `Object0012` above is flush against the top of the stimulus space, and a little left of center.

We make the simplifying assumption that the entire array of lights is constantly within the model's "visual field". This cannot be literally true for our subjects because of the large visual angle subtended by the display (16°), but was more true for Seibel's subjects who worked with a display covering 7° of arc. Because the model is assumed to be staring at the lights at all times during performance of the task, the stimulus objects are inserted into working memory on every cycle of the production system (see Section 5.3.3 for how this is done).

### 6.1.2. The response space

The response space is constructed analogously to the stimulus space; it is a rectangle just bounding the array of buttons. This is a response space (as opposed to a stimulus space) because the objects in it represent patterns of modifications to be made to the environment, rather than patterns of stimuli perceived in the environment. Objects in this space represent locations at which the model is going to press (or not press). The fingers are not explicitly modelled; it is assumed that some other portion of the organism enables finger movement according to the combination of location and action.

Response objects look much like stimulus objects. For example, the response object corresponding to stimulus object `Object0012` might look like:

(**External-Response** `Object0141` [COMPONENT-PATTERN *Press*] [SPATIAL-PATTERN *One*]
[MINIMUM-X *0.3*] [MAXIMUM-X *0.36*]
[MINIMUM-Y *0.0*] [MAXIMUM-Y *0.2*])

The only differences are the type (**External-Response**), the identifier, which is unique to this instance of this response, and the value of COMPONENT-PATTERN, which is *Press* rather than *On*.

Response objects are created dynamically by the model as they are needed. Once they are created, response objects hang around in working memory until competition from newer ones causes them to drop out.

## 6.2. The control structure: a goal hierarchy

The control structure imposed upon the ***Xaps2*** architecture is that of a goal hierarchy. This control structure is totally serial at the level of the goal. Information about proposed subgoals and suspended supergoals can coexist with the processing of a goal instance, but only one such instance can be actively pursued at a time. The choice of this tightly controlled structure is not forced by the nature of the architecture. Instead, it came from the following three motivations.

- The control structure provides the bottleneck required by the bottleneck constraint. Though this satisfies the constraint, it does so only in a weak sense because it is not an architectural limitation. This contrasts with ***Hpsa77*** (Newell, 1980), in which the mechanism of variable binding creates a structural bottleneck in the architecture.

- The bottleneck is only across goals, not within goals. During the processing of goal instances, productions are free to execute in parallel. The parallel constraint is therefore still met. In addition, the cognitive-parallelism constraint is met; goals are employed all through the performance system, so the locus of parallelism is not limited to just the sensory and motor components.

- Complicated execution paths (such as iterative loops) are difficult to construct in loosely controlled systems. While such systems may be logically adequate, convincing activation-based

control schemes to loop, solely on the basis of activation values, has proven difficult to accomplish.

The first requirement of a system that employs a goal hierarchy is a representation for the goals. As stated earlier, each goal is represented as an object type, and a goal instance is represented as an object with a unique identifier. `Object134` in Section 5.1 represents a typical goal instance — the goal name is **OneTrial**, and the identifier is `Object134`. Because goals can be distinguished by their types, and multiple instances of the same goal can be distinguished by their identifiers, it is possible to maintain information about a number of goal instances simultaneously.

The goal hierarchy is processed in a depth-first fashion, so the second requirement is a *stack* in which the current state of execution can be represented. In ***Xaps2***, working memory does not behave as a stack; more recent objects will tend to be more highly activated, but this is not sufficient for the implementation of a goal hierarchy. The primary difficulty involves simultaneously keeping the goals in the stack active and maintaining the proper ordering among them. If the stack is just left alone, subgoal activity causes the objects in the stack to decay. The oldest objects may very well decay right out of working memory. If the stack is continually refreshed by reassertion of its elements into working memory, then the ordering, which depends on activation levels, will be disturbed. Some variation on this scheme may still work, but we have instead pursued a more symbolic representation of the goal stack.

Each goal instance has a STATUS attribute. Together, the STATUS attributes (that is, the dominant value of the STATUS attributes) of the active goal instances completely determine the control state of the model. Three common STATUS values are *Start*, *Started*, and *Done*. *Start* means that the goal is being initialized; *Started* signifies that initialization is complete and that the goal is being pursued; and *Done* signifies that the goal has completed. The stack is implemented by pointing to the current subgoal of a suspended supergoal via the STATUS attribute of the supergoal. Notice that any goal instance whose STATUS is the identifier of some other goal instance must be suspended by definition, because its STATUS is no longer *Started*. A goal can therefore be interrupted at any time by a production that changes its STATUS from *Started* to some other value. Execution of the goal resumes when the STATUS is changed back to *Started*.

Activating a subgoal of a currently active goal is a multi-step operation. The first step is for the goal to signify that it wants to activate a subgoal of a particular type. This is accomplished by changing the STATUS of the goal to the type of the subgoal that should be started. This enables the productions that create the new subgoal instance. Four tasks must be performed whenever a new subgoal is started.

1. The current goal instance must be blocked from further execution, until the subgoal is completed.

2. A new instance of the subgoal must be created. This is a new object with its own unique identifier.

3. The parameters, if any, must be passed from the current goal instance to the subgoal instance.

4. A link, implementing the stack, must be created between the current goal instance and the new subgoal instance.

As noted above, the supergoal is suspended as soon as the desire for the subgoal is expressed (task 1). Changing the STATUS of the current goal instance effectively blocks further effort on the goal. The other three tasks are performed by a set of three productions. Because the type of an object (in this case, the name of the goal) cannot be matched by a variable, a distinct set of productions is required for each combination of goal and subgoal. One benefit of this restriction is that the goal-instance creation productions can perform parameter passing from goal to subgoal as part of the creation of the new instance. The first production of the three, performs precisely these two tasks: (2) subgoal creation, and (3) parameter passing. Schematically, these productions take the following form.

Production schema Start<Goal name>:

*If* the current goal instance has a subgoal name as its STATUS
*then* generate a new instance of the subgoal with STATUS *Start*
(parameters to the subgoal are passed as other attributes).

When such a production executes, it generates a new symbol to be used as the identifier of the object representing the goal instance. The second production builds a stack link from a goal to its subgoal (task 4), by copying this new identifier into the STATUS attribute of the current goal instance. This must be done after the first production fires, because this production must examine the newly created object to determine the identifier of the new goal instance.

Production schema CreateStackLink<Goal name>:

*If* the current goal instance has a subgoal name as its STATUS
*and* there is an active object of that type with STATUS *Start*
*then* replace the goal's STATUS with the subgoal's identifier.

The third production checks that all four tasks have been correctly performed before enabling work on the subgoal:

Production schema Started<Goal name>:

*If* the current goal instance has a subgoal identifier as its STATUS
*and* that subgoal has STATUS *Start*
*then* change the STATUS of the subgoal to *Started.*

At first glance, it would appear that the action of this third production could just be added to the second

production. In most production systems this would work fine, but in ***Xaps2*** it doesn't. One production would be changing the values of two attributes at once. Since there is no guarantee that both alterations would happen in one cycle, a race condition would ensue. If the subgoal is *Started*, before the stack link is created, the link will never be created. Generally in ***Xaps2***, separate productions are required to make a modification and test that the modification has been performed.

It generally takes one cycle of the production system to express the desire for a subgoal, and three cycles to activate the subgoal (one cycle for each of the three productions), for a total of four cycles of overhead for each new subgoal. This number may be slightly larger when any of the modifications requires more than one cycle to be completed.

Goals are terminated by *test* productions that sense appropriate conditions and change the STATUS of the goal instance to *Done*. If the subgoal is to return a result, then an intermediate STATUS of *Result* is generated by the test productions, and additional productions are employed to return the result to the parent goal and change the STATUS of the subgoal instance to *Done*, once it has checked that the result has actually been returned. The standard way of returning a result in ***Xaps2*** is to assert it as the new value for some attribute of the parent goal instance. It may take several cycles before it becomes the dominant value, so the production that changes the STATUS to *Done* waits until the result has become the dominant value before firing. Terminating a goal instance generally requires one cycle, plus between zero and four cycles to return a result.

The parent goal senses that the subgoal has completed by looking for an object of the subgoal type whose identifier is identical to the parent's STATUS, and whose own STATUS is *Done*. Once this condition is detected, the parent goal is free to request the next subgoal, or to continue in any way that it sees fit.

The mechanism described so far solves the problem of maintaining the order of stacked goal instances. However, it does not prevent the objects representing these instances from decaying out of working memory. This is resolved by an additional production for each goal-subgoal combination that passes activation from the goal type to the subgoal type. The topmost goal type passes activation to itself and downwards to the next level. All of the other goal types simply pass activation to their subgoal types. These productions fire on every cycle.

Keeping the types (goals) active insures that at least one instance of each goal can be retained on the stack. Multiple instances of the same goal, such as would be generated by recursion, would result in lossage of instances through competition. In order for recursion to work, either the architecture would have to be changed to fire all instantiations of a production (one goal instance per instantiation) instead of only the "best", or a separate production would be required for each instance (which must be created dynamically, as are the goal instances). The five goal types are discussed in the following sections.

### 6.2.1. The Seibel goal

Seibel is the top level goal type for the task. It enters the working memory as a stimulus from the outside world (see Section 5.3.3 for a discussion of stimuli), corresponding to a request to perform the task. The Seibel goal type is used solely to keep the **OneTrial** goal active.

### 6.2.2. The OneTrial goal

A desire for a new instance of the **OneTrial** goal is generated exogenously each time the stimulus array is changed, that is, once each trial. Both this desire, and the new stimulus array are inserted into working memory as stimuli. The **Seibel** goal could have detected the presence of a new stimulus array and generated the **OneTrial** goal directly, but we have taken this simpler approach for the time being because we wanted to focus our attention on within-trial processing.

The **OneTrial** goal implements the following aspects of the performance algorithm (Section 3):

**Focus** a point to the left of the leftmost light.
**While** there is an *On*-light to the right of the focal point **Do**
<Goal **OnePattern**>

The point of focus is modelled as the value of an attribute (INITIAL-LOCATION) of the **OneTrial** goal instance. This should be thought of as the focus of attention within the visual field, rather than as the locus of eye-fixation. Setting the initial focus takes two cycles. First the goal's STATUS is changed to *Initialize*, and then a production which triggers off of that STATUS, sets the value of the INITIAL-LOCATION to *0.0* (the left edge of the stimulus space).

The entire body of the **While** loop has been moved inside of a single goal (**OnePattern**), so the loop is implemented by repeatedly starting up **OnePattern** goal instances. The first instance is created when a test production has determined that the INITIAL-LOCATION has been set. Subsequent instances are established whenever the active **OnePattern** instance has completed. The focal point gets updated between iterations because the **OnePattern** goal returns as its result the right edge of the light pattern that it processed. This result is assigned to the INITIAL-LOCATION attribute.

What we have described so far is an infinite loop; new instances of the **OnePattern** goal are generated endlessly. This is converted into a **While** loop with the addition of a single production of the following form.

Production DoneOneTrial:

*If* there is a **OneTrial** goal with STATUS **OnePattern**
*and* there is no *On*-light to the right of its INITIAL-LOCATION
*then* the **OneTrial** goal is *Done*.

The test for an *On*-light to the right of the INITIAL-LOCATION is performed by a one-sided (greater than) real-number match to the MINIMUM-X values of the stimulus objects. The expected value is the INITIAL-LOCATION, and the interval is 1.0. The match will succeed if there is another light to the right, and fail otherwise. The production above, therefore has this test negated.

The reaction time for the model on Seibel's task is computed from the total number of cycles required to complete (STATUS of *Done*) one instance of the **OneTrial** goal. Experimentally, this has been determined to be a fixed overhead of approximately 13 cycles per trial, plus approximately 31 cycles for each *On*-light — an instance of the **OnePattern** goal (see Section 8.1). These numbers, and those for the following goals, are from the full performance model, which is the initial performance model with some additional overhead for the integration of chunking into the control structure (Sections 7.1.1.2 and 7.1.2).

### 6.2.3. The OnePattern goal

The **OnePattern** goals control the four steps inside the **While** loop of the performance strategy:

**Locate** the *On*-light.
**Map** the light location into the location of the button under it.
**Press** the button.
**Focus** the right edge of the light.

Two of these steps (**Map** and **Focus**) are performed directly by the goal instance, and two (**Locate** and **Press**) are performed by subgoals (**OneStimulusPattern** and **OneResponsePattern**).

At the start, a typical instance of the **OnePattern** goal looks like[11]:

(**OnePattern** `Object45` [INITIAL-LOCATION *0.0*])

The first step is to locate the next stimulus pattern to process. This is accomplished by a subgoal, **OneStimulusPattern**, which receives as a parameter the INITIAL-LOCATION, and returns the attributes of the first *On*-light to the right of the INITIAL-LOCATION. These attributes are added to the **OnePattern** instance, to yield an object like:

(**OnePattern** `Object45`
[INITIAL-LOCATION *0.0*]
[STIMULUS-COMPONENT-PATTERN *On*] [STIMULUS-SPATIAL-PATTERN *One*]
[STIMULUS-MINIMUM-X *0.21*] [STIMULUS-MAXIMUM-X *0.36*]
[STIMULUS-MINIMUM-Y *0.00*] [STIMULUS-MAXIMUM-Y *0.30*])

The mapping between stimulus and response is currently wired directly into the performance algorithm. This is sufficient but not essential for the current model. In some follow-on work we are investigating the relationship between this model and stimulus-response compatibility. In these systems, the mapping is

[11] For simplicity of presentation, an additional STATUS attribute in the following three examples is not shown.

performed in a separate sub-goal. This provides flexibility, and the ability to perform a considerable amount of processing during the mapping.

The mapping employed in the current model is a minimal one; all that is required is turning the stimulus attributes into response attributes, and changing the COMPONENT-PATTERN from *On* to *Press*. This mapping is performed by a single production to yield an object of the following form.

```
(OnePattern Object45
            [INITIAL-LOCATION 0.0]
            [STIMULUS-COMPONENT-PATTERN On] [STIMULUS-SPATIAL-PATTERN One]
            [STIMULUS-MINIMUM-X 0.21] [STIMULUS-MAXIMUM-X 0.36]
            [STIMULUS-MINIMUM-Y 0.00] [STIMULUS-MAXIMUM-Y 0.30])
            [RESPONSE-COMPONENT-PATTERN Press] [RESPONSE-SPATIAL-PATTERN One]
            [RESPONSE-MINIMUM-X 0.21] [RESPONSE-MAXIMUM-X 0.36]
            [RESPONSE-MINIMUM-Y 0.00] [RESPONSE-MAXIMUM-Y 0.30])
```

These response parameters are passed to a subgoal, **OneResponsePattern**, that converts them into a new response object. Following completion of this subgoal, the **OnePattern** goal terminates, passing the coordinate of the right edge of the selected stimulus pattern as its result (to be used as the focal point for the next search).

Not counting the time to perform its two subgoals, a typical instance of this goal requires 12 cycles of the production system, including the overhead involved in starting and finishing the goal.

### 6.2.4. The OneStimulusPattern goal

The **OneStimulusPattern** goal is responsible for finding the next *On*-light to the right of the INITIAL-LOCATION, which it receives as a parameter from its parent **OnePattern** goal. This selection is made by a single production employing the same type of one-sided real-number match used to determine when the trial has completed (Section 6.2.2). It looks for an *On*-light to the right of the INITIAL-LOCATION for which there are no other *On*-lights between it and the INITIAL-LOCATION. (recall that *Off*-lights are ignored in this algorithm). As long as there is some *On*-light to the right of the INITIAL-LOCATION it will be found. If there is more than one, the closest one to the INITIAL-LOCATION is selected. On completion, the goal instance returns the values of the six attributes describing the selected stimulus pattern to the parent **OnePattern** goal.

A typical instance of this goal requires 9 cycles of the production system, including the overhead involved in starting and finishing the goal, though it may be higher.

### 6.2.5. The OneResponsePattern goal

Conceptually, the **OneResponsePattern** goal is the mirror image of the **OneStimulusPattern** goal. Given the parameters of a response object, its task is to create a new response object with those values. We assume that the motor system automatically latches on to response objects as they are created, so creation of the object is the last step actually simulated in the model.

A typical instance of this goal requires 10 cycles of the production system, including the overhead involved in starting and finishing the goal, though it may be higher.

## 7. The Chunking Process

The initial performance model executes an instance of the **OnePattern** goal for each *On*-light in the stimulus array. Patterns consisting of a single *On*-light are *primitive* patterns for the model; they are at the smallest grain at which the perceptual system is being modelled. Larger, or *higher-level*, patterns can be built out of combinations of these primitive patterns. For example, a single higher-level pattern could represent the fact that four particular lights are all *On*. The same holds true for response patterns, where the primitive patterns are single key presses. Higher-level response patterns that specify a combination of key presses can be built out of these primitive response patterns.

According to the chunking theory of learning, chunks represent patterns experienced in the task environment. They improve performance because it is more efficient to deal with a single large pattern than a set of smaller patterns. The remainder of this section describes the design of this chunking process — how chunks represent environmental patterns, and how chunks are acquired from task performance. As currently constituted, this is an error free design; chunks are always acquired and used correctly. Rather than model errors directly by a bug-laden final model, the problem of errors is tackled by discussing the types of errors simple versions of the model naturally make, and the mechanisms implemented to ensure that these errors do not occur.

### 7.1. The representation of chunks

We propose that a chunk consists of three components: (1) a stimulus pattern, (2) a response pattern, and (3) a connection between the two. In contrast to systems that treat chunks as static data structures, we consider a chunk to be an active structure. A chunk is the productions that process it. The obvious implementation of this proposal involves the creation of one production per chunk. The production would have one condition for each primitive component of the stimulus pattern, and one action for each primitive component of the response pattern. The connection is implemented directly by the production's condition-action link. This implementation is straightforward enough, but it is inadequate for the following reasons.

- These productions violate the control structure of the model by linking stimuli to responses directly, without passing through the intervening bottleneck. If such productions could be created, then it should also be possible to create the optimal algorithm of ten parallel productions, one for each light-button combination.

- The chunking mechanism implied by these productions is non-hierarchical; a chunk is always defined directly in terms of the set of primitive patterns that it covers.

- The direct connection of stimulus to response implies that it is impossible for the cognitive system to intervene in the middle. The mapping of stimulus to response is wired in and unchangeable.

These problems can all be solved by implementing each chunk as three productions, one for each component. The first production *encodes* a set of stimulus patterns into a higher-level stimulus pattern; the second production *decodes* a higher-level response pattern into a set of smaller response patterns; and the third production indirectly (see below) *connects* the higher-level stimulus pattern to the higher-level response pattern.

For the acquisition of a chunk to improve the performance of the model, these productions must help overcome the bottleneck caused by the model's inability to process more than one pattern at a time. This bottleneck can be precisely located within the **OnePattern** goal — between the termination of the **OneStimulusPattern** goal and the beginning of the **OneResponsePattern** goal. According to the encoding constraint, encoding must occur before the bottleneck, that is, before the **OneStimulusPattern** goal completes and selects the pattern to use. Likewise, the decoding constraint implies that decoding must occur after the bottleneck, that is, after the start of the **OneResponsePattern** goal. The connection production must appear somewhere in between.

The model must execute an instance of the **OnePattern** goal for each pattern processed — approximately 31 production-system cycles (Section 6.2.2). If there are four *On*-lights in the stimulus, then the initial performance model requires four iterations, or about 124 cycles. If one pattern can cover all four *On*-lights, only one iteration is required, cutting the time down to 31 cycles. If instead we had two patterns of two *On*-lights each, it would take two iterations, or about 62 cycles. Just as the chunking theory of learning proposes, performance can be improved through the acquisition of patterns experienced during task performance.

For simplicity, the current system works only with chunks that are built out of exactly two subchunks. This is not a limitation on the theory; it is merely the simplest assumption that still lets us investigate most of interesting phenomena. The remainder of this section describes how the three components of a chunk are represented and how they are integrated into the model's control structure. We delay until the following section the description of how a chunk is built.

### 7.1.1. The encoding component

The initial performance model perceives the world only in terms of primitive stimulus patterns consisting of either a single *On*-light or a single *Off*-light. The encoding component of a chunk examines the currently perceived patterns, as reflected by the contents of working memory, and based on what it sees, may assert a new higher-level stimulus pattern. When this new object appears in working memory, it can form the basis for the recognition of even higher-level patterns. The entire set of encoding productions thus performs a hierarchical parsing process on the stimuli.

Encoding is a goal-free data-driven process in which productions fire whenever they perceive their pattern. This process is asynchronous with the goal-directed computations that make up most of the system. This works because the perceived patterns interact with the rest of the system through a filter of goal-directed selection productions. As an example, the selection production in Section 6.2.4 chooses one pattern from the stimulus space based on its location and COMPONENT-PATTERN.

In essence, we are proposing that the traditional distinction between parallel data-driven *perception* and serial goal-directed cognition be modified to be a distinction between parallel data-driven *chunk encoding* and serial goal-directed cognition. In the remainder of this section we describe the details of this chunk-encoding process.

#### 7.1.1.1. Representation of higher-level stimulus patterns

All stimulus patterns, be they primitive or higher-level, are represented as working-memory objects of type External-Stimulus. For purposes of comparison, here are objects representing a primitive pattern, and a higher-level pattern.

```
(External-Stimulus Primitive-Example [COMPONENT-PATTERN On]
                                     [SPATIAL-PATTERN One]
                                     [MINIMUM-X 0.21] [MAXIMUM-X 0.36]
                                     [MINIMUM-Y 0.00] [MAXIMUM-Y 0.30])

(External-Stimulus Higher-Level-Example [COMPONENT-PATTERN On]
                                        [SPATIAL-PATTERN Spatial-Pattern-0145]
                                        [MINIMUM-X 0.21] [MAXIMUM-X 0.78]
                                        [MINIMUM-Y 0.00] [MAXIMUM-Y 0.64])
```

They are almost identical, what differs is the values of some attributes. The four attributes defining the bounding box are interpreted in the same fashion for all patterns. They always define the rectangle just bounding the pattern. For primitive chunks, this is a rectangle just large enough to contain the light. For higher-level chunks, it is the smallest rectangle that contains all of the lights in the pattern.

The COMPONENT-PATTERN of primitive patterns is always *On* or *Off*, signifying the type of light contained in the pattern. For higher-level patterns, a value of *On* is interpreted to mean that all of the lights contained in the pattern are *On*. Other values are possible for higher-level patterns, but in the current task we only deal with patterns composed solely of *On*-lights. This means that the *Off*-lights are dealt with by ignoring them — not that the *Off*-lights can't be there.

The primary difference between primitive and higher-level patterns is in the value of the SPATIAL-PATTERN attribute. For primitive patterns it always has the value *One*, signifying that the entire bounding box contains just a single light. For higher level patterns, the value must indicate how many *On*-lights there are within the box, and what their positions are. One alternative for representing this information is to store it explicitly in

the object in terms of a pattern language. The pattern language amounts to a strong constraint on the variety of patterns that can be perceived. This is the tactic employed in most concept formation programs (for example, Evans, 1968, and Mitchell, Utgoff, Nudell, & Banerji, 1981). It is a powerful technique within the domain of the pattern language, but useless outside of it.

We have taken the less constrained approach pioneered by Uhr & Vossler (1963), in which there is little to no precommitment as to the nature of the patterns to be learned. A unique symbol is created to represent each newly perceived pattern. This symbol is stored as the value of the SPATIAL-PATTERN attribute — *Spatial-Pattern-0145* in the example above. Instead of the meaning being determined in terms of a hard-wired pattern language, it is determined by the productions that act on the symbol. The encoding production knows to create an object with this symbol when it perceives the appropriate lower-level patterns. Likewise, the connection production knows how to create the appropriate response object for this symbol. With this scheme, any pattern can be represented, but other operations on patterns, such as generalization, become difficult.

#### 7.1.1.2. Integration of the encoding component into the model

When the concept of chunking is added to the initial performance model, changes in the control structure are needed for the model to make use of the newly generated higher-level patterns. The initial performance model iterates through the lights by repeatedly selecting the first *On*-light to the right of the focal point, and then shifting the focal point to the right of the selected light. When there are higher-level patterns, this algorithm must be modified to select the *largest* pattern that starts with the first *On*-light to the right of the focal point, while shifting the focal point to the right of the pattern's bounding box. Accomplishing this involves simply changing the selection production so that it does not care about the SPATIAL-PATTERN of the object that it selects. It then selects the most highly activated stimulus object consisting of only *On*-lights, with no other such object between it and the INITIAL-LOCATION. The largest pattern is selected because a higher-level pattern containing $n$ components will be more highly activated than its components. If a production has $n$ equally activated conditions, call the activation $a$, then its actions will be asserted with an activation level of $(\sqrt{n})\cdot a$ (derived from $n\cdot a / \sqrt{n}$).

Originally, it was intended that this selection be based solely on the match activation of the competing instantiations. The effect of size was added (via the combination of activation) to the effect of nearness to the INITIAL-LOCATION (via a real-number match). This often worked, but it did lead lead to omission errors in which a large pattern was preferred to a near pattern, skipping over intermediate *On*-lights without processing them. To avoid these errors, the more explicit location comparison process described in Section 6.2.4 is currently employed.

Selection now works correctly, that is, if the encoding process has completed by the time the selection is made. Since encoding is an asynchronous, logarithmic process, determining the time of its completion is problematic. This problem is partly solved by the data-driven nature of the encoding productions. Encoding starts as soon as the stimuli become available, not just after the **OneStimulusPattern** goal has started. This head start allows encoding usually to finish in time.

For the cases when this is insufficient, a pseudo-clock is implemented by the combination of an **Always** production and a **Decay** production. Encoding takes an amount of time dependent on the height of the chunk hierarchy, so waiting a fixed amount of time does not work. Instead, the clock keeps track of the time between successive assertions of new stimulus patterns by encoding productions. If it has been too long since the last new one, encoding is assumed to be done. The clock is based on the relative activation levels of two particular values of an attribute. One value remains at a moderate level; the other value is reset to a high level on cycles in which a new pattern is perceived, and decays during the remainder. When the activation of this value decays below the other value, because no new encoding productions have fired, encoding is considered to be done. This mechanism is clumsy but adequate.

#### 7.1.1.3. The encoding productions

Encoding productions all have the same structure, consisting of three conditions and one action. The three conditions look for the two stimulus patterns that make up the new pattern, and the absence of other *On* patterns between the two desired ones. The action creates a new object in working memory representing the appropriate higher-level pattern.

At first glance, only the first two conditions would seem to be necessary, but absence of the third condition can lead to errors of omission. Suppose that an encoding production is created for a pattern consisting of a pair of *On*-lights separated by an *Off*-light. If the middle light is *Off* the next time the two lights are *On*, there is no problem. The problem occurs when all three lights are *On*. Without the third condition, the production would match and the higher-level pattern would be recognized. If that pattern is then used by the performance system, it would press the buttons corresponding to the two outer lights, and then move the focal point past the right edge of the pattern's bounding box. The middle *On*-light would never be processed, resulting in a missing key press. By adding the third condition, the pattern is not recognized unless there is no *On*-light embedded between the two subpatterns. These errors are therefore ruled out.

Let's look at a couple of concrete examples. In this first example we encode two primitive patterns (*On*-lights) separated by an *Off*-light. The relevant portion of working memory is:

(External-Stimulus Object0141 [COMPONENT-PATTERN *On*]
[SPATIAL-PATTERN *One*]
[MINIMUM-X *0.21*] [MAXIMUM-X *0.36*]
[MINIMUM-Y *0.00*] [MAXIMUM-Y *0.30*])

(External-Stimulus Object0142 [COMPONENT-PATTERN *Off*]
[SPATIAL-PATTERN *One*]
[MINIMUM-X *0.42*] [MAXIMUM-X *0.57*]
[MINIMUM-Y *0.00*] [MAXIMUM-Y *0.30*])

(External-Stimulus Object0143 [COMPONENT-PATTERN *On*]
[SPATIAL-PATTERN *One*]
[MINIMUM-X *0.63*] [MAXIMUM-X *0.78*]
[MINIMUM-Y *0.34*] [MAXIMUM-Y *0.64*])

Encoding the two *On*-lights yields a new higher-level stimulus pattern with a bounding box just big enough to contain the two *On*-lights; the *Off*-light is simply ignored. The COMPONENT-PATTERN remains *On*, and a new symbol is created to represent the SPATIAL-PATTERN. The object representing the pattern looks like:

(External-Stimulus Object0144 [COMPONENT-PATTERN *On*]
[SPATIAL-PATTERN *Spatial-Pattern-0145*]
[MINIMUM-X *0.21*] [MAXIMUM-X *0.78*]
[MINIMUM-Y *0.00*] [MAXIMUM-Y *0.64*])

The production that performs this encoding operation has the form:

Production Encode1:

*If* there is an **External-Stimulus** object
consisting of just one *On*-light
whose left edge is 0.21 (within 0.15), right edge is 0.36 (within 0.15),
top edge is 0.00 (within 0.30), bottom edge is 0.30 (within 0.30)
*and* there is an **External-Stimulus** object
consisting of just one *On*-Light
whose left edge is 0.63 (within 0.15), right edge is 0.78 (within 0.15),
top edge is 0.34 (within 0.30), bottom edge is 0.64 (within 0.30)
*and* there is *No* **External-Stimulus** object
consisting of *On*-lights in any spatial pattern
whose left edge is left of 0.63 (within 0.27)
*then* create a new **External-Stimulus** object
consisting of *On*-lights in configuration *Spatial-Pattern-0145*
whose left edge is 0.21, right edge is 0.78
top edge is 0.0, bottom edge is 0.64.

The first condition looks for an *On*-light bounded by [0.21, 0.36] horizontally, and [0.00, 0.30] vertically. The bounding box is matched by four two-sided real-number condition patterns. The lights may not always be positioned exactly as they were when the production was created, so the match is set up to succeed over a range of values (the interval of the real-number match). The sizes of the intervals are based on the notion that the accuracy required is proportional to the size of the pattern. The horizontal intervals are therefore set to

the width of the pattern ($0.36 - 0.21 = 0.15$), and the vertical intervals are set to the height of the pattern ($0.30 - 0.00 = 0.30$).

The second condition works identically to the first, with only the location of the light changed. The third condition insures that there are no intervening *On*-lights. This last condition is actually testing that no *On* pattern starts between the right edge of the first sub-pattern and the left edge of the second sub-pattern. That this works depends on the fact that the lights are being processed horizontally, and that there is no horizontal overlap between adjacent lights. Currently, this knowledge is built directly into the chunking mechanism; a situation which is tolerable when only one task is being explored, but intolerable in a more general mechanism.

The above example chunked two primitive patterns together to yield a higher-level pattern, but the same technique works if the subpatterns are higher-level patterns themselves, or even if there is a mixture. In the following example, a higher-level pattern is combined with a primitive pattern. Suppose the situation is the same as in the previous example, plus there is an additional *On*-light to the right. After the encoding production fires, working memory consists of the four objects mentioned above (three primitive ones, and one higher-level one), plus the following object for the extra light.

(**External-Stimulus** `Object0146` [COMPONENT-PATTERN *On*]
[SPATIAL-PATTERN *One*]
[MINIMUM-X *0.84*] [MAXIMUM-X *0.99*]
[MINIMUM-Y *0.68*] [MAXIMUM-Y *0.98*])

A higher-level pattern can be generated from this pattern and `Object0144`. The new pattern covers the entire bounding box for the four lights. The encoding production for this is:

Production Encode2:

*If* there is an **External-Stimulus** object
consisting of *On*-lights in configuration *Spatial-Pattern-0145*
whose left edge is 0.21 (within 0.57), right edge is 0.78 (within 0.57),
top edge is 0.00 (within 0.64), bottom edge is 0.64 (within 0.64)
*and* there is an **External-Stimulus** object
consisting of just one *On*-Light
whose left edge is 0.84 (within 0.15), right edge is 0.99 (within 0.15),
top edge is 0.68 (within 0.30), bottom edge is 0.98 (within 0.30)
*and* there is *No* **External-Stimulus** object
consisting of *On*-lights in any spatial pattern
whose left edge is left of 0.84 (within 0.06)
*then* create a new **External-Stimulus** object
consisting of *On*-lights in configuration *Spatial-Pattern-0147*
whose left edge is 0.21, right edge is 0.99
top edge is 0.0, bottom edge is 0.98.

As should be clear, this production is basically the same as production Encode1. The bounding boxes are appropriately changed, and the SPATIAL-PATTERN of one of the subpatterns is *Spatial-Pattern-0145*, the name for the higher-level pattern generated by production Encode1, and not *One* (signified in the productions by the phrase "consisting of just one *On*-light"). When production Encode2 fires, it creates a stimulus object of the following form.

(**External-Stimulus** `Object0148` [COMPONENT-PATTERN *On*]
[SPATIAL-PATTERN *Spatial-Pattern-0147*]
[MINIMUM-X *0.21*] [MAXIMUM-X *0.99*]
[MINIMUM-Y *0.00*] [MAXIMUM-Y *0.98*])

### 7.1.2. The decoding component

Decoding productions perform the inverse operation of encoding productions. When one matches to a higher-level pattern, it generates that pattern's two subpatterns. Because decoding must occur after the start of the **OneResponsePattern** Goal (after the bottleneck), it is defined on response patterns, rather than stimulus patterns. We assume that decoding occurs because the motor system only responds to primitive **External-Response** objects. When the response is specified by a higher-level pattern, it must be decoded down to its component primitives before the response can occur.

The entire set of decoding productions acts as a hierarchical decoding network for higher-level response patterns. Unlike encoding, decoding is initiated under goal-directed control. The **OneResponsePattern** goal's parameters describe a response pattern that is to be executed. From this description, the goal builds the appropriate **External-Response** object, and decoding begins. Decoding can't begin until the goal has built this object, but once it has begun, it continues to completion without further need of direction from the goal. Integrating the decoding component into the performance model is thus trivial; whenever an object representing a higher-level response pattern is generated, the appropriate decoding productions will fire. The one complication is that, as with encoding, decoding requires a variable number of cycles to complete. The problem of determining when decoding is done, is solved by the use of a second pseudo-clock (Section 7.1.1.2). In fact, this mechanism is inadequate for this purpose, but the problem does not affect the execution of the remainder of the model, so the current scheme is being employed until a better alternative is devised.

The following decoding production is the analogue of production Encode2 in Section 7.1.1.3. It has one condition that matches the higher-level response pattern corresponding to the stimulus pattern generated by production Encode2, and it has two actions which generate response patterns corresponding to the two stimulus subpatterns of production Encode2. One of the subpatterns is primitive, while the other one is a higher-level pattern that must be decoded further by another production.

Production Decode2:

*If* there is an **External-Response** object
consisting of *Press*-keys in configuration *Spatial-Pattern-0151*
whose left edge is 0.21 (within 0.78), right edge is 0.99 (within 0.78)
top edge is 0.0 (within 0.98), bottom edge is 0.98 (within 0.98)
*then* create a new **External-Response** object
consisting of *Press*-keys in configuration *Spatial-Pattern-0150*
whose left edge is 0.21, right edge is 0.78,
top edge is 0.00, bottom edge is 0.64
*and* create a new **External-Response** object
consisting of just one *Press*-key
whose left edge is 0.84, right edge is 0.99,
top edge is 0.68, bottom edge is 0.98.

### 7.1.3. The connection component

A connection production links a higher-level stimulus pattern with its appropriate higher-level response pattern. The entire set of connection productions defines the *stimulus-response mapping* for the task. This mapping must occur under goal direction so that the type of mapping can vary according to the task being performed. It would not be a very adaptive model if it were locked into always responding the same way to the same stimulus.

The connection productions need to be located before the encoding component and after the decoding component — between the end of the **OneStimulusPattern** goal and the start of the **OneResponsePattern** goal. They are situated in, and under the control of, the **OnePattern** goal. This goal already contains a general mechanism for mapping the description of a primitive stimulus pattern to the description of the appropriate primitive response pattern. These descriptions are local to the **OnePattern** goal, and are stored as attributes of the object representing the goal (Section 6.2.3).

The connection productions extend this existing mechanism so that higher-level patterns can also be mapped. Whether a connection production fires, or the initial mechanism executes, is completely determined by the SPATIAL-PATTERN of the stimulus pattern. If it is *One*, the initial mechanism is used, otherwise a connection production is required. Integration of the connection productions into the performance model is therefore straightforward. The following production connects a higher-level stimulus pattern with SPATIAL-PATTERN *Spatial-Pattern-0147* (Section 7.1.1.3), to the corresponding higher-level response pattern (Section 7.1.2).

Production Map-Spatial-Pattern-0147:

*If* there is a **OnePattern** goal whose STATUS is *MapOnePattern*
containing the description of a stimulus pattern
of *On*-lights in configuration *Spatial-Pattern-0147*
whose left edge is 0.21 (within 0.78), right edge is 0.99 (within 0.78)
top edge is 0.0 (within 0.98), bottom edge is 0.98 (within 0.98)
*then* add the description of a response pattern
consisting of *Press*-keys in configuration *Spatial-Pattern-0151*
whose left edge is 0.21, right edge is 0.99,
top edge is 0.00, bottom edge is 0.98

The key to making the proper connection is that the production matches to the unique SPATIAL-PATTERN specified by the stimulus pattern (*Spatial-Pattern-0147*), and generates the unique SPATIAL-PATTERN for the response pattern (*Spatial-Pattern-0151*). As an example, suppose working memory contains an object of the form:

(**OnePattern** `Object131` [STATUS *MapOnePattern*]
[STIMULUS-COMPONENT-PATTERN *On*]
[STIMULUS-SPATIAL-PATTERN *Spatial-Pattern-0147*]
[STIMULUS-MINIMUM-X *0.21*] [STIMULUS-MAXIMUM-X *0.99*]
[STIMULUS-MINIMUM-Y *0.00*] [STIMULUS-MAXIMUM-Y *0.98*])

The connection production would modify this element by adding the description of the corresponding response pattern. The object would then have the form:

(**OnePattern** `Object131` [STATUS *MapOnePattern*]
[STIMULUS-COMPONENT-PATTERN *On*]
[STIMULUS-SPATIAL-PATTERN *Spatial-Pattern-147*]
[STIMULUS-MINIMUM-X *0.21*] [STIMULUS-MAXIMUM-X *0.99*]
[STIMULUS-MINIMUM-Y *0.00*] [STIMULUS-MAXIMUM-Y *0.98*])
[RESPONSE-COMPONENT-PATTERN *Press*]
[RESPONSE-SPATIAL-PATTERN *Spatial-Pattern-151*]
[RESPONSE-MINIMUM-X *0.21*] [RESPONSE-MAXIMUM-X *0.99*]
[RESPONSE-MINIMUM-Y *0.00*] [RESPONSE-MAXIMUM-Y *0.98*])

## 7.2. The acquisition of chunks

Chunk acquisition is a task-independent, primitive capability of the architecture. The acquisition mechanism is therefore implemented as ***Lisp*** code, rather than as a set of productions within the architecture. The mechanism continually monitors the execution of the performance model, and acquires new chunks from the objects appearing in working memory. It accomplishes this by building productions for the three components of the chunk. There are two principal structural alternatives for this mechanism: (1) the components can be created all at once; or (2) they can be created independently. There are clear trade-offs involved.

With the all-at-once alternative, the components of a chunk are all created at the same time. The primary

advantage of this approach is simplicity in creating the connection component. In order to create a correct connection production, the corresponding stimulus and response SPATIAL-PATTERNs must be known. With the all-at-once alternative, the SPATIAL-PATTERNs are directly available because the connection production is created concurrently with the encoding and decoding productions. With the independent alternative, making this connection is more difficult. The connection production must determine the appropriate SPATIAL-PATTERNs, even though they are denoted by distinct symbols, and may not be in working memory at the time. This is difficult, but if possible, it does lead to two advantages over the all-at-once approach. First, it places only a small demand on the capacity of working memory. When the stimulus information is around, the encoding component can be created, and likewise with the decoding component. All of the information does not have to be active at once. Second, transfer of training is possible at a smaller grain size. With the all-at-once alternative, transfer of training occurs only when the entire chunk is usable in another task. With the independent alternative, individual encoding and decoding components can be shared, because a new connection production can be created during the transfer task that makes use of stimulus and response patterns from the training task.

Implementing the independent alternative looked hard enough that the all-at-once alternative was chosen for this initial attempt at building a chunking mechanism. Creating all of the components at once eliminates the problems of the independent alternative by forcing all of the information to be in working memory at the same time. This information exists within the instances of the **OnePattern** goal (Section 6.2.3). Each instance describes a stimulus pattern and its associated response pattern. Given two of these instances, we have all of the information required to create a chunk. Built into the current chunking mechanism is the knowledge that chunks are based on the data in these goal instances, and how patterns are encoded as attributes of the **OnePattern** objects.

Basing the acquisition of chunks on the information in **OnePattern** goal instances, rather than on the raw stimuli and responses, has the consequence of limiting chunk acquisition to only those patterns that are actually employed by the model during performance of the task. The potentially explosive number of possibilities for chunking is thus constrained to the relatively small set of patterns to which the subject actually attends. Many incidental patterns may be perceived in the process, but practice only improves performance on those components of the task actually performed.

Chunks are built out of the two most highly activated instances of the **OnePattern** goal in working memory, assuming that there are at least two present. These instances represent the two most recently processed patterns. Two of the architectural choices made in ***Xaps2*** were motivated by the need to have two instances of this goal simultaneously active.

- Competition among objects is limited to within types so that pursuance of other goals would not cause old instances of the **OnePattern** goal to disappear from working memory.

- The working-memory threshold is set at .0001 so that competition from the current instance of the **OnePattern** goal does not wipe out the previous instance before there is a chance to chunk them together. This is adequate for the current model, but will not be for cases where the patterns take longer to process. This limitation amounts to a reasonable restriction on the length of time over which the chunking process can combine two patterns.

In order to assure that the two most highly activated **OnePattern** instances are both from the same trial — we don't want cross-trial chunks — working memory is flushed between trials. This is a kludge intended to simulate the effects of inter-trial activity.

Once a chunk has been created, we want the model to use it when appropriate, but not to recreate it. If the model were continually recreating the same chunks, production memory would quickly fill up with useless information. This problem breaks down into two subproblems: within-trial duplications, and across-trial duplications. First, consider the problem of within-trial duplication. Suppose a chunk was just created from the two most highly activated **OnePattern** objects; what is to stop the system from continually recreating the same chunk as long as those two objects are the most activated? To avoid this, the chunking mechanism keeps track of the identifiers of the last two instances that it chunked together. It only creates a new chunk if the identifiers of the two most highly activated instances differ from the stored identifiers. This also is an ad hoc solution necessary until we understand better what the true constraint should be.

Across-trial duplications occur when a chunk is created during one trial, and then recreated when similar circumstances arise on a later trial. As currently constructed the model will never produce a duplicate of this type. If a chunk already exists that combines two patterns into a higher-level pattern, then the encoding component of the chunk assures that whenever those two patterns are perceived, the higher-level pattern will also be perceived. The higher-level pattern will be selected for processing instead of the two smaller ones, so there is no possibility of them ever again being the two most recently used (most highly activated) patterns. This does assume error free performance by the model, a condition that we have taken pains to assure holds.

## 8. The Results

In this section we present and analyze results from simulations of the complete model, consisting of the production system architecture, the performance model, and the chunking mechanism. These simulations demonstrate that the model works; the chunking theory can form the basis of a practice mechanism for production system architectures. In addition, these simulations provide a detailed look at the acquisition and use of chunks, and verify that the model does produce power-law practice curves. In Section 2.3.1 we showed that the chunking equation — an approximation of the full model — produces curves that are well matched by a power law. Now we can demonstrate it directly, though not analytically, for the exact model of one task.

### 8.1. The results of a simulation

The complete model has been run successfully on a specially selected sequence of nine trials for the left hand (five lights only). This sequence was devised especially to illustrate important aspects of the model. For each trial in this sequence, Table 8-1 shows the task to be performed, the chunks used, the chunks acquired, and the reaction time in number of production system cycles. Figure 8-1 presents an alternative organization of the chunks acquired during this sequence of trials — the chunk hierarchy. Each node in this hierarchy represents one chunk that was acquired. The node's children represent the two subchunks from which that chunk was created.

| Trial | Type | Chunks Used | Chunks Acquired | Cycles |
|---|---|---|---|---|
| 1 | o●●●● | -●--- --●-- ----● | -●●-- --●●● | 106 |
| 2 | ●o●●● | ●---- --●●● | ●o●●● | 75 |
| 3 | ●●o●● | ●---- -●--- | ●●--- | 72 |
| 4 | ●●●o● | ●●--- --●●● | ●●●●● | 74 |
| 5 | ●o●●● | ●o●●● | | 44 |
| 6 | ●●o●● | ●---- ---●- ----● | ●●●●- ---●● | 105 |
| 7 | ●●o●● | ●●●●- ----● | ●●o●● | 74 |
| 8 | ●●o●● | ●●o●● | | 44 |
| 9 | ●●●o● | ●●--- ---●● | ●●●●● | 75 |

**Table 8-1:** The nine trial sequence simulated by the model.
● is *On*, o is *Off*, and - is *don't care*.

At the most global level, these results demonstrate directly that the task was performed successfully, chunks were acquired, and they did improve the model's performance. Looking in more detail, first examine the relationship between the last column of Table 8-1, the number of cycles per trial, and the third column, the chunks used. We can see that the time to perform a trial is approximately given by:

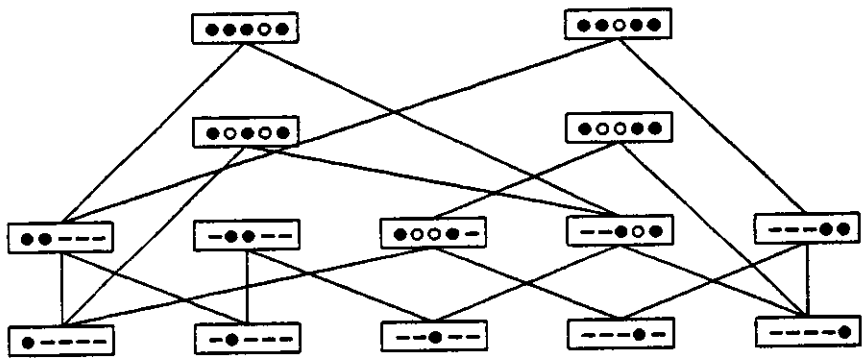


**Figure 8-1:** The tree of chunks created during the nine trial simulation.

$$NumberOfCycles = 13 + (31 \times NumberOfPatternsProcessed) \quad (9)$$

A three-pattern trial takes about 106 cycles (105-106 in the data), a two-pattern trial takes about 75 cycles (72-75 in the data), and a one-pattern trial takes 44 cycles (44 in the data).

A chunk is acquired for the first and second patterns used, the second and third patterns used, and so forth up to the number of patterns in the trial. The number of chunks acquired on a trial is therefore given by:

$$NumberOfChunksAcquired = NumberOfPatternsProcessed - 1 \quad (10)$$

The rate of acquisition of chunks is one every 31 cycles, once the constant overhead of 44 cycles per trial (13 plus the time to process the first pattern on the trial) has been removed, satisfying the learning assumption (Section 2.3).

This learning is demonstrably too fast. For the ten-light task environment, the entire task environment can be learned within $\log_2(10)$, between three and four, iterations through the task environment (at 1023 trials per iteration). This could be remedied in one of two ways. The first possibility is to propose that there are in fact more chunks to be learned than we have described. For example, the level at which primitive patterns are defined could be too high, or there may be other features of the environment that we are not capturing. The second alternative is that chunks are not learned at every opportunity. Gilmartin (1974) computed a rate of chunk acquisition of about one every eight to nine seconds — less than one chunk per trial in this task. Without speculating as to the cause of this slow down, we could model it by adding a parameter for the probability ( $< 1$) that a chunk is learned when the opportunity exists. We do not know which alternative is correct, but would not be surprised to find both of them implicated in the final solution.

One point clearly illustrated by this sequence of trials is that chunking is hierarchical, without having a strong notion of *level*. Chunks can be based on primitive patterns, higher-level patterns, or a mixture. The

sequence illustrates the following combinations: (1) the creation of chunks from primitive patterns (trials 1, 3, and 6); (2) the creation of chunks from higher-level patterns (trials 4 and 9); (3) the creation of chunks from one primitive pattern and one higher-level pattern (trials 2 and 7); and (4) the creation of no chunks (trials 5 and 8). The *Off*-lights in the chunks represent the regions in which no *On*-light should appear (Section 7.1.1.3).

Also illustrated is how the chunks created on one trial can be used on later trials. As one example, look at trials 6 through 8 in Table 8-1. All three trials employ the identical task, containing three *On*-lights. On trial 6, the three *On*-lights are processed serially (105 cycles), and two chunks are acquired for the two combinations of two successive *On*-lights. Notice that the two chunks share the middle *On*-light as a subpattern. On the following trial, trial 7, the first chunk created on trial 6 is used, taking care of the first two *On*-lights. All that is left is the third *On*-light, which is a primitive pattern. The time for trial 7 is 74 cycles, a savings of 31 over trial 6. During trial 7, a chunk is created that covers all three *On*-lights by combining the two patterns employed during the trial. On trial 8, only one pattern is required, and the trial takes only 44 cycles.

Chunks not only improve performance on trials that are exact repetitions of earlier trials, but they can also be transferred to trials that merely share a subpattern. Thorndike first described transfer along these lines: "A change in one function alters any other only in so far as the two functions have as factors identical elements." (Thorndike, 1913). Trials 1 and 2 illustrate this variety of transfer of training. Both trials have the third and fifth lights *On* and the fourth light *Off*, but differ in the first two lights. Nonetheless, the chunk created in the first trial is used to speed up the performance of the second trial. The same chunk is also reused in trial 4.

The complete model has also been run successfully on a sequence of twenty ten-light trials, with results comparable to those for the five-light sequence.

## 8.2. Simulated practice curves

The model is too costly computationally to run the long trial sequences required for the generation of practice curves. The execution time varies with the number of productions in the system — slowing down as chunks are added — but in the middle of the twenty trial sequence, the model took an average of 22 CPU minutes to process each pattern (approximately 31 production system cycles) on a *DecSystem 2060*. This deficiency is overcome through the use of a meta-simulation — a more abstract simulation of the simulation. The meta-simulation is faster than the simulation because it ignores the details of the performance system. It merely keeps track of the chunks that would be created by the model and the patterns that would be used

during performance. From this information, and Equation 9, it estimates the number of cycles that the production-system model would execute.

Via this meta-simulation, extensive practice curves have been generated. As a start, Figure 8-2 shows the practice curve generated by the meta-simulation for the 408 trial sequence used for subject 3 (Section 3). Comparing this curve with the curve for the human subject (Figure 3-3), we see a basic similarity, though the human's curve is steeper and has more variability.

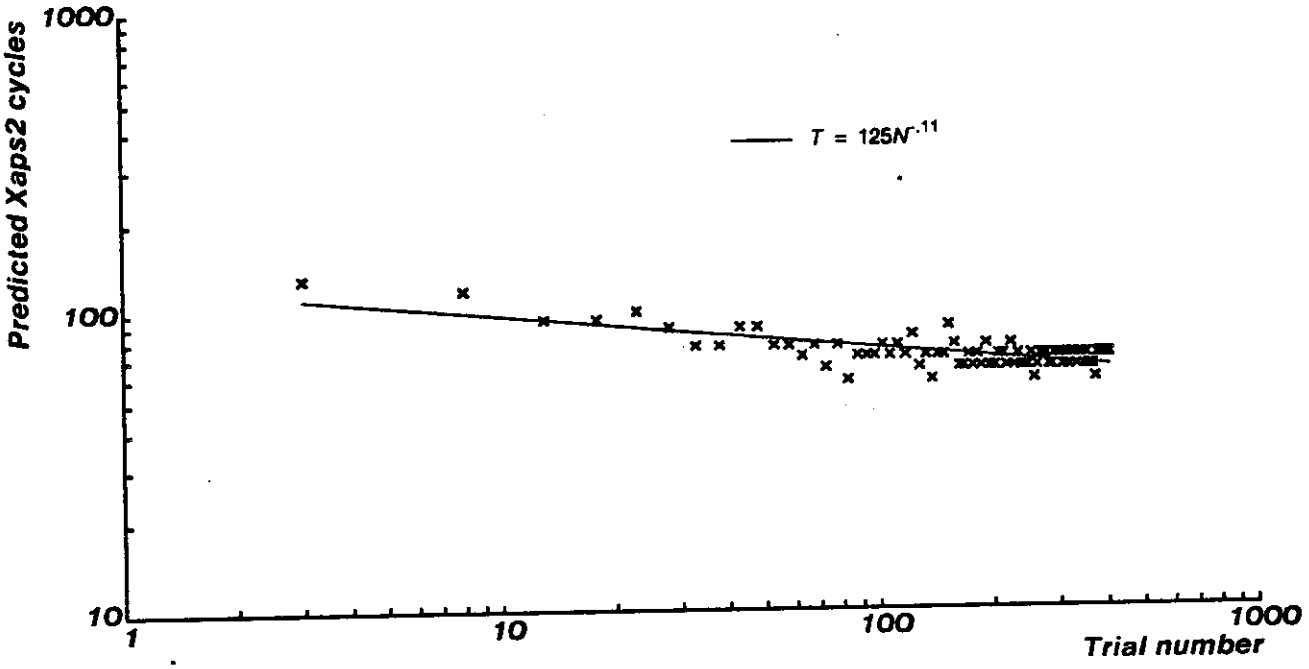



**Figure 8-2:** Practice curve predicted by the meta-simulation (log-log coordinates). The 408 trial sequence performed by Subject 3 (aggregated by five trials).

Seibel ran his subjects for 75 blocks of 1023 trials each (Seibel, 1963). To compare the model with this extensive data, the meta-simulator was run for the same number of trials. A single random permutation of the 1023 trials was processed 75 times by the meta-simulation. Figure 8-3 shows the practice curve generated by the meta-simulation for this sequence of trials. It is clear from this curve that creating a chunk at every opportunity leads to perfect performance much too rapidly — by the third block of trials.

A much better curve can be obtained by slowing down the rate of chunk acquisition, per the second suggestion in Section 8.1. We can make a quick, back-of-the-envelope calculation to find a reasonable value for the probability of acquiring a chunk, given the opportunity. To do this we will make three assumptions.

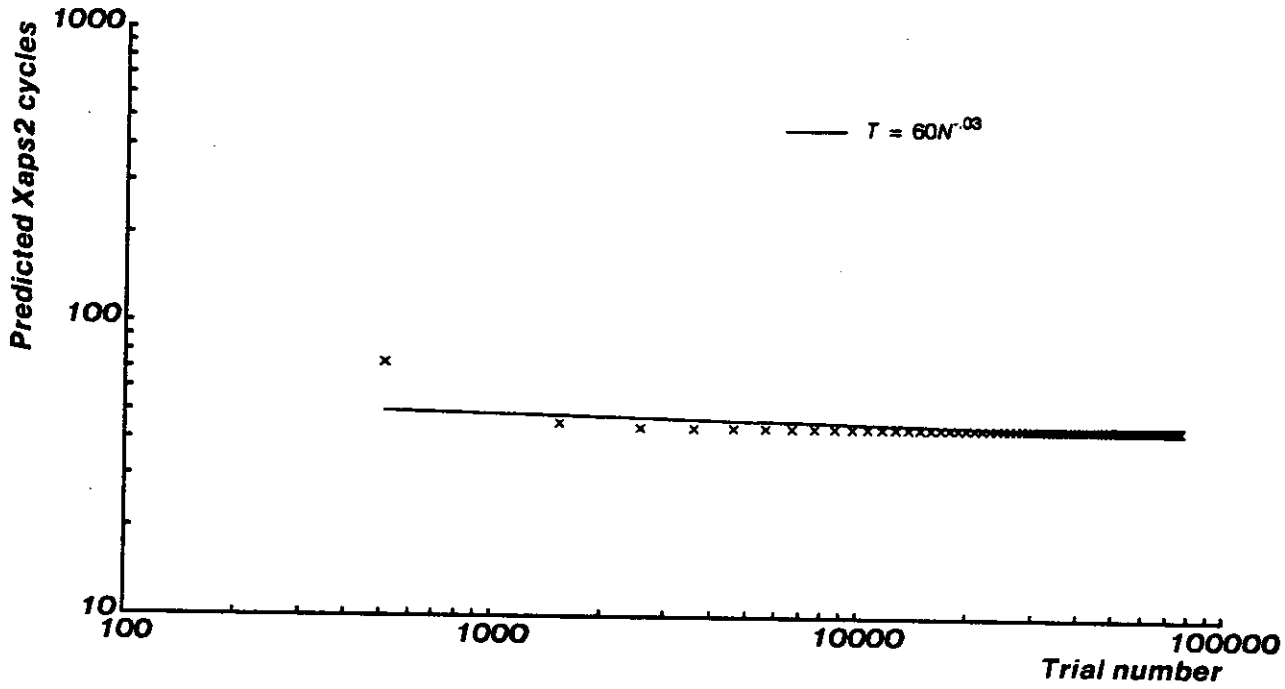



**Figure 8-3:** Practice curve predicted by the meta-simulation (log-log coordinates). Seventy five data points, each averaged over a block of 1023 trials.

- Assume that the model has the opportunity to acquire a chunk each time a pattern is processed, and that there is no overhead time.
- Assume that the time to process a pattern is in the range of times for a *simple reaction time* — 100 to 400 msec (Card, Moran, & Newell, In press).
- Assume that it takes 8-9 seconds to acquire a chunk (Gilmartin, 1974).

The probability ($p$) of acquiring a chunk is essentially the rate of chunking, as measured in chunks per pattern. This rate can be computed by dividing the time per pattern (0.1 - 0.4 seconds) by the time per chunk (8.0 - 9.0 seconds). Using the extreme values for the two parameters, we find that the probability should be in the interval [0.01, 0.05]. We have chosen to use one value in this interval — $p = 0.02$.

Figure 8-4 shows the results of a meta-simulation in which chunk acquisition is slowed down by this factor. This curve is linear in log-log coordinates over the entire range of trials ($r^2 = 0.993$). A slight wave in the points is still detectable, but the linearity is not significantly improved by resorting to the generalized power law ($r^2$ is still 0.993). We currently have no explanation for this phenomenon. We can only comment that the deviations are indeed small, and that similar waves appear to exist in the general power law fit to Seibel's data (Figure 2-2), though they are somewhat obscured by noise.

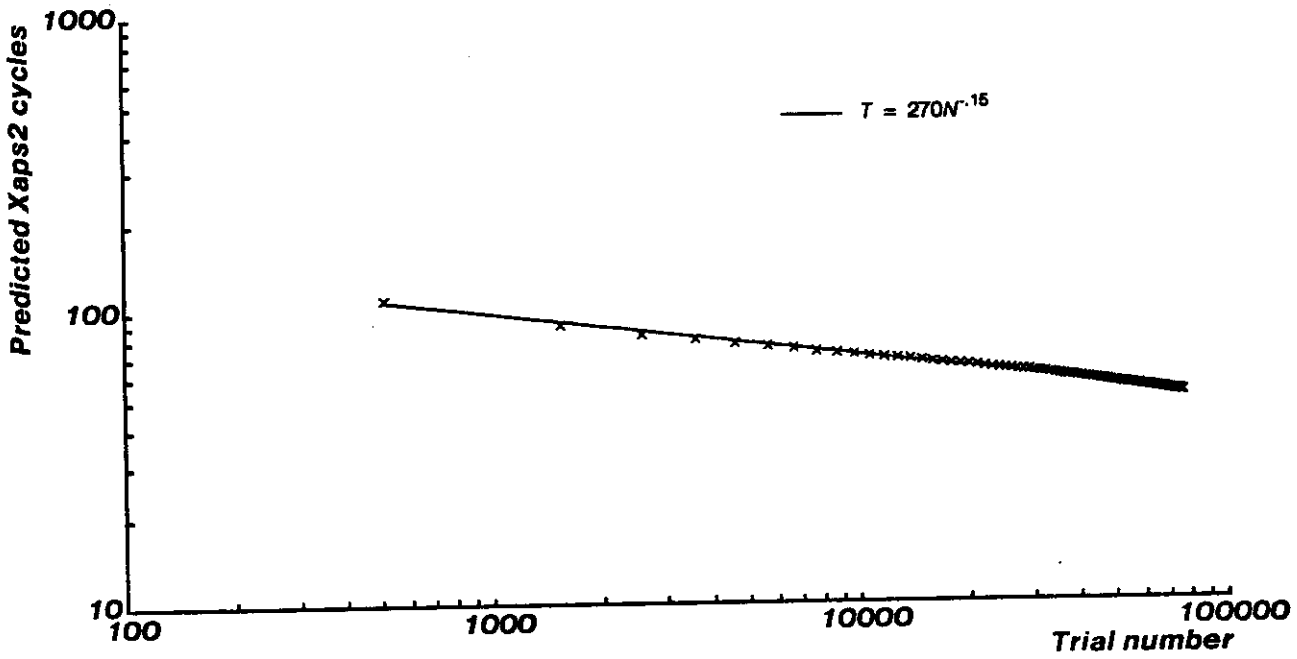



**Figure 8-4:** Practice curve predicted by the meta-simulation (log-log coordinates).
Seventy five data points, each averaged over a block of 1023 trials.
The probability of creating a chunk when there is an opportunity, is 0.02.

If a low probability of chunk acquisition is required in order to model adequately highly aggregated long sequences of trials (Figure 8-4), and a high probability is required for an adequate fit to less aggregated, short trial sequences (Figure 8-2), then there would be a major problem with the model. Fortunately, the one value of 0.02 is sufficient for both cases. Figure 8-5 shows the same 408 trial sequence as Figure 8-2, with the only difference being the reduced probability of chunk acquisition. Thus, given a reasonable value for $p$, the chunking model produces good power-law curves over both small and large trial ranges.

The most important way in which Figure 8-4 differs from the human data (Figure 2-1), is that the power ($\alpha$) of the power-law fit is lower for the meta-simulation — 0.15 for the meta-simulation versus 0.32 for the central linear portion of the subject's curve. One approach to resolving this discrepancy is to examine the meta-simulation for parameters that can be modified to produce larger powers. Modification of $p$, the one parameter mentioned so far, can cause small perturbations in $\alpha$, but is incapable of causing the large increase required. When $p$ was varied over [0.001, 1.0][12], $\alpha$ only varied over the range [0.03, 0.15].

[12] The range was sampled at 0.001, 0.01, 0.02, 0.1, and 1.0.

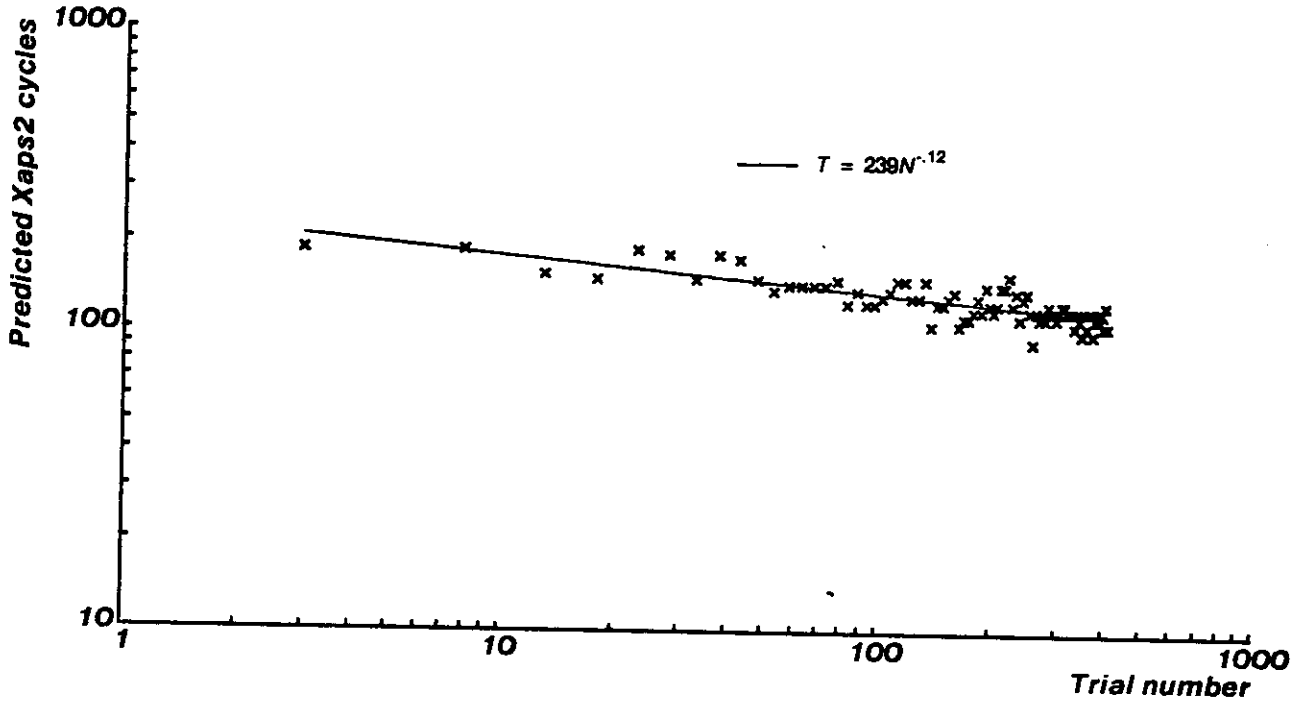



**Figure 8-5:** Practice curve predicted by the meta-simulation (log-log coordinates). The 408 trial sequence performed by Subject 3 (aggregated by five trials). The probability of creating a chunk when there is an opportunity, is 0.02.

One parameter that can effect $\alpha$, is the number of lights (and buttons) in the task environment. Increasing this number can significantly raise $\alpha$. With twenty lights and buttons[13], the meta-simulation produced a practice curve with an $\alpha$ of 0.26. For the shorter 408 trial sequence, an $\alpha$ of 0.16 was generated, compared with 0.17 for Subject 3 (Figure 3-3). While this manipulation yields good results, it is still true that those ten extra lights and buttons don't actually exist in the task environment. An alternative interpretation is required in which these ten virtual lights and buttons are thought of as modelling unspecified other features of the task environment (see Section 8.1).

Given the simulation results in this section, a rough estimate of the cycle time of the ***Xaps2*** production-system architecture can be computed. One method is to compute the mean response time for the human data; remove some portion of it, say half, as an estimate of the task time outside the scope of the model; and divide the remainder by the mean number of cycles per trial. The value varies with the number of lights used in the simulation (10 or 20) and whether a long simulation is being compared with the Seibel data (Figure

[13] Each ten-light trial, in a single random permutation of the 1023 trials, had an additional ten random lights appended to it, to yield 1023 twenty-light trials. This block of trials was then repeated 75 times.

2-1), or a short simulation is being compared to Subject 3 (Figure 3-3), but all four computations yield a value between 3 and 6 msec. The average value is 4 msec.

One cycle every 4 msecs is a very fast rate. The accepted value for production-system architectures is generally thought to be on the order of the *cognitive cycle time* — between 25 and 170 msec (Card, Moran, & Newell, In press). Note, however, that the model simulates the cognitive system at a smaller grain size than is normally done. The cognitive cycle is more appropriately identified with the complete processing of one pattern (one iteration through the **OnePattern** goal). If we ignore the implementation of the model's goal structure as productions, and just look at this level of goal-directed processing, the architecture looks remarkably like a conventional serial production system. During each cycle of this higher-level "production system" (a **OnePattern** goal), we *recognize* a single pattern (a **OneStimulusPattern** goal) and *act* accordingly (a **OneResponsePattern** goal) — approximately 31 cycles. The ***Xaps2*** cycle time of 3 to 6 msec per cycle yields a time estimate for this higher-level cycle of between 93 and 186 msec, with a mean of 124 msec. These times are well within the accepted range for the cognitive cycle time.

## 9. Conclusion

This paper has reported on an investigation into the implementation of the chunking theory of learning as a model of practice within a production-system architecture. Starting from the outlines of a theory, a working model capable of producing power-law practice curves has been produced. This model has been successfully simulated for one task — a 1023-choice reaction-time task.

During this research we have developed a novel highly-parallel production system architecture — ***Xaps2*** — combining both symbolic and activation notions of processing. The design of this architecture was driven by the needs of this work, but the resulting system is a fully general production-system architecture. Most importantly, it meets a set of constraints derived from an analysis of the chunking theory. These constraints must be met by any other architecture in which the chunking theory is embedded.

A performance model for the reaction-time task has been implemented as a set of productions within this architecture. Though the architecture provides parallel execution of productions, the control structure of the model is a serially-processed goal hierarchy — yielding a blend of serial and parallel processing. The goal hierarchy controls a loop through three tasks: (1) select a stimulus pattern to process; (2) map the stimulus pattern into an appropriate response pattern; and (3) execute the response pattern. Two of these tasks, 1 and 3, require the model to communicate with the outside world. The required stimulus and response interfaces are modelled as two-dimensional euclidean spaces of patterns. The model perceives patterns in the stimulus space and produces patterns in the response space. As with the production system architecture, the designs of the control structure and interfaces have been driven by the needs of this work. A second look shows that there is very little actual task dependence in these designs. The control structure, or a slightly more general variant, works for a large class of reaction-time tasks.

To this model is added the chunking mechanism. Chunks are acquired from pairs of patterns dealt with by the performance model. Each chunk is composed of a triple of productions: (1) an encoding production that combines a pair of stimulus patterns into a more complex pattern; (2) a decoding production which decomposes a complex response pattern into its simpler components; and (3) a connection production which links the complex stimulus pattern with the complex response pattern. Chunks improve the model's performance by reducing the number of times the system must execute the control loop. Both simulations and meta-simulations (simulations of the simulations) of the model have been run. The result is that chunking can improve performance, and it does so according to a power-law function of the number of trials.

The results of this investigation have been promising, but there is much work still to be done. One open question is whether these results will hold up for other tasks. As long as the task can be modelled within the control structure described in this article, power-law learning by chunking is to be expected. For radically

different tasks, the answer is less certain. To investigate this, the scope of the model needs to be extended to a wider class of tasks.

A number of aspects of the model need improvement as well. The production system architecture needs to be better understood, especially in relation to the chunking theory and the task models. Oversimplifications in the implementation of the chunking theory — such as allowing only pairwise chunking — need to be replaced by more general assumptions. In addition, a number of ad hoc decisions and mechanisms need to be replaced by more well reasoned and supported alternatives.

## References

Anderson, J. R. *Language, Memory, and thought.* Hillsdale, N.J.: Lawrence Erlbaum Associates, 1976.

Anderson, J. A. Neural models with cognitive implications. In D. LaBerge & S. J. Samuels (Ed.), *Basic Processes in Reading.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1977.

Anderson, J. R. Acquisition of cognitive skill. *Psychological Review,* 1982, *89,* 369-406.

Anderson, J. A., & Hinton, G. E. Models of information processing in the brain. In G. E. Hinton & J. A. Anderson (Ed.), *Parallel Models of Associative Memory.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1981.

Bower, G. H. & Winzenz, D. Group structure, coding, and memory for digit series. *Experimental Psychology Mongraph,* 1969, *80,* 1-17. (May, Pt. 2).

Card, S. K., Moran, T. P, & Newell, A. *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates, In press.

Chase, W. G. & Simon, H. A. Perception in chess. *Cognitive Psychology,* 1973, *4,* 55-81.

DeGroot, A. D. *Thought and Choice in Chess.* The Hague: Mouton, 1965.

Evans, T. G. A program for the solution of geometric-analogy intelligence test questions. In M. Minsky (Ed.), *Semantic Information Processing.* Cambridge, Mass.: MIT Press, 1968.

Forgy, C. & McDermott, J. *The Ops2 Reference Manual.* Pittsburgh, Pa.: Department of Computer Science, Carnegie-Mellon University, 1977. IPS Note #77-50.

Gilmartin, K. J. *An Information Processing Model of Short-Term Memory.* Doctoral dissertation, Carnegie-Mellon University, 1974.

Johnson, N. F. Organization and the concept of a memory code. In Melton, A. W. & Martin, E (Eds.), *Coding Processes in Human Memory.* Washington, D.C.: Winston, 1972.

Joshi, A. K. Some extensions of a system for inference on partial information. In D. A. Waterman & F. Hayes-Roth (Ed.), *Pattern-Directed Inference Systems.* New York: Academic Press, 1978.

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

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

Mitchell, T. M., Utgofff, P. E., Nudel, B., & Banerji, R. Learning problem-solving heuristics through practice. In *Proceedings of the Seventh IJCAI.*, 1981.

Moran, T. P. *Compiling cognitive skill* (AIP Memo 150). Xerox PARC, 1980.

Neisser, U., Novick, R., Lazar, R. Searching for ten targets simultaneously. *Perceptual and Motor Skills,* 1963, *17,* 427-432.

Neves, D. M. & Anderson, J. R. Knowledge compilation: Mechanisms for the automatization of cognitive skills. In Anderson, J. R. (Ed.), *Cognitive Skills and their Acquisition.* Hillsdale, NJ: Erlbaum, 1981.

Newell, A. Production systems: Models of control structures. In Chase, W. G. (Ed.), *Visual Information Processing.* New York: Academic Press, 1973.

Newell, A. Harpy, production systems and human cognition. In Cole, R. (Ed.), *Perception and Production of Fluent Speech.* Hillsdale, NJ: Erlbaum, 1980. (Also available as CMU CSD Technical Report, Sep 1978).

Newell, A. & Rosenbloom, P. S. Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive Skills and Their Acquisition.* Hillsdale, NJ: Erlbaum, 1981.

Newell, A. & Simon, H. A. *Human Problem Solving.* Englewood Cliffs: Prentice-Hall, 1972.

Norman, D. A. Categorization of action slips. *Psychological Review,* 1981, *88,* 1-15.

Rosenbloom, P. S. The XAPS Reference Manual. 1979.

Rosenbloom, P. S., & Newell, A. Learning by chunking: Summary of a task and a model. In *Proceedings of AAAI-82, National Conference on Artificial Intelligence.* American Association for Artificial Intelligence, 1982.

Rumelhart, D. E., & McClelland, J. L. An interactive activation model of context effects in letter perception: Part 2. The contextual enhancement effect and some tests and extensions of the model. *Psychological Review,* 1982, *89,* 60-94.

Rumelhart, D. E., & Norman, D. A. Simulating a skilled typist: A study of skilled cognitive-motor performance. *Cognitive Science,* 1982, *6,* 1-36.

Seibel, R. Discrimination reaction time for a 1,023-alternative task. *Journal of Experimental Psychology,* 1963, *66*(3), 215-226.

Shiffrin, R. M. & Schneider, W. Controlled and automatic human information processing: II. Perceptual learning, automatic attending, and a general theory. *Psychological Review,* 1977, *84,* 127-190.

Snoddy, G. S. Learning and stability. *Journal of Applied Psychology,* 1926, *10,* 1-36.

Thacker, C. P., McCreight, E. M., Lampson, B. W., Sproull, R. F., & Boggs, D. R. Alto: a personal computer. In D. P. Sieworek, C. G. Bell, & A. Newell (Ed.), *Computer Structures: Principles and Examples.* New York: McGraw Hill, 1982.

Thibadeau, R., Just, M. A., & Carpenter, P. A. A model of the time course and content of reading. *Cognitive Science,* 1982, *6,* 157-203.

Thorndike, E. L. *Educational Psychology. II: The Psychology of Learning.* New York: Bureau of Publications, Teachers College, Columbia University, 1913.

Uhr, L., & Vossler, C. A pattern-recognition program that generates, evaluates, and adjusts its own operators. In E. Feigenbaum & J. Feldman (Ed.), *Computers and Thought.* New York: McGraw-Hill, 1963.

VanLehn, K. *On the representation of procedures in repair theory* (Tech. Rep. CIS-16). Xerox PARC, October 1981.

Welford, A. T. Learning curves for sensory-motor performance. In *Proceedings of the Human Factors Society -- 25th Annual Meeting.*, 1981.