Newell, A. Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), *Attention and Performance VIII*. Hillsdale, NJ: Erlbaum, 1980.

Newell, A. & Rosenbloom, P. Mechanisms of skill acquisition and the law of practice. In Anderson, J. A. (Ed.), *Learning and Cognition*. Hillsdale, NJ: Erlbaum, 1981.

Newell, A. & Simon, H. A. GPS, a program that simulates human thought. In Feigenbaum, E. A. & Feldman, J. (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963.

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

Newell, A. & Simon, H. A. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 1976, *19*(3), 113-126.

Newell, A., McDermott, J. & Forgy, C. L. *Artifical Intelligence: A self-paced introductory course* (Tech. Rep.). Computer Science Department, Carnegie-Mellon University, September 1977.

Nilsson, N. *Problem-solving Methods in Artificial Intelligence*. New York: McGraw-Hill, 1971.

Nilsson, N. *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga, 1980.

Polya, G. *How to Solve It*. Princeton, NJ: Princeton University Press, 1945.

Polya, G. *Mathematical Discovery, 2 vols.*. New York: Wiley, 1962.

Rosenbloom, P. S. & Newell, A. *Learning by chunking: A production-system model of practice* (Tech. Rep.). Computer Science Department, Carnegie-Mellon University, Oct 1982.

Rulifson, J. F., Derksen, J. A. & Waldinger, R. J. *QA4: A procedural calculus for intuitive reasoning* (Tech. Rep.). Stanford Research Institute Artificial Intelligence Center, 1972.

Sacerdoti, E. D. *A Structure for Plans and Behavior*. New York: Elsevier, 1977.

Wason, P. C. & Johnson-Laird, P. N. *Psychology of Reasoning: Structure and content*. Cambridge, MA: Harvard, 1972.

Waterman, D. A. & Hayes-Roth, F., (Eds.). *Pattern Directed Inference Systems*. New York: Academic Press, 1978.

Winston, P. *Artificial Intelligence*. Reading, MA: Addison-Wesley, 1977.

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.

CHAPTER 7

The Chunking of Goal Hierarchies: A Generalized Model of Practice

P. S. Rosenbloom, Stanford University, and A. Newell, Carnegie Mellon University

Abstract

This chapter describes recent advances in the specification and implementation of a model of practice. In previous work the authors showed that there is a ubiquitous regularity underlying human practice, referred to as the *power law of practice*. They also developed an abstract model of practice, called the *chunking theory of learning*. This previous work established the feasibility of the chunking theory for a single 1023-choice reaction-time task, but the implementation was specific to that one task. In the current work a modified formulation of the chunking theory is developed that allows a more general implementation. In this formulation, task algorithms are expressed in terms of hierarchical goal structures. These algorithms are simulated within a goal-based production-system architecture designed for this purpose. *Chunking* occurs during task performance in terms of the parameters and results of the goals experienced. It improves the performance of the system by gradually reducing the need to decompose goals into their subgoals. This model has been successfully applied to the task employed in the previous work and to a set of stimulus-response compatibility tasks.

10.1 INTRODUCTION

How can systems—both natural and artificial—improve their own performance? At least for natural systems (people, for example), we know that *practice* is effective. A system is engaged in practice when it repeatedly performs one task or a set of similar tasks. Recently, Newell and Rosenbloom (1981) brought together the evidence that there is a ubiquitous law—the *power law of practice*—that characterizes the improvements in human performance during practice. The law states that when human performance is measured in terms of the time needed to perform a task, it improves as a power-law function of the number of times the task has been performed (called the *trial number*). This result holds over the entire domain of human performance, including both purely perceptual tasks, such as target detection (Neisser, Novick, and Lazar, 1963), and purely cognitive tasks, such as supplying justifications for geometric proofs (Neves and Anderson, 1981).

The ubiquity of the power law of practice suggests that it may reflect something in the underlying *cognitive architecture.* The nature of the architecture is of fundamental importance for both artificial intelligence and psychology (in fact, for all of cognitive science; see Newell, 1973; Anderson, 1983a). It provides the control structure within which thought occurs, determining which computations are easy and inexpensive as well as what errors will be made and when. Two important ingredients in the recent success of expert systems come from fundamental work on the cognitive architecture; specifically, the development of *production systems* (Newell and Simon, 1972; Newell, 1973) and *goal-structured problem solving* (Ernst and Newell, 1969; Newell and Simon, 1972). This chapter discusses recent efforts to take advantage of the power law of practice by using it as a generator for a general production-system practice mechanism. This is a highly constrained task because of the paucity of plausible practice models that can produce power-law practice curves (Newell and Rosenbloom, 1981).

As a beginning, Newell and Rosenbloom (1981) developed an abstract model of practice based on the concept of *chunking*—a concept already established to be ubiquitous in human performance—and derived from it a practice equation capable of closely mimicking a power law. It was hypothesized that this model formed the basis for the performance improvements brought about by practice. In sections 10.2 and 10.3 this work on the power law and the abstract formulation of the chunking theory is briefly summarized.

Rosenbloom and Newell (1982a, 1982b) took the analysis one step further by showing how the *chunking theory of learning* could be implemented for a single psychological task within a highly parallel, activation-based production system called XAPS2. This work established more securely that the theory is a viable model of human practice by showing how it could actually be applied to a task to produce power-law practice curves. By producing a working system, it also established the theory's viability as a practice mechanism for artificial systems.

The principal weakness of the work done up to that point was the heavy task dependence of the implementation. Both the representation used for describing the task to be performed and the chunking mechanism itself had built into them knowledge about the specific task and how it should be performed. The work reported here is focused on the removal of this weakness by the development of generalized, task-independent models of performance and practice.

This generalization process has been driven by the use of a set of tasks that fall within a neighborhood around the task previously modeled. That task sits within an experimental domain widely used in psychology, the domain of *reaction-time* tasks (Woodworth and Schlosberg, 1954). Reaction-time tasks involve the presentation to a subject of stimulus display—such as an array of lights, a string of characters on a computer terminal, or a spoken word—for which a specific "correct" response is expected. The response may be spoken, manual (such as pressing a button, pushing a lever, or typing), or something quite different. From the subject's reaction time—the time it takes to make the response—and error rate, it is possible to draw conclusions about the nature of the subject's cognitive processing.

The particular task, as performed by Seibel (1963), was a 1023-choice reaction-time task. It involved ten lights (strung out horizontally) and ten buttons, with each button right below a light. On each trial of the task some of the lights came on while the rest remained off. The subject's task was to respond as rapidly as possible by pressing the buttons corresponding to the lights that were on. There were $2^{10} - 1$, or 1023, possible situations with which the subject had to deal (excluding the one in which all ten lights were off). Rosenbloom (1983) showed that a general task representation based on the concept of *goal hierarchies* (discussed in section 10.4) could be developed for the performance of this task.

In a goal hierarchy, the root node expresses a desire to do a task. At each level further down in the hierarchy, the goals at the level above are decomposed into a set of smaller goals to be achieved. Decomposition continues until the goals at some level can be achieved directly. This is a common control structure for the kinds of complex problem-solving systems found in artificial intelligence, but this is the first time they have been applied to the domain of reaction-time tasks.

Goal hierarchies also provided the basis for models of a set of related reaction-time tasks known as *stimulus-response compatibility* tasks. These tasks involve fixed sets of stimuli and responses and a mapping between them that is manipulated. The main phenomenon is that more complex and/or "counterintuitive" relationships between stimuli and responses lead to longer reaction times and more error. A model of stimulus-response compatibility based on goal hierarchies provides excellent fits to the human reaction-time data (Rosenbloom, 1983).

The generalized practice mechanism (described in section 10.5) is grounded in this goal-based representation of task performance. It resembles a form of store-versus-compute trade-off, in which composite structures (chunks) are created that relate patterns of goal parameters to patterns of goal results.

These chunking and goal-processing mechanisms are evaluated by implementing them as part of the architecture of the XAPS3 production system. The XAPS3 architecture is an evolutionary development from the XAPS2 architecture. Only those changes required by the needs of chunking and goal processing have been made. This architecture is described in section 10.6. From this implementation simulated practice results have been generated and analyzed for the Seibel and compatibility experiments (section 10.7).[1]

Following the analysis of the model, this work is placed in perspective by relating the chunking theory to previous work on learning mechanisms (section 10.8). The theory stakes out an intermediary position among four previously disparate mechanisms, bringing out an unexpected commonality among them.

Before concluding and summarizing (section 10.10), some final comments are presented on ways in which the scope of the chunking theory can be expanded to cover more than just the speeding up of existing algorithms (section 10.9). Specifically, the authors describe a way in which chunking might be led to perform other varieties of learning, such as generalization, discrimination, and method acquisition.

10.2 THE POWER LAW OF PRACTICE

Performance improves with practice. More precisely, the time needed to perform a task decreases as a power-law function of the number of times the task has been performed. This basic law, the power law of practice, has been known since Snoddy (1926). This law was originally recognized in the domain of motor skills, but it has recently become clear that it holds over a much wider range of human tasks, possibly extending to the full range of human performance. Newell and Rosenbloom (1981) brought together the evidence for this law from tasks involving perceptual-motor skills (Snoddy, 1926; Crossman, 1959), perception (Kolers, 1975; Neisser, Novick, and Lazar, 1963), motor behavior (Card, English, and Burr, 1978), elementary decisions (Seibel, 1963), memory (Anderson, 1980), routine cognitive skill (Moran, 1980), and problem solving (Neves and Anderson, 1981; Newell and Rosenbloom, 1981).

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

[1] A more comprehensive presentation and discussion of these results can be found in Rosenbloom (1983).

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):

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

As shown by the following log transform of Equation 1, power-law functions plot as straight lines on log-log paper:

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

Figure 10-1 shows the practice curve from one subject in Kolers' study (1975) of reading inverted texts—each line of text on the page was turned upside down—as plotted on log-log paper. The solid line represents the power-law fit to this data. Its linearity is clear ($r^2 = 0.932$).

Many practice curves are linear (in log-log coordinates) over much of their range but show a flattening at their 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 can be greater than zero. In general, there is a nonzero minimum bound on performance time, determined by basic physiological limitations 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

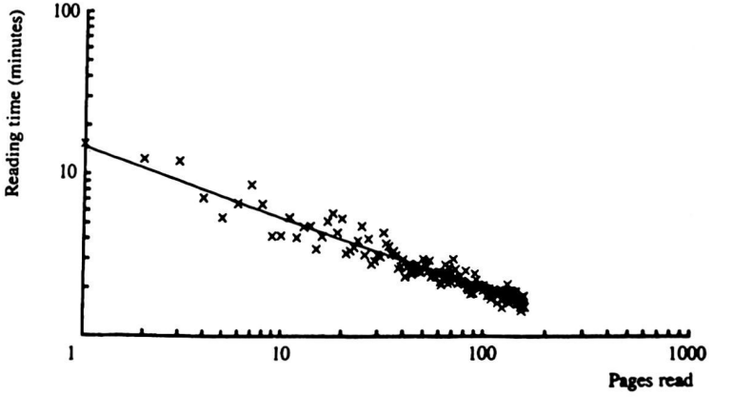


Figure 10-1: Learning to read inverted text (log-log coordinates). Plotted from the original data for Subject HA (Kolers, 1975).

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 10-2 shows a practice curve from the Seibel task (Seibel, 1963), as fit by a generalized power-law function (each data point represents the mean reaction time over a block of 1023 trials). This curve, which shows flattening at both ends when plotted as a simple power law, is now linear over the whole range of trials. As stated earlier, similar fits are found across all dimensions of human performance. 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.

10.3 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

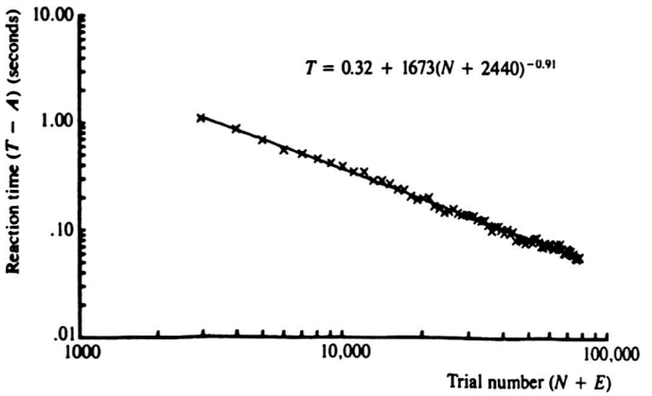


Figure 10-2: Optimal general power-law fit to the Seibel data (log-log coordinates).

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*, that 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, and so on) rooted in a set of preexisting *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.

The existence of chunks does not need to be justified solely on the basis of the practice curves. Chunks stand on their own as a thoroughly documented component of human performance (Miller, 1956; DeGroot, 1965; Bower and Winzenz, 1969; Johnson, 1972; Chase and Simon, 1973; Chase and Ericsson, 1981). The traditional view of chunks is that they are data structures representing a combination of several items. For example, in one set of classic experiments, Bower and colleagues (Bower, 1972; Bower and Springston, 1970; Bower and Winzenz, 1969) showed that recall of strings of numbers or letters is strongly influenced by the segmentation of the string. If the segmentation corresponds to a previously learned grouping of the items (for example, FBI-PHD-TWA-IBM), performance is better than if no such relation obtains (FB-IPH-DTW-AIB-M). These results were interpreted as evidence for segmentation-guided chunking of familiar strings. By replacing a string of several letters with a single chunk, the subject's memory load is reduced, allowing more letters to be remembered. At recall time the chunks are decoded to yield the original items to be recalled.

The chunking theory of learning proposes two modifications to this classical view. The first change is the assumption that there is not a single symbol (chunk) to which the items are encoded and from which they can later be decoded. As a simple example, the process of reading the string IBM out loud is more than just the encoding of the three letters into a chunk, followed by the subsequent decoding to the three letters. What needs to be decoded is not the visual representation of IBM, but the articulatory representation—allowing the subject to *say* "IBM."

Based on this consideration, the chunking theory assumes that there are two symbols for each chunk—a *stimulus* symbol and a *response* symbol. The process of using a chunk consists of encoding the stimulus items to the stimulus symbol (a many-one mapping), mapping the stimulus symbol to the response symbol (a one-one mapping), and decoding the response symbol to the response items (a one-many mapping) (see figure 10-3). Encoding and decoding are fast parallel hierarchical processes, and the mapping serves as a (serial) point of control at which the choice of response can be made. Acquisition of a chunk speeds up performance by reducing the number of mappings to be performed [illegible] the example in figure 10-3, be[illegible] the chunk

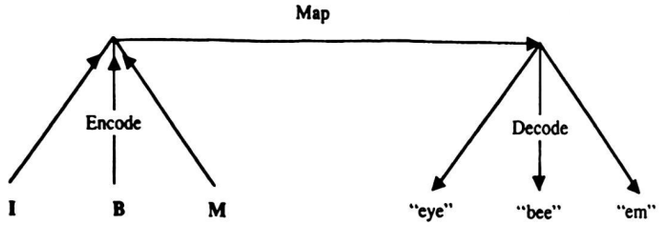


Figure 10-3: A three-part chunk for the articulation of the visual string "IBM."

is acquired, three mappings (one for each letter) are required. After acquisition of the chunk, one mapping suffices.

The second difference between this proposal and the classical view of chunking is the idea that the chunk consists of the three processes (encoding, mapping, and decoding), not just the symbols. This is really just a shift of emphasis; chunks are viewed as the processes rather than just as the results of the processes.

10.4 GOAL-STRUCTURED PERFORMANCE MODELS

In its current formulation, chunking explains how performance on a task can be sped up with practice. It does not explain how the organism first learns to do the task (but see the discussion in section 10.9). Consequently, each task simulation must be initialized with a performance model for the task. What is needed—and is provided by the notion of a *goal hierarchy*—is a general, task-independent representation for these performance models. Goal hierarchies are frequently found in artificial intelligence systems, but they have not previously been employed in the modeling of the kinds of reaction-time tasks dealt with here.

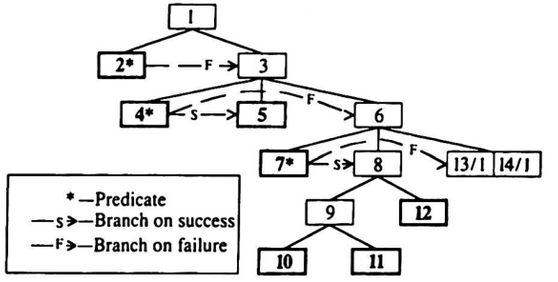
Goal hierarchies are built out of *goals;* each goal is a data structure representing a desired state of affairs. A goal is not a procedure for bringing about that state; it is only a description of the state. In order for the goal state to be brought about, there must be a *method* associated with the goal. The method could be a rigid algorithm, or it could be one of the more flexible *weak methods* (Newell, 1969), such as means-ends analysis (Ernst and Newell, 1969) or heuristic search (Nilsson, 1971). In the discussion that follows, the properly distinct notions of goal and method will be conflated together into a single active concept for the sake of convenience. These conflated "goals" are active processes that take a set of *parameters* and return a set of *results*.

When a goal can be decomposed into a set of simpler goals (Nilsson, 1971) and those goals can be decomposed even further, a goal hierarchy results. In its simplest form, as an AND hierarchy, a goal is successful if all of its subgoals are successful. The structure to be described here more closely resembles an AND/OR hierarchy, in which some goals succeed only if all of their subgoals succeed, and others succeed if

any one of their subgoals succeed. A *terminal* goal is reached when the goal can be fulfilled directly, without the need for further decomposition.

We use a *depth-first* strategy for processing a goal hierarchy, in which the most recently generated (the deepest) goal is always selected as the next goal to process. With a depth-first paradigm there is always exactly one goal being actively worked on at any point in time. We will refer to this goal as the *active* or *current* goal. When a subgoal becomes the active goal, the parent goal of that subgoal is *suspended* until control is returned to it by completion of the subgoal, at which point it again becomes the current goal. On completion, the subgoal will have either *succeeded* or *failed.* The *control stack* specifies the location in the hierarchy at which control currently resides. It does this by maintaining the path from the root goal of the hierarchy to the current goal. This path consists of the active goal and all of its suspended ancestors.

Figure 10-4 shows two different representations of a goal hierarchy for the Seibel (1963) 1023-choice reaction-time task. At the top of the figure, the hierarchy is shown as a tree structure. At the bottom of the figure, the goals are shown in their depth-first processing order. In both representations the boldface goals are the



1. Do-Lights-If-Any (Min-X, Max-X)
  2. **No-Light-On?** (Min-X, Max-X)
  IF-FAILED *No-Light-On?* THEN
    3. Do-Lights (Min-X, Max-X)
      4. **No-Light-Off?** (Min-X, Max-X)
      IF-SUCCEEDED *No-Light-Off?* THEN
        5. **Do-Press-All-Buttons** (Min-X, Max-X)
      IF-FAILED *No-Light-Off?* THEN
        6. Do-Off-And-On (Min-X, Max-X)
          7. **One-Light-On?** (Min-X, Max-X)
          IF-SUCCEEDED *One-Light-On?* THEN
            8. Press-Button-Under-On-Light (Min-X, Max-X)
              9. Get-On-Light-X (Min-X, Max-X)
                10. **Get-On-Light-Stimulus** (Min-X, Max-X)
                11. **Get-Stimulus-X**
              12. **Press-Button-At-X**
          IF-FAILED *One-Light-On?* THEN
            13. Do-Lights-If-Any (Min-X, [Min-X + Max-X]/2)
            14. Do-Lights-If-Any ([Min-X + Max-X]/2, Max-X)

Figure 10-4: Goal hierarchy for the Seibel (1963) task.

terminals—those goals that can be fulfilled directly. The labeled arrows (either S or F) represent *branch* points in the hierarchy. With simple depth-first processing, the children of a node are processed in a strict left-to-right fashion. In this work, this style of control has been augmented by allowing the left-to-right processing to be conditioned on the success or failure of the previous child (actually, of any previous goal for which this information is still available). For example, goal 2 (**No-Light-On?**) tests whether there are any lights on within a specific region of the display of lights. Only if this goal fails should goal 3 (Do-Lights) be attempted. If goal 2 succeeds, then there are no lights on that need to be processed and the parent goal (goal 1: Do-Lights-If-Any) can be terminated successfully. Those goals that do not return a result are called *predicates* (denoted by an asterisk in figure 10-4) and are used to test the truth of various conditions. It is only the success or failure of predicates that matters (as the basis for a branch). If the predicate succeeds, then the condition is true. If it fails, the reason could be either that the condition is false or that something went wrong during the attempt.

The goal structure in figure 10-4 is based on a recursive divide-and-conquer algorithm in which the stimulus display is broken up into smaller and smaller horizontal segments until *manageable* pieces are generated. There are three types of horizontal segments that have been defined as manageable. The first type of manageable segment is one in which no lights are on. Such segments require no explicit processing, so the goal just returns with success. The opposite of the first type of segment—one in which no lights are off—is also manageable. For such a segment, the system generates a single response specifying that a *press* action must occur in the entire region defined by the segment (using the **Do-Press-All-Buttons** goal). Specifying a single button press is actually a special case of this, in which the region is just large enough to contain the one button. Allowing multi-on-light segments to be manageable implies that sequences of adjacent on-lights can be pressed simultaneously even before chunking has begun. Such behavior is seen very early in the trial sequence for some subjects. The remaining manageable segments are those that contain exactly one light on. These segments are processed (using the Press-Button-Under-On-Light goal) by finding the location of that light and generating a button press at that location. If a generated segment does not meet any of these three criteria, it is unmanageable and is split into two smaller segments.

The recursive aspect of the algorithm implies that many different instances of each goal will be simultaneously represented in the system, although at most one can actually be active. It is necessary to keep track of which goal instance is relevant to which segment of the stimulus display, so the segment (in terms of its minimum and maximum X values) is an *explicit* parameter to the goals. In addition to the explicit parameters, a goal can have *implicitly* defined parameters. Any object existing before the activation of the goal (i.e., as part of the goal's *initial state*) that is examined during the processing of the goal—such as a stimulus light—is an implicit parameter of the goal. Implicit parameters are dynamically determined by the actual processing of the goal.

A second implication of the recursion in this hierarchy is that the system does not start off with a constant built-in goal hierarchy. Instead, it has a generator of goal hierarchies. That chunking works on such a structure is important for any claims about the potential generality of the mechanism (for more on this, see section 10.9).

The recursion occurs at goals 13 and 14 in figure 10-4. They are repetitions of the topmost goal in the hierarchy (Do-Lights-If-Any), but the scope of each is limited to one-half of the display currently being processed. The numeric computation to obtain the middle of the segment (involving an addition and a division), while on the surface too powerful a computation to appear where it does, is only intended as an approximation to a process that divides the stimulus display into two (or three) roughly equal parts.

In addition to the goal hierarchy, the model assumes a *working memory* for the short-term storage of information relevant to the processing that is going on. For each goal, the working memory is logically partitioned into two components—the *initial state* and the *local state*. The initial state consists of the data existing at the time the goal is first activated. The remainder of the working memory—consisting of those pieces of data created during the processing of the goal—makes up its local state. Only the local state of a goal can be modified during the processing of that goal; the initial state can be examined, but it cannot be modified. The modularity resulting from this scoping rule increases the likelihood that an arbitrary set of goals can be pursued without their interfering with each other. This modularity is also important in insuring correct performance of the chunking mechanism.

Each datum in the working memory is relevant in a particular temporal *processing context*. As long as the datum's creator goal is in the control stack, the system is working either on that goal or on one of its descendants. The datum may be relevant at any point in this processing. However, once the goal disappears from the control stack, it is no longer being pursued. The datum will most likely no longer be relevant. The processing context of a datum is that period of processing during which its creator goal is in the control stack. Once a piece of information becomes *out of context*—its creator is no longer part of the control stack—it usually can be safely deleted from the working memory. Data that cannot be deleted—because they are needed outside of the context in which they were created—are called the *results* of the goal. Their continued presence in the working memory is insured by changing their context to be the context of the parent goal.

10.5 CHUNKING ON GOAL HIERARCHIES

Given a task-independent organization for performance models, it is possible to return to the original objective of describing a task-independent formulation of the chunking theory. In the chunking mechanism described by Rosenbloom and Newell (1982a, 1982b), chunks related patterns of stimuli (lights) to patterns of responses

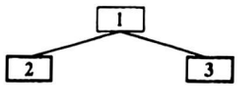
(button presses). The goal-oriented formulation is obtained by altering this definition slightly so that chunks relate patterns of goal parameters to patterns of goal results. Each chunk improves the performance of the system by eliminating the need to process fully a specific instance (a combination of parameter values) of a particular goal. It replaces the normal processing (decomposition into subgoals for nonterminal goals and direct execution of an action for terminal goals) with a direct connection between the relevant parameter values and results. A goal can (and almost certainly will) have more than one possible chunk; each combination of parameter values requires its own chunk.

As with the abstract characterization of the chunking theory, each chunk consists of three components: encoding, decoding, and connection (or mapping). The goal's parameter values form the basis for the encoding component. Given the presence of those values in the working memory, the encoding component generates a new object representing their combination. Encoding is a parallel, goal-independent, data-driven process. Every encoding component executes as soon as appropriate, irrespective of whatever else is happening in the system. The results of encoding components can themselves become parameter values of other goals, leading to a hierarchical encoding process.

The results of the goal form the basis for the decoding component. Given the presence of an encoded result-object in the working memory, the decoding component generates the actual results returned by the goal. Decoding occurs when the results are needed. As with encoding, decoding is a parallel, goal-independent process. The set of decoding components forms a hierarchical structure in which complex results are decoded to simpler ones, which are then decoded even further.

The connection component of the chunk generates the encoded result from the encoded parameter. Connections provide a locus of control by occurring serially, under the control of the goals. Thus a connection can be made only when the system is working on the goal for which the chunk was formed (and after the encoding component has executed). This insures that, even though encoding and decoding are uncontrolled, only appropriate results are generated.

A chunk can be created for a goal when the following two conditions are met: (1) the goal has just completed successfully, and (2) all of the goal's subgoals were themselves processed by chunks. The first condition insures both that chunks are created for appropriate goals and that the chunk is created at a time when the information required for the chunk is available. The second condition causes chunks to be created bottom up in the goal hierarchy. It is this bottom-up aspect of chunking that leads to hierarchical encoding and decoding networks. However, notice that bottom-up chunking does not imply that all low-level chunks are learned before any high-level chunks are learned, or even that all of the chunks must be learned for a subgoal before any can be learned for its parent goal. The second condition on chunk creation merely states that chunks must exist for the goal's subgoals *in the current situation*. Whether other chunks exist or do not exist for the subgoals is irrelevant.



1. Compute-Average-Of-Two-Numbers
2. Compute-Sum-Of-Two-Numbers
3. Divide-Sum-By-2

Figure 10-5: A simple three-goal hierarchy for the averaging of two numbers.

Given enough practice with enough task variations, all of the situations for all of the goals in the hierarchy will be chunked, and asymptotic behavior will be reached for the task. The amount of time this takes depends on the number of goals, the number of situations for each goal, how frequently the different situations arise, and whether chunks are created whenever they can be.

The three-goal hierarchy shown in figure 10-5 provides a simple example of how chunking works. This structure computes the average of two numbers. The top-level goal (Computer-Average-Of-Two-Numbers) takes as parameters the two numbers to be averaged and returns to a single result, which is their mean. The first subgoal (**Compute-Sum-Of-Two-Numbers**) performs the first half of the computation. It takes the two numbers as parameters and returns their sum as its result. The second subgoal finishes the computation by taking the sum as a parameter and returning half of it as its result.

Suppose that the first task is to average the numbers 3 and 7. Control would pass from goal 1 to goal 2. When goal 2 finishes and returns its result of 10, a chunk of three components is created (bottom left of figure 10-6). An encoding component is created that encodes the two parameters (3 and 7) into a new symbol (E1). It executes as soon as it is created, because the parameters are in the working memory. A decoding component is created that decodes from a second new symbol (D1) to the result (10). A connection component (the horizontal line with the goal name above it and goal number below it) is created that generates the result symbol (D1) when it detects both the presence of the encoded parameter (E1) and that goal 2 is the active goal. The connection does not execute immediately because goal 2 is already complete when the chunk is created.

Following the termination of goal 2, goal 1 is reactivated but then is suspended in favor of goal 3 (**Divide-Sum-By-2**). When this goal terminates successfully (returning the number 5), a chunk is created for it (bottom right of figure 10-6). The encoding component encodes the number 10 into the symbol E2; the decoding component decodes from the symbol D2 to the number 5; and the connection component connects E2 to D2 (in the presence of an active goal 3). In contrast to the chunk for goal 1, this chunk can be used in more than one task situation. It can be used whenever goal 1 generates a sum of 10, whether it does it by adding 3 and 7, 5 and 5, or any other pair of numbers. This is a form of transfer of training.

Following the termination of goal 3, goal 1 is reactivated and terminates successfully (returning the number 5). No chunk is created for goal 1 because its subgoals were not processed by chunks. At this point, the task is complete.

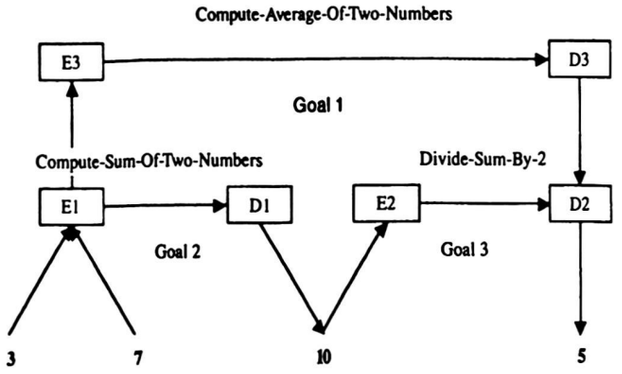


Figure 10-6: Sample chunks created for the hierarchy in figure 10-5.

Given what was learned during the performance of this task, the next time the same task is performed things will go differently. As soon as the task is restarted (again with the values 3 and 7), the encoding component from the chunk for goal 2 executes, placing E1 in the working memory. Goal 1 is activated and then suspended in favor of goal 2. At this point, the connection component for goal 2 executes, generating D1 and successfully completing goal 2. D1 is decoded to the number 10, which is then immediately reencoded to E2 by the encoding component for goal 3. Following the subsequent reactivation and suspension of goal 1, goal 3 is activated. The connection component for goal 3 executes, generating D2 and returning D2 as the result to goal 1. This time, when goal 1 terminates, a chunk is created (top of figure 10-6), because both of the subgoals were processed by chunks.

The encoding component for this chunk builds upon the existing encodings by encoding E1 to a new symbol (E3); it does not go straight from goal 1's primitive parameters (3 and 7). This happens (and causes hierarchical encoding) because, for this instance of goal 1, E1 is the implicit parameter, not 3 and 7. Recall from section 10.4 that the implicit parameters of a goal consist of those pieces of the goal's initial state that are examined during the goal's performance. E1 is generated before goal 1 is activated (so it is part of the goal's initial state) and examined by the connection component for goal 2. On the other hand, neither of the objects representing the numbers 3 and 7 is examined during the processing of goal 1. Therefore, E1 is an implicit parameter (and included in the chunk), and the numbers 3 and 7 are not.

The decoding component is created in an analogous hierarchical fashion. It decodes from a new symbol (D3) to D2. This occurs because D2 (and not the number 5) is the result of goal 1. It never became necessary to decode D2, so it was passed directly up as the result of both goals 3 and 1. The connection component of this chunk links E3 to D3 in a straightforward manner.

If the same task is performed yet again, the encoding components immediately generate E1, followed by E3. Goal 1 is activated, and its connection component executes, generating D3 and completing goal 1. If the result is needed by some part of the system outside of the hierarchy, it will be decoded to D2 and then to the number 5.

The example that we have just gone through outlines the basics of how the chunking mechanism works. The next step is to look at the more complex situation of the Seibel task (figure 10-4). Chunking starts in this structure with the terminal goals (numbers 2, 4, 5, 7, 10, 11, and 12). Consider goal 11 (**Get-Stimulus-X**), for example. Successful completion of this goal requires retrieving from the working memory the representation of a stimulus that has been perceived and then generating a piece of information representing the horizontal location (X) of that stimulus. The parameter for this goal is the stimulus—an implicit parameter—and the result is the location. In the chunk for this situation, the encoding and decoding components are trivial. They simply recode from the single parameter to a new symbol and from a second new symbol to the result. The connection component tests for the presence of an active **Get-Stimulus-X** goal and the encoding symbol and produces the decoding symbol.

Goal 10 (**Get-On-Light-Stimulus**) presents a slightly more complicated case than goal 11. The goal has both explicit and implicit parameters. The explicit parameters (Min-X and Max-X) define the region of the stimulus display in which an on-light should be found. The implicit parameter is the actual external representation (in the stimulus display) of the on-light that is found. The result of this goal is the internal representation of the on-light as it appears in the working memory. Just as before, the implicit parameter and the result form the basis for trivial encoding and decoding components. However, the explicit parameters are different. They do not exist as separate entities in the working memory; instead they act as if they were augmentations to the name of the goal. Therefore, they appear along with the name of the goal as part of the connection component.

As a final example of the workings of the chunking mechanism, consider how it creates chunks consisting of sequences of light-button pairs—the kind of chunks produced by the chunking mechanism in Rosenbloom and Newell (1982a). The locus of these chunks can be found at the recursive step in the goal hierarchy. The root (and recursive) goal in the hierarchy (Do-Lights-If-Any) has implicit parameters that represent lights in the stimulus display, and it generates results that are the button presses for those lights. The earliest chunks that can be created for this goal are those at the bottom of the recursion—that is, goals to process manageable segments of the display. Each of these chunks will represent a single on-light in a region, a region of solid on-lights, or a region with no on-lights. Once the chunks exist for goals 13 and 14 (and their sibling goal 7, the predicate **One-Light-On?**) in a single situation, the parent goal (goal 6: Do-Off-And-On) can be chunked. This yields a new chunk for the combined situation in both segments. This process continues up the hierarchy until goal 1 is chunked for that level of recursion. But goal 1 at that level is just goal 13 or 14 at the next level up. Therefore, gradually (since these chunks are acquired one at a time), the level of aggregation of segments covered by chunks increases.

This process always leads to light-button chunks for contiguous light-button pairs. It does not lead to chunks for disjoint patterns such as the two extreme right and left light-button pairs. This is not a limitation on the generality of the chunking mechanism. Instead, it is a function of the goal structure employed. A different goal structure (reflecting a different processing strategy) could lead to the creation of such disjoint chunk patterns.

One of the strong task dependencies present in the Rosenbloom and Newell (1982a) chunking mechanism was that encoding productions had to have a condition added to them that insured that no on-light appeared between the two patterns being chunked together. This was so even though the information that this condition was necessary appeared nowhere in the task algorithm. In the hierarchy in figure 10-4, such conditions are generated naturally from the goals for processing segments of the display with no on-lights in them. We see that the task dependencies show up in the goal hierarchy, not in the chunking mechanism.

The following list of points summarizes the key aspects of chunking as it applies to goal hierarchies:

- Each chunk represents a specific goal with a specific set of parameter values. It relates the parameter values to the results of the goal.
- Chunks are created through experience with the goals processed.
- Chunks are created bottom up in the goal hierarchy.
- Chunks consist of encoding, connection, and decoding components.
- Chunk encoding and decoding are hierarchical, parallel, goal-asynchronous processes that operate on goal parameters and results, respectively.
- Chunk connection is a serial, goal-synchronous process that generates (possibly encoded) results from (possibly encoded) parameters.
- Chunks improve performance by replacing the normal processing of a goal (and its subgoals) with the faster processes of encoding, connection, and decoding.

10.6 THE XAPS3 ARCHITECTURE

Modeling goal hierarchies and chunking requires an architecture with specific properties. The XAPS3 production-system architecture is one such architecture.[2] It is a new architecture, but it builds upon the work done in the development of the

[2]See Rosenbloom (1983) for a more detailed description of the XAPS3 architecture. A general introduction to production systems can be found in Waterman and Hayes-Roth (1978).

XAPS2 architecture (Rosenbloom and Newell, 1982a). The presentation of XAPS2 begins with a discussion of a set of *constraints* that must be met by any architecture within which the chunking theory of learning is implemented. These constraints state that the model must contain both parallelism and a bottleneck (the *parallel* and *bottleneck* constraints). The parallelism must appear in all aspects of the system's performance including cognitive processing (the *cognitive-parallelism* constraint), and the bottleneck must occur after chunk encoding and before chunk decoding (the *encoding* and *decoding* constraints). It is not claimed that these constraints are known to be necessary—the arguments are not that tight—but this permits (and encourages) attempts to show how the constraints can be circumvented.

The same constraints still hold—as they must if they are really constraints—for the design of the XAPS3 architecture. More recently, two new constraints have been formulated from the consequences of the chunking theory. These new constraints rule out XAPS2 and have led to the design of the XAPS3 architecture. Although XAPS3 is a direct descendent of XAPS2, the constraints have forced a number of significant changes in the XAPS3 design.

The first new constraint is the *crypto-information constraint:*

> **The crypto-information constraint:** Any system that learns through experience cannot use crypto-information if it wants to guarantee correct learning.

Crypto-information—hidden information—is information that the architecture accesses while making performance decisions but that is not accessible to programs running within the architecture. The XAPS2 architecture employs a form of crypto-information in its use of *activation*—real-valued weights associated to portions of working memory. These activation values are used by the architecture to decide which of multiple instantiations of a production should execute on a cycle, among other purposes. It is crypto-information because knowledge about activation values cannot be represented in the productions executed within the architecture. Another example of crypto-information is the information about working memory recency used for conflict resolution in the OPS languages (Forgy, 1981).

The use of activation in XAPS2 is a good example of how crypto-information can lead to incorrect performance. Suppose there is an object *A* with an activation of 0.5 and an object *b* with an activation 0.3. Suppose also that there is a production that matches both *A* and *B*, generating two instantiations. When this situation first occurs, the instantiation that matches *A* will execute because of its greater activation. From this experience, the system learns what action to perform under these circumstances—that is, the action associated with the *A* instantiation. At some later point both *A* and *B* may again be represented, but this time with the activations reversed. Even though the action associated with *B* is now the correct one, its previous experience tells the system to do the action associated with *A*, because the information about relative levels of activation could not be represented in the record of the previous experience.

Not only would the previous information be incorrect, but there would in fact be no way ever to learn the correct information.

It is important to note that this constraint does not rule out all activation-based production-system architectures. If the activation is not used as a basis for an architectural decision, or if sufficient knowledge about activation levels is representable within productions, then activation is not a a problem, because it is no longer a form of crypto-information.

Though activation was a focal point of the XAPS2 design, XAPS3 has been designed to work without it to meet the demands of this constraint. It is thus a more traditional, purely symbolic, production-system architecture.

The second new constraint is the *goal-architecturality constraint:*

> **The goal-architecturality constraint:** The representation and processing of goals must be defined in the architecture itself; it cannot be left up to the discretion of productions.[3]

The reason behind this constraint can be traced to the chunking mechanism's need to understand how goals are represented and processed. This requirement implies that goals must be understood by the architecture, because the chunking mechanism is itself an architectural mechanism.

One way for the architecture to understand the processing of goals is for the goal-processing algorithm to be defined within the architecture itself. The obvious alternative—production-defined processing of goals—requires the chunking mechanism to be able to abstract the algorithm and representation from the productions and tune itself to whatever scheme is being used. This requires considerably more intelligence than the authors are willing to ascribe to the chunking mechanism or for that matter to any mechanism built into the architecture. For a system to exhibit truly adaptive behavior, it must be able to apply its full knowledge and learning capabilities to any task requiring intelligence. This is feasible at the program level but not at the architectural level.

In response to the goal-architecturality constraint, the goal processing that occurred at the level of productions in XAPS2 has been moved down into the architecture of XAPS3.

XAPS3 is one architecture sitting within the design space delimited by the constraints so far determined. The remainder of this section describes the XAPS3 architecture (for more details, see Rosenbloom, 1983). It is like XAPS2 in the structure of working memory objects and in the fact that it allows parallelism at the level of production execution, but it differs in its lack of activation and in its built-in mechanisms

[3]Anderson (1982a) has previously made a similar argument for the placement of goal processing in the architecture of the ACT* production system (Anderson, 1983a).

for goal processing and chunking.[4] This description is divided into sections on the standard components of a production system—working memory, production memory, and the cycle of execution—followed by sections on the processing of goals and the chunking mechanism.

10.6.1 Working Memory

The XAPS3 working memory consists of an unordered set of *objects* representing all of the types of information required in an information-processing system (except for the types of information represented in productions): goals, patterns of stimuli and responses, intermediate computations, and so forth. Each object is a token, in the sense of the classical type-token distinction. It is uniquely specified by a general *type* and a unique *identifier.* The type is a symbol that can be common among several objects, such as **Goal.** The identifier is a unique symbol for the object, which allows multiple objects of the same type to be present simultaneously in working memory. A new identifier is generated dynamically by the architecture whenever a new object is created.

XAPS3 objects also have an optional set of *attributes,* each of which has exactly one *value.* A successfully completed instance of a **Press-Button-At-Horizontal-Location** goal (with a type of **Goal** and an identifier of `Object127`) looks like the following example.[5]

(**Goal** `Object127`
[NAME *Press-Button-At-Horizontal-Location*]
[STATUS *Succeeded*] [RESULT-TYPE *Response*])

The interpretation of the two attributes (NAME and STATUS) should be obvious. They tell the system that this goal is an instance of the **Press-Button-At-Horizontal-Location** goal that has completed successfully. The third attribute (RESULT-TYPE) specifies the type of object to be returned as the result of this goal (see section 10.6.4 for the details).

During the lifetime of a working memory object, two types of auxiliary tags are kept. The *created-by* tag marks the identifier of the goal that was active—there being one at most—when the object was created (if there was no goal active at the time, then the special symbol `<None>` is used). This tag is used for determining the object's status as part of either the local or the initial state of the current goal. The *examined-by* tag marks the identifier of the goal active when the object was last

[4]See Moran (1973), Anderson (1983a), and Sauers and Farrell (1982) for other attempts at integrating the processing of goals into the architecture of a production system.

[5]The format of this object and of subsequent objects and productions has been altered slightly for clarity of presentation.

examined by a production. This tag, in combination with the first one, allows the chunking mechanism to determine the implicit parameters of goals.

10.6.2 Production Memory

There is a single homogeneous production memory in XAPS3. The productions in it are similar to the productions in XAPS2 and to those in the OPS languages. They have three parts: a name, a list of one or more conditions, and a list of one or more actions. Here is an example production taken from the implementation of the Seibel task (figure 10-4):

```
(DefProd SubGoal/Do-Lights/Do-Press-All-Buttons
    ((Goal <Exists> [NAME Do-Lights] [STATUS Active]
            [MINIMUM-LOCATION = ?Min-Loc]
            [MAXIMUM-LOCATION = ?Max-Loc])
     (Goal {<Local> <Exists>} [NAME No-Light-Off?]
            [STATUS Succeeded]))
    -
    ((Goal <New-Object> [Name Do-Press-All-Buttons]
            [STATUS Want] [RESULT-TYPE Response]
            [MINIMUM-LOCATION = ?Min-Loc]
            [MAXIMUM-LOCATION = ?Max-Loc])))
```

The name (SubGoal/Do-Lights/Do-Press-All-Buttons) is purely for the convenience of the programmer; it doesn't affect the processing of the system in any way. Each condition is a pattern to be matched against the objects in working memory. A condition pattern contains a type field (specifying the type of object to be matched), an identifier field (containing several different types of information), and an optional set of patterns for attributes. In the first condition in production SubGoal/Do-Lights/Do-Press-All-Buttons the type is **Goal**, the identifier is <Exists>, and there are four attributes (NAME, STATUS, MINIMUM-LOCATION, and MAXIMUM-LOCATION) with associated values (*Do-Lights*, *Active*, = *?Min-Loc*, and = *?Max-Loc*, respectively).

Condition patterns are built primarily from *constants* and *variables*. Constants are signified by the presence of the appropriate symbol in the pattern and only match objects that contain that symbol in that role. Some example constants in production SubGoal/Do-Lights/Do-Press-All-Buttons are **Goal**, MINIMUM-LOCATION, and *Succeeded*. Variables are signified by a symbol (the name of the variable) preceded by an equal sign. They match anything appearing in their role in an object. An example is = *?Min-Loc* in the sample production. All instances of the same variable within a production must be bound to the same value in order for the match to succeed.

The identifier in the first condition of production SubGoal/Do-Lights/Do-Press-All-Buttons is specified by the special symbol <Exists>. Such a condition succeeds if there is any object in working memory that matches the condition. If there is more than one such object, the choice is conceptually arbitrary (actually, the first object found is used). Only one instantiation is generated. If the identifier is specified as a variable, the condition still acts like an exists condition, but the identifier can be retrieved, allowing the object to be modified by the actions of the production.

The complete opposite of an exists condition is a not-exists condition—commonly called a negated condition. When the symbol <Not-Exists> appears in the identifier field, it signifies that the match succeeds only if there is no object in working memory that matches the remainder of the condition pattern.

The second condition in the sample production contains the symbol <Local> as well as <Exists> in the identifier field (they are conjoined syntactically by braces). When this symbol is added to an identifier pattern of any type—constant, variable, exists, or not-exists—it signifies that the condition should be matched only against those objects in working memory that are local to the active goal. This information—provided by the objects' created-by tags—is a means by which the production that works on a goal can determine which objects are part of their local context.

The remainder of the condition specifies a pattern that must be met by the attribute-value pairs of working memory objects. These patterns are built out of constants, variables, built-in predicates (such as <Greater-Than>), and general LISP computations (via an escape mechanism). In addition, any of the above forms can be negated, denoting that the condition only matches objects that do not match the pattern.

There is only one kind of action in XAPS3—modifying working memory. The interface to the outside world is through working memory rather than through production actions. Actions can create new objects in working memory and, under certain circumstances, modify existing objects. When the action has an identifier of <New-Object>, a new object is created by replacing the identifier with a newly generated symbol, instantiating the variables with their values computed during the match, and replacing calls to LISP functions with their values (via another escape mechanism). An existing object can be modified by passing its identifier as a parameter from a condition. As discussed in section 10.4, only objects local to the current goal can be modified.

There are no production actions that lead to the deletion of values or objects from working memory. A value can be removed only if it is superseded by another value. As discussed in section 10.4, objects go away when they are no longer part of the current context.[6] No explicit mechanism for deletion has proved necessary, so none has been included.

[6]This mechanism is similar to the *dampening* mechanism in the ACT architecture (Anderson, 1976).

10.6.3 The Cycle of Execution

XAPS3 follows the traditional *recognize-act* cycle of production-system architectures, with a few twists thrown in by the need to process goals. The recognition phase begins with the *match* and finishes up with *conflict resolution*. The *cycle number*—simulated time—is incremented after the recognition phase, whether any productions are executed or not. During the act phase, productions are executed.

10.6.3.1 Recognition

The match phase is quite simple. All of the productions in the system are matched against working memory in parallel (conceptually). This process yields a set of legal instantiations with at most one instantiation per production, because each condition generates at most one instantiation. Each instantiation consists of a production name and a set of variable bindings for that production that yield a successful match. This set of instantiations is then passed in its entirety to the conflict resolution phase,[7] where they are winnowed down to the set to be executed on the current cycle. This winnowing is accomplished via a pair of conflict resolution rules.

The first rule is *goal-context refraction.* A production instantiation can fire only once within any particular goal context. It is a form of the standard OPS refractory inhibition rule (Forgy, 1981), differing only in how the inhibition on firing is released. With the standard rule, the inhibition is released whenever one of the working memory objects on which the instantiation is predicated has been modified. With goal-context refraction, inhibition is released whenever the system leaves the context in which the instantiation fired. If the instantiation could not legally fire before the context was established but could fire both while the context was active and after the context was terminated, then the instantiation must be based, at least in part, on a result generated during the context and returned when the context was left. Therefore, the instantiation should be free to fire again to reestablish the still-relevant information.

The second rule—the *parameter-passing bottleneck*—states that only one parameter-passing instantiation can execute on a cycle (conceptually selected arbitrarily). This rule first appeared in a slightly different form as an assumption in the HPSA77 architecture (Newell, 1980a). It will be justified in section 10.6.5.

10.6.3.2 Action

All of the instantiations that make it through the conflict resolution phase are fired (conceptually) in parallel. Firing a production instantiation consists of

[7]For purposes of efficiency, these two phases are actually intermingled. However, this does not change the behavior of the system at the level at which we are interested.

(1) overwriting the *examined-by* tags of the working memory objects matched by the instantiation, with the identifier of the active goal; (2) replacing all variables in the actions by the values determined during the match phase; (3) evaluating any LISP forms; and (4) performing the actions. This can result in the addition of new objects to working memory or the modification of existing local objects. If conflicting values are simultaneously asserted, the winner is selected arbitrarily. This does not violate the crypto-information constraint because the architecture is using no information in making the decision. If the performance system works correctly, even when it can't depend on the outcome of the decision, then the learned information will not lead to incorrect performance.

10.6.4 Goal Processing

In section 10.4, the processing of goals was described at an abstract level. In this section how that processing is implemented within the XAPS3 architecture is described. The goals themselves, unlike chunks, are just data structures in working memory. A typical XAPS3 goal goes through four phases in its life. The current phase of a goal is represented explicitly at all times by the value associated with the STATUS attribute of the working memory object representing the goal.

In the first stage of its life, the goal is *desired:* at some point, but not necessarily right then, the goal should be processed. Productions create goal desires by generating a new object of type **Goal.** Each new goal object must have a NAME attribute, a STATUS attribute with a value of *Want,* and a RESULT-TYPE attribute. In addition, it may have any number of other attributes, specifying explicit parameters to the goal. The value of the RESULT-TYPE attribute specifies the type of the results that are to be returned on successful completion of the goal. All local objects of that type are considered to be results of the goal. Results are marked explicitly so they won't be flushed when the context is left and so the chunking mechanism will know to include them in the chunks for the goal.

Leaving the expression of goal desires under the control of productions allows goals to be processed by the architecture, while the structure of the hierarchy is still left under program (that is, production) control. The architecture controls the transition to the second, *active* phase of the goal's life. At most, one goal is active at any point in time. The architecture attempts to activate a new goal whenever the system is at a loss about how to continue with the current goal. This occurs when there is an empty conflict set; that is, there is no production instantiation that can legally fire on the current cycle. When this happens, the system looks in working memory to determine if there are any subgoals of the current goal—those goals created while the current goal was active—that are desired. If such a subgoal is found, it is made the active goal, and the parent goal is *suspended* by replacing its STATUS with the identifier of the newly activated subgoal. If more than one desired subgoal is found, one is arbitrarily selected (actually, the last one found is used).

Suspension is the third phase in the life of a goal (it occurs only for nonterminal goals). Replacing the STATUS of the parent goal with the subgoal's identifier accomplishes two things: it halts work on the goal, because the productions that process goals all check for a STATUS of *Active,* and it maintains the control stack for the goal hierarchy. A suspended goal remains suspended until its active subgoal terminates, at which time it returns to being active. If a goal has more than one subgoal, the goal will oscillate between the active and suspended states.

If no progress can be made on the active goal and there are no desired subgoals, then the system has no idea how to continue making progress; it therefore terminates the active goal with a STATUS of *Failed.* Following termination, the goal is in its fourth and final phase of life. In addition to a failure termination, goals can be terminated with a STATUS of *Succeeded.* There are no uniform criteria for determining when an arbitrary goal has completed successfully, so it has been left to productions to assert that this has happened. This is done via the creation of an object of type **Succeeded.** When the architecture detects the presence of such an object in working memory, it terminates the current goal and reactivates its parent.

At goal termination time a number of activities occur in addition to the obvious one of changing the active goal. The first two activities occur only on the successful termination of a goal. As will be discussed in the following section, the first step is the possible creation of a chunk. The second step is to return the results of the terminating goal to its parent goal. This is accomplished by altering the created-by tags of the results so that it looks as if they were created by the parent goal. The third step is to delete all of the objects from working memory created during the processing of this goal. The fourth and final step is to enable result decoding if it is appropriate.

10.6.5 Chunking

As was seen in section 10.5, chunking improves performance by enabling the system to use its experience with previous instances of a goal to avoid expanding the goal tree below it. In this section is detailed how this has been implemented within the XAPS3 architecture—yielding a working, task-independent production-system practice mechanism. This section begins with a description of how chunks are used and concludes with a description of how they are acquired.

10.6.5.1 The Use of Chunks

The key to the behavior of a chunk lies in its connection component. When a goal is proposed in a situation in which a chunk has already been created for it, the connection component of that chunk substitutes for the normal processing of the goal. This is accomplished in XAPS3 by having the connection component check

that the goal's status is *desired.* The connection is a production containing a condition testing for the existence of a desired goal, a condition testing for the encoding of the goal's parameters, and any relevant negated (`<Not-Exists>`) conditions. It has two actions: one marks the goal as having succeeded, and the other asserts the goal's encoded result. If there is a desired goal in working memory in a situation for which a connection production exists, the connection will be eligible to fire. Whether (and when) it does fire depends on conflict resolution. Connection productions are subject to the parameter-passing-bottleneck conflict resolution rule because they pass the identifier of the desired goal as a parameter to the action that marks the goal as having succeeded. This conflict resolution rule serves a dual purpose: (1) it implements part of the bottleneck constraint by insuring that only one goal can be connected at a time, and (2) it removes a source of possible error by insuring that only one connection can execute for any particular goal instance.

If the connection does fire, it removes the need to activate and expand the goal, because the goal's results will be generated directly by the connection (and decoding) and the goal will be marked as having succeeded. If instead no connection production is available for the current situation of a desired goal, then eventually the production system will reach an impasse—no productions eligible to fire—and the goal will be activated and expanded. Therefore, we have just the behavior required of chunks—they replace goal activation and expansion, if they exist.

This behavior is, of course, predicated on the workings of the encoding and decoding components of the chunk.[8] The encoding component of a chunk must execute before the associated connection can. In fact, it should execute even before the parent goal is activated, because the subgoal's encoded symbol should be part of the parent goal's initial state. Recall that encodings are built up hierarchically. If the parent goal shares all of the parameters of one of its subgoals, then the encoding of the parent goal's parameters should be based on the encoding generated for the subgoal. This behavior occurs in XAPS3 because the encoding components are implemented as goal-free productions that do not pass parameters. They fire (concurrently with whatever else is happening in the system) whenever the appropriate parameter values for their goal are in working memory (subject to refraction). If the parameters exist before the parent goal becomes active, as they must if they are parameters to it, then the encoded symbol becomes part of the parent goal's initial state.

As stated in section 10.4, the decoding component must decode an encoded result when it will be needed. Each decoding component is a production that keys off the nonprimitive result pattern generated by the connection production and off an

[8]For efficiency, the encoding and decoding components are not created if there is only one parameter or result, respectively.

object of type **Decode**. When the architecture determines that decoding should occur, it places a **Decode** object in working memory, with the type of the object to be decoded specified by the TYPE attribute.[9] The actions of the decoding production generate the component results out of which the nonprimitive pattern was composed.

10.6.5.2 The Acquisition of Chunks

A complete specification of the chunk acquisition process must include the details of when chunks can be acquired and from what information they are built. A chunk can be acquired when three conditions are met. The first condition is that some goal must have just been completed. The system can't create a chunk for a goal that terminated at some point in the distant past, because the information on which the chunk must be based is no longer available. Chunks also are not created prior to goal completion (on partial results). Chunks are simple to create in part because they summarize all of the effects of a goal. If chunks were created partway through the processing of a goal—for partial results of the goal—then a sophisticated analysis might be required in order to determine which parameters affect which results and how. This is not really a limitation on what can be chunked, because any isolable portion of the performance can be made into a goal.

The second condition is that the goal must have completed successfully. Part of the essence of goal failure is that the system does not know why the goal failed. This means that the system does not know which parameter values have lead to the failure; thus, it can't create a chunk that correctly summarizes the situation. Chunking is success-driven learning, as opposed to failure-driven learning (see for example, Winston, 1975).

The third and final condition for chunk creation is that all of the working memory modifications occurring since the goal was first activated must be attributable to that goal, rather than to one of its subgoals. This condition is implemented by insuring that no productions were fired while any of the goal's subgoals were active. All of the subgoals must either be processed by a chunk or fail immediately after activation—failure of a subgoal, particularly of predicates, does not necessarily imply failure of the parent goal.

To summarize, a chunk is created after the system has decided to terminate a goal successfully but before anything is done about it (such as marking the goal succeeded, returning the results, or flushing the local objects from working memory). At that point the goal is still active, and all of its information is readily available.

Most of the information on which the chunk is based can be found in working memory (but see below). The first piece of information needed for the creation of a chunk is the name (and identifier) of the goal that is being chunked. This information

[9]Matters are actually somewhat more complicated (Rosenbloom, 1983).

is found by retrieving the object representing the active goal. The goal's explicit parameters are also available as attribute-value pairs on the goal object. Given the goal's identifier, the system finds its implicit parameters by retrieving all of the objects in working memory that were part of the goal's initial state—that is, their created-by tag contains an identifier different from that of the active goal—and that were examined by a production during the processing of the active goal. This last type of information is contained in the objects' examined-by tags. The goal's results are equally easy to find. The architecture simply retrieves all of the goal's local objects that have a type equal to the goal's RESULT-TYPE.

Because the goal parameter and result information is determined from the constant objects in working memory, chunks themselves are not parameterized. Each chunk represents a specific situation for a specific goal. However, two forms of abstraction are performed during the creation of a chunk: (1) the inclusion of only the implicit parameters of a goal and not the entire initial state, and (2) the replacement of constant identifiers (found in the working memory objects) with neutral `<Exists>` specifications. These abstractions allow the chunks to be applicable in any situation that is relevantly identical, not merely totally identical. Different chunks are needed only for relevant differences.

The one complication in the clean picture of chunk acquisition so far presented involves the use of negated conditions during the processing of a goal. When a negated condition successfully matches working memory, there is no working memory object that can be marked as having been examined. Therefore, some of the information required for chunk creation cannot be represented in working memory. The current solution for this problem is not elegant, but it works. A temporary auxiliary memory is maintained, into which is placed each nonlocal negated condition occurring on productions that fire during the processing of the goal (local negated conditions can be ignored because they do not test the initial state). This memory is reinitialized whenever the goal that is eligible to be chunked changes. Before the conditions are placed in the memory they are fully instantiated with the values bound to their variables by the other conditions in their production. As discussed in Rosenbloom (1983), including a negated condition in an encoding production can lead to performance errors, so these conditions are all included in the associated connection production.

10.7 RESULTS

In this section some results derived from applying the XAPS3 architecture to a set of reaction-time tasks will be presented. A more complete presentation of these results can be found in Rosenbloom (1983). The first experiment described here is the Seibel task. Two different sequences of trials were simulated, of which the first

sequence is the same as the one used in Rosenbloom and Newell (1982a). The simulation completed 268 trials before it was terminated.[10] A total of 682 productions was learned. On the second sequence of trials—from a newly generated random permutation of the 1023 possibilities—259 trials were completed before termination. For this sequence, 652 productions were learned.

Figure 10-7 shows the first sequence as fit by a general power law. Each point in the figure represents the mean value over five data points (except for the last one, which only includes three).[11] For this curve, the asymptote parameter (A) has no effect. Only E, the correction for previous practice, is required to straighten out the curve. At first glance, it seems nonsensical to talk about previous practice for such a simulation, but a plausible interpretation is possible. In fact, there are two independent explanations—either or both may be responsible.

The first possibility is that the level at which the terminal goals are defined is too high (complex). If the "true" terminals are more primitive, then chunking starts at a lower level in the hierarchy. One view of what chunks are doing is that they are turning their associated (possibly nonterminal) goals into terminal goals for particular parameter situations. During preliminary bottom-up chunking, the system would eventually reach the lowest level in the current hierarchy. All of the practice prior to that point is effectively previous practice for the current simulation.

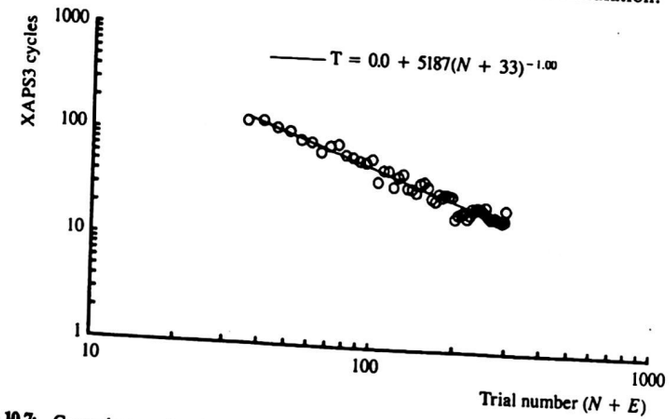


Figure 10-7: General power-law fit to 268 simulated trials of the Seibel (1963) task.

[10] At this point the FRANZLISP system—which appeared not to be garbage collecting in the first place—refused to allocate any more memory for the simulation.

[11] These data appear noisier than the human data from Seibel (1963) shown in figure 10-2. This is accounted for by the fact that each point in figure 10-2 was the mean of 1023 data points and each point in this figure is the mean of five data points.

The other source of previous practice is the goal hierarchy itself. This structure is posited at the beginning of the simulation, hence it is already known perfectly. However, there must exist some process of *method acquisition* by which the subject goes from the written (or oral) instructions to an internal goal hierarchy. Though method acquisition does not fall within the domain of what is here defined as "practice," a scheme will be proposed in section 10.9 whereby chunking may lead to a mechanism for method acquisition.

In addition to the Seibel task, the system so far has been applied to fourteen tasks from three different stimulus-response compatibility experiments (Fitts and Seeger, 1953; Morin and Forrin, 1962; Duncan, 1977). As a sample of these results, figure 10-8 shows a pair of simulated practice curves for two tasks from Fitts and Seeger (1953). These curves contain fifty trials each, aggregated by five trials per data point.

This chapter is not the appropriate place to discuss the issues surrounding whether the simulated practice curves are truly power laws or something slightly different (such as exponentials). Suffice it to say that a mixture of exponential, power law, and ambiguous curves is obtained. These results roughly follow the predictions of the approximate mathematical analysis of the chunking theory appearing in Rosenbloom (1983). They also fit with the observation that the human curves tend to be most exponential for the simplest tasks—the compatibility tasks are among the simplest tasks for which we have human practice data. For more on this issue, see Newell and Rosenbloom (1981) and Rosenbloom (1983).

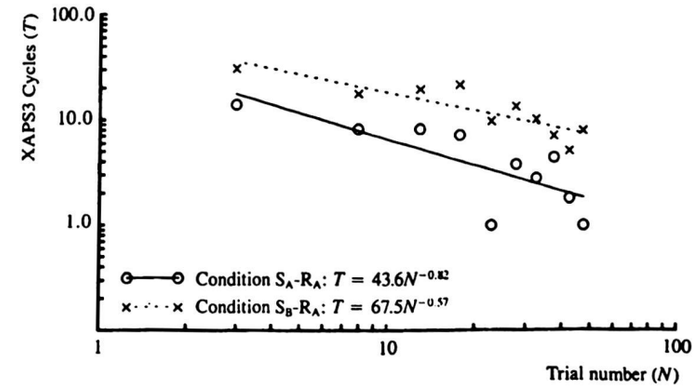


Figure 10-8: Simulated practice curves for conditions S_A-R_A and S_B-R_A from Fitts and Seeger (1953). The latter curve is the average over two hierarchy variations.

10.8 RELATIONSHIP TO PREVIOUS WORK

The current formulation of the chunking theory of learning provides an interesting point of contact among four previously disparate concepts: (1) classical chunking; (2) production composition (Lewis, 1978; Neves and Anderson, 1981; Anderson, 1982b); (3) table look-up—memo functions (Michie, 1968) and signature tables (Samuel, 1967); and (4) macro-operators (Fikes, Hart and Nilsson, 1972; Korf, 1983). Classical chunking has already been discussed in section 10.3, so this section covers only the latter three ideas, followed by a proposal about the underlying commonality among these concepts.

10.8.1 Production Composition

Production composition (Lewis, 1978; Neves and Anderson, 1981; Anderson, 1982b) is a learning scheme whereby new productions are created through the combination of old ones. Given a pair of productions that execute successively, the system creates their composition from their conditions and actions (figure 10-9). The condition side of the new production consists of all of the conditions of the first production (C_1, C_2, and C_3), plus those conditions from the second production that do not match actions of the first production (C_5). The conditions of the second production that match actions of the first production (C_4 matches A_4) are not included in the composition (removing the serial dependency between the two productions). All of the actions from both productions are combined in the action side of the new production (A_4 and A_6). The resulting composition is a single production that accomplishes the combined effects of the older pair of productions. As learning continues, composed productions can themselves be composed, until there is a single production for an entire task.

In some recent work with the GRAPES system (Saures and Farrell, 1982; Anderson, Farrell, and Saurers, 1982; Anderson, 1983b), production composition was integrated with goal-based processing. In GRAPES, specific goals are designated by the programmer to be ones for which composition will occur. When such a

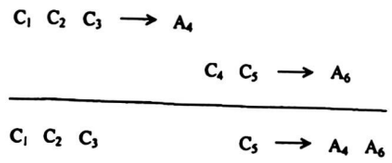


Figure 10-9: An example of production composition.

goal completes successfully, all of the productions that executed during that time are composed together, yielding a single production that accomplishes the goal.

Because the main effects of chunking and goal-based composition are the same—the short-circuiting of goals by composite productions—it is probably too early to know which mechanism will turn out to be the correct one for a general practice mechanism. However, there are a number of differences between them worth noting. We will focus on the three most important: (1) the knowledge required by the learning procedure: (2) the generality of what is learned; and (3) the hierarchical nature of what is learned.

10.8.1.1 Knowledge-Source Differences

With chunking, all of the information required for learning can be found in working memory (modulo negated conditions). With production composition, the information comes from production memory (and possibly from working memory). Being able to ignore the structure of productions has two advantages. The first advantage is that the chunking mechanism can be much simpler. This is both because working memory is much simpler than production memory—productions contain conditions, actions, variables, function calls, negations, and other structures and information—and because, with composition, the complex process of matching conditions of later productions to actions of previous productions is required, in order that conditions that test intermediate products not be included in the composition. Chunking accomplishes this by only including objects that are part of the goal's initial state.

The second advantage of the chunking strategy, of learning from working memory, is that chunking is applicable to any goal, no matter what its internal implementation is (productions or something else). As long as the processing of the goal leaves marks on the working memory objects that it examines, chunking can work.

10.8.1.2 Generalization Differences

The products of chunking are always constant productions (except for the identifiers of objects) that apply only for the situation in which they were created (although, as already discussed, two forms of abstractions are performed). With production composition, the variables existing in the productions to be composed are retained in the new production. The newly learned material is thus more general than that learned by chunking. The chunking mechanism definitely has more of a table look-up flavor. Section 10.8.2 contains a more thorough discussion of chunking as table look-up, and section 10.9 discusses how a chunking mechanism could possibly learn parameterized information.

10.8.1.3 Hierarchical Differences

Both mechanisms learn hierarchically in that they learn for goals in a hierarchy. They differ in how they decide about which goals to learn and in whether the learned material is itself hierarchical.

Chunking occurs bottom up in the goal hierarchy. Production composition—in GRAPES at least—works in isolation on any single goal in the hierarchy. For this to work, subgoals are kept as actions in the new productions. The composition approach is more flexible, but the chunking approach has two compensating advantages. The first advantage is that, with chunking, the encoding and decoding components can be themselves hierarchical, based on the encoding and decoding components of the previously chunked subgoals. Productions produced by composition tend to accumulate huge numbers of conditions and actions because they are flat structures.

The second advantage is again simplicity. When information is learned about a goal at an arbitrary position in the hierarchy, its execution is intermingled with the execution of its subgoals. Knowing which information belongs in which context requires a complete historical trace of the changes made to working memory and the goals that made the changes.

10.8.2 Table Look-up

It has been seen that from one point of view chunking resembles production composition. From another point of view it resembles a table look-up scheme, in which a table of input parameters versus results is gradually learned for each goal in the system. As such, it has two important predecessors—memo functions (Michie, 1968; Marsh, 1970) and signature tables (Samuel, 1967).

10.8.2.1 Memo Functions

A memo function[12] is a function with an auxiliary table added. Whenever the function is evaluated, the table is first checked to see if there is a result stored with the current set of parameter values. If there is, it is retrieved as the value of the function. Otherwise, the function is computed and the arguments and result are stored in the table. Memo functions have been used to increase the efficiency of mathematical functions (Michie, 1968; Marsh, 1970) and of tree searches (Marsh, 1970).

[12]Memo functions themselves are derived from the earlier work by Samuel (1959) on creating a rote memory for the values of board positions in the game of checkers.

Chunking can be seen as generating memo functions for goals. But these are hierarchical memo functions, and ones in which the arguments need not be specified explicitly. Chunking also provides a cleaner implementation of the ideas behind memo functions because the table is not simply an add-on to a different processing structure. It is implemented by the same "stuff" (productions) as is used to represent the other types of processing in the system.

10.8.2.2 Signature Tables

Signature tables were developed as a means of implementing nonlinearities in an evaluation function for checkers (Samuel, 1967). The evaluation function is represented as a hierarchical mosaic of signature tables. Each signature table had between two and four parameters, each of which had between three and fifteen possible values. The parameters to the lowest-level tables were measures computed on the checkerboard. For each combination of parameter values a number was stored in the table representing how good that combination was. There were nine of these tables arranged in a three-level hierarchy. The values generated by lower tables were fed into higher tables. The final value of the evaluation function was the number generated by the root (top) table.

Signature tables capture the table look-up and hierarchical aspects of chunking, though only for encoding. There is no decoding because signature tables are not a model of action; they act simply as a classifier of board positions. Another difference between chunking and signature tables is that information is stored in the latter not as a direct function of experience, but as correlations over a number of experiences.

10.8.3 Macro-Operators

A macro-operator is a sequence of operators that can be viewed as a single operator. One classical system that makes use of macro-operators is STRIPS (Fikes, Hart, and Nilsson, 1972). STRIPS works by taking a task and performing a search to find a sequence of operators that will accomplish the task. Given a solution, STRIPS first generates a highly specific macro-operator from the sequence of operators and then generalizes it by figuring out which constants can be replaced by variables. The generalized macro-operator is used as a plan to guide the performance of the task, and it can be used as a primitive operator in the generation of a macro-operator for another task.

Each STRIPS operator is much like a production: it has a set of conditions (a *precondition wff*) and a set of actions (consisting of an *add list* and a *delete list*). Each macro-operator is represented as a *triangle table* representing the conditions and actions for all subsequences of the operators in the macro-operator (preserving the

order of execution). This process is very much like a production composition scheme that takes all of the productions that fire during the processing of a goal and creates a composition for every possible subsequence of them. STRIPS differs from the mechanisms described above in exactly how it represents, selects, and uses what it learns, but it shares a common strategy of storing, with experience, meaningful (based on the task/goal) composites that can reduce the amount of processing required by subsequent tasks.

Another form of macro-operators can be found in Korf's (1983) work on macro-operator–based problem solving. Korf presents a methodology by which a table of macro-operators can be found that can collectively solve all variations on a problem. For example, one table of macro-operators is sufficient for solving any initial configuration of Rubik's cube. Korf's technique is based on having a set of differences between the goal state and the current state. The differences are ordered and then solved one at a time. During the solution of a difference, solutions to previous differences can be destroyed, but they must be reinstated before the current difference is considered solved. Rather than learn by experience, Korf's system preprocesses the task to learn the macro-operator table capable of handling all variations of the task. It does this in time proportional to what it would take to search for a solution to one variation of the task without the table of macro-operators.

Even though the macro-operators are nonvariabilized, a single table with size proportional to the product of the number of differences and the number of values per difference is totally sufficient. This is because at each point in the solution what is to be done depends only on the value of the current difference and not on any of the other differences. It is possible to be more concrete and to bring out the relationship of this mechanism to chunking by viewing the sequence of differences as a goal hierarchy. The top-level goal is to solve all of the differences. In general, to solve the first $x + 1$ differences one first processes a subgoal for solving the first x differences; one then processes a subgoal for solving the first $x + 1$ differences given that the first x differences have been solved. These latter conditional goals are the terminal goals in the hierarchy. Moreover, each one has only one parameter that can vary—the value of difference $x + 1$—so only a very few macro-operators need be created for the goal (the number of values that the difference can take). Korf's macro-operators are essentially chunks for these terminal goals. They relate the parameters of a goal (the set of differences already solved and the value of the next difference) to the composite result (the sequence of operators to be performed). Korf avoids the combinatorial explosion implicit in the tasks by creating macro-operators only for these limited terminal goals. If the chunking mechanism were doing the learning, it would begin by chunking the terminals, but it would then proceed to learn about the nonterminal goals as well. Korf's work is a good example of how choosing the right goal hierarchy (and limiting the set of goals for which learning occurs) can enable a small set of nonvariabilized macro-operators (or chunks) to solve a large class of problems.

10.8.4 Summary

Although classical chunking, production composition, table look-up, and macro-operators were proposed in quite different contexts and bear little obvious relationship to each other, the current formulation of the chunking theory of learning has strong ties to all four. (1) It explains how classical chunks can be created and used; (2) it results in productions similar to those generated by goal-directed production composition; (3) it caches the results of computations, as in a table look-up scheme; and (4) it unitizes sequences of operators into higher-level macro-operators. The chunking theory differs from these four mechanisms in a number of ways, but at the same time it occupies a common ground among them. This leads the authors to propose that all five mechanisms are different manifestations of a single underlying idea centered on the storage of composite information rather than its recomputation. The chunking theory has a number of useful features, but it is probably too early to know what formulation will turn out to be the correct one in the long run for general practice mechanism.

10.9 EXPANDING THE SCOPE OF CHUNKING

In this work it has been shown how chunking can provide a model of practice for tasks that can already be accomplished. Performance is sped up but not qualitatively changed. It is interesting to ask whether chunking can be used to implement any of the other, more complex forms of learning, such as method acquisition, concept formation, generalization, discrimination, learning by being told, and expectation-driven learning. Chunking does not directly accomplish any of these forms of learning, and on first glance the table look-up nature of chunking would seem to preclude its use in such sophisticated ways. However, four considerations suggest the possibility that the scope of chunking may extend much further.

The first two considerations derive from the ubiquitous presence of both chunking and the power law of practice in human performance. Chunking is already implicated at least in higher-level cognitive processes, and the power law of practice has been shown to occur over *all* levels of human performance. Thus, if the current chunking theory turns out not to be extendable, the limitation will probably be in the details of the implementation itself rather than in the more global formulation of the theory.

The remaining two considerations stem from the interaction of chunking with problem solving. The combination of these two mechanisms has the potential for generating interesting forms of learning. The strongest evidence of this potential to date can be found in the work of Anderson (1983b). He has demonstrated how

production composition (a mechanism that is quite similar to chunking, as has been shown here), when combined with specific forms of problem solving, can effectively perform both production generalization and discrimination. Generalization comes about through the composition of an analogy process, and discrimination comes from the composition of an error-correction procedure.

The final line of evidence comes from the work of Newell and Laird on the structure of a general problem-solving system based on the *problem space hypothesis* (Newell, 1980b), a *universal weak method* (Laird and Newell, 1983), and *universal subgoaling* (Laird, 1983). The problem space hypothesis states that intelligent agents are always performing in a problem space. At any instant, the agent will have a *goal* that it is attempting to fulfill. Associated with that goal is a *problem space* in which the goal can be pursued. The problem space consists of a set of *states*, a *problem* (initial and desired states), a set of *operators* that move the agent between states, and *search control* information that assists in guiding the agent efficiently from the initial to the desired state. Added to this problem space structure are (1) a universal weak method, which allows the basic problem-solving methods, such as generate-and-test and depth-first search, to arise trivially out of the knowledge of the task being performed; and (2) universal subgoaling, which enables a problem solver automatically to create subgoals for any difficulties that can arise during problem solving. The result of this line of work has been a problem-solving production-system architecture called SOAR2 that implements these three concepts (Laird, 1983).

To see the power of integrating chunking with such a problem-solving system, consider the problem of *method acquisition;* given an arbitrary task or problem, how does the system first construct a method (goal structure) for it? This is the prototypical case of "hard" learning. There are at least two ways in which chunking can assist SOAR2 in method acquisition: (1) by compiling complex problem-solving processes into efficient operators, and (2) by acquiring search control information that eliminates irrelevant search paths. A single goal-chunking mechanism can acquire both of these types of information; the difference is in the types of goals that are chunked.

The compilation process is analogous to the kinds of chunks that are created in XAPS3: inefficient subgoal processing is replaced by efficient operator application. Given a task along with its associated goals and problem spaces, SOAR2 attempts to fulfill the task goal through a repeated process of elaborating the current situation with information and selecting a new goal, problem space, state, or operator. The system applies operators to a state by creating a new state, elaborating it with the results of the operator, and selecting the new state. If the application of the operator requires problem solving itself, it will not be possible to apply it to a state directly via a set of elaborations. Instead, a difficulty will arise for which SOAR2 will create a subgoal.

One way this subgoal can be pursued is by the selection of a problem space within which the task of applying the problematic operator to its state can be accomplished. The subgoal is fulfilled when the operator has been applied and a new state generated. A chunk could be created for this subgoal that would be applicable in any state that defines the same set of parameters for that operator. The next time the operator is tried it will be applied directly, so the difficulty will not occur and the subgoal will not be needed.

Another way that a difficulty can arise in SOAR2 is if there is uncertainty about which operator to apply to a state. As with all such difficulties, SOAR2 automatically creates a subgoal to work on this problem. It then employs an *operator selection* problem space within which it can evaluate and compare the set of candidate operators. The difficulty is finally resolved when a set of *preferences*—statements about which operators are preferred to which other operators—has been created that uniquely determines which of the original operators should be selected.

A subgoal that deals with an operator selection problem has a set of parameters—those aspects of the goal and state that were examined during the generation of the preferences. It also has a set of results—the preferences. Should a chunk be created for this goal, it would be a piece of search control for the problem space that allows it to pick the appropriate operator directly. As the system acquires more search control, the method becomes more efficient because of the resulting reduction in the amount of search required.

One limitation of the current chunking mechanism that such a method acquisition scheme could alleviate is the inability of chunks to implement parameterized operators. Chunking always creates a totally specified piece of knowledge. As currently formulated, it cannot create the kinds of parameterized operators used as terminal nodes in the goal hierarchies. We have seen that chunking does create abstracted knowledge, and Korf's (1983) work shows that nonvariabilized macro-operators can attain a good deal of generality from the goal hierarchy itself (see section 10.8.3), but fully parameterized operators are outside the current scope. On the other hand, problem spaces are inherently parameterized by their initial and desired states. Therefore it may be that it is not necessary for chunks to create parameterized operators. These operators can come from another source (problem spaces). Chunks would only be responsible for making these operators more efficient.

In summary, the ubiquity of both chunking and power-law learning indicates that the chunking model may not be limited in its scope to simple speedups. Examining three "hard" types of learning reveals that generalization and discrimination are possible via the combination of a similar learning mechanism (production composition) and specific types of problem solving; additionally, method acquisition appears feasible via chunking in problem spaces. If this does work out, it may prove possible to be able to formulate a number of the other difficult learning problems within this paradigm. The complications would appear as problem solving in problem spaces, and the chunking mechanism would remain simple, merely recording the results generated by the problem-solving system.

10.10 CONCLUSION

At the beginning of this investigation the authors set out to develop a generalized, task-independent model of practice, capable of producing power-law practice curves. The model was to be based on the concept of chunking, and it was to be used as the basis (and a source of constraint) for a production-system practice mechanism. All of this has been accomplished. The generalized model that has been developed is based on a goal-structured representation of reaction-time tasks. Each task has its own goal hierarchy, representing an initial performance algorithm.

When a goal is successfully completed, a three-part chunk can be created for it. The chunk is based on the parameters and results of the goal. The encoding component of the chunk encodes the parameters of the goal, yielding a new symbol representing their combination. The connection component of the chunk ties the encoded parameter symbol to an encoded symbol for the results of the goal. The decoding component of the chunk decodes the new result symbol to the results out of which it is composed.

The chunk improves the performance of the system by eliminating the need to process the goal fully; the chunk takes care of it. The process of chunking proceeds bottom up in the goal hierarchy. Once chunks are created for all of a goal's subgoals in a specific situation, it is possible to create a chunk for the goal. This process proceeds up the hierarchy until there is a chunk for the top-level goal for every situation that it could face.

Mechanisms for goal processing and chunking have been built into a new production-system architecture that fits within a set of contraints developed for the architecture of cognition. This architecture has been applied to a number of different reaction-time tasks (though not all of these results are presented here). It is capable of producing power-law practice curves.

As currently formulated, the chunking theory stakes out a position that is intermediary among four previous disparate mechanisms: classical chunking, memo functions, production composition, and macro-operators. These five ideas are different manifestations of a single underlying idea centered on the storage of composite information rather than its recomputation.

And finally, a research path has been outlined by which the chunking theory, when integrated with a problem-solving system, can potentially be expanded to cover aspects of learning, such as method acquisition, outside of the domain of pure practice.[13]

[13]For follow-up work along this path, see Laird, Rosenbloom, and Newell (1984).

ACKNOWLEDGMENTS

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 U.S. Government.

The authors would like to thank John Laird for innumerable helpful discussions about this material.

References

Anderson, J. R., *Language, Memory, and Thought*, Erlbaum, Hillsdale, N.J., 1976.

———, Private communication, 1980.

———, Private communication, 1982a.

———, "Acquisition of Cognitive Skill," *Psychological Review*, Vol. 89, pp. 369–406, 1982b.

———, *The Architecture of Cognition*, Harvard University Press, Cambridge, 1983a.

———, "Knowledge Compilation: The General Learning Mechanism," *Proceedings of the Machine Learning Workshop*, R. S. Michalski (Ed.), Allerton House, University of Illinois at Urbana-Champaign, pp. 203–12, June 22–24, 1983b. (An updated version of this paper appears as chap. 11 of this volume.)

Anderson, J. R., Farrell, R., and Sauers, R., "Learning to Plan in LISP," Technical Report, Department of Psychology, Carnegie-Mellon University, 1982.

Bower, G. H., "Perceptual Groups as Coding Units in Immediate Memory," *Psychonomic Science*, Vol. 27, pp. 217–19, 1972.

Bower, G. H., and Springston, F., "Pauses as Recoding Points in Letter Series," *Journal of Experimental Psychology*, Vol. 83. pp. 421–30, 1970.

Bower, G. H., and Winzenz, D., "Group Structure, Coding, and Memory for Digit Series," *Journal of Experimental Psychology Monograph*, Vol. 80, Pt. 2, pp. 1–17, May 1969.

Card, S. K., English, W. K., and Burr, B., "Evaluation of Mouse, Rate Controlled Isometric Joystick, Step Keys, and Text Keys for Text Selection on a CRT," *Ergonomics*, Vol. 21, pp. 601–13, 1978.

Chase, W. G., and Ericsson, K. A., "Skilled Memory," in *Cognitive Skills and Their Acquisition*, J. R. Anderson (Ed.), Erlbaum, Hillsdale, N.J., 1981.

Chase, W. G. and Simon, H. A., "Perception in Chess," *Cognitive Psychology*, Vol. 4, pp. 55–81, 1973.

Crossman, E. R. F. W., "A Theory of the Acquisition of Speed-Skill," *Ergonomics*, Vol. 2, pp. 153–66, 1959.

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

Duncan, J., "Response Selection Rules in Spatial Choice Reaction Tasks," *Attention and Performance VI*, Erlbaum, Hillsdale, N.J., 1977.

Ernst, G. W., and Newell, A., *GPS: A Case Study in Generality and Problem Solving*, ACM Monograph, Academic Press, New York, 1969.

Fikes, R. E., Hart, P. E., and Nilsson, N. J., "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, Vol. 3, pp. 251–88, 1972.

Fitts, P. M., and Seeger, C. M., "S-R Compatibility: Spatial Characteristics of Stimulus and Response Codes," *Journal of Experimental Psychology*, Vol. 46, pp. 199–210, 1953.

Forgy, C. L., "OPS5 User's Manual," Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, July 1981.

Johnson, N. F., "Organization and the Concept of a Memory Code," in *Coding Processes in Human Memory*, A. W. Melton and E. Martin (Eds.), Winston, Washington, D.C., 1972.

Kolers, P. A., "Memorial Consequences of Automatized Encoding," *Journal of Experimental Psychology: Human Learning and Memory*, Vol. 1, No. 6, pp. 689–701, 1975.

Korf, R. E., "Learning to Solve Problems by Searching for Macro-Operators," Ph.D. diss., Carnegie-Mellon University, 1983. (Available as Technical Report No. 83-138, Department of Computer Science, Carnegie-Mellon University, 1983.)

Laird, J. E., "Universal Subgoaling," Ph.D. diss., Carnegie-Mellon University, 1983.

Laird, J. E., and Newell, A., "A Universal Weak Method," Technical Report No. 83-141, Department of Computer Science, Carnegie-Mellon University, 1983.

Laird, J. E., Rosenbloom, P. S., and Newell, A., "Towards Chunking as a General Learning Mechanism," in *Proceedings of AAAI-84*, Austin, Tex., pp. 188–192, 1984.

Lewis, C. H., "Production System Models of Practice Effects," Ph.D. diss., University of Michigan, 1978.

Marsh, D., "Memo Functions, the Graph Traverser, and a Simple Control Situation," in *Machine Intelligence 5*, B. Meltzer and D. Michie (Eds.), American Elsevier, New York, 1970.

Michie, D., "'Memo' Functions and Machine Learning," *Nature*, Vol. 218, pp. 19–22, 1968.

Miller, G. A., "The Magic Number Seven Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review*, Vol. 63, pp. 81–97, 1956.

Moran, T. P., "The Symbolic Imagery Hypothesis: An Empirical Investigation via a Production System Simulation of Human Behavior in a Visualization Task," Ph.D. diss., Carnegie-Mellon University, 1973.

———, "Compiling Cognitive Skill," AIP Memo No. 150, Xerox PARC, 1980.

Morin, R. E., and Forrin, B., "Mixing Two Types of S-R Association in a Choice Reaction Time Task," *Journal of Experimental Psychology*, Vol. 64, pp. 137–41, 1962.

Neisser, U., Novick, R., and Lazar, R., "Searching for Ten Targets Simultaneously," *Perceptual and Motor Skills*, Vol. 17, pp. 427–32, 1963.

Neves, D. M., and Anderson, J. R., "Knowledge Compilation: Mechanisms for the Automatization of Cognitive Skills," in *Cognitive Skills and Their Acquisition*, J. R. Anderson (Ed.), Erlbaum, Hillsdale, N.J., 1981.

Newell, A., "Heuristic Programming: Ill-Structured Problems," in *Progress in Operations Research*, Vol. 3, J. Aronofsky (Ed.), Wiley, New York, 1969.

———, "Production Systems: Models of Control Structures," in *Visual Information Processing*, W. G. Chase (Ed.), Academic Press, New York, 1973.

———, "Harpy, Production Systems and Human Cognition," in *Perception and Production of Fluent Speech*, R. Cole (Ed.), Erlbaum, Hillsdale, N.J., 1980a. (Also available as Technical Report No. CMU-CS-78-140, Department of Computer Science, Carnegie-Mellon University, 1978.)

———, "Reasoning, Problem Solving and Decision Processes: The Problem Space as a Fundamental Category," in *Attention and Performance VIII*, R. Nickerson (Ed.), Erlbaum, Hillsdale, N.J., 1980b. (Also available as Technical Report CMU CSD, Department of Computer Science, Carnegie-Mellon University, 1979.)

Newell, A. and Rosenbloom, P. S., "Mechanisms of Skill Acquisition and the Law of Practice," in *Cognitive Skills and Their Acquisition*, J. R. Anderson (Ed.), Erlbaum, Hillsdale, N.J., 1981.

Newell, A., and Simon, H. A., *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

Nilsson, N. J., *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.

Rosenbloom, P. S., "The Chunking of Goal Hierarchies: A Model of Practice and Stimulus-Response Compatibility," Ph.D. diss., Carnegie-Mellon University, 1983. (Available as Technical Report No. 83-148, Department of Computer Science, Carnegie-Mellon University, 1983.)

Rosenbloom, P. S., and Newell, A., "Learning by Chunking: A Production-System Model of Practice," Technical Report No. 82-135, Department of Computer Science, Carnegie-Mellon University, 1982a.

———, "Learning by Chunking: Summary of a Task and a Model," *Proceedings of AAAI-82*, Pittsburgh, Pa., pp. 255–257, 1982b.

Samuel, A. L., "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, Vol. 3, pp. 210–29, 1959.

———, "Some Studies in Machine Learning Using the Game of Checkers, II—Recent Progress," *IBM Journal of Research and Development*, Vol. 11, pp. 601–17, 1967.

Sauers, R., and Farrell, R., "GRAPES User's Manual," Technical Report, Department of Psychology, Carnegie-Mellon University, 1982.

Seibel, R., "Discrimination Reaction Time for a 1,023-Alternative Task," *Journal of Experimental Psychology*, Vol. 66, No. 3, pp. 215–26, 1963.

Snoddy, G. S., "Learning and Stability," *Journal of Applied Psychology*, Vol. 10, pp. 1–36, 1926.

Waterman, D. A., and Hayes-Roth, F. (Eds.), *Pattern-Directed Inference Systems*, Academic Press, New York, 1978.

Winston, P. H., "Learning Structural Descriptions from Examples," in *The Psychology of Computer Vision*, P. H. Winston (Ed.), McGraw-Hill, New York, 1975.

Woodworth, R. S., and Schlosberg, H., *Experimental Psychology*, rev. ed., Holt, Rinehart and Winston, New York, 1954.

CHAPTER 8

Towards Chunking as a General Learning Mechanism

J. E. Laird, P. S. Rosenbloom, and A. Newell, Carnegie Mellon University

ABSTRACT

Chunks have long been proposed as a basic organizational unit for human memory. More recently chunks have been used to model human learning on simple perceptual-motor skills. In this paper we describe recent progress in extending chunking to be a general learning mechanism by implementing it within a general problem solver. Using the *Soar* problem-solving architecture, we take significant steps toward a general problem solver that can learn about all aspects of its behavior. We demonstrate chunking in *Soar* on three tasks: the Eight Puzzle, Tic-Tac-Toe, and a part of the *R1* computer-configuration task. Not only is there improvement with practice, but chunking also produces significant transfer of learned behavior, and strategy acquisition.

1 Introduction

Chunking was first proposed as a model of human memory by Miller [8], and has since become a major component of theories of cognition. More recently it has been proposed that a theory of human learning based on chunking could model the ubiquitous power law of practice [12]. In demonstrating that a practice mechanism based on chunking is capable of speeding up task performance, it was speculated that chunking, when combined with a general problem solver, might be capable of more interesting forms of learning than just simple speed ups [14]. In this paper we describe an initial investigation into chunking as a general learning mechanism.

Our approach to developing a general learning mechanism is based on the hypothesis that all complex behavior — which includes behavior concerned with learning — occurs as search in problem spaces [11]. One image of a system meeting this requirement consists of the combination of a performance system based on search in problem spaces, and a complex, analytical, learning system also based on search in problem spaces [10]. An alternative, and the one we adopt here, is to propose that all complex behavior occurs in the problem-space-based performance system. The learning component is simply a recorder of experience. It is the experience that determines the form of what is learned.

Chunking is well suited to be such a learning mechanism because it is a recorder of goal-based experience [13, 14]. It caches the processing of a subgoal in such a way that a chunk can substitute for the normal (possibly complex) processing of the subgoal the next time the same subgoal (or a suitably similar one) is generated. It is a task-independent mechanism that can be applied to all subgoals of any task in a system. Chunks are created during performance, through experience with the goals processed. No extensive analysis is required either during or after performance.

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 F33615-81-K-1539. 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.

The essential step in turning chunking into a general learning mechanism is to combine it with a general problem-space problem solver. One candidate is *Soar*, a reflective problem-solving architecture that has a uniform representation and can create goals to reason about any aspect of its problem-solving behavior [5]. Implementing chunking within *Soar* yields four contributions towards chunking as a general learning mechanism.

1. Chunking can be applied to a general problem solver to speed up its performance.

2. Chunking can improve *all* aspects of a problem solver's behavior.

3. Significant transfer of chunked knowledge is possible via the implicit generalization of chunks.

4. Chunking can perform strategy acquisition, leading to qualitatively new behavior.

Other systems have tackled individual points, but this is the first attempt to do all of them. Other work on strategy acquisition deals with the learning of qualitatively new behavior [6, 10], but it is limited to learning only one type of knowledge. These systems end up with the *wandering bottle-neck* problem — removal of a performance bottleneck from one part of a system means that some other locale becomes the bottleneck [10]. Anderson [1] has recently proposed a scheme of knowledge compilation to be a general learning mechanism to be applied to all of cognition, although it has not yet been used on complex problem solving or reasoning tasks that require learning about all aspects of behavior.

2 Soar — A General Problem-Solving Architecture

Soar is a problem solving system that is based on formulating all activity (both problems and routine tasks) as heuristic search in problem spaces. A problem space consists of a set of *states* and a set of *operators* that transform one state into another. Starting from an initial state the problem solver applies a sequence of operators in an attempt to reach a desired state. *Soar* uses a production system[1] to implement elementary operators, tests for goal satisfaction and failure, and *search control* — information relevant to the selection of goals, problem spaces, states, and operators. It is possible to use a problem space that has no search control, only operators and goal recognizers. Such a space will work correctly, but will be slow because of the amount of search required.

In many cases, the directly available knowledge may be insufficient for making a search-control decision or applying an operator to a state. When this happens, a *difficulty* occurs that results in the automatic creation of a subgoal to perform the necessary function. In the subgoal, *Soar* treats the difficulty as just another problem to solve; it selects a problem space for the subgoal

[1] A modified versions of Ops5 [3], which admits parallel execution of all satisfied productions.