An Integrated Cognitive Architecture for Autonomous Agents

Pat Langley ◇
Kevin Thompson ◇
Wayne Iba
John H. Gennari
John A. Allen

Department of Information & Computer Science
University of California, Irvine, CA 92717

Technical Report 89-28

September 15, 1989

◇ Current address: AI Research Branch, Mail Stop 244-17, NASA Ames Research Center, Moffett Field, CA 94035.

This research was supported by Contract MDA 903-85-C-0324 from the Army Research Institute. We would like to thank members and alumni of the UCI machine learning group – especially Doug Fisher, David Benjamin, and Patrick Young – for useful discussions that led to many of the ideas in this paper. We also thank Mike Pazzani for his comments on an earlier draft.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Technical Report No. 4	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* An Integrated Cognitive Architecture for Autonomous Agents		5. TYPE OF REPORT & PERIOD COVERED Annual Report 7/88–6/89 6. PERFORMING ORG. REPORT NUMBER UCI–ICS Technical Report 89–**
7. AUTHOR*(s)* Pat Langley, Kevin Thompson, Wayne F. Iba, John Gennari, and John A. Allen		8. CONTRACT OR GRANT NUMBER*(s)* MDA 903-85-C-0324
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Information & Computer Science University of California, Irvine, CA 92717		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Army Research Institute 5001 Eisenhower Avenue Alexandria, Virginia 22333		12. REPORT DATE September 15, 1989 13. NUMBER OF PAGES 46
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

Portions of this report appeared as separate papers in *Proceedings of the Sixth International Workshop on Machine Learning.* Ithaca, NY: Morgan Kaufmann, 1989.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

cognitive architectures	incremental hill climbing
probabilistic concepts	concept formation
heuristic classification	plan acquisition
spatial knowledge	motor learning

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

OVER

20. ABSTRACT

In this paper we describe ICARUS, a cognitive architecture designed to control the behavior of an integrated intelligent agent. The framework assumes that all long-term knowledge is organized in a *probabilistic concept hierarchy*, that *heuristic classification* is the central performance mechanism, and that a process of *concept formation* underlies all learning. We present the details of ICARUS' three main components: LABYRINTH, which classifies composite objects and acquires object concepts; DÆDALUS, which generates plans and acquires plan knowledge; and MÆANDER, which recognizes, executes, and acquires motor schemas. We also describe CLASSIT, a module for classification and concept formation that underlies the other three components. In addition, we relate ICARUS to other cognitive architectures, discuss some open issues, and describe our plans for extending and evaluating the system.

1. Introduction

In order to exist in the world, an intelligent agent must have access to large amounts of knowledge about physical objects, plans, and motor skills, and it must also be able to acquire and organize new knowledge from experience. Traditional research in artificial intelligence has focused on high-level aspects of cognition, making few efforts to develop integrated systems that interact with the physical environment. In this paper we describe ICARUS, an integrated cognitive architecture that we have designed with these issues in mind.

Our long-term goal is an integrated intelligent agent that acts on internal drives, acquires knowledge from experience, and uses this knowledge to achieve its goals. We will focus on three main types of knowledge – physical object concepts, planning knowledge, and motor schemas – as crucial for intelligent behavior in a physical environment. In solving a problem, the agent should recognize similar problems it has solved before. In recognizing an object, it should use knowledge of objects it has seen in the past. In performing a motor operation, it should take into account previous operations of a similar nature. The working hypotheses of ICARUS are that a single long-term memory – a probabilistic *concept hierarchy* – can represent knowledge in all these domains, that a single performance mechanism – *heuristic classification* – underlies all these abilities, and that a single learning mechanism – *concept formation* – is sufficient to acquire this knowledge from experience.

1.1 Issues for Cognitive Architectures

We have designed ICARUS to address six basic issues. Other researchers have dealt with some of these independently, but not within the context of a single research project. These issues include:

- *Interaction with the environment*: An ICARUS agent constructs a model of its environment from sensory input and interacts with this environment through effectors. Although we are currently working with simulated worlds, we have the long-term goal of attaching agents to physical robots. We assume that early vision and primitive motor control are solved problems, but we model cognition at a 'lower', more primitive level than most AI researchers.
- *Grounded symbols*: Most research in AI has assumed high-level symbolic representations that are disconnected from sensori-motor issues. In contrast, ICARUS assumes that all symbols are ultimately *grounded* in some sensori-motor description (Harnad, 1989). For instance, the symbol THROW would be described in terms of an agent's arm, the thrown object, and the manner in which both change over time. For simplicity, most of our examples in this paper will involve symbolic primitives like shape and color, but we believe that real-valued attributes – such as size, position, and velocity – are necessary to describe much of the complexity in the physical world.
- *Learning as incremental hill climbing*: Learning is essential for intelligent action in an uncertain, complex world, and we feel that much 'everyday' learning in humans (e.g., concept formation and skill acquisition) occurs in a gradual, unconscious fashion. We do

not believe that physical agents can afford extensive search through a space of hypotheses, nor can they afford extensive reprocessing of previous instances. An ICARUS agent starts with a weak model of the world, which it improves with experience using an incremental hill-climbing strategy (Langley, Gennari, & Iba, 1987). The learning component 'searches' a space of long-term memory structures using simple-minded, local operators that adjust memory in response to new experience.

- *Organization and indexing of knowledge*: Unlike much AI research, our work on ICARUS emphasizes the importance of organizing knowledge in long-term memory. We believe that intelligent action emerges not only from large amounts of domain knowledge, but also from the ability to efficiently find the "best" knowledge for a given goal and situation. To this end, we organize long-term memory into a hierarchy of concepts that are used for indexing and retrieval.
- *Psychological constraints*: Although we do not intend ICARUS as a complete psychological model, its characteristics are constrained by coarse-level psychological phenomena. This bias has been useful in generating research ideas, and we feel that many properties of the human information-processing system are useful for interacting with the world. For example, our approach to classification and concept formation has been influenced by studies of basic-level effects in human classification (Mervis & Rosch, 1981). This phenomenon may prove to be not just a robust psychological invariant, but an important constraint on indexing concepts.
- *Integrated architecture*: Much of ICARUS' promise lies not in its components, but in their symbiotic organization. Any planning system is incomplete without some mechanism for executing its operators, and any approach to motor behavior is incomplete without some higher-level control component. Neither can occur without some mechanism for recognizing objects and states in the world. The components of ICARUS will work together, accessing the same long-term memory structures in order to take advantage of previous experience.

Taken together, our responses to these issues have significantly constrained our design of ICARUS, and they have provided the basic assumptions on which we base our approach. In the following pages, we describe the directions in which these assumptions have led.

1.2 An Overview of ICARUS

Before turning to the details of ICARUS, we should give a brief overview of the architecture's memories and processes. The system gains information about its environment through a *sensory buffer*, which contains descriptions of external objects and the agent's own internal states. This buffer is very short-lived, but it is constantly refreshed with updated descriptions of the world. ICARUS makes no claims about the nature of early vision or other senses. Marr (1982) has argued that the human visual system produces three-dimensional descriptions of objects in the environment, and we will assume that such information is available to the agent without explaining its origin. Thus, we explicitly abstract away from problems of

multi-sensor fusion and object delineation, while remaining concerned with the higher-level problem of object *recognition*.

We make similar simplifying assumptions about motor control. ICARUS affects its environment by placing commands in a *motor buffer*, which causes body parts to move to specified positions at given velocities. Translating such commands to muscle contractions or voltage changes is an active research problem in robotics, but we will assume that this task can be separated from the generation of motor commands. Clearly, there exist some limits on this independence assumption. For instance, one cannot lift an object beyond a given mass, but under reasonable load conditions and in the absence of obstacles, we will ignore issues of low-level motor control.

The main repository of ICARUS' knowledge is *long-term memory*,[1] which consists of nodes organized into an 'is-a' hierarchy. Each node can be viewed as a 'symbol' or 'chunk' that is ultimately grounded in a sensori-motor description. We will generally refer to these nodes as *concepts*, regardless of their purpose. Long-term memory is effectively infinite in size, retaining all items stored in it indefinitely, though 'forgetting' can occur by losing access to an item. ICARUS distinguishes between simple and composite concepts. The former are described directly in terms of sensory-level features like length, width, texture, and hue; whereas the latter are composed of simple concepts or lower-level composites. ICARUS does not rely on a small, predetermined set of 'primitive' symbolic concepts; the system can acquire an indefinite number of concepts from experience. In the following sections, we describe the representation and organization of concepts in more detail.

Figure 1 presents a graphic representation of ICARUS' main processes and memories, with memories shown as circles and processing modules as rectangles. The architecture has no single "top-level" module, but instead has several cooperating processes that act in parallel. Information from the environment enters the sensory buffer, where the process of object recognition decides on the categories of observed objects and stores them in long-term memory, based on earlier experience with similar objects. The LABYRINTH system implements this process and its associated learning mechanism, as we describe in Section 3.

The planning process operates solely on long-term memory, noting unsolved problems (goals to transform one state into another), generating plans to solve them, and storing traces of this process in memory. Section 4 describes DÆDALUS, a system that instantiates our approach to plan generation. At the lowest level, plans specify operators that should be applied, and a third ICARUS component generates motor programs from descriptions of these operators in long-term memory. These motor commands alter the motor buffer, which directly affects the environment. The MÆANDER system implements the process of motor control, as we describe in Section 5. This component can also recognize and store instances of motor schemas.[2]

---

[1] One can view short-term memory as the active portion of long-term memory, but we delay discussion of such issues until Section 6.2.

[2] We assume that perceptual traces are parsed into appropriate temporal chunks before being given to the recognition process. We do not yet have a theory of this segmentation process.

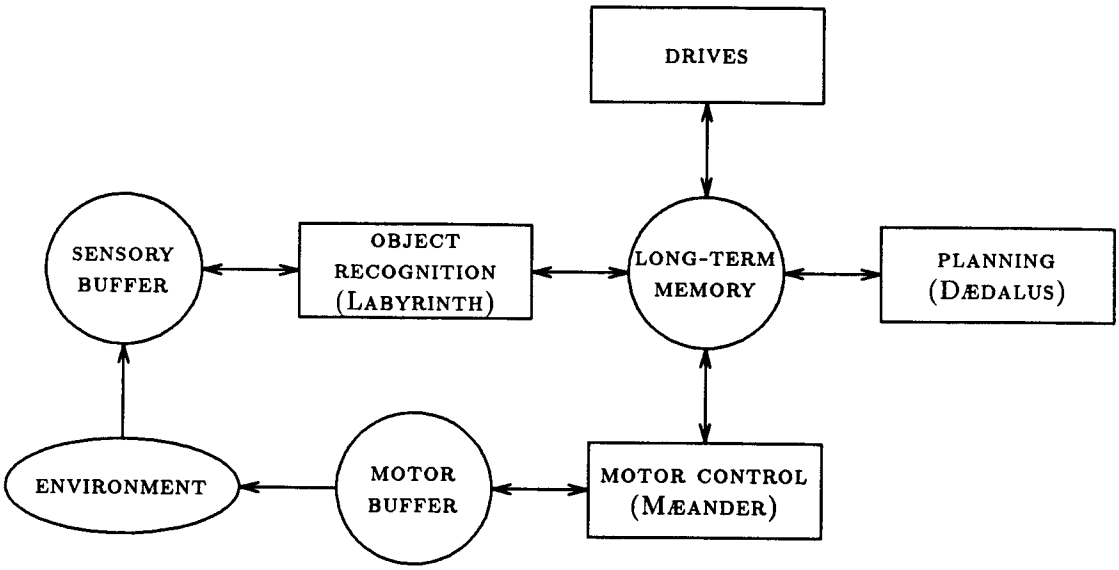

Figure 1. Memories and processes of the ICARUS architecture.

Given a top-level problem or goal, ICARUS' planning module can generate subproblems that help achieve that goal. However, the system needs some means to generate its top-level goals, and this is the function of internal *drives*. These are structures in long-term memory that match against key situations, such as noticeable hunger or extreme fatigue. When a drive is activated, it adds a new goal to memory, where (if it is sufficiently active) it attracts the attention of the planning module. In some cases, this means interrupting processing of the current problem. Although we believe drives should play a major role in controlling ICARUS' behavior, they constitute the least specified aspect of the architecture, and we will mention them only in passing.

Figure 1 obscures a central feature of ICARUS, notably that a single mechanism underlies the diverse processes of object recognition, planning, and motor control. This mechanism provides the higher-level components with access to long-term memory, modifying this memory in the act of retrieval. Thus, this process tightly integrates learning with performance, leading to incremental changes in the long-term store. We discuss this mechanism in the following section, delaying discussion of the components that use it until later.

2. Classification and Concept Formation

The basic operation in ICARUS involves retrieving relevant knowledge through a process of *heuristic classification* (Clancey, 1985). This task can be stated as:

- *Given:* An unclassified instance that may be only partially described;
- *Find:* The category in which that instance should be placed.

Having retrieved a category, one can also use the stored description of that class to make predictions about unobserved aspects of the instance. For example, a doctor may diagnose a disease on the basis of a few symptoms, and then prescribe medicine that should cure

the disease or slow its progression. Many other aspects of human behavior can be viewed in these terms, and our work with ICARUS assumes that heuristic classification is the basic process underlying all cognition.

This approach to intelligence implies that one has access to a large knowledge base, and this knowledge must be acquired and organized in some fashion. Our work on ICARUS further assumes that this occurs through an incremental process of *concept formation*, which involves clustering a sequence of observed instances into categories, forming intensional descriptions for each category, and creating a hierarchical organization for the categories. Concept formation differs from the task of *learning from examples* (e.g., Quinlan, 1986) in that learning is unsupervised. Thus, it can be viewed as a form of *conceptual clustering* (e.g., Michalski & Stepp, 1983), but it differs from most work on this topic in that learning must be incremental. Examples of concept formation systems include Feigenbaum's (1963) EPAM, Kolodner's (1980, 1983) CYRUS, Lebowitz's (1980, 1987) UNIMEM, and Fisher's (1987a, 1987b) COBWEB. Although our approach has much in common with all of these systems, it borrows most heavily from Fisher's work.

As we detail in Sections 3, 4, and 5, ICARUS applies the same basic processes of classification and concept formation to the retrieval and acquisition of object concepts, plan knowledge, and motor schemas. However, before turning to these aspects of cognition, we must first describe the basic processes, which we have implemented in a system called CLASSIT (Gennari, Langley, & Fisher, 1989). Below we summarize the nature and organization of the system's memory, then describe the algorithm, its evaluation function, and some measures of performance improvement. In the following treatment, we will use the term *concept* to refer to any node in long-term memory that summarizes one or more instances, whether these instances refer to objects, plans, motor behavior, or some other event.

2.1 Probabilistic Representation of Concepts

CLASSIT assumes that each instance is described as a conjunction of attribute-value pairs, and it employs a probabilistic representation for concepts (Smith & Medin, 1981). A probabilistic scheme associates a probability with each attribute value of a concept description, thus subsuming 'logical' representations that specify concepts as conjunctions of necessary attributes. In particular, CLASSIT represents each concept C_k as a set of attributes A_i and a subset of their possible values V_{ij}. Associated with each value is the conditional probability of that value given membership in the class, $P(A_i = V_{ij}|C_k)$. In addition, each concept has an associated probability of occurrence, $P(C_k)$. For example, the attribute BIRTH for the MAMMAL concept would have LIVE with very high probability and EGGS with low probability; the vast majority of mammals give live birth, with only a few laying eggs.

Although most of our examples will involve nominal (symbolic) representations, CLASSIT can also handle real-valued attributes. In the nominal case, the system effectively stores a discrete probability distribution for each attribute associated with a concept. Thus, a natural analog for a real-valued attribute would be to store a continuous probability distribution. CLASSIT assumes that the values of such real-valued attributes follow a *normal* distribution,

which it can conveniently summarize by its mean μ and its standard deviation σ. In this scheme, more general concepts (those covering more instances) tend to have attributes with higher σ values, and more specific concepts have attributes with lower standard deviations. Fried and Holyoak (1984) give arguments for positing a normal distribution in numeric domains.

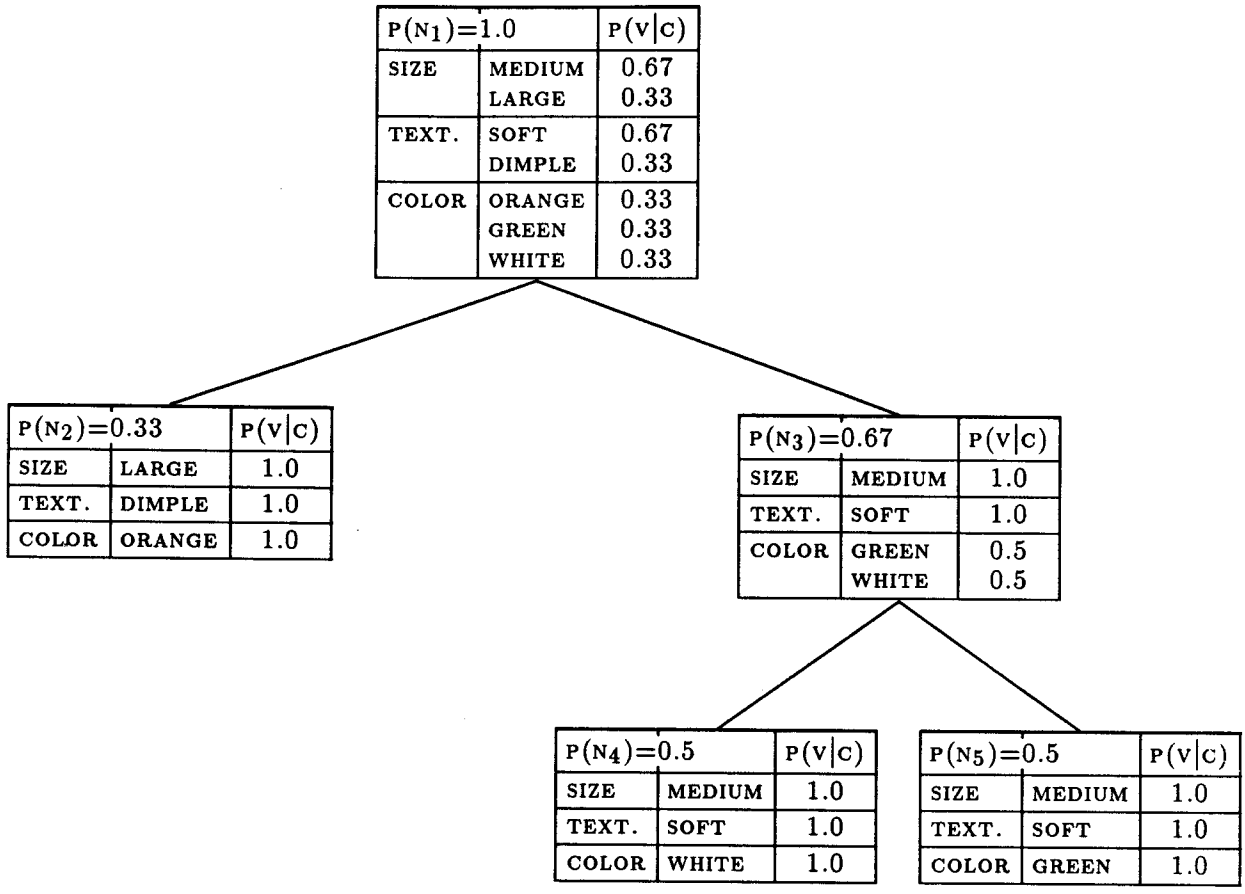

Figure 2. A CLASSIT concept hierarchy that organizes three balls.

2.2 Organizing Knowledge in a Concept Hierarchy

Like earlier approaches to concept formation, CLASSIT organizes its knowledge into a *hierarchy* of concepts. Nodes in this hierarchy are partially ordered according to their generality, with concepts lower in their hierarchy being more specific than their ancestors. Thus, the root node summarizes all instances that have been observed, terminal nodes often correspond to single instances, and intermediate nodes summarize clusters of observations. Fisher and Langley (in press) review arguments for organizing probabilistic concepts in a hierarchy.

Figure 2 presents a simple concept hierarchy that organizes three instances of balls, along with the probabilities for each concept and those for the attribute values. For example, the root node (N_1) has an associated probability of one and states that its members have a

$\frac{2}{3}$ chance of being MEDIUM in size and a $\frac{1}{3}$ chance of being LARGE, the same probabilities for having a SOFT and DIMPLED texture, and equal probabilities of being ORANGE, GREEN, or WHITE in color. Concept N_3 has a $\frac{2}{3}$ chance of occurring and its members are always MEDIUM sized and SOFT, but they are evenly split among GREEN and WHITE colors. The terminal nodes in the hierarchy – N_2 (a basketball), N_4 (a softball), and N_5 (a tennis ball) – have less interesting probabilistic descriptions, since each is based on a single instance. Note that the probability of each node's occurrence is specified relative to its parent, rather than with respect to the entire distribution.

There is a strong similarity between CLASSIT's concept hierarchies and those occurring in Fisher's (1987a) COBWEB. Both differ from EPAM, UNIMEM, and CYRUS, which labeled the links from parent nodes to their children with explicit indices. In contrast, CLASSIT and COBWEB connect parents to their children only through IS-A links, treating the concept nodes themselves as indices. In addition, both systems divide instances into disjoint classes, so that each observation is summarized by nodes along a single path through the hierarchy; this differs from UNIMEM and CYRUS, which allow non-disjoint hierarchies. However, CLASSIT diverges from COBWEB in that it does not store all observed instances; in some cases, terminal nodes themselves may contain abstractions, as in UNIMEM. Also, Fisher's system stores all attributes with every node in the hierarchy, whereas CLASSIT (Gennari, 1989) stores only the most diagnostic attributes, as we discuss in Section 2.5.

2.3 Classification and Learning in CLASSIT

Table 1 presents the basic CLASSIT algorithm, which classifies observations and forms a concept hierarchy in the process. Upon encountering a new instance I, the system starts at the root and sorts the instance down the hierarchy, using an evaluation function (described in Section 2.4) to decide which action to take at each level. At a given node N, it retrieves all children and considers placing the instance in each child C in turn; CLASSIT also considers creating a new child based on the instance. The algorithm uses its evaluation function to determine which of the resulting partitions is 'best',[3] and then carries out the appropriate action, which in turn modifies memory. Thus, the processes of classification and learning are inextricably intertwined.

More specifically, if the instance I is sufficiently different from all the concepts in a given partition, the evaluation function recommends placing I into a singleton class rather than incorporating it into an existing concept. In this case, CLASSIT creates a new child of the current parent node and bases its initial description on that of the instance. The classification process halts at this point, since the new node has no children.

If CLASSIT instead decides to incorporate the instance I into an existing child C, it modifies the probability distribution for each attribute in C based on the instance's values, thus updating the concept definition. The system also updates the probability of the selected

[3] This lets the system avoid the need for explicit attribute tests or indices at each node. At the level we have described it, the CLASSIT algorithm is identical to that used in Fisher's COBWEB.

Table 1. The top-level CLASSIT algorithm.

```
Input: The current node N of the concept hierarchy.
       An unclassified (attribute-value) instance I.
Side effects: A concept hierarchy that classifies the instance.
Top-level call: Classit(Top-node, I).
Variables: C, L, M, P, Q, and R are nodes in the hierarchy.
           U, V, W, and X are clustering (partition) scores.

Classit(N, I)

  If N is a terminal node,
     Then Create a new child L of node N.
          Initialize L's probabilities to those for N.
          Create a new child M of node N.
          Initialize M's probabilities using I's values.
          Incorporate(N, I).
  Else Incorporate(N, I).
       For each child C of node N,
           Compute the score for placing I in C.
       Let P be the node with the highest score W.
       Let R be the node with the second highest score.
       Let X be the score for placing I in a new node Q.
       Let Y be the score for merging P and R into one node.
       Let Z be the score for splitting P into all its children.
       If W is the best score,
              Then if W is high enough,
                   Then Classit(P, I) (place I in category P).
       Else if X is the best score,
          Then initialize Q's probabilities using I's values
               (place I by itself in the new category Q).
       Else if Y is the best score,
              Then let O be Merge(P, R, N).
                   Classit(O, I).
       Else if Z is the best score,
              Then Split(P, N).
                   Classit(N, I).
```

category. Next, CLASSIT decides whether to recurse on the children of the concept, continuing down the hierarchy only if I is different enough (according to a system parameter) from its description.

If the current concept C is a terminal node and if I is different enough from C, the system extends the hierarchy downward by creating two new children, one based on C and another based on I. Thus, the system retains specific cases only if they are sufficiently different from previous experiences. This strategy slows the growth of the concept hierarchy, and preliminary experiments (Gennari et al., 1989) suggest that it also reduces overfitting of the data (Quinlan, 1986).

Some examples based on the hierarchy in Figure 2 will clarify the effect of these operators. Given a new instance I (a colored golf ball) described as SIZE SMALL, COLOR ORANGE, and TEXTURE DIMPLED, CLASSIT first incorporates the observation into the root node $N1$, giving a new set of probabilities. The system then considers the two children of this node, deciding that the instance best matches $N2$, and that adding the instance to this class would be better than creating a new disjunct. As a result, it incorporates I into $N2$ (giving new scores) and creates two children for $N2$, one ($N6$) based on the original version of $N2$ and the other ($N7$) based on the instance. Figure 3 presents the hierarchy after sorting is complete.

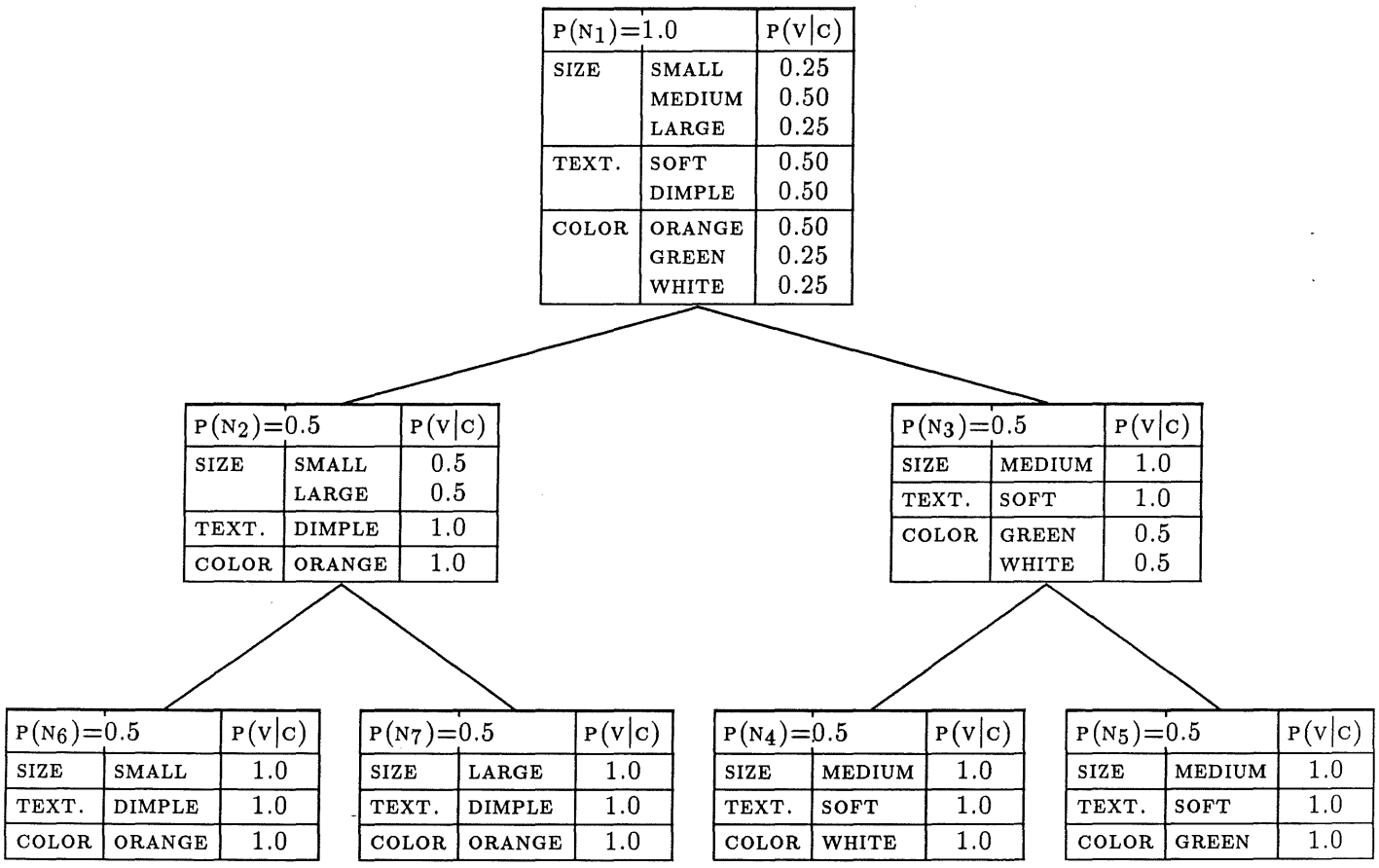

Figure 3. The concept hierarchy after incorporating a fourth ball.

Now suppose CLASSIT encounters another instance J (a marble), which is described as SIZE SMALL, COLOR CLEAR, and TEXTURE SMOOTH. Again the system incorporates the description into the root node $N1$, altering the probabilities. However, when it considers incorporating J into $N2$ and $N3$, it finds the instance sufficiently different from both that it creates a new singleton concept ($N8$). CLASSIT stores this new node as a third child of $N1$, basing its initial counts on the values in the instance. Figure 4 shows the structure of the hierarchy after this step.

Any incremental learning system is sensitive to the order in which it encounters instances. Schlimmer and Fisher (1986) have argued that, to mitigate these effects, one should include *bidirectional* learning operators that can reverse the effects of previous learning should new instances suggest the need. Such operators give the effect of backtracking without the memory overhead required by explicitly storing previous hypotheses. To this end, CLASSIT incorporates two additional operators – *merging* and *splitting* – to recover from poor clusterings that might decrease accuracy and increase retrieval time.

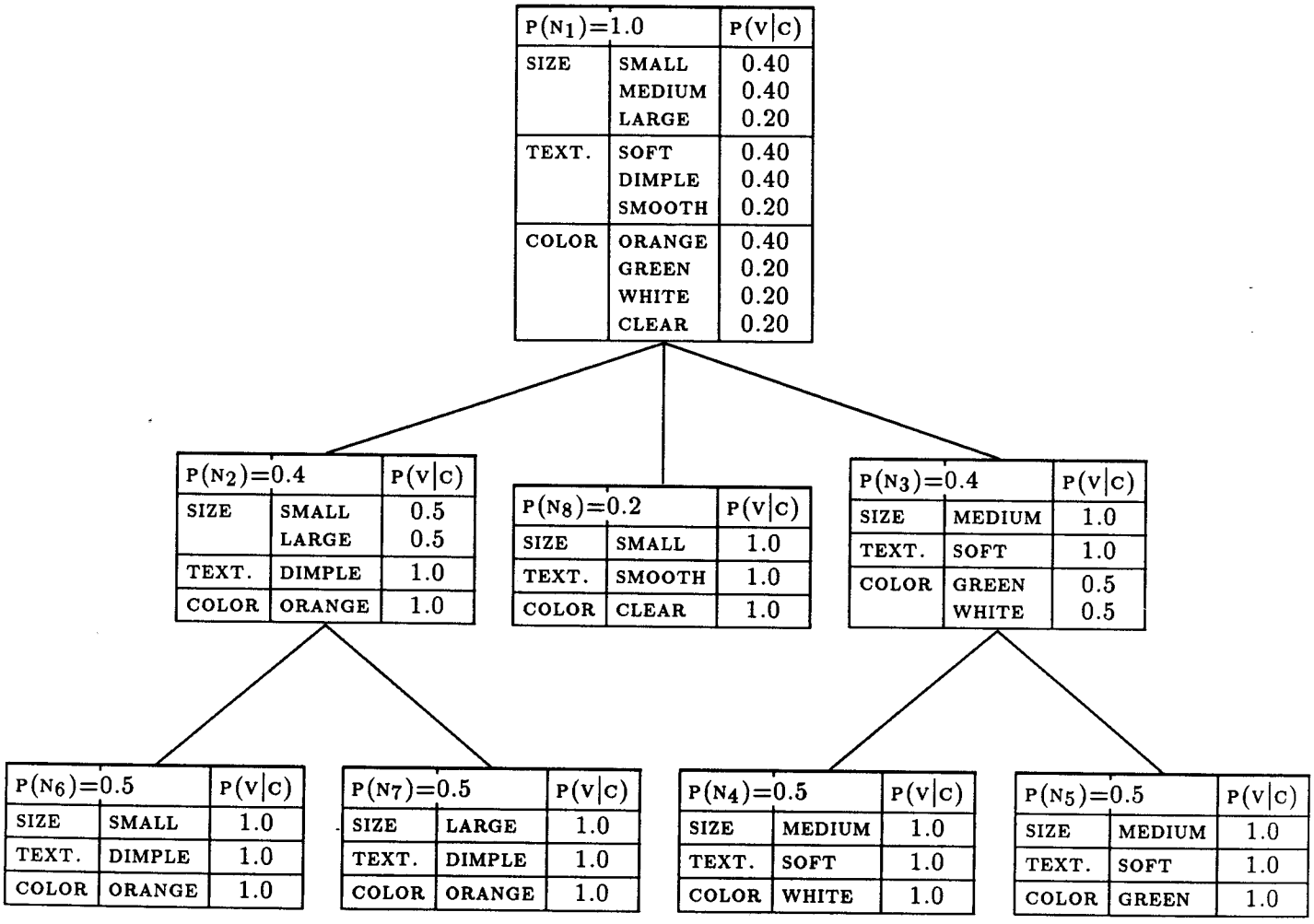

Figure 4. The concept hierarchy after a fifth ball leads to a new disjunct.

At each level of the classification process, CLASSIT considers merging the two nodes that best classify the new instance. It also considers the inverse operation (splitting), which deletes the best candidate node and promotes its children to the current level. If either of these operators gives a superior partition according to the evaluation function, CLASSIT adjusts the structure of memory accordingly. Note that all the learning operators are local in their effects, so that at each level the system considers only relevant concepts, as determined

by the classification process up to that point. Our preliminary studies suggest that the merge and split operators can improve the retrieval efficiency of acquired concept hierarchies, but the effect on accuracy remains less clear.

2.4 Category Utility

We have mentioned that CLASSIT uses an evaluation function to determine the appropriate action during classification. Since a major goal of concept formation is to let the agent categorize new experience and make predictions, the system employs *category utility* – an evaluation function that attempts to maximize predictive ability. Gluck and Corter (1985) originally derived this measure from both game theory and information theory in order to predict basic-level effects in psychological experiments, and Fisher (1987b) adapted it for use in his COBWEB model of concept formation. The measure assumes that concept descriptions are probabilistic in nature, and it favors clusterings that maximize a tradeoff between intra-class similarity and inter-class differences.

One can define category utility as the *increase* in the expected number of attribute values that can be correctly predicted, given a set of K categories, over the expected number of correct predictions without such knowledge, normalized by the size of the partition. The complete expression for category utility is

$$\frac{\sum_{k=1}^{K} P(C_k) \sum_i \sum_j P(A_i = V_{ij}|C_k)^2 - \sum_i \sum_j P(A_i = V_{ij}|C_p)^2}{K} \quad . \tag{1}$$

The first subexpression in the numerator of equation (1) represents a tradeoff between *predictability*, the ability to predict a feature given an instance of a concept, and *predictiveness*, the ability to predict a concept given a feature, summed across all classes (k), attributes (i), and values (j). The probability $P(C_k)$ weights each concept in the partition, so that frequently occurring concepts play a more important role than those occurring less frequently. The second subexpression represents the same information at the parent C_p of a partition, using the same measure without knowledge of categories in the partition. This difference is divided by the number of categories, K, to allow comparison of different size partitions.

When dealing with nominal attributes, CLASSIT uses the above expression to decide among operators when sorting an instance. However, real-valued attributes require one to restate category utility in a different form. The two innermost summations in equation (1) can be modified for real-valued attributes by changing summation to integration. Thus, the probability of a particular attribute value is the height of the curve at that value, so the summation of the square of all probabilities becomes the integral of the squared normal distribution. After discarding irrelevant constants, one arrives at a version of category utility for real-valued attributes:

$$\frac{\sum_{k}^{K} P(C_k) \sum_{i}^{I} 1/\sigma_{ik} \quad - \quad \sum_{i}^{I} 1/\sigma_{ip}}{K} \quad , \tag{2}$$

where σ_{ik} is the standard deviation for a given attribute in a given class, and σ_{ip} is the standard deviation for a given attribute in the parent node.[4] CLASSIT can also deal with mixed data, using the discrete version for nominal attributes and the continuous one for numeric features, though we have not yet tested it on such cases.

2.5 Attention in Classification

As we have described the CLASSIT algorithm, it inspects all attributes of an instance, matching them against concept descriptions in long-term memory as though in parallel. However, any agent in the physical world will have limits on its perceptual power, being able to process in detail only a few features of an object at a given time. As a result, there must be a sequential aspect to perception; in humans, this emerges as the phenomena of selective attention and eye movements.

The current version of CLASSIT differs from earlier versions (Gennari et al., 1989) in incorporating a mechanism for attention. At each level of the classification process, the system inspects only one attribute at a time, halting when it has enough information to make a decision. In ordering attributes for inspection, CLASSIT uses a 'salience' score stored at the parent of the concept nodes it is selecting among. An attribute's score is simply its contribution to the category utility measure over the entire partition; thus, the salience of attribute i over a set of K concepts is

$$\frac{\sum_{k}^{K} P(C_k) 1/\sigma_{ik} \;-\; 1/\sigma_{ip}}{K} \quad , \tag{3}$$

which can be stored at the parent for use in ordering attributes. To make the algorithm less sensitive to order effects, selection is only a probabilistic function of salience. Thus, CLASSIT occasionally samples attributes that do not appear diagnostically useful, though the chances of this on any given step are small.

In addition to ordering attributes, the system also needs a stopping criterion to determine the number of attributes it should inspect before making a decision. CLASSIT resolves this problem by imagining a worst-case scenario. At a given step, the category utility score for some concept C will be superior to that of its siblings. The system computes the scores that would result if the unobserved attributes matched one of these sibling concepts perfectly. If C would still emerge as the leader, there is no reason to inspect additional attributes, because they cannot change the outcome. If C still might lose, then the algorithm continues sampling attributes until a clear winner emerges.

---

[4] In our implementation, the attribute summations are divided by I, because instances may have missing attributes. Also, note that for any concept based on a single instance, the value of $1/\sigma$ is infinite. To resolve this problem, we have introduced the notion of *acuity*, a system parameter that specifies the minimum σ value. This limit corresponds to the notion of a 'just noticeable difference' in psychophysics – the lower limit on one's perceptual ability. This parameter indirectly controls the breadth of the trees created by CLASSIT, affecting the score of new disjuncts. See Gennari et al. (1989) for details.

Because CLASSIT expresses category utility as a sum over all attributes, this attention algorithm can be completely incremental. As the system inspects each attribute, it adds that feature's score into the current sum, avoiding the need to reprocess previously sampled attributes. Of course, an attribute may be reexamined at lower levels in the hierarchy if it is still diagnostically useful; the results of attention do not carry across levels.

This strategy leads to an interesting change in CLASSIT's behavior as a function of experience. Upon first encountering members of a general class, it has little information and all attributes appear equally salient. As a result, the system must inspect most of the attributes before assigning an instance to a subcategory. In domains with little regularity, CLASSIT never moves beyond this stage, since many attributes are needed to distinguish between classes. However, in more regular domains the system's behavior changes as it observes more instances. In this case, some attributes begin to contribute more heavily to the total category utility metric, and as the salience scores become more disparate, CLASSIT tends to inspect only those attributes with high scores. The less salient attributes can be ignored, allowing more efficient classification with little decrease in predictive accuracy (Gennari, 1989).

2.6 Measuring Performance in CLASSIT

Learning involves some change in performance, and one can measure CLASSIT's behavior on a variety of performance tasks. The simplest of these is a *recognition* task, in which the system must decide whether it has seen an instance before. One can interpret the algorithm as recognizing an instance if it decides there is no reason to store the experience as a new node in the concept hierarchy. This is effectively a rote memorization task, and CLASSIT's recognition ability should improve in domains where instances are repeated, since it can store them as terminal nodes in memory.

A more interesting task is *recall*, in which the system must fill in the values of missing attributes given the values of observed ones. This is effectively a *prediction* task, and most of our tests of CLASSIT to date have focused on this ability. Gennari et al. (1989) report evidence that the system's predictive accuracy improves over time, though the learning rate and asymptotic performance are affected both by the environment and by system parameters like acuity. These tests have focused on prediction when single attributes are omitted, and future studies should examine CLASSIT's recall ability when less information is available.

The prediction task provides some measure for the quality of learned concepts, but it does not evaluate the organization of memory. For instance, runs with CLASSIT variants lacking the merge and split operators produce quite different (and skewed) concept hierarchies, but we have observed almost no degradation in predictive capacity. A more sensible evaluation of hierarchy quality would measure *retrieval time*. One can define this measure in terms of the total number of attributes inspected during classification, which will be affected by attention, by the number of nodes examined at each level, and by the number of levels. In general, we expect CLASSIT's retrieval time to increase as a logarithmic function of the

number of instances encountered, with the constant determined by factors like attention, use of merging and splitting, and regularity of the domain.

2.7 Comments on CLASSIT

In summary, the CLASSIT algorithm sorts new instances through a concept hierarchy that represents long-term memory, changing this memory in the process. Concepts are probabilistic in nature, and the evaluation function (category utility) uses probabilities in selecting categories in which to place an instance. The learning method is incremental and fully integrated with performance. The system borrows ideas from earlier work on incremental concept formation, particularly from Fisher's (1987a) COBWEB, but it also extends this work to handle numeric attributes, to store fewer instances in memory, and to selectively attend to attributes.

An important direction for future research involves making CLASSIT more consistent with results from the psychological literature. Although the evaluation function was designed with basic-level effects in mind (Gluck & Corter, 1985), the current system does not model the reaction-time aspects of this phenomenon. However, Fisher (1988) has shown that one can extend the basic COBWEB hierarchy to allow direct indexing of concepts, and that this explains the relative retrieval times observed in humans for certain concepts. This approach also promises a more efficient retrieval mechanism, which may reduce degradation in efficiency as a function of experience. Thus, the basic approach we have taken in CLASSIT shows promise for modeling the details of human classification and concept formation, and as we show in later sections, can be extended to a variety of more complex domains.

3. Recognition and Concept Formation for Composite Objects

One can use the CLASSIT algorithm to acquire and organize concepts that can be described in terms of attribute-values, and most work on conceptual clustering and numerical taxonomy (e.g., Michalski & Stepp, 1983; Everitt, 1974) has focused on such domains. However, objects in the physical world often have a complex relational structure, and in this section we describe LABYRINTH (Thompson & Langley, 1989), a component of ICARUS that carries out incremental concept formation over *composite* objects, i.e., objects for which the attribute values may themselves be objects that can be further decomposed. We begin by describing the representation of composite objects and concepts, and then turn to LABYRINTH's performance and learning algorithms.

3.1 Representation and Organization in LABYRINTH

LABYRINTH borrows from COBWEB and CLASSIT the basic principle of probabilistic concepts organized in a disjoint hierarchy, but it extends the representation to composite instances and concepts. Each composite instance is described as a set of components that are linked to their parent by PART-OF relations. Each component may itself be a composite

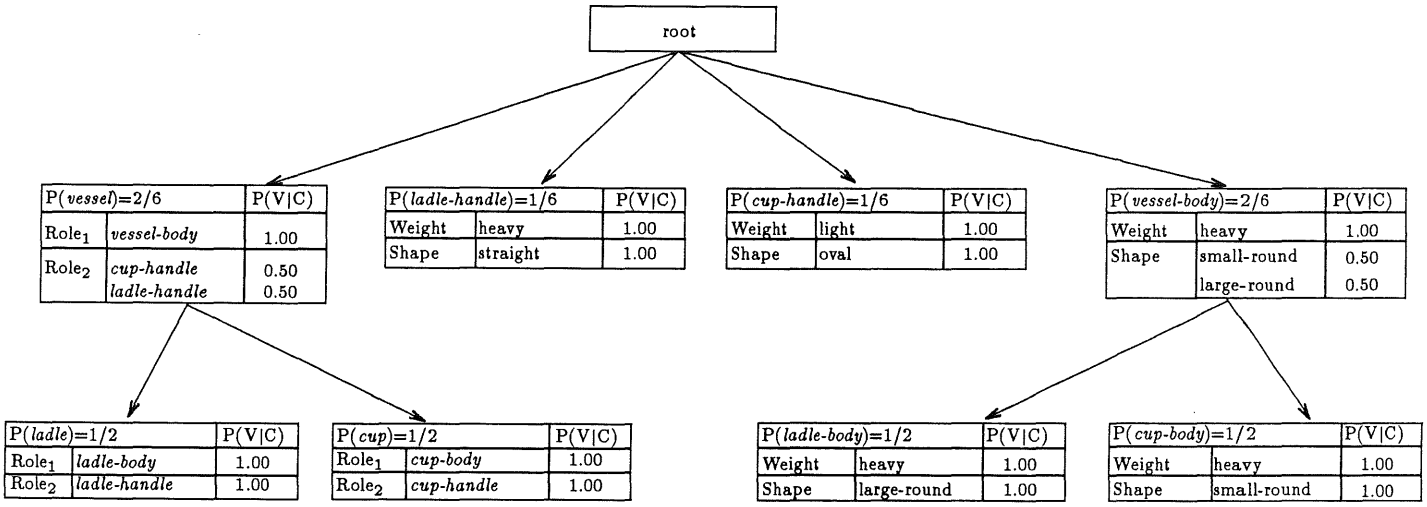

Figure 5. A portion of LABYRINTH's memory.

object, with components of its own. Simple components (the leaves of the PART-OF tree) are described with primitive attribute values, like those used by CLASSIT.

However, one can also view composite objects (nonterminal nodes in the PART-OF tree) as having attributes, whose values are component objects. Thus, we will use 'attribute' to refer both to components (in the case of composite concepts) and to descriptive features (in the case of simple concepts). Still, there is a major difference between attributes of simple objects and those of composite objects. In primitive objects, the correspondence between attributes of two instances is given in the input. However, in composite objects, the attributes are unordered, so that LABYRINTH must determine this correspondence itself.

Composite object concepts are similar to instances in that they consist of nodes connected by PART-OF links. At each level of a given concept's tree, there exists a set of associated probabilistic attributes; however, these attributes represent components rather than observable attributes, except at the lowest level. The 'values' associated with these 'attributes' refer to other nodes in the concept hierarchy, giving an interleaved memory structure that, as we discuss in Section 6.1, is similar to that proposed in Schank's (1982) theory of dynamic memory.

For example, Figure 5 presents a partial LABYRINTH hierarchy containing three composite concepts and five simple concepts. The values of each composite concept refer to simple concepts, which are represented exactly as in CLASSIT. One can view each composite attribute as specifying a *role* that can be filled by its possible values (i.e., components). In some cases, an attribute takes on a single value that corresponds to an abstract concept high in the hierarchy. For instance, the first role of VESSEL has the single value VESSEL-BODY. This has two more specific children – CUP-BODY and LADLE-BODY – which have similar features and which occupy the analogous role in the composite concepts CUP and LADLE.

Table 2. The basic LABYRINTH algorithm.

```
Input: O is a composite object.
       S is the set of O's simple components.
       R is the root node of the concept (is-a) hierarchy.
Side effects: Labels O and all its components with class names.

Procedure Labyrinth(O, S, R)
  For each simple component A of composite object O,
      Let C be Classit(A, R).
      Return Labyrinth'(O, A, C, R).

Procedure Labyrinth'(O, A, C, R)
  Label object A as an instance of category C.
  If A is not the top-level object O,
     Then let B be the composite object of which A is a component.
          If all components of B are labeled,
             Then let D be Classit'(B, R).
                  Return Labyrinth'(O, B, D, R).

Note: Classit' is a variant of Classit that considers different mappings between
      components of the instance and components of the concept, selecting the best
      according to category utility.
```

However, LABYRINTH can also represent cases in which the objects that can fill a given role are quite different. For instance, the second role of VESSEL specifies two possible values – CUP-HANDLE and LADLE-HANDLE. Because these two concepts have quite different features, the root node is their only common parent in the concept hierarchy. Nevertheless, the handle of a cup and the handle of a ladle occupy the same role in their respective composite concepts, so they are both included as values in the VESSEL concept. Such sets of concepts are very similar to the *internal disjuncts* that occur in some inductive approaches to learning (e.g., Michalski, 1983).

3.2 Classification and Learning with Composite Concepts

Now that we have described LABYRINTH's memory structures, we can demonstrate its behavior on a simple two-level instance of a CUP with the components HANDLE and BODY, each described in terms of primitive features. As specified in Table 2, the system processes composite objects in a "component-first" style, classifying first the primitive objects and then the composites. For instances involving more than two levels, the process can be extended indefinitely by proceeding until it classifies all the composites contained in the instance, including the instance itself.

In our example, the system first classifies the handle based on its primitive attribute-value description, labeling the component object as an instance of the concept CUP-HANDLE. The same procedure leads the system to label the body as an instance of CUP-BODY. When both components of the overall object have been labeled, the composite instance has been

transformed so that its attributes' values are no longer simple components, but labels of concepts in memory. The CLASSIT′ subroutine (described below) treats these labels as simple nominal values, letting it classify the composite instance as if it were a simple instance. In this case, LABYRINTH labels the top-level object as a member of the composite concept CUP. The result is that each sub-tree of the instance is classified and labeled, starting with the simple components and ending with the entire instance.

To classify composite instances, CLASSIT′ extends CLASSIT in two ways, each of which resolves complications in the process of incorporating an object into a concept. As we noted earlier, the nonterminal attributes of a composite object are unlabeled, necessitating an extra search to 'map' attributes in the instance to those in the composite concept. For example, given a new instance of the VESSEL concept (say a BUCKET), the system must determine which component (the body) should occupy the first role and which component (the handle) should occupy the second. To determine the best mapping, CLASSIT′ employs a heuristic method, using a simplified form of category utility to estimate the predictiveness of a single concept. To determine the overall score of incorporating an instance into a concept, it first generates all possible mappings of components into roles and determines which mapping is best. It then uses the best score in comparing this action with other alternatives.

In addition, CLASSIT′ uses a new operator, *attribute generalization*, to avoid composite concepts with overly-specific labels on their attributes. Upon incorporating a new composite instance (e.g., incorporating the instance into the concept VESSEL), LABYRINTH must evaluate whether the attributes in the updated concept description should simply add the label from the current instance to the attribute-value list (e.g., add CUP-HANDLE to LADLE-HANDLE in ROLE_2), or instead point to a more general common ancestor in the concept hierarchy (e.g., to VESSEL-BODY instead of the internal disjunct <CUP-BODY, LADLE-BODY> in ROLE_1). Each time CLASSIT′ incorporates an instance into an existing composite concept, it considers both of these options, iterating over all attributes and their values to determine which action to take; Fisher (personal communication, 1989) has proposed a metric to evaluate the quality of alternatives.

This decision involves a choice between extending an internal disjunct on the basis of functional roles and generalizing on the basis of common structure. The second option corresponds to learning with *structured attributes* through climbing a generalization tree (Michalski, 1983). However, LABYRINTH differs from most earlier approaches in that it is constantly revising the structure of these 'attributes'. This results from the fact that the descriptions of composite concepts refer to other nodes in the concept hierarchy, which it defines itself. In effect, LABYRINTH is *dynamically changing* the representation used to describe composite concepts. This suggests that, to the extent component concepts are shared across many composite concepts, the system should learn more rapidly than one without this ability. Testing this prediction will be a priority in our future work.

3.3 Comments on LABYRINTH

To date, most research on concept formation has focused on attribute-value representations. However, some work has dealt with unsupervised learning in more complex languages. For instance, Stepp and Michalski's (1986) CLUSTER/S incorporates background knowledge and the "goal" of classification into the learning process. However, their clustering method appears to be inherently nonincremental, and seems limited to logical, 'all-or-none' concepts. Wasserman's (1985) MERGE incrementally forms a concept hierarchy from structured instances similar to those used by LABYRINTH, but it shares many of the *ad hoc* mechanisms and arbitrary thresholds of its predecessor, UNIMEM.

Fisher (personal communication, 1989) has recently proposed an alternative adaptation of CLASSIT to learn composite instances. The most important difference involves the classification strategy, which starts by attempting to classify the overall object, then its components, and finally the simple objects. Feigenbaum's (1963) EPAM used a similar strategy to classify composite instances. Of course, this approach does not actually label an object until its components have been labeled, any more than does the LABYRINTH scheme. However, it relies on knowing at the outset the mapping between components of the instance and components of the concepts. In contrast, our approach attempts to determine the best mapping dynamically at each level. We predict that this method will prove more efficient and more general, but these are empirical questions that remain to be tested.

We plan to evaluate LABYRINTH along the same dimensions that we are studying with CLASSIT, notably predictive accuracy and retrieval time. However, the system's ability to handle composite objects suggest some additional tests. For example, LABYRINTH should be able to predict not only missing attributes, but also missing components and even missing structure. We will test LABYRINTH on its ability to acquire composite concepts with different numbers of levels and with different amounts of redundancy (e.g., how often a component concept is used in composite concepts). We also plan to examine order effects, not only in terms of instances, but in terms of the order in which one examines simple components.

The current version of LABYRINTH can be extended in a variety of ways. One direction involves adding relational (multi-argument) predicates to the description language, using the attribute-mapping process to constrain search. In this scheme, one would treat each relational descriptor as a separate 'feature' with an associated conditional probability. We also plan to incorporate CLASSIT's mechanism for selective attention into LABYRINTH, which would would let the system inspect components serially. Third, we hope to explore alternative search schemes that combine the current bottom-up approach with the top-down method used by EPAM; each process tells only part of the story. Finally, the existing system was designed for comparing objects with similar structures, and we plan to examine ways to classify and organize objects with significantly different structures, making the approach more relevant to real-world objects. Taken together, these extensions should make LABYRINTH a robust framework for the recognition, prediction, and formation of structured concepts.

4. The Generation and Acquisition of Plans

In order to achieve its goals, an intelligent agent must be able *plan*; that is, to order its actions in the world. We can briefly state the planning task as:

- *Given*: A goal to transform an initial world state I into a desired state D;
- *Given*: A set of primitive operators O that let one directly transform states of the world;
- *Find*: A sequence of operator instances from O that will transform I into D.

This formulation makes a number of assumptions. First, one must have some explicit description of the desired state, even though this may be only partially specified. It also assumes that the planning task involves a single agent, making timing less important than in multi-agent planning tasks. Third, it makes the closed-world assumption – that all aspects of the world remain constant unless the agent applies some operator. Finally, it assumes that one can separate the process of plan generation and plan execution.[5]

4.1 Approaches to Planning

One can identify three distinct paradigms within the AI planning literature. The earliest approach uses weak, domain-independent methods like means-ends analysis (e.g., Newell, Shaw, & Simon, 1960; Fikes, Hart, & Nilsson, 1971) to select relevant operators and create subgoals.[6] A second framework incorporates domain-specific goal decompositions or schemas, which specify useful orders on operators or useful subgoals (e.g., Bresina, 1988). A third approach – case-based reasoning – retrieves specific plans from memory and uses them to constrain the planning process.

Because one cannot predict interactions among operators, planning in novel domains may require *search*. This makes many planning methods impractical for use in controlling real-time robotic agents (Georgeff, 1987). One natural response is to employ machine learning techniques to acquire domain-specific planning knowledge, and to use this knowledge to reduce or eliminate search on future problems. Researchers have applied machine learning techniques to all three of the planning paradigms described above. For instance, Minton (1988) has studied learning within a means-ends planner, DeJong and Mooney (1986) have used a schema-based method, and Hammond (1986) and Kolodner (1987) have examined case-based approaches.

In this section, we describe the planning component of ICARUS – as implemented in a system called DÆDALUS – which views these paradigms as 'stages' in the development of planning expertise. The system begins with knowledge of the operators for a domain and, like Minton's (1988) PRODIGY, uses means-ends analysis to construct plans. However,

[5] In future work we hope to loosen these assumptions, and to address the generation and acquisition of nonlinear plans (Mooney, 1988).

[6] More recent work in this tradition has invoked more powerful (and more expensive) search methods (Wilkins, 1982), but has continued to use general, domain-independent techniques.

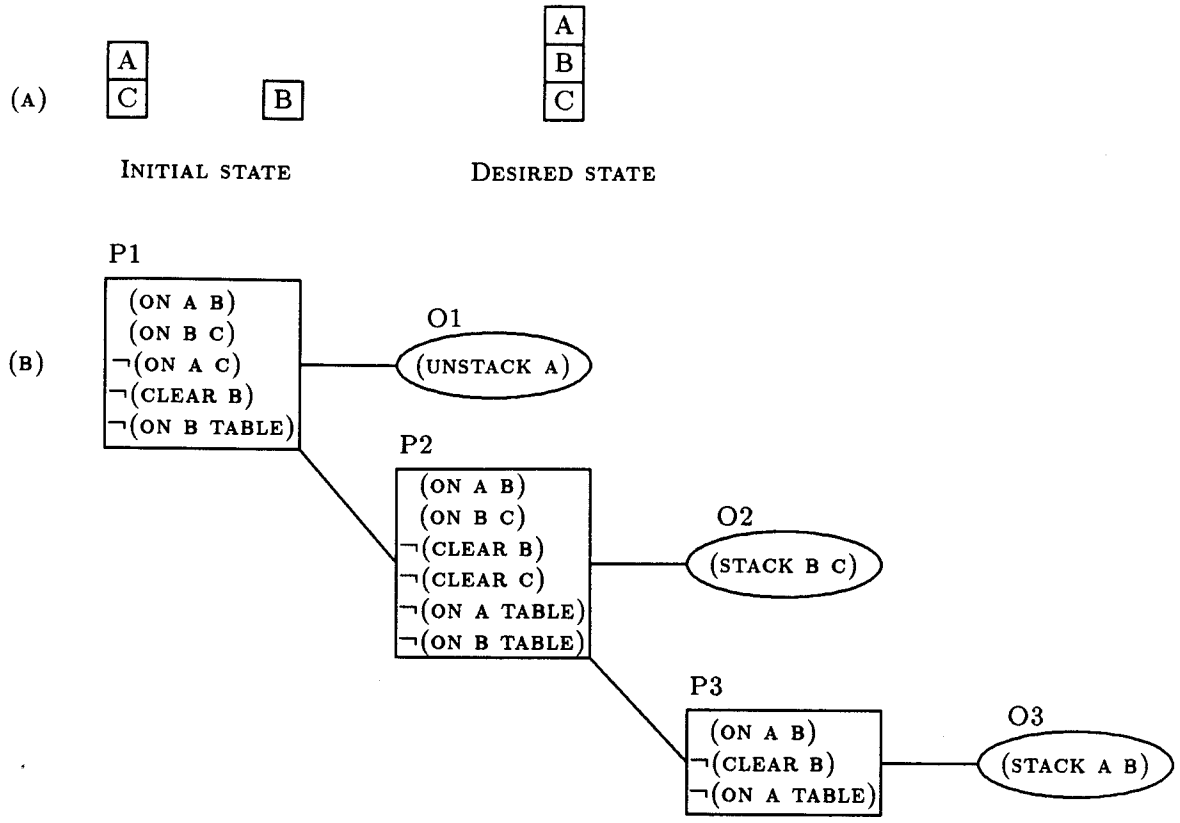

Figure 6. A DÆDALUS plan generated with initial knowledge of operators.

DÆDALUS stores these plans (cases) in a probabilistic concept hierarchy, describing them by the differences they reduce. Upon encountering a previously unseen problem, the system retrieves a relevant plan (one with analogous differences) and uses it to select operators for the new task. The retrieval process leads DÆDALUS to generalize its stored plans so that it gradually shifts from a case-based mode to one using abstractions, while still retaining the ability to employ means-ends analysis when necessary. Below we describe DÆDALUS' representation and organization of plans, its performance and learning components, and its overall behavior.

4.2 Representing States, Problems, and Operators

DÆDALUS acts on data structures of three types: states, problems, and operators. In general, a *state* consists of some description of the world, possibly including the internal state of the agent. States can be viewed as composite objects of the type processed by LABYRINTH, as described in Section 3. For our examples of planning, we will use a simple STRIPS-like state representation (Fikes et al., 1971), with each state described as a set of objects and symbolic relations that hold among them.

A *problem* consists of an initial state and a desired state that the agent wants to achieve. Each state may contain only *partial* descriptions of the world. For instance, Figure 6 (a) presents a graphical description of a simple problem in the blocks-world domain. One can also describe a problem in terms of the *differences* between the initial and desired state. Node P1 in Figure 6 (b) summarizes the problem in this manner. The notion of representing problems as differences is central to our approach.

Most work on planning assumes that operators have known preconditions and effects, and that they should be reasonably efficient (i.e., require no search). To this end, DÆDALUS assumes that operators correspond to compiled motor skills, which we discuss in Section 5. For the purposes of planning, this means that one can treat operators as 'black boxes', describing them in terms of preconditions and postconditions. However, from this information one can derive a set of *differences* that exist between states before and after application, giving a description similar to that used for problems.

4.3 The Organization of Plan Memory

DÆDALUS uses the notion of differences to organize its memory for planning knowledge. As in CLASSIT and LABYRINTH, long-term memory takes the form of a probabilistic concept hierarchy. Initially, the terminal nodes in this hierarchy consist only of general operators described in terms of the differences between their initial and final states. Internal nodes correspond to *classes* of operators that have some overlap in their difference descriptions. Essentially, this hierarchy is an efficiently organized difference table (Newell et al., 1960) that changes with experience.

Figure 7 (a) presents an initial difference hierarchy for the blocks-world domain. Each node has an associated name N (top), a set of associated differences D_i (center), and a set of one or more associated operators (bottom). Moreover, each node has a certain probability of occurrence $P(N)$, and each difference has a conditional probability $P(D_i|N)$ of occurrence given the concept, as does each operator. For clarity, Figure 7 (b) shows a traditional STRIPS representation for each operator. Thus, node `N3` depicts DÆDALUS's summary description for the operator `(stack ?x ?y)`, which has preconditions `(clear ?x)` `(clear ?y)` `(on ?x table)`, where question marks indicate pattern-match variables. Its actions include adding `(on ?x ?y)` and deleting `(on ?x table)` `(clear ?y)`. Thus, it can be summarized by the set of differences `(on ?x ?y)` `¬(clear ?y)` `¬(on ?x table)`, as shown in the terminal node `N3`. The 'hierarchy' in this figure has only two levels, but domains involving more actions would contain internal nodes that index and summarize subsets of the operators.

The system represents a *plan* for solving a particular problem in terms of a derivational trace (Carbonell, 1986) that states the reasons for each step in the operator sequence. This trace consists of a binary tree of problems and subproblems, with the original task as the top node and with trivial (one-step) subproblems as the terminal nodes. Each node (problem) in this derivational trace is described by differences between its initial and final state, along with the operator instance that was selected to transform one into the other.

(A)

P(N1) = 1.0	P(D|C)
¬(ON ?X ?Y)	0.67
(CLEAR ?Y)	0.67
(ON ?X ?Z)	0.67
¬(CLEAR ?Z)	0.67
¬(ON ?X TABLE)	0.33
(ON ?X TABLE)	0.33
(PUTON ?X ?Z)	0.33
(STACK ?X ?Z)	0.33
(UNSTACK ?X)	0.33

P(N2) = 0.33	P(D|C)
¬(ON ?X ?Y)	1.0
(CLEAR ?Y)	1.0
(ON ?X ?Z)	1.0
¬(CLEAR ?Z)	1.0
(PUTON ?X ?Z)	1.0

P(N3) = 0.33	P(D|C)
(ON ?X ?Y)	1.0
¬(CLEAR ?Y)	1.0
¬(ON ?X TABLE)	1.0
(STACK ?X ?Y)	1.0

P(N4) = 0.33	P(D|C)
¬(ON ?X ?Y)	1.0
(CLEAR ?Y)	1.0
(ON ?X TABLE)	1.0
(UNSTACK ?X)	1.0

(B)

NAME:	(PUTON ?X ?Z)
CONDS:	(ON ?X ?Y)
	(CLEAR ?X)
	(CLEAR ?Z)
ADD:	(ON ?X ?Z)
	(CLEAR ?Y)
DELETE:	(ON ?X ?Y)
	(CLEAR ?Z)

NAME:	(STACK ?X ?Y)
CONDS:	(CLEAR ?X)
	(CLEAR ?Y)
	(ON ?X TABLE)
ADD:	(ON ?X ?Y)
DELETE:	(ON ?X TABLE)
	(CLEAR ?Y)

NAME:	(UNSTACK ?X ?Y)
CONDS:	(ON ?X ?Y)
	(CLEAR ?X)
ADD:	(ON ?X TABLE)
	(CLEAR ?Y)
DELETE:	(ON ?X ?Y)

Figure 7. An initial difference hierarchy (A) that organizes three operators (B).

Figure 6 (b) presents a simple derivational trace for the task in Figure 6 (a) that involves three problems and three operators. This trace only branches downward, since each operator can be applied as soon as it is selected. However, the simple structure does not mean the problem is trivial. The first time it attempts this task, DÆDALUS selects `(stack a b)` and `(puton a b)` in preference to `(unstack a)`, considering the latter only after its first two choices lead to failed plans that involve loops.[7]

DÆDALUS uses its 'concept' hierarchy to store information about the problems and subproblems it has encountered, along with the operators that led to their successful solution. Because each problem can be described as a set of differences, they can be stored in the

[7] Note that this trace contains no information about failed operators, though future versions of DÆDALUS may retain results of this sort as well. We will draw traces from left to right, to distinguish them from concept hierarchies.

same format as the original operators. In this case, one interprets nodes in the hierarchy as problems DÆDALUS has solved, and the associated operators as the ones that led the system to a solution. Thus, nodes correspond to probabilistic 'selection rules' for deciding among operators, but the program does not retain the derivational trace itself in memory. Figure 8 presents a modified version of the difference hierarchy from Figure 7, after the system has incorporated the problem-operator pairs from the derivational trace in Figure 6.

4.4 Using and Acquiring Plan Knowledge

As shown in Table 3, the planning component of DÆDALUS uses a variant of means-ends analysis (Newell et al., 1960). In this framework, solving a problem (transforming a current state into a desired one) involves the recursive generation of subproblems. The standard means-ends approach determines all differences between the current and desired state, selects the most important difference (using some predefined criteria), and then retrieves an operator that reduces the difference. If the selected operator cannot be applied, a subproblem is generated to change the current state into one that satisfies the operator's preconditions. Applying the operator produces a new state, along with a new subproblem to transform this into the desired state; the algorithm is then called recursively to solve this task.

DÆDALUS differs from most means-ends planners in the way it retrieves operators from memory. First the system computes all differences between the current and goal states. It then uses a variant on CLASSIT to sort the difference structure, D, down the difference hierarchy, looking for the best match between the differences of D and the differences stored in the hierarchy.[8] DÆDALUS selects the operator associated with the difference node retrieved through this process. Should this operator lead to an unsuccessful plan (e.g., if its preconditions cannot be achieved), DÆDALUS backtracks, retrieving the operator with next best match, and continues.

This strategy also differs from earlier methods in placing an *ordering* on operators, rather than dividing them into relevant and irrelevant sets. One result is that it prefers operators that reduce multiple differences in the current problem, which should make it more selective than traditional techniques. More important, although DÆDALUS prefers operators that reduce problem differences, it is not restricted to this set. If none of the 'relevant' operators are successful, it falls back on operators that match none of the current differences. This gives it the potential to break out of impasses that can occur on 'trick problems'.

DÆDALUS integrates learning into its planning process, using the derivational traces described above. Whenever it finds a plan that achieves a problem or subproblem, it stores the description of that problem in its concept hierarchy. This involves storing the problem description (the differences and the selected operator) as a new terminal node (case) in the

[8] This variant treats each difference as a separate feature that takes on the value PRESENT or ABSENT. (For simplicity, we have shown only the probabilities for the PRESENT value.) The current version computes all maximal partial matches between the differences at a node and those in an instance, computes a simplified version of category utility for each match, and selects the one with the highest score. One can imagine more efficient greedy and attentional approaches to this matching problem, but we have not implemented them.

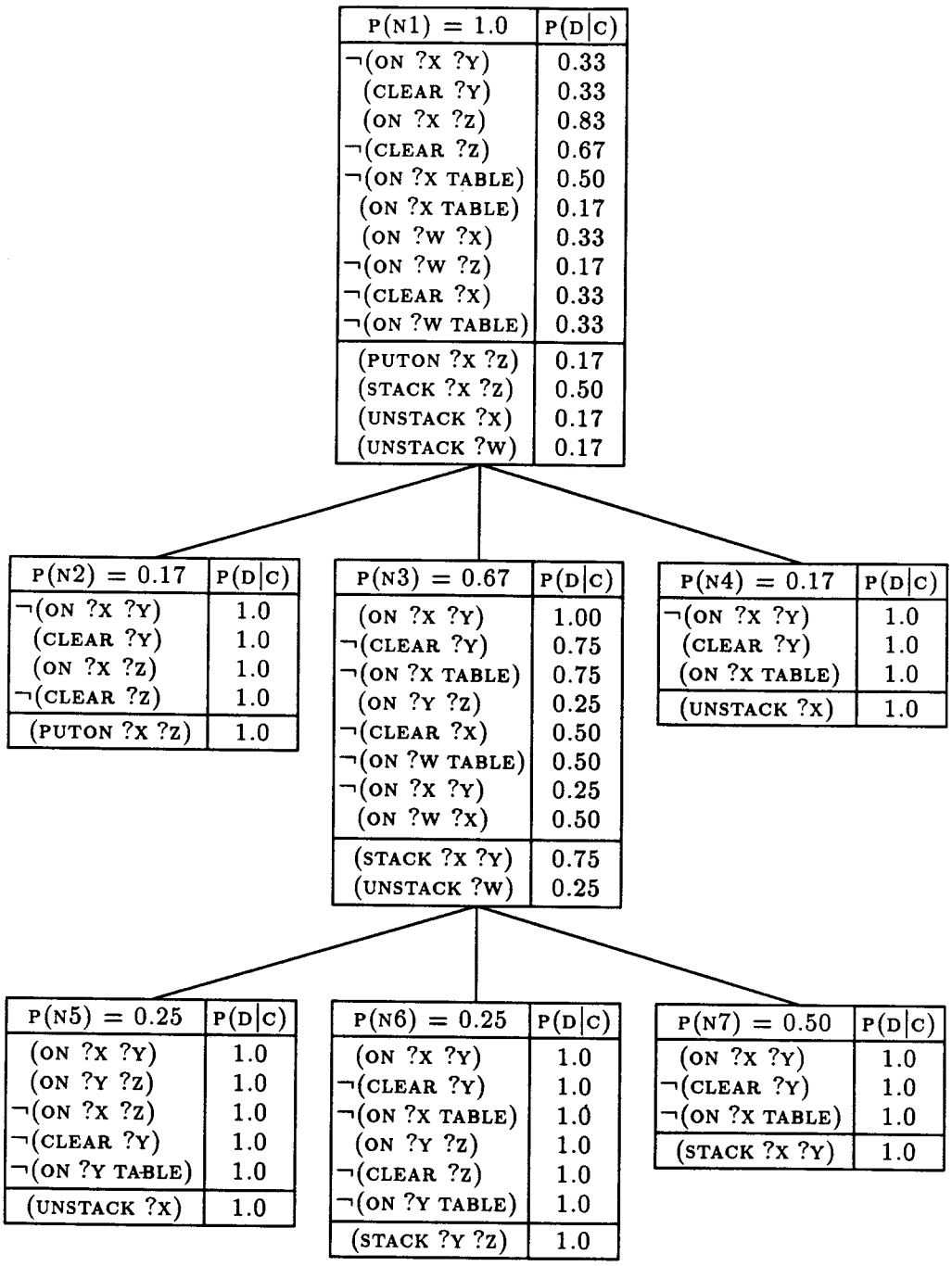

Figure 8. A revised difference hierarchy that incorporates the problem-operator pairs from Figure 7.

hierarchy, making it a sibling of the node that was first retrieved. In addition, DÆDALUS updates the summary descriptions of the nodes (indices) by revising the probabilities on all nodes along the path that the problem was originally sorted. The system invokes this process for each subproblem as it is solved, effectively storing (and indexing) a 'selection rule' (Minton, 1988) describing the operator to use for that problem.

Table 3. DÆDALUS' means-ends planning algorithm.

```
Inputs:  STATE is a (partially described) initial state.
         GOAL is a (partially described) desired state.
Outputs: A final state that matches the description of GOAL.
Side effects: A modified concept hierarchy that includes this problem.

Procedure Transform(STATE, GOAL)

  If STATE matches GOAL,
     Then return STATE.
     Else let L be the null list.
          Let D be the differences between STATE and GOAL.
          Repeat until FLAG = true.
              Let FLAG be fail.
              Retrieve the operator O that best matches differences D and
                                  that is not a member of L.
              If O ≠ fail,
                 Then let L be Insert O into L.
                      Let NEW be Apply(O, STATE).
                      If NEW ≠ fail,
                         Then let NSTATE be Transform(NEW, GOAL).
                              If NSTATE ≠ fail,
                                 Then let FLAG be true.
                 Else let FLAG be true.
          If O = fail,
             Then return fail.
             Else incorporate difference-operator pair (D, O) into memory.
                  Return NSTATE.

Procedure Apply(O, STATE)

  Let C be the preconditions on operator O.
  Let R be the results of operator O.
  If R is pathological (e.g., if R has been seen during this problem),
     Then return fail.
     Else if STATE does not match C,
             Then let NEW be Transform(STATE, C) and return NEW.
             Else return R.
```

Upon encountering a new problem, DÆDALUS uses its memory of past successes to select operators in a more discriminating fashion. Specific problems (described by differences and operators) are stored in the same concept hierarchy as the original operators, and the same sorting process is used to retrieve them. If a stored case matches a new problem or subproblem more closely (according to category utility) than one of the original operator descriptions (because it has more differences in common), DÆDALUS retrieves this case and attempts to apply the associated operator. In some situations, a problem may be sufficiently unique that the system does not sort it all the way down to a terminal node, instead using a

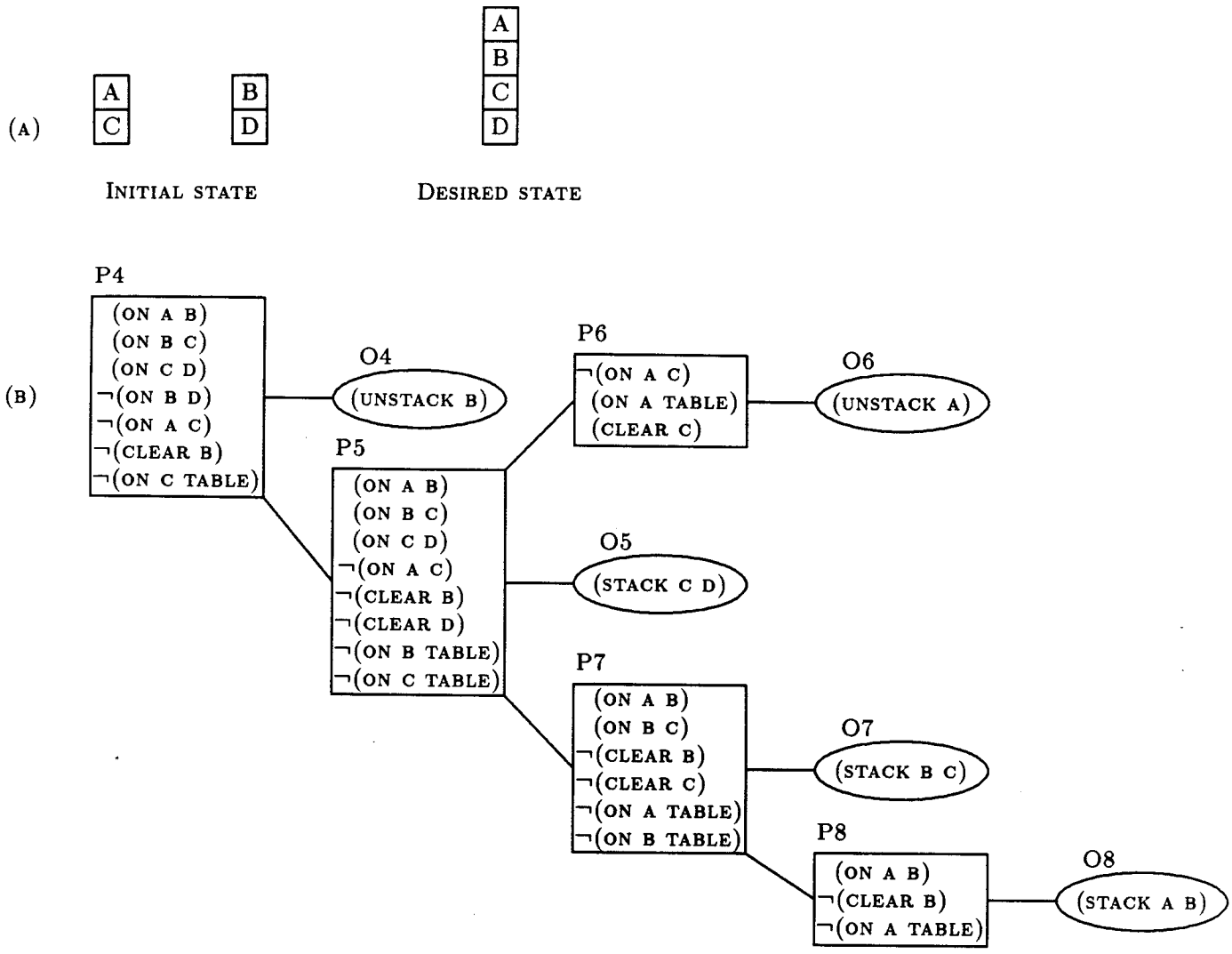

Figure 9. A DÆDALUS plan guided by knowledge stored in the difference hierarchy.

more abstract node. This retrieved problem description may specify more than one operator; in such cases, DÆDALUS selects the operator with the highest conditional probability.

For example, given the top-level problem P4 in Figure 9 (a), the system sorts its difference description through the hierarchy in Figure 8. In this case, the instance passes from node N1 through node N2, eventually reaching the terminal node N6, which describes the top-level problem P1 from Figure 6. DÆDALUS retrieves the operator associated with the case – giving the instantiation `(unstack b)` – and attempts to apply it to the current state. Without knowledge of the right operator to apply for this case, the system would have selected `stack` or `puton`, searching down faulty paths and being forced to backtrack, as it was on the earlier problem. Instead, the system selects the right operator at the outset.

The remainder of the example is less interesting. Since the preconditions of `(unstack b)` are met, DÆDALUS applies the operator and creates subproblem P5 to transform the resulting state into the desired one. The system sorts the differences for P5 through memory,

this time retrieving `(stack c d)` as the preferred action. However, the preconditions of this operator instance are not met, so DÆDALUS first attempts to generate a state that satisfies them, giving the new subproblem P6. In this case it selects `(unstack a)`, which solves P6 in a single step and lets the system apply `(stack c d)`. This process continues, with the program selecting `(stack b c)` and then `(stack a b)`, both of which can be applied without additional subproblems. The last application produces a state with no differences from the desired state, so DÆDALUS halts, having solved the top-level problem P4. For the sake of simplicity, we have not shown how the system updates its difference hierarchy after solving this problem.

4.5 The Behavior of DÆDALUS

Our approach to planning in DÆDALUS bears similarities to certain work in case-based reasoning. Like Kolodner's (1987) JULIA and Hammond's (1986) CHEF, our system stores specific cases and retrieves them when solving new problems. However, our approach to organizing plan memory and indexing cases is significantly different from their methods, focusing on the differences occurring in each problem. DÆDALUS is most similar to Veloso and Carbonell's (1988) approach to derivational analogy, sharing the notion of derivational traces and a means-ends planner. However, our system organizes plan knowledge into a probabilistic concept hierarchy, whereas their work to date has not focused on issues of indexing and retrieval. Finally, our approach only retains knowledge of successful plans and does not store failed cases.

DÆDALUS also differs from all three systems in another way. Rather than storing cases as monolithic data structures, it stores only the operator selected for each problem or subproblem. This is similar to the use of preference rules in Laird, Rosenbloom, and Newell's (1986) SOAR and in Minton's (1988) PRODIGY. This means DÆDALUS retains no memory of the relation between problems and their subproblems, and it must sort each new subproblem through memory even if a similar problem-subproblem pair has previously occurred. Thus, the system cannot retrieve entire plans from memory, as in JULIA and CHEF, but it can effectively regenerate them using the difference-operator cases stored in memory. Laird et al. (1986) have argued that SOAR's distributed knowledge structures lead to greater transfer than storing macro-operators or entire cases, and we expect this to hold for DÆDALUS as well.

As noted earlier, one emergent effect of our approach to learning should be a three-stage development of planning expertise. Initially, DÆDALUS has access only to the domain operators stored in its concept hierarchy. As a result, it will sometimes select a poor operator and be forced to backtrack. In this stage, it behaves much like any knowledge-lean means-ends planning system. However, as DÆDALUS gains experience in the domain, it stores specific cases that specify useful operators and the situations in which they should be applied. In this stage, the system will behave like a case-based planner, retrieving particular problems it has solved in the past as a guide to its actions on new tasks.

As it gains more experience in a domain, DÆDALUS will begin to encounter problems that are similar to earlier ones. When this occurs, the system does not bother to store the new problem by extending its concept hierarchy downward to retain the new instance. Instead, it simply 'averages' the new case into the existing problem description, changing its probabilities and increasing its chance of being retrieved in the future. Gradually, many of the terminal nodes will 'blur together' previous experiences, and DÆDALUS will move from a case-based reasoning mode into a one relying on abstractions. However, at each stage in its development, the system can employ earlier stages when necessary. If a case is unusual, it will be retained as a separate terminal node in memory and will be retrieved when needed. Similarly, if DÆDALUS encounters a truly unfamiliar problem, it still has access to its original operators and can solve that problem using means-ends analysis and search.

4.6 Comments on DÆDALUS

We plan to test DÆDALUS's learning ability using two performance measures – optimality of the generated plans and the amount of search during the planning process. We expect that retrieval of previous cases will let DÆDALUS avoid operator-selection errors that it made on previous runs, and that with experience its search will become much more selective than that of a non-learning means-ends planner. However, Minton (1988) has demonstrated that in some cases the cost of matching preference rules can exceed their benefit, so we must address this issue in our experiments. In particular, we predict that DÆDALUS' indexing scheme will be very efficient, increasing in cost only logarithmically with the number of stored problems and subproblems.[9] In addition, the system will be selective in the cases it stores, retaining only those that are sufficiently different (according to category utility) from the retrieved case.

We also hope to explore variants and extensions to the basic approach, implementing a version that stores entire plans (derivational traces) rather than problem-operator combinations. As in the current approach, DÆDALUS would retrieve a previous problem based on its difference description. However, rather than sorting new subproblems through the difference hierarchy, the system would simply use subproblems associated with the retrieved plan, checking to make sure they are still necessary. In some cases, the new problem will require additional steps, forcing DÆDALUS to generate novel subproblems and sort them through memory. In these ways, the system could adapt a stored plan to a similar but slightly different problem. This approach would require us to integrate DÆDALUS with LABYRINTH, because the latter would be needed to support the retrieval and update of structures like derivational traces.

[9] Testing this claim would involve comparing the total number of nodes visited and the differences inspected during the traversal of memory during planning in both learning and non-learning versions of the system.

An ideal intelligent agent learns not only from successes, but also from failures. The current version of DÆDALUS learns from failures in the sense that it stores operators found during backtracking search, but it retains no information about operators retrieved earlier in that search which led to failed paths. We hope to augment the system in this manner, storing with difference nodes the operators that one should *not* select. As with the rejection rules in SOAR and PRODIGY, this knowledge should help constrain search by eliminating possible candidates even when no positive advice is available. However, it is not clear whether this information should be stored in the same hierarchy as the selection rules or in a separate one entirely.

In future work, we hope to implement a tight coupling between planning and motor control, which we describe in the following section. We currently treat operators as compiled motor skills that one can execute without creating subgoals, but the dividing line between planning and motor behavior should be a function of the agent's goals and experience. Presumably, DÆDALUS would come to a new domain with only a few general operators, but practice in the domain would lead to new domain-specific motor schemas that augment the repertoire of operators available for planning. We describe one method for acquiring such motor skills in the next section. We also hope to incorporate an 'automatization' process (Schneider & Fisk, 1983) that gradually transforms planning expertise into compiled motor skills with practice, but our ideas on this mechanism remain somewhat vague.

Although our examples to date have focused on STRIPS-like symbolic states and operators, we hope to extend DÆDALUS to handle more realistic, numeric descriptions of the physical world. Although the same basic planning and learning methods should apply, one important issue remains to be resolved. In means-ends analysis, a problem is solved only when no differences remain between the current and desired state. This decision is clear-cut in a STRIPS framework, but given real-valued state descriptions, one must use some form of partial-matching scheme to decide when the remaining differences are insignificant. We hope to use some principled metric like category utility to address the issue, but the details remain open.

Finally, we plan to incorporate a *priority queue* into DÆDALUS, which would provide a more realistic model of attention in planning and let the system handle multiple independent goals. Problems could enter this queue from high-level drives (e.g., hunger or sleep), from the planner as subproblems, or from an execution monitoring component when a plan fails. Each problem would be ordered by its associated priority, which generally decreases over time, so that old problems would be gradually forgotten. If a new problem were passed to the queue with a higher priority than the currently active one, the current problem would be set aside in lieu of the more important one. The extended system would work on this task until it was solved or until another problem (possibly the original one) becomes more important. In summary, DÆDALUS provides a fertile framework within which to formulate and test our ideas about the relations among memory, planning, and learning.

5. The Execution and Improvement of Motor Skills

Planning techniques like those used in DÆDALUS seem well-suited to solving high-level problems, but they halt at the level of primitive operators. In order to interact with a physical environment, one requires some way to represent and execute such operators. We can state this task of motor control as:

- *Given*: A goal to transform an initial world state I into a desired state D, obeying path constraints P;
- *Given*: A primitive operator O that lets one directly transform states of the world;
- *Generate*: A *motor program*, specifying the locations, velocities, or forces for body parts, over time, that achieves this goal.

As in other areas, one can also define a related learning task, which involves improvement in the execution of motor skills. In this section, we examine MÆANDER, the component of ICARUS that deals with issues of motor behavior.[10] As before, we describe its representation and organization of memory, then turn to its performance and learning components.

Very little AI research has focused on the generation and improvement of motor skills. 'Classical' planning systems (e.g., Sacerdoti 1977; Segre, 1987) have focused on generating a sequence of abstract operators described in terms of high-level preconditions and effects, without reference to how they accomplish these effects. Research in robotics deals with issues of grounding desired actions in appropriate torques and voltages (e.g., Swartz, 1984), and some work in AI has attempted to integrate high-level planning with agents in the world (e.g., Nilsson, 1984; Laird et al., in press). However, research on motor behavior has generally avoided issues of skill acquisition and improvement. As we will see, MÆANDER's approach to motor behavior operates at a lower level than the symbolic descriptions like `(puton a b)` that are used in most planning systems, but it operates at a higher level than the actions used in robots like SHAKEY (Nilsson, 1984). In addition, our approach differs from most robotics work in its emphasis on the role of knowledge in motor control.

5.1 Representation and Organization of Motor Skills

Our approach to motor behavior assumes that the agent has knowledge of its limbs, and that, for any given time, it can specify the position and velocity of each joint. We further assume that, if physically possible, the low-level motor system responds to such commands by manipulating torques and voltages as needed. In this framework, the task of motor control involves specifying joint positions and velocities so as to generate appropriate behavior.

MÆANDER represents its domain knowledge in terms of *motor schemas*, which represent different movement patterns for sets of limbs. Each schema specifies the state of one or more joints at several points in time during the course of an action, and can be viewed as the

[10] Iba and Langley (1987) describe MAGGIE, an earlier system that has much in common with MÆANDER. The current system differs primarily in its organization of motor schemas into a concept hierarchy at different levels of abstraction.

memory structure that encodes that action. Thus, a schema consists of a temporal sequence of states, $(S_1, S_2, \ldots, S_n)$, where each state, $S_i = (t_i, \{\langle J_k, \mathbf{p}, \mathbf{v}\rangle, \ldots\})$, contains a time value t_i and a set of 3-tuples. The states (S_i) are ordered so that the time values (t_i) are in an increasing sequence, $t_i < t_j$ for $i < j$. Each 3-tuple includes a joint name J_k, the expected position $\mathbf{p}$ of the joint in three-space at time t_i, and the desired velocity vector $\mathbf{v}$ of the joint upon reaching the position $\mathbf{p}$. All numeric attributes have an associated mean and variance that summarize previous experience with the schema. Each state contains a *set* of such 3-tuples, one for each of the agent's joints, though not all joints need be specified.

We distinguish between two sorts of motor schemas. The first type – *viewer-centered* schemas – represent the position and velocity vectors using Cartesian three-space coordinates, with the origin centered at the agent. This representation describes all joints in terms of a single Cartesian coordinate system. We assume this information is available as visual feedback during execution of a skill; it can also be used to describe another agent's actions. The second type – *joint-centered* schemas – represent information in their own local, joint-centered coordinate system. Each local coordinate system is spherical, being defined with the previous joint in the effector as the origin. We assume this information is available as proprioceptive feedback during execution; MÆANDER uses this representation to directly control its motor behavior.

As we will see, MÆANDER can initially acquire motor skills in viewer-centered form, by observing another agent performing that skill. In addition, DÆDALUS (when run in physical domains) describes problems and subproblems in terms of initial and final states using the viewer-centered scheme. However, an agent needs a joint-centered schema in order to execute an action. Intuitively, viewer-centered schemas are better suited to how things 'look', whereas joint-centered schemas are better suited to controlling limbs. MÆANDER moves from a viewer-centered schema to a joint-centered one by applying an *inverse kinematic transform* (Wylie, 1975). In general, this process will generate errors in the joint-centered representation. This results from the differing representational power of the two coordinate systems and the sparse representation of the schema. Actions that may appear simple in one coordinate system can be quite complex from the other's point of view, and the translation process is inherently imperfect. As we will see, learning can be used to overcome these limitations.

Like other parts of ICARUS, the MÆANDER component organizes its domain knowledge in a probabilistic concept hierarchy. We have already mentioned that schema attributes have associated means and variances. This lets the system represent motor knowledge at different levels of abstraction, with nodes higher in the hierarchy tending toward higher variances. For example, different types of throwing actions might be stored as children of a more general THROW concept; similarly, each type of throw might have children representing specific cases of throwing behavior. Each schema also has an associated probability, summarizing the percentage of times it has been used relative to its siblings. MÆANDER's memory organization differs from other facets of ICARUS in that viewer-centered and joint-centered schemas always occur in pairs, being stored in the same place in the hierarchy.

5.2 Retrieving, Executing, and Monitoring Motor Skills

MÆANDER's retrieval process has much in common with the one used by DÆDALUS. The system indexes motor schemas by the differences they reduce, so that retrieving an operator to solve a problem is equivalent to retrieving an operator for planning. However, MÆANDER can also be used to *recognize* instances of motor skills when another agent executes them.[11] In such cases, the system also takes the internal structure of motor schemas into account, using category utility to determine the degree of match between the observed sequence of states and viewer-centered schemas stored in memory. MÆANDER can recognize similar cases of its own joint-centered behavior by sorting a trace through memory of its past actions.

Having retrieved a relevant motor schema based on a problem described in viewer-centered coordinates, MÆANDER attempts to execute this schema. If no joint-centered description is associated with the retrieved schema, the system must first translate it into a joint-centered representation, using an inverse kinematic transform. If such a description is already stored, MÆANDER uses the stored version instead. Both forms of knowledge structure are sparse, containing joint descriptions for only a few points in time. In order to generate movement, the system must "fill in" the intermediate points through a process of interpolation. We will refer to the resulting dense representation as a *motor program*; in contrast to schemas, which represent positions and velocities only at selected times, programs specify joint information at many points. MÆANDER never stores its motor programs in long-term memory; instead, it generates them "on the fly" by computing a spline for each joint, based on successive pairs of states specified in the joint-centered schema.

MÆANDER then 'runs' the resulting program by placing the joints at the specified positions with the stated velocities. However, recall that the translation between representations can introduce errors. Figure 11 shows viewer-centered and joint-centered schemas for drawing a straight line, along with the interpolated behaviors each would generate of its own accord. The joint-centered motor program (on the right) gives a poor approximation of the desired behavior given by the viewer-centered schema (on the left).

To alleviate this problem, MÆANDER monitors the environmental states it generates, comparing actual positions with the intended positions as given in the viewer-centered schema. Execution and monitoring proceed in parallel until the system detects an error. The monitoring process occurs at a constant rate, checking whether the results of executing the motor program diverge from the desired (viewer-centered) action by more than a threshold. If the error is significant, the monitoring process calls an error-recovery mechanism that, in an attempt to reduce the error, applies a correction function that modifies the velocity of the joint for a short period of time.

In this manner, MÆANDER can approximate the desired action even though its joint-centered description has inherent errors. However, the degree to which it accomplishes this goal depends on its rate of monitoring and the speed at which it executes the skill. Thus,

[11] In principle, one could also use DÆDALUS for the purpose of plan recognition, but we have not applied the system to this problem.

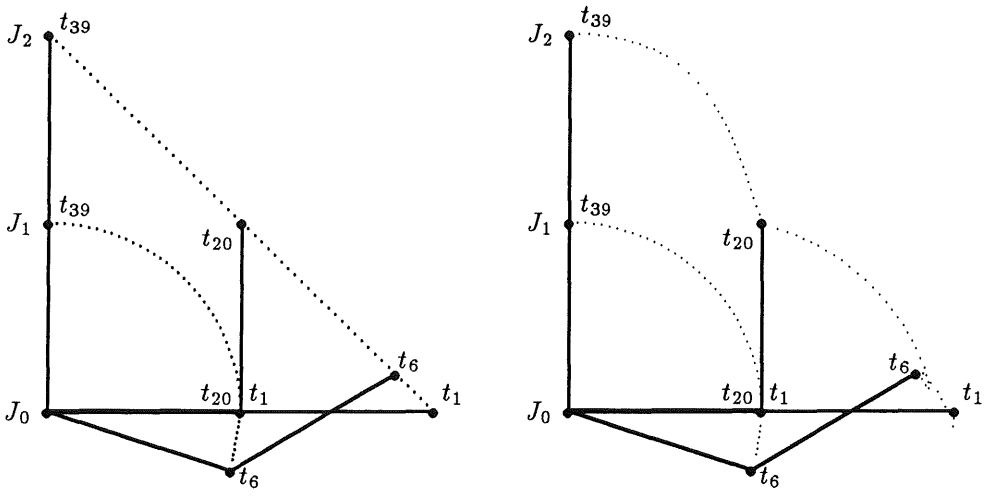

Figure 11. Traces of the behavior generated by a viewer-centered and joint-centered motor schema, with arm shown in the positions it occupies at times specified by states in the schemas (from Iba & Langley, 1987).

the system exhibits a tradeoff between speed and accuracy (Iba & Langley, 1987) similar to that observed in humans. Moreover, if one assumes that the monitoring rate corresponds to the attention level, then MÆANDER is more accurate when it attends more carefully to the task at hand. Both of these effects suggest an important role for learning in motor behavior, which we discuss below.

5.3 Acquisition and Improvement of Motor Skills

In principle, MÆANDER can acquire new joint-centered schemas through play or through active experimentation on existing schemas. However, a simpler strategy involves observing the motor behavior of other agents. In this scenario, the system sorts observed behavior through its memory of viewer-centered schemas, extending and revising a concept hierarchy in the same fashion as other components of ICARUS. Thus, MÆANDER can accumulate considerable expertise about possible actions before it ever attempts to carry them out. Once the system has an accurate viewer-centered description of a motor skill, it can use this summary to generate an initial joint-centered schema and to recover from errors during its execution.

However, MÆANDER's learning does not end at this point. The monitoring process provides data about execution errors, and the system attempts to learn from these errors so it can avoid them in the future without the need for monitoring. To this end, MÆANDER retains information about the largest error that it detects during execution monitoring. After the trial during which this error occurred, the system produces a modified joint-centered description that should more closely approximate the viewer-centered schema from which it was derived. Table 4 summarizes the algorithm used to generate the variant description.

Table 4. MÆANDER's algorithm for altering joint-centered schemas.

```
Inputs:  S is the schema that one wants to modify.
         F is the state in the motor buffer with the largest error.
         R is the root node for the hierarchy of motor schemas.
Side effect: A modified version of the schema S.

Procedure Alter-Schema(S, F, R)
  Let M be the best possible modification to the state values of schema S.
  Let P be the percentage improvement of M over the current version of S.
  If the improvement P is larger than the bias B,
     Then modify S by adding M.
     Else add failure state F to S.
  Classit(S, R).
```

Given that a stored schema is specified as a sequence of states, there are two natural approaches to improving its accuracy. One can modify the velocity for a given joint, or one can change the schema's structure by adding a new state. MÆANDER first determines possible changes to the velocity of the joint involved in the error, selecting the best change according to an evaluation function; we describe the details of this process elsewhere (Iba & Langley, 1987). The system estimates the percentage improvement that would occur if it altered the velocity in this manner. If the percentage is greater than a 'bias' parameter, it selects this change; otherwise, the system decides to add the state with the largest error as a new state in the schema. The bias parameter models the tradeoff between adding a new state (structural change) and modifying an existing state (fine tuning). Preliminary experiments show that an intermediate setting leads to the most rapid learning.

Having decided on a modification, one might alter the joint-centered schema directly. However, recall that MÆANDER relies on probabilistic descriptions of its schemas for retrieval, and these are modified gradually in response to experience. Thus, in order to 'implement' the desired change, the system must 'practice' the modified joint-centered behavior. This requires sorting the modified joint-centered description through memory, giving the effect of 'mental practice'. Alternatively, MÆANDER can run the modified schema in the physical world, sorting actual traces through the joint-centered hierarchy. In either case, the sorting process alters schemas along the retrieval path, and may alter the organization of memory by causing disjuncts, merges, or splits. As a result, MÆANDER gradually gains experience with the modified behavior, until it can be reliably retrieved and executed. In this view, the learning method shown in Table 4 corresponds to a conscious strategy, designed to take advantage of the unconscious learning that occurs during the retrieval and execution of motor schemas.

5.4 Comments on MÆANDER

Of the four ICARUS modules, we have spent the most effort in making MÆANDER consistent with knowledge of human behavior. The system improves its motor skills in a gradual manner, following a learning curve roughly similar to the power laws observed in human

skill acquisition. This occurs because the learning algorithm always focuses first on the state causing the largest error, leading the system to overcome larger errors before smaller ones. Like humans, MÆANDER exhibits a tradeoff between speed and accuracy, and this tradeoff decreases with practice. We also predict that the system will exhibit effects of *practice variability* (Schmidt, 1975), though we have not yet demonstrated this phenomenon.

To date, we have tested MÆANDER primarily on simple motor skills like drawing straight lines, since these predominate in the literature on human motor behavior. However, we plan to test the system on more complex skills, such as handwriting, throwing objects into a basket, and juggling. The latter two tasks raise issues of hand-eye coordination and manipulating objects over which one has only partial control. We believe these can be cast within the framework of viewer-centered schemas, making them accessible to the same monitoring and error recovery methods that MÆANDER uses for simpler motor skills.

In the longer term, we hope to integrate MÆANDER's execution process more fully into DÆDALUS, letting ICARUS interleave planning and execution in a principled way. We also hope to account for *automatization* in terms of the compilation of plan knowledge into motor schemas. This process would gradually transform derivational traces stored in the plan hierarchy, eliminating the reasons for actions and thus increasing efficiency at the expense of flexibility. The details of this mechanism remain an issue for future research.

6. Discussion

In the previous sections we described ICARUS in terms of its various components. With this as background, we can examine its relation to other cognitive architectures, discuss some aspects of the overall framework, and present our plans for integration and evaluation. Below we address each of these in turn.

6.1 Related Work on Cognitive Architectures

Because the ICARUS model attempts to span a substantial portion of cognitive behavior, it bears clear relations to many aspects of earlier AI research. We have cited some relevant work in the context of individual components, but here we briefly discuss related learning work on cognitive architectures for learning. The best known examples of such architectures are Anderson's (1983) ACT*, Laird et al.'s (1986) SOAR, and Minton's (1988) PRODIGY.[12]

Like ICARUS, these architectures attempt to cover a broad range of behaviors within a unified theoretical framework, though they differ in their generality and theoretical content. For instance, PRODIGY makes strong claims about the nature of problem solving, but is largely limited to this facet of cognition. At the other extreme, ACT* takes a weaker stance on the organization of thought processes, but has been applied to domains as diverse

[12] Another recent example is Mitchell et al.'s (in press) THEO architecture, which differs significantly from both ICARUS and earlier systems. Briefly, it organizes memory into frames, uses a backward-chaining mechanism to control reasoning, and employs a caching technique to improve retrieval efficiency. Unlike other architectures, THEO has no explicit short-term memory.

as problem solving, natural language, and concept recognition. ICARUS attempts to make strong theoretical statements about the nature of cognition, but also aims for broad coverage by including specific modules for object recognition, planning, and motor control.

The design of each architecture has been constrained by knowledge of human cognition, though each focuses on different phenomena. Anderson's framework incorporates psychological results mainly into its spreading-activation retrieval mechanism and into its strengthening processes. Laird et al.'s framework incorporates ideas on chunking from Rosenbloom's (1986) model of human learning. Minton's system shows less concern with psychological issues, though its use of means-ends analysis was influenced by Newell et al.'s (1960) GPS, an early model of human problem solving. Some aspects of ICARUS have been significantly constrained by psychological results, though its focus on categorization and motor behavior distinguishes it from the other frameworks.

The four architectures clearly describe a set of memories and their characteristics, with domain knowledge residing in a permanent long-term memory. SOAR, PRODIGY, and ACT* all represent this knowledge in the form of production rules, though the latter also includes a separate declarative memory. Traditionally, work on production systems has assumed that condition-sides can be matched in parallel, and this has discouraged researchers from addressing issues of memory organization and indexing. In contrast, ICARUS' use of interleaved concept hierarchies makes these issues central; thus, it takes a clear stance on the structure of memory, whereas the other architectures sidestep the problem.

All four frameworks identify a set of primitive processes that are supported at the architectural level. In the systems based on production systems, the primitive actions involve matching condition sides and applying one or more of the matched rules. In SOAR and PRODIGY, this occurs during an elaboration cycle, in which selection, rejection, and preference rules 'vote' in favor of particular states, operators, and goals; the architecture then makes a decision based on these votes. ACT* makes less commitment about the nature of its rules, with some acting as operators, others as goal generators, and others as inference makers. ICARUS diverges from these systems, with heuristic classification as its primitive operation and with other processes, such as means-ends analysis and motor execution, built on top of this basic mechanism.[13]

In addition, all the architectures incorporate a single basic learning mechanism that constitutes a form of incremental hill climbing. Chunking in SOAR, knowledge compilation in ACT*, and explanation-based learning in PRODIGY have much in common, effectively caching the results of rule or operator applications to simplify future processing. In contrast, the central learning mechanism in ICARUS is concept formation, the process of updating probabilistic descriptions and altering the structure of the concept hierarchy. This scheme is primarily concerned with improving *accuracy* in terms of recognition and prediction, whereas chunking and its relatives focus mainly on improving *efficiency*. Of course, the hierarchical

[13] Each of the architectures incorporates Newell's (1980) problem-space hypothesis – that all cognitive behavior can be viewed as search through a problem space. Also, each implements this process using some more primitive mechanisms – classification in ICARUS and the recognize-act cycle in the other three.

organization of memory has implications for efficiency, and Rosenbloom (1987) has shown that SOAR can exhibit a form of 'data chunking' that shares features with LABYRINTH's formation of object concepts. However, the systems differ in what they treat as underlying processes and what they treat as emergent phenomena.

On many dimensions, ICARUS is most similar to 'analogical' or 'case-based' approaches to cognition (e.g., Carbonell, 1986; Kolodner, 1987). We have noted that the system's basic data structures share aspects of Schank's (1982) theory of dynamic memory, and in its early stages, the DÆDALUS component exhibits a form of 'case-based reasoning' (Kolodner, 1987). Like dynamic memory, ICARUS relies on interleaved hierarchies of different concepts, with complex concepts being specified in terms of their components and with more abstract concepts stored above their specific children. Our framework differs from dynamic memory (as it does from SOAR, PRODIGY, and ACT*) in its emphasis on probabilistic descriptions, in its focus on grounded symbols and interaction with physical environments, and in its concern with sensori-motor phenomena in addition to high-level cognition. However, both ICARUS and dynamic memory emphasize retrieval from an organized memory, and both can be viewed as direct descendants of EPAM (Feigenbaum, 1963; Feigenbaum & Simon, 1984), the earliest model of incremental concept formation.

6.2 Attention and Short-Term Memory in ICARUS

One central finding of cognitive psychology is that the human information-processing system contains sequential 'bottlenecks' that require some form of selective *attention*. In describing ICARUS' components, we revealed three different forms of attention. The first dealt with the basic processes of retrieval and object recognition as modeled by CLASSIT and LABYRINTH. This variant corresponds roughly to the notion of perceptual attention in the psychological literature (Treisman, 1969), though we have not attempted to account for such attentional phenomena.

A second form of attention occurs in MÆANDER's process of execution monitoring. This corresponds roughly to the notion of closed-loop processing in theories of human motor behavior (e.g., Adams, 1971), and it underlies the system's ability to model the tradeoff between speed and accuracy in motor control. However, MÆANDER currently uses a system parameter to describe the level of attention, and does not reflect the conscious nature of this process. Future versions should model execution monitoring in more detail, possibly borrowing from the method used in CLASSIT.

The attentional bottleneck is not limited to the sensori-motor level, and one can cast DÆDALUS (like any means-ends system) as modeling the cognitive aspects of attention. In this framework, the agent generates new goals (subproblems) sequentially, and the focus of attention is on one goal at any given time. In future work, we plan to associate levels of attention with each subproblem that reflect their priority, and to integrate the planning process with the application of internal drives. We also hope to use goal information to direct the attentional processes in object recognition and motor control.

Attention is closely related to short-term memory, a feature that has been notably lacking in our description of ICARUS. Production-system architectures like SOAR and ACT* incorporate a declarative working memory that is distinct from knowledge stored in production rules, but this dichotomy makes little sense in the ICARUS framework. One model would have the classification process 'activate' nodes in ICARUS' long-term memory, with the set of active nodes constituting short-term memory. As in ACT*, activation levels would decay over time, requiring explicit rehearsal or external stimuli to keep a goal or concept active. However, working out the details of this memory model remains an issue for future work.

6.3 Action and Perception in ICARUS

The distinction between *action* and *perception* cuts across all aspects of intelligent behavior. Action, whether imagined or real, can range from the production of sequential plans to the invocation of motor schemas to the design of physical devices. Similarly, perception can range from simple recognition of an object or motor behavior to complex understanding of another agent's plan. The current version of ICARUS associates perception mainly with object recognition and action only with planning and motor control, but this is a clear oversimplification. To this end, we should briefly consider the relation between action and perception.

Our current view is that action-oriented behavior is connected directly to *goals*, whereas perception and understanding are linked to *beliefs*. This is not a distinction between the *mechanisms* used in perception and action, but between the interpretation of structures in memory. In fact, we intend that future versions of ICARUS more fully integrate the mechanisms for perception and action, generating beliefs in some cases and goals in others. We have already seen that MÆANDER can use the same mechanism to retrieve motor schemas for execution and to recognize another's behavior. Similarly, DÆDALUS should be able to infer another agent's plan from an observed sequence of actions; the task of plan understanding can be viewed as a more constrained version of the plan generation task. To distinguish between such alternative interpretations, the system will place clear labels on its data structures, stating whether they describe the agent's goals (which should lead to action) or describe events in the world (which should simply be summarized).

One might also make finer distinctions within the broad categories of goals and beliefs. For instance, one could separate goals that should actually be implemented from those that involve wishful thinking; this might be handled with notions of goal priority. More important, one might divide beliefs into two classes: *observations*, which one knows are true, and *predictions*, which one may or may not confirm. The current versions of CLASSIT and LABYRINTH are able to make predictions, but they do not store these in memory along with observations. This certainly seems desirable, but it is less clear whether to assume a strong dichotomy or to allow for many sources of beliefs, which may or may not disagree. Identifying appropriate roles for prediction and observation remains an important goal of our research program.

6.4 Spatial Knowledge in ICARUS

An intelligent agent exists in some physical environment larger than its sensors can encompass. In order to reason about such surroundings, it must be able to represent familiar locations, organize them in memory, access them from that memory, and acquire them from experience. Before we can herald ICARUS as a successful model of such an agent, we must show it can handle all these aspects of spatial knowledge, and here we discuss the architecture's potential in this context.

We will use the term *place* to refer to a location and its associated sense data (i.e., the visible objects it contains and their features). LABYRINTH's ability to deal with composite concepts suggests an approach to handling knowledge of places. Briefly, one can represent each place as a complex 'object concept', with objects visible from that location (e.g., boulders, posts) as its component objects. Upon receiving a sensory description of its current surroundings, ICARUS would pass this information to LABYRINTH for incorporation into long-term memory. This process would involve classifying and storing component objects, redescribing the overall scene by treating the component labels as its 'features', and finally classifying and storing the scene in memory, described in terms of its components. If the scene is genuinely new, LABYRINTH would store it as a new place concept; otherwise it would store the experience as a child of a familiar place.

This approach to place description also supports the notion of *landmarks*, which are generally viewed as useful in navigation and exploration. In this framework, landmarks are objects that can be seen from many positions, making them components of many place concepts, and are sufficiently unique to aid in distinguishing places from each other. The complexity of spatial information requires that we incorporate the CLASSIT attention mechanism into LABYRINTH, and the augmented version should tend to focus on landmarks early in the process of place recognition. It should also use landmarks in identifying sequences of places as it moves along, which leads us to a different but related aspect of spatial knowledge.

In addition to recognizing familiar places, an intelligent agent should also be able to *navigate* from one place to another, and knowledge of *routes* is generally viewed as useful to this end. In ICARUS, it seems natural to view routes as sequences of places that occur in the context of *plans*. Thus, generating (or retrieving) a route involves generating (or retrieving) a plan for moving between two places, and acquiring a route involves storing a successful plan for moving from one place to another. We hope to use DÆDALUS to handle the task of navigation and the acquisition of route knowledge, letting the agent's navigational abilities improve with experience of a particular area.

However, before any system can formulate routes, it must have some basic information about spatial relations between different places. We assume that LABYRINTH will recognize familiar places and store them in memory, but we need some additional mechanism for linking nearby places. To this end, we plan to incorporate a drive for *exploration* into ICARUS. Given no high-priority goals, this will lead the agent toward novel objects and places in its environment, so as to examine them in more detail. It will also lead the agent

away from familiar places in search of novel ones. These goals should be interrupted by other drives if the agent runs low on fuel or energy, or if it is about to collide with an obstacle, causing the agent to return to a fuel source or avoid the object. But lacking other goals, the exploration drive will lead the agent to new experiences, letting it store information about new places and connections between those places.

As we anticipate implementing them in ICARUS, drives will be evoked by stimuli rather than goals. As a result, exploratory behavior would not produce any coherent plan that can be stored in memory for future retrieval. Instead, it would lead to local actions, which ICARUS would store as instances of particular operators. However, the preconditions on a given operator instance would be the place in which the agent applied that operator, and the postcondition would be the place resulting from its application. Thus, the drive for exploration should lead ICARUS to store information about connections between specific places in its hierarchy of motor schemas or operators.

Once the agent has stored connections in this form, it can invoke DÆDALUS to solve particular navigation problems. Given the task of moving from place A to place B, it will retrieve operators that let it transform one into another. If the two places share some component (landmark) object C, then DÆDALUS can use information about the differences between the current and desired description of C to select appropriate operators. If there are no shared components, then the system must resort to a strategy of depth-first search until it finds some sequence of operators that transform A into B. DÆDALUS will succeed in this effort only if memory contains some path that connects the two places, and even if it does find a path, the process may take considerable search.

However, once means-ends analysis has found a route between A and B, the system will store information about the successful plan in long-term memory. Given the same problem or another task with shared subproblems, DÆDALUS will use its previous experience to constrain the search process. As it gains experience in a given environment, the system will construct a repertoire of route plans that it can use to efficiently navigate between various places. This information will be topological rather than metrical in nature, but the former constitutes an important aspect of spatial knowledge (Kuipers, 1982).

6.5 The Status and Evaluation of ICARUS

The ICARUS architecture does not yet exist as an integrated entity, but all of its components have been implemented. An initial version of CLASSIT (Gennari et al., 1989) has been tested on simulated physical domains, and we have extended the system to include selective attention. MÆANDER's predecessor, MAGGIE (Iba & Langley, 1987), was tested on simple motor tasks, and the current version extends this approach by organizing motor schemas into a concept hierarchy and employing concept formation as the main learning mechanism. LABYRINTH, DÆDALUS, and MÆANDER have all been implemented and are currently undergoing initial tests.

Combining these four systems into a single, integrated architecture will be a challenging task, but we have made some progress in this direction. For instance, LABYRINTH, DÆDALUS, and MÆANDER already use the CLASSIT system as their primary subroutine. We forsee few problems with connecting DÆDALUS and MÆANDER; the former will generate plans that the latter can execute, and we will not attempt a tight integration of planning and execution in the near future. One major issue that remains is the interaction between internal drives and the planning process. Another concern is the interaction between planning and object recognition, but we hope to delay this in the first version of ICARUS, assuming that these processes run independently and in parallel.

We plan to evaluate each of these systems experimentally using artificial domains, some purely symbolic in nature and others designed to mimic the continuous nature of the physical world. As in previous studies, we plan to vary aspects of the environment, such as the regularity of object classes and the complexity of problems. We also plan to vary aspects of the systems, such as their parameter settings and evaluation functions, and we hope to carry out lesion studies, comparing the behavior of each system to that when certain components are omitted. In each case, we plan on measuring learning in terms of performance improvement over time, examining both accuracy and efficiency as dependent variables. A central hypothesis is that retrieval time will not degrade with large amounts of experience; to this end, we plan to run the systems on training sets of many instances.

Evaluating the overall architecture will be more difficult, but we will attempt this task only after studying the components in some detail. For this purpose we plan to use a simulated environment such as the World Modeler's System,[14] which simulates a three-dimensional world obeying the laws of Newtonian physics. This simulation models time in discrete steps, updating the positions of objects based on their previous positions, their velocities, and the forces applied to them, including gravity, torque, and friction. The simulator computes the effects of elastic collisions among rigid objects, and alerts the agent when it touches other objects. The agent controls its effectors by placing them in desired positions, and can pick up objects as a primitive action.

We have designed two initial domains within this testbed. The first involves a blocks world in which the agent consists of a robot hand that must pick up blocks and place them in specified locations. The second domain involves a mobile agent that can wander around, learning about its environment while subject to conflicting drives like hunger and curiosity. Both domains are relatively simple, but they should provide initial tests of the overall architecture and let us compare variants to each other. For instance, we expect the innate drives to have a major influence on both initial performance and learning. The simulated world also lets one control levels of uncertainty in sensori-motor data, and we will vary this as well. In the longer term, we hope to test ICARUS in complex worlds, and our ultimate goals include connecting the system to a physical robot for more realistic tests.

[14] This software was developed by researchers at Carnegie Mellon University and the University of California, Irvine.

6.6 Summary

In this paper we have described our designs for ICARUS, an integrated cognitive architecture for controlling autonomous agents situated in the physical world. We considered the architecture's four main components in some detail. These modules included: LABYRINTH, which classifies objects and acquires complex object concepts; DÆDALUS, which generates plans and acquires plan expertise; and MÆANDER, which executes motor programs and acquires motor schemas. All three components invoke CLASSIT, which classifies instances described in terms of primitive attributes and organizes them in a probabilistic concept hierarchy.

Although we have detailed ideas about ICARUS' components, the overall architecture must still be finalized and tested. Many issues still remain open, but we believe ICARUS constitutes a promising theory of intelligent behavior that deserves further exploration. The initial design responds to the six goals stated at the outset of the paper, and the basic framework continues to provide fertile ideas for improvements and extensions. It may be some time before we develop a complete architecture capable of controlling a physical robot, but we feel that we are rapidly making progress in that direction.

References

Adams, J. A. (1971). A closed-loop theory of motor learning. *Journal of Motor Behavior, 3*, 111–149.

Anderson, J. R. (1983). *The architecture of cognition.* Cambridge: Harvard University Press.

Bresina, J. L. (1988). *REAPPR – An expert system shell for planning* (Technical Report LCSR-TR-119). New Brunswick, NJ: Rutgers University, Busch Campus, Hill Center for the Mathematical Sciences, Laboratory for Computer Science Research.

Carbonell, J. G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Carbonell, J. G., & Veloso, M. (1988). Integrating derivational analogy into a general problem solving architecture. *Proceedings of the DARPA Workshop on Case-based Reasoning* (pp. 104–121). Clearwater Beach, FL: Morgan Kaufmann.

Clancey, W. J. (1985). Heuristic classification. *Artificial Intelligence, 27*, 289–350.

DeJong, G. F., & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning, 1*, 145–176.

Everitt, B. (1974). *Cluster analysis.* London: Heinemann Educational.

Feigenbaum, E. A. (1963). The simulation of verbal learning behavior. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and thought.* New York: McGraw–Hill.

Feigenbaum, E. A., & Simon, H. A. (1984). EPAM-like models of recognition and learning. *Cognitive Science, 8*, 305–336.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2*, 189–208.

Fisher, D. (1987a). Knowledge acquisition via incremental conceptual clustering. *Machine Learning, 2*, 139–172.

Fisher, D. (1987b). *Knowledge acquisition via incremental conceptual clustering.* Doctoral dissertation, Department of Information & Computer Science, University of California, Irvine.

Fisher, D. H. (1988). A computational account of basic level and typicality effects. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 233–238). St. Paul, MN: Morgan Kaufmann.

Fisher, D. H., & Langley, P. (in press). The structure and formation of natural categories. In G. H. Bower (Ed.), *The psychology of learning and motivation: Advances in research and theory* (Vol. 26). Cambridge, MA: Academic Press.

Fried, L. S., & Holyoak, K. J. (1984). Induction of category distributions: A framework for classification learning. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *10*, 234–257.

Gennari, J. (1989). Focused concept formation. *Proceedings of the Six International Workshop on Machine Learning* (pp. 379–382). Cornell, NY: Morgan Kaufmann.

Gennari, J. H., Langley, P., & Fisher, D. (1989). Models of incremental concept formation. *Artificial Intelligence*, *40*, 11–61.

Georgeff, M. (1987). Planning. In J. F. Traub (Ed.), *Annual review of computer science.* Palo Alto, CA: Annual Reviews, Inc.

Gluck, M., & Corter, J. (1985). Information, uncertainty and the utility of categories. *Proceedings of the Seventh Annual Conference of the Cognitive Science Society* (pp. 283–287). Irvine, CA: Lawrence Erlbaum.

Hammond, K. (1986). *Case-based planning: An integrated theory of planning, learning, and memory.* Doctoral dissertation, Department of Computer Science, Yale University, New Haven, CT.

Harnad, S. (1989). *The symbol grounding problem.* Unpublished manuscript, Department of Psychology, Princeton University, Princeton, NJ.

Iba, W., & Langley, P. (1987). A computational theory of human motor learning. *Computational Intelligence*, *3*, 338–350.

Kolodner, J. L. (1980). *Retrieval and organizational strategies in conceptual memory: A computer model.* Doctoral dissertation, Department of Computer Science, Yale University, New Haven, CT.

Kolodner, J. L. (1983). Reconstructive memory: A computer model. *Cognitive Science*, *7*, 281–328.

Kolodner, J. L. (1987). Extending problem solver capabilities through case-based inference. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 167–178). Irvine, CA: Morgan Kaufmann.

Laird, J., Rosenbloom, P., & Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, *1*, 11–46.

Laird, J., Yager, E. S., Tuck, C. M., & Hucka, M. (in press). Learning in teleautonomous systems using SOAR. *Proceedings of the 1989 NASA Conference on Space Telrobotics.*

Langley, P., Gennari, J., & Iba, W. (1987). Hill-climbing theories of learning. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 312–323). Irvine, CA: Morgan Kaufmann.

Lebowitz, M. (1980). *Generalization and memory in an integrated understanding system.* Doctoral dissertation, Department of Computer Science, Yale University, New Haven, CT.

Lebowitz, M. (1987). Experiments with incremental concept formation: UNIMEM. *Machine Learning*, *2*, 103–138.

Marr, D. (1982). *Vision: A computational investigation into the human representation and processing of visual information.* San Francisco, CA: W. H. Freeman.

Mervis, C., & Rosch, E. (1981). Categorization of natural objects. *Annual Review of Psychology*, *32*, 89–115.

Michalski, R. S., & Stepp, R. (1983). Learning from observation: Conceptual clustering. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 564–569). St. Paul, MN: Morgan Kaufmann.

Mitchell, T. M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., & Schlimmer, J. C. (in press). THEO: A framework for self-improving systems. In K. VanLehn, (Ed)., *Architectures for Intelligence.* Hillsdale, N.J.: Lawrence Elrbaum.

Mooney, R. (1988). Generalizing the order of operators in macro-operators. *Proceedings of the Fifth International Workshop on Machine Learning* (pp. 270–283). Ann Arbor, MI: Morgan Kaufmann.

Newell, A., Shaw, J. C., & Simon, H. A. (1960). Report on a general problem-solving program for a computer. *Information Processing: Proceedings of the International Conference on Information Processing* (pp. 256–264).

Newell, A. (1980). Reasoning, problem solving, and decision processes: The problem space hypothesis. In R. Nickerson (Ed.), *Attention and performance VIII.* Hillsdale, NJ: Lawrence Erlbaum.

Nilsson, N. (1984). *Shakey the robot* (Technical Note 323). Menlo Park, CA: SRI International.

Rosenbloom, P. S. (1986). The chunking of goal hierarchies. In J. Laird, P. Rosenbloom, & A. Newell (Eds.), *Universal subgoaling and chunking.* Boston, MA: Kluwer Academic Publishers.

Rosenbloom, P. S., Laird, J. E., & Newell, A. (1987). Knowledge level learning in SOAR. *Proceedings of the Sixth National Conference on Artificial Intelligence* (pp. 499–504). Seattle, WA: Morgan Kaufmann.

Sacerdoti, E. D. (1977). *A structure for plans and behavior.* New York: Elsevier North-Holland.

Schank, R. C. (1982). *Dynamic memory.* Cambridge, UK: Cambridge University Press.

Schneider, W., & Fisk, A. D. (1983). Attention theory and mechanisms for skilled performance. In R. A. Magill (Ed.), *Memory and control of action.* Amsterdam: North-Holland Publishing Company.

Schlimmer, J., & Fisher, D. (1986). A case study of incremental concept induction. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 496–501). Philadelphia, PA: Morgan Kaufmann.

Schmidt, R. A. (1975b). A schema theory of discrete motor skill learning. *Psychological Review, 82,* 225–260.

Segre, A. M. (1987). *Explanation-based learning of generalized robot assembly plans* (Technical Report UILU-ENG-2208). Urbana: University of Illinois, Department of Electrical and Computer Engineering.

Smith, E., & Medin, D. (1981). *Categories and concepts.* Cambridge, MA: Harvard University Press.

Stepp, R. E., & Michalski, R. S. (1986). Conceptual clustering of structured objects: A goal-oriented approach. *Artificial Intelligence, 28,* 43–69.

Swartz, N. M. (1984). Arm dynamics simulation. *Journal of Robotic Systems, 1,* 83–100.

Thompson, K., & Langley, K. (1989). Incremental concept formation with composite objects. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 371–374). Ithaca, NY: Morgan Kaufmann.

Treisman, A. M. (1969). Strategies and models of selective attention. *Psychological Review, 76,* 282–299.

Wasserman, K. (1985). *Unifying representation and generalization: Understanding hierarchically structured objects.* Doctoral dissertation, Department of Computer Science, Columbia University, New York, NY.

Wilkins, D. (1984). Domain independent planning: Representation and plan generation. *Artificial Intelligence, 22,* 269–301.

Wylie, C. R. (1975). *Advanced engineering mathematics* (4th ed.). New York: McGraw-Hill.