Report 80-29
Stanford -- KSL

An Introduction to Knowledge Engineering, Blackboard Model, and AGE.
H. Penny Nii,
Mar 1980

Heuristic Programming Project
Report No. HPP-80-29

March 1980

An Introduction to Knowledge Engineering, Blackboard Model, and AGE

H. Penny Nii

Heuristic Programming Project
Departments of Computer Science and Medicine
Stanford University
Stanford , Calilfornia 94305

This work was supported in parts by the Defense Advanced Research Projects under ARPA contract no. MDA 903-77-C-0322 and National Institutes of Health grant no. RR-00758.

Table of Contents

PREFACE

This document is intended for people who are interested in applying artificial intelligence methods and techniques to real world problems. The content is focussed on knowledge engineering, an approach to building high performance programs by codifying and using the "expertise" of experts in some specific domain.

The document covers three separate topics: knowledge engineering, its description and practice; the Blackboard model, viewed as a problem solving paradigm, with examples of programs that have their roots in this paradigm; and AGE, a generalized software tool for building application programs. It is not intended to be a scientific discussion of artificial intelligence methods or the Blackboard model. Nor does it pretend to define the emerging field of knowledge engineering. Rather, it is a collection of practical advice and issues to consider when building expert programs.

BLANK PAGE

1 Introduction to Knowledge Engineering

In recent years the term *knowledge engineering* is being used more frequently to refer to the various activities involved in writing application programs using methods and techniques developed in artificial intelligence [Feigenbaum]. This section is an attempt to describe the scope of these activities and to introduce to those unfamiliar with knowledge engineering some lore in this emerging field.

1.1 What is Knowledge Engineering?

"Programming"--like "loving"--is a single word that encompasses an infinitude of activities. [Weinberg]

Knowledge engineering is the art of designing and building computer programs that perform some specific task in a domain by combining methods of symbolic inference and representation with knowledge of that domain. The resulting programs are often called *knowledge-based*, or *expert*, programs.

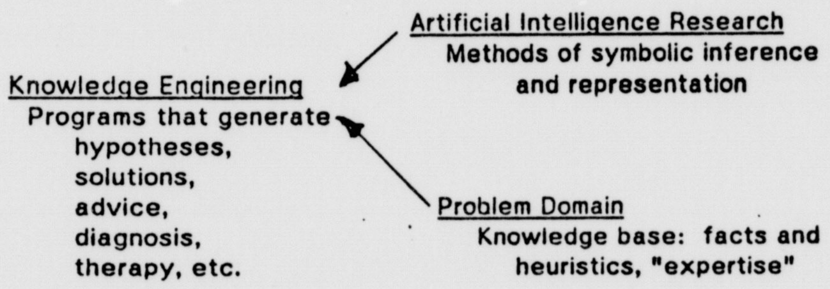

Artificial Intelligence Research
Methods of symbolic inference and representation
Knowledge Engineering
Programs that generate hypotheses, solutions, advice, diagnosis, therapy, etc.
Problem Domain
Knowledge base: facts and heuristics, "expertise"

Figure 1.1 Knowledge Engineering Task

The discovery and cumulation of techniques of symbolic inference and representation is generally the work of the field of artificial intelligence research. But knowledge engineers must often invent techniques when existing ones are not sufficient for the task at hand. The knowledge of a problem domain is held by experts of the domain. This knowledge consists of both formal, textbook knowledge and experiential knowledge--the expertise of the experts. What then, do knowledge engineers know and what do they do?

A *knowledge engineer* is the person who integrates AI methods with knowledge about a problem domain to create a high performance program. Like other experts, a knowledge engineer brings to bear on the problem of building a knowledge-based program many, diverse sources of knowledge. Much of the knowledge is "hard" knowledge, like programming. Some is heuristic knowledge, like, "design a program that reasons like the human expert." He must know the inference methods and representational forms arising from AI research--their strengths and weaknesses; their scope of applicability to different types of problems; and how they can be implemented. He must be able to interact with the expert on the expert's terms, by learning something about the problem domain; in order to get the expert to verbalize knowledge that he himself is not aware that he has; and to keep him interested over the long period of time it takes to build an expert system.

A knowledge engineer works in a space that contains concepts like "heuristic search" and "semantic nets", methods of software design and management, and actual program codes. One possible way to view this space is shown below. It is a multi-dimensional space containing concepts and techniques from a variety of computer science subfields (AI, system design, data structures and data bases, algorithms), and methods and techniques from software engineering (software design, management, and development techniques). A knowledge engineer searches the space and makes choices appropriate to the goals of the application. He chooses an AI paradigm best suited to the task at hand; he chooses a representation compatible with and adequate for the knowledge and the objects in the task domain <u>and</u> the AI paradigm. Before coding, he designs a program and chooses some software management technique that will enable smooth program development. At the same time he works as a transducer of knowledge between the domain expert and the emerging program.

Once the appropriate paradigm and representation are chosen and a program is designed, the actual coding begins. But as the implementation progresses and knowledge accumulates, changes invariably need to be made. At times this involves simple recoding; at other times it involves shifting to another paradigm and starting over. Building a knowledge-based program is an iterative process of extracting a human expert's knowledge, transferring it to a program, and testing and modifying the knowledge base.

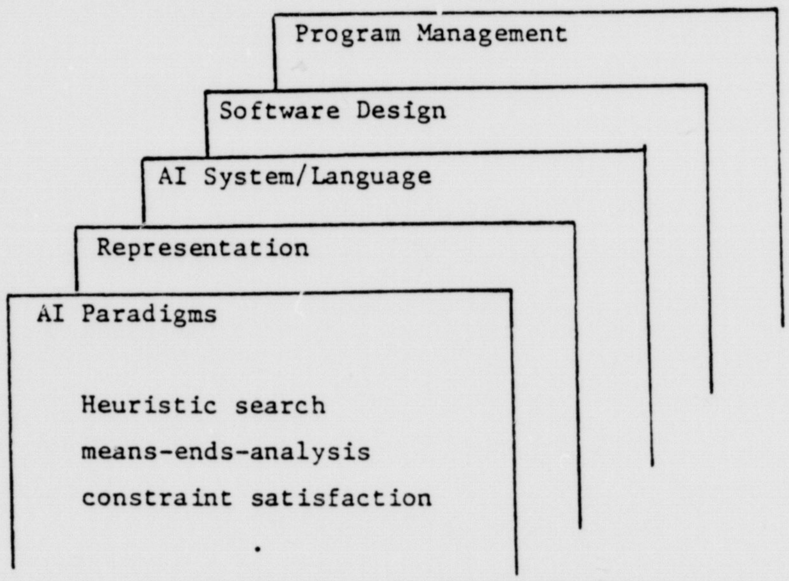


Figure 1.2 Knowledge Engineering Space

1.2 What Knowledge Engineers need to know

Knowledge Engineering is a complex activity directed at understanding and solving complex and often ill-defined problems. The general process and the problems associated with the development of knowledge-based programs are, in many respects, no different than those of building any large, complex software system. The experienced program developers might ask at this point, "Are there any problems that arise in building knowledge-based programs that are unique?" The answer is, "Yes, a few."

First, there are conflicting requirements between the need to understand and plan a system before programming, and the need to have a running program immediately (in a few days) for the expert.

Second, it is impossible to know in advance the programming requirements. The statement by Weinberg "...writing a program is a process of learning--both for the programmer and the person who commissions the program," is compounded in difficulties by the need to work with ill-defined goals, ill-structured problems [Newell 69], and large search spaces.

Third, often the knowledge bases are, in contrast to algorithms, unsystematic and unreliable--that is, the knowledge obtained from the experts is often heuristics, or "rules of thumb" that are judgmental and uncertain.

Fourth, the knowledge base must be constantly modified and updated as the knowledge of the domain grows and changes; the new knowledge may be in conflict with, or contradictory to, the existing knowledge.

It may be argued that many of the same conditions hold true for writing any program; however, if all the conditions hold for what you are doing, then you are probably building a knowledge-based program. We have listed some rules that potential knowledge engineers may find useful in Appendix A. The list is by no means complete, but reflects things we have learned through experience.

1.3 Knowledge Engineering Team

Ideally, a knowledge engineering team should consist of an expert, a knowledge engineer, and one or more programmers. The task of the expert is twofold: (1) to divulge his expertise either informally by solving many example problems, or more formally by writing down the rules of inference used to solve the problem; and (2) to test the program when new knowledge is added or old ones corrected. The task of the knowledge engineer is also twofold: to be the "chief surgeon" of the team [see Brooks]; and (2) to be a liaison between the expert and the programmers. As a "surgeon" he performs all the tasks described in Section 1.1--he is the final arbiter and decision maker.

The number of programmers and their tasks differ depending on the size and complexity of the program. In applications developed thus far, with the exception of speech and vision programs, the number of programmers in a team has been small. In many projects the knowledge engineer also serves the programmer function. Ideally, the team should follow "Mill's surgical model" as described by Brooks.

2 Blackboard Model as an AI Problem-solving Paradigm

The current AGE system provides the user with a framework to be used for incremental, opportunistic hypothesis formation based on the Blackboard model, originally developed for HEARSAY-II Speech Understanding Project (H-II). The current AGE is intended for tasks in which the problem-solving process involves finding a first-order solution and incrementally improving it.

The primary objective of this guide is to help the potential user of AGE determine whether this framework, together with various tools currently available in AGE, is suitable for his problem. We have listed below some common characteristics of tasks that have utilized the Blackboard model, together with case studies of programs that have their roots in the Blackboard model. We have included analyses of some of their different features, and whenever possible, an explanation of why specific design features and implementations were chosen. We hope that these case studies will serve as a guide in evaluating and constructing new problems. Once a problem is formulated in terms of the Blackboard model, there are two additional documents that describe how to use the AGE system: *The Joy of AGE-ing: User's Guide to AGE* and *An AGE Reference manual.*

2.1 Task Characteristics

"Psychologists know that human performance on a given task is a function of the task itself <u>and</u> as understood by the subject." [Weinberg]

1. A large search space; where it is infeasible or impossible to generate all the elements in the solution space, even as a prelude to heuristic pruning.

2. Many independent, or semi-independent, sources of knowledge that must "work together" to form a solution. No one knowledge source is sufficient by itself to solve the problem.

3. The task domain can be represented in a hierarchy or a set of interrelated hierarchies. The levels of hierarchy can represent levels of conceptual abstractions, information aggregation, or analysis.

4. A knowledge source uses information or data from one (or more) levels to infer information for another level.

5. Hypothesis formation uses an "opportunistic" strategy. That is, there is no computationally feasible "legal move generator" that defines the solution space in which pruning and steering can take place toward finding the correct solution. Rather, by reasoning about bits and pieces of available evidence, one can incrementally generate partial hypotheses that will eventually lead to more global solution hypotheses.

6. Although not necessarily true for all cases, the solutions are built incrementally in

"go-forward" manner without backtracking. The "current-best-hypothesis" is used as a means for determining what is to be done next to improve the hypothesis. Note that this in itself does not rule out the possibility of keeping alternative hypotheses.

In addition to these basic characteristics, which may or may not characterize your problem, AGE imposes certain restrictions by its choice of representation and inferential mechanisms.

7. Informal knowledge, or heuristics, are represented in a production rule formalism. The rules are designed to allow arbitrarily complex matching on the left-hand-side, but restricts the right hand side to building and modifying the data structure on the blackboard.

8. The more concrete and descriptive knowledge can be represented in an object-centered, frame representation using a program package called Units. Information represented in Units can be used and modified by production rules. Although the general Units representation allows procedural and rule attachments, AGE does not support this.

9. The control structure is generally oriented towards choosing the appropriate KSs based on the changes made on the blackboard. It allows for data- (or event) driven, expectation- (or model-driven), and goal-driven processing. However, there are limitations imposed on how these control mechanisms can be used.

10. KSs can be used to establish links (support) between hypotheses. Facilities provided in AGE for traversing these links are minimal compared to what can be found in a general semantic-net representation package.

2.2 Inappropriate Problems

"It is common sense to take a method and try it.
If it fails, admit it frankly and try another.
But above all, try something." [F. D. Roosevelt]

Given an application problem, there are many practical issues to be considered--some of these are presented in the appendix in a form of lore of knowledge-engineering. The questions of how knowledge should be represented and what inference methods should be applied are difficult ones to answer--they are precisely the questions being asked in AI research. However, given that one has chosen the "right" problem fitting the guidelines stated in rule 3 above, one needs to know how to proceed in choosing the appropriate representation and paradigm. We can make some conjectures along this line based on the few knowledge-based programs that have been build thus far. Since we have already listed the problem characteristics that are appropriate to using the Blackboard model, we'll now state these conjectures in terms of when the use of Blackboard model is not necessary.

The most important rule to remember is this: if an algorithm can be written to solve your problem, there is no need to insist on using AI methods. Of course, if the problem definition tends to change often, one needs to design a robust algorithm. Or, if explanations comprehensible to experts, who generally think heuristically, are needed, an algorithm may not be the method of choice.

In general, if a problem is well-structured, the use of Blackboard model is not necessary. By "well-structured", we mean problems that have :

1. sharp goal criterion,
2. well defined path to the solution, and
3. well defined knowledge.

Ill-structured problems are those in which one or more of the above criteria is relaxed. One can view the Blackboard model as a method of dealing with ill-structured problems by extracting well defined sub-problems, one at a time. Tentative solutions to the sub-problems are put on the blackboard. What sub-problem to solve next is determined to a large extent by what is on the blackboard and a general plan. In general, ill-structured problems can be dealt with by building a large knowledge base and adding a blackboard. [from lecture by H. Simon, 1977].

Even in well-structured problems, there are many task and domain characteristics that need to be considered before one can choose from the few problem solving paradigms currently available. If one views problem solving as search in some space, then one needs to consider the size of the space; whether there is a systematic way of generating the space; and whether there are methods of evaluating the generated objects in the space. If the space is large, there must be enough knowledge to prune and reduce the space. There are also trade offs to be considered between search, knowledge and strength of inference methods.

The following is a set of high level in the appendix heuristics one can apply to in order to choose an initial paradigm. As mentioned in Rule 1, many of the properties of the application problem are not known until some work has been done. Once the problem is clearer, one can switch to a more appropriate paradigm if the first one does not work. We'll begin with weak methods that are easy to understand conceptually, and to program.

```
 If there is a systematic generator of the solution space AND
    there are rules to prune the space AND
    there are evaluators to determine when the goal(s) has
       been reached,
then  try the heuristic search paradigm first.
   [see Heuristic-DENDRAL or GAMMA as examples]
```

This paradigm is improved if there are rules that can rank order the generator output in order to do "best-first-search." What happens if, instead of a legal move generator, there is only a a plausible move generator? Heuristic search can still be used, but there would be no guarantee of finding all the solutions, nor even of finding one. However, a plausible move generator implies that there exists a fairly strong model of the domain. One can sometimes use the model to improve on the unguided generate-and-test search.

```
  If there is a known goal state and an initial state AND
     there are methods of finding differences between two
        states AND
     there are operators to reduce the differences,
then try means-ends analysis.
```

[see GPS for numerous examples]

Although this is a very attractive paradigm, in real problems the exact goals are rarely known, nor are there obvious ways of determining the differences or distances between solution states.

```
  If the search space is relatively small (in the thousands
        or less) AND
       the space can be represented as an AND/OR tree
        (i.e. goals that can be reached by generating
         independent subgoals),
then try exhaustive search.
```

If there is in addition, a strong model (or theory) in the domain, stronger inference methods can be used. MYCIN's backchaining of production rules fits this model. Note that the premises in the left-hand-sides of the rules can be independently achieved by further firing of other rules. That is, the left-hand-sides generate subgoals that can be satisfied by the firing of the right-hand-sides. [See Shortliffe 76 for details of backchaining.]

2.3 The Blackboard Model

In its most abstract and simplest form, the Blackboard model can be described as follows [Lesser and Erman 1977]:

1. There are diverse **knowledge sources** (KSs) that are kept separate and independent;

2. There is a **global data base**, the blackboard, that is used as a means of communications and interaction among the KSs; and

3. The KSs **respond to changes** in the Blackboard.

The model itself does not specify the structure of the data base, the representational forms of the KSs, nor the mechanisms for response. Different applications have chosen different system designs and implementations, as well as a variety of representations and control mechanisms. As a result, the programs have different organizations and behavior, even though they have the same conceptual origin. However, all programs use variants of generate-and-test, or hypothesize-and-test paradigm--generation of hypotheses on the blackboard followed by evaluation of the plausibility or correctness of the hypotheses. The hypotheses are proposed by the KSs using formal and informal knowledge.

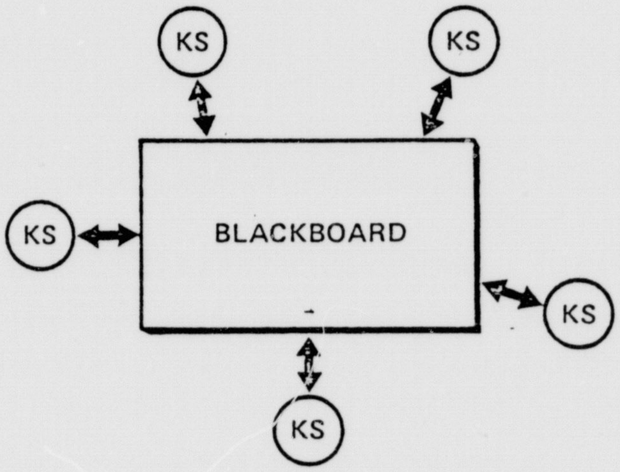
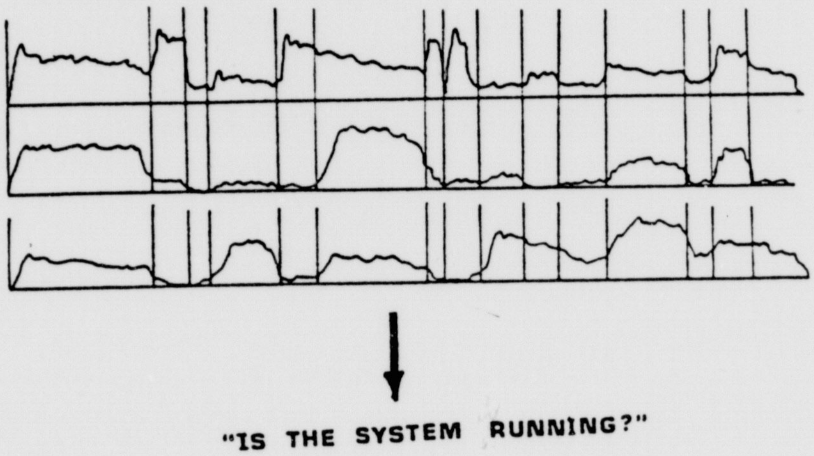


Figure 2.1 The Blackboard Model

2.4 Case Studies in the Use of Blackboard Model

2.4.1 CASE I: HEARSAY-II Speech Understanding System

[Lesser and Erman 1977]



Task Description

The task of the Speech Understanding project is to design and implement a system that "accepts continuous speech from many cooperative speakers of the general American dialect, in a quiet room over a good quality microphone, allow a slight tuning of the system per speaker, by requiring only natural adaptation by the user, permitting a slightly selected vocabulary of 1,000 words, with a highly artificial syntax...in a few times real time..." [Newell, et.al. 1973] Hearsay-II was developed at Carnegie-Mellon University to meet these challenges.

There are several characteristics in the speech problem that led this group to develop the Blackboard model:

1. Large search space.
2. Diverse sources of knowledge.
3. Error and variability, both in the data and in the performance of the KSs.
4. Need for an experimental approach, to assess the contribution of the individual KSs, and to iterate over large amounts of data, and
5. Need for accuracy and speed.

The Blackboard Structure

The blackboard is partitioned into 6 - 8 (depending on the configuration) levels of analysis (see figure 2.2). The sequence of levels forms a hierarchy in which elements on each level could be loosely described as forming abstractions of the lower level. One such hierarchy is, from the lowest to the highest level: parametric, segment, syllable, word, word-sequence and phrase levels. An element represents an hypothesis; for example, on the word level each element represents an hypothesized word consisting of group of syllables on the syllable level. The blackboard can be viewed as a two dimensional problem space with time and information level as its coordinates.

Each hypothesis, no matter on what level, uses a uniform attribute-value structure--some attributes are required for all levels (e.g., its level name). The attributes include "validity rating", an estimate of the "truth" of the hypothesis represented as some integer value. The structural relationship between the hypotheses are represented by links, forming an AND/OR tree over the entire hierarchy. Alternative solutions can be formed by expanding along the OR paths. Because segmentation of speech signals along word boundaries is fuzzy, the blackboard could contain a large number of alternatives.

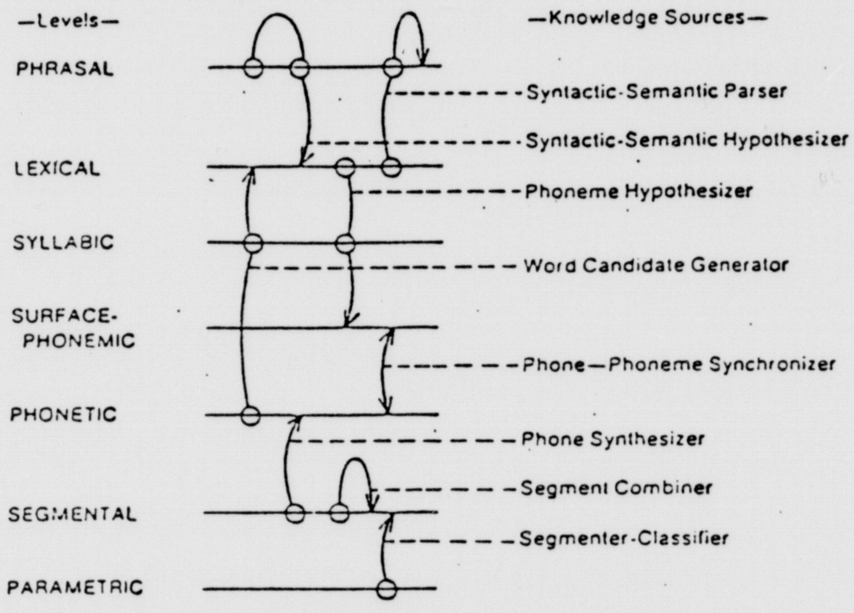


Figure 2.2 A Blackboard Configuration and KSs

Knowledge Source Structure

Each KS has two major components: a precondition and an action. The function of the precondition is to monitor the blackboard for patterns (hypotheses, attributes, values, and links). When a matching pattern is found, the KS is invoked. The data pattern that matches the precondition is called the stimulus frame. When the KS is executed, the action part changes the hypotheses in the vicinity of the stimulus frame. The changes brought about by the action is called the **response frame.**

Both the precondition and action are written as SAIL procedures. Several KSs can be grouped into "modules", and the KSs within a module can share codes and long-term, built-in data.

Control Structure

Although HEARSAY-II is designed to run on distributed, parallel machines, it is easier to understand its organization if we view the control structure serially. The basic components and actions of the control mechanisms are described below:

1. Given a current hypotheses state, execute the preconditions of "selected" KSs. To keep from firing all the preconditions continually, each precondition declares a priori the kind of blackboard changes it is interested in.

2. The precondition firing results in a set of stimulus frames and "invoked" KSs. Associated with each stimulus frame is a descriptive summary of the response frame; i.e. a potential contribution the KS can make to the current hypotheses.

3. A scheduler selects from the numerous alternatives of potential changes, a response frame to be executed. The problem of "focus-of-attention" is defined, in the context of this structure, as a problem of developing a method of minimizing the total number of KS executions to achieve a relatively low rate of error. Thus, the problem of knowledge selection can be viewed as a problem of resource allocation.

4. KS execution modifies the vicinity of the associated stimulus frame, and the operation is repeated with the execution of preconditions.

Miscellaneous Notes

1. The focus-of-attention problem is framed as a scheduling problem. Although various general solutions have been suggested [Hayes-Roth & Lesser 1976], the scheduler needs to know the goals and strategies of the task to be able to evaluate the next best move. It would appear that ultimately one would need a knowledge-based scheduler for effective utilization of the KSs.

2. The preconditions are complex, cycle-intensive procedures that need to search large areas of the blackboard. Each KS needs to determine what changes were made since the last time it viewed the blackboard.

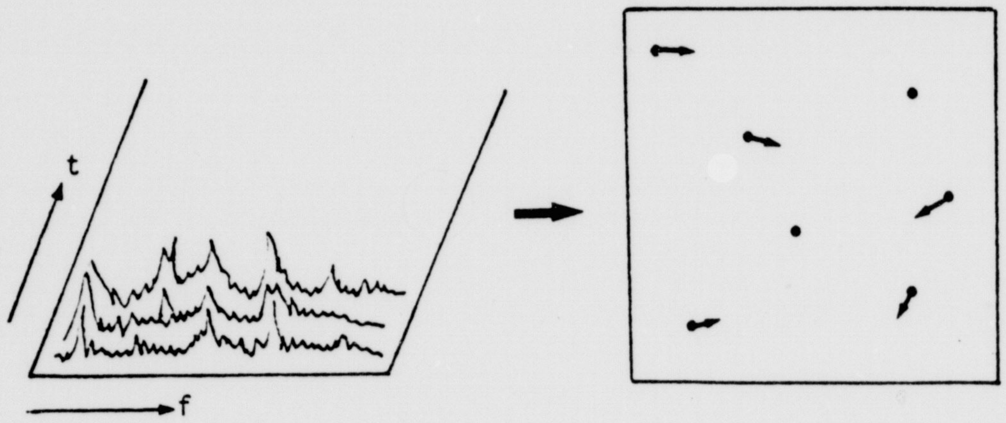
3. H-II maintains alternative hypothesis structures. There is no notion of keeping only the "best" hypothesis. This alleviates some of the problems to be encountered in later systems.

4. Confidence in the hypotheses are represented by an integer between 1 to 100.

Cumulation of confidence is by simple addition. When the confidence in an hypothesis node is changed, the change is propagated up and down the entire structure.

2.4.2 CASE II: SU/X

[Nii and Feigenbaum 1978]



Task Description

The task of the SU/X project was to develop a program that performed analysis of a digitized plot of continuous acoustic signals produced by objects in space. The data was a plot of frequencies over time. The output of the program was a situation plot of the objects that produced the sound as they moved around. (This type of problem is often characterized as a "cocktail party problem" in which the location of each person is to be tracked by listening to their voices.) The front-end signal processing hardware and software detect energy peaks appearing at various spectral frequencies, and follow these peaks over time and space. The spatial configuration is hypothesized by analysis of signals from various listening posts. SU/X was designed to analyze digitized description of these data.

The characteristics of the SU/X problem had some similarities to the speech understanding problem:

1. a large search space,
2. no legal move generator that defined the space of solutions,
3. a low signal-to-noise ratio,
4. many knowledge sources each of which could contribute only bits and pieces to the solution, and
5. real-time processing (but the changes occurred very slowly.)

In H-II, the solution is a sentence on the highest level of the hierarchy; in SU/X the answer was the whole hierarchic structure viewed as the best understanding of the situation at various levels of detail.

Structure of the Blackboard

The blackboard contained a hierarchically organized **current-best-hypothesis** (CBH). In contrast to H-II, each element could have alternative values for the various attributes, but no alternative links. In other words, the hierarchy was structured only as an AND tree, with possible alternative values in each node. The hierarchy represented the situation plot with the highest level representing the scene. The next and subsequent levels in decreasing order represented: the identified objects and their attributes including spatial location; the component sources of sound of the objects; the signal feature groups of the sources; the signals features; and the signal segments which were the input to the program (see figure 2.3).

Because from the point of view of the situation assessment, the entire structure represented the best hypothesis of an unfolding situation, the data structure objects were referred to as elements of the CBH, or **hypothesis elements.** The hypothesis elements formed a linked network, each element representing a meaningful aggregation of lower level hypothesis elements. There was no attempt made to maintain uniformity of attributes across the levels. Each KS knew the descriptor vocabulary on the levels of interest.

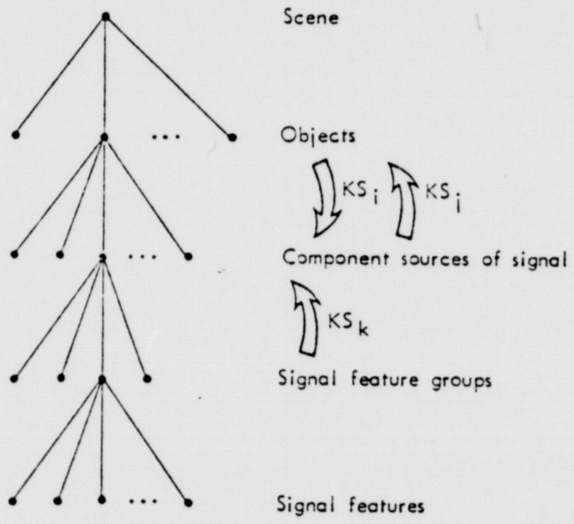


Figure 2.3 The Hypothesis Structure and KSs in SU/X

In addition to the CBH, the blackboard contained five other types of information generated by, and available to, the KSs (see figure 2.4):

1. **Event-list:** All changes made to the CBH, together with its type, were posted on the Event-list.

2. **Expectation-list:** The input often included symbolic information about the objects which may, or may not, have been detected acoustically. This information, together with knowledge about the characteristics of the objects, was used to generate expectations about the signal features to be detected and confirmed.

3. **Problems-list:** This list contained a description of various problems the KSs encountered. For example, a non-firing of rules in a KS that meant, "I should know, but don't"; it also contained information needed from the user.

4. **Clock-events-list:** Because some time-oriented behaviors of the objects were known at each level of analysis, check lists were posted to be periodically reviewed by various KSs. The postings by the KSs were of the form, "Call me at time x, I need to follow up on y."

5. **History-list:** All the processing events were kept, and they were used to generate an explanation of how the CBH came about. Since the program processed events in a breadth-first order, and humans had difficulties in following this processing order, the history-list was used to construct a text that made it appear as though the processing had been in depth-first order. The result was an explanation of the elements in the CBH with all of the support elements, a form much easier for people to understand.

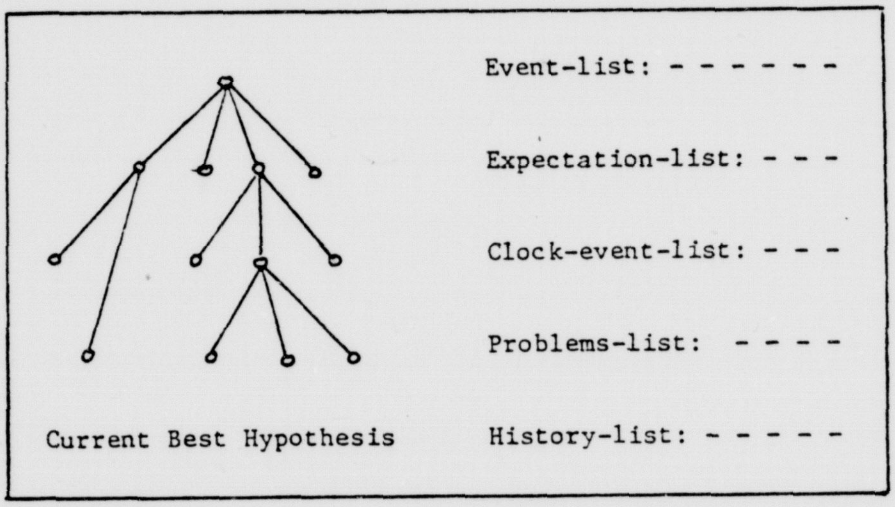


Figure 2.4 Contents of the Blackboard in SU/X

Knowledge Source Structure

Each KS consisted of a precondition and a set of production rules that represented the domain specific knowledge. In contrast to the complex H-II preconditions, a precondition in SU/X consisted of two atoms, a name of an event type and its modifier (new, old, or modified). An event type was one of several predefined types of changes that could be made to the hypothesis elements. In addition to the precondition and the rules, a KS contained a section for binding variables that were used often in rules to minimize recomputation during the match phase. This also served to "freeze" the context until all the appropriate rules in a KS were fired.

The production rules were usually ordered so that rules for the most specific situations were guaranteed to fire before the more general weaker rules. Rules were allowed to change the values in the hypothesis elements and the links between the elements. In addition, the rules posted the type of changes they made on the Event-list, expected events on the Expectations-list, or asked to be recalled later by posting on the Clock-events-list. The variety of event types were used by a control-oriented KS to determine the next action.

Control related functions were represented uniformly as KSs. In addition to the KSs containing domain-specific knowledge, called Hypothesis-formation-KSs, there were two other types of KSs: a KS-activator decided which hypothesis-formation KS was relevant to the currently focussed event and activated it; and the Strategy-KS chose an event from the various event lists to be the next "focussed event". These KSs, the Hypothesis-formation KSs, KS-activators, and Strategy-KS, formed a hierarchy of management structure (see figure 2.5).

Control Structure

Different kinds of activities necessary for problem solving were represented as hierarchically organized control components. The lowest level contained KSs whose tasks were to make primary inferences directly effecting the CBH. The next level held meta-KSs that knew about the capabilities of each of the KSs on the Hypothesis-formation level. The highest level held a KS that determined what hypothesis element to process next.

This control hierarchy should be clearly distinguished from the hypothesis hierarchy. The hypothesis hierarchy represents an a priori established plan for the solution presented by a "natural" decomposition of the solution space. The control hierarchy, on the other hand, represents the arrangement of various problem-solving activities necessary to solve the problem.

Miscellaneous Notes

1. One of the major differences between H-II and SU/X is the in the way KSs are selected for invocation. The type of knowledge contained in the preconditions of individual KSs of H-II were centralized into one KS in SU/X.

2. Because of the differences described in 1, H-II structure is more amenable to distributed computation than SU/X.

3. The hierarchical control in SU/X was an attempt to separate the domain specific knowledge and knowledge about the utilization of these knowledge. It was the first attempt at such an organization, and as such, the structure was rather simplistic. A little richer structure is implemented in AGE.

4. In SU/X the contents of the blackboard was expanded specifically to include control-

type information as a part of the global data structure. Having this type of data globally accessible made the explicit representation of control functions in production-rule form a natural extension to the KS structure.

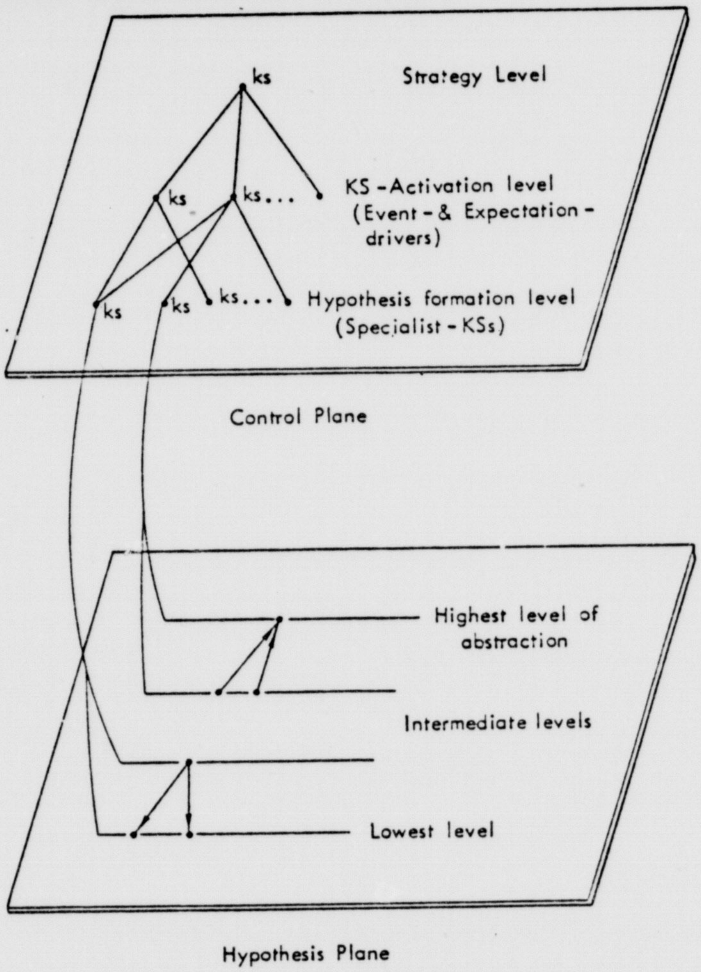
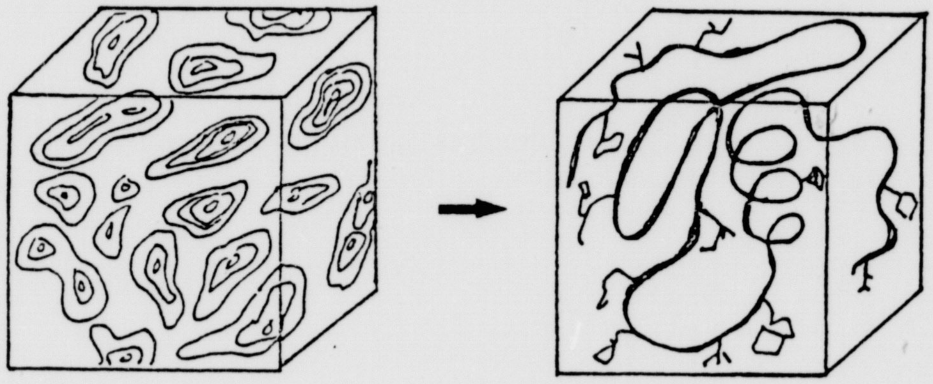

ks
Strategy Level
ks
ks...
KS-Activation level (Event-& Expectation-drivers)
ks
ks
ks...
Hypothesis formation level (Specialist-KSs)
Control Plane
Highest level of abstraction
Intermediate levels
Lowest level
Hypothesis Plane

Figure 2.5 Control Hierarchy in SU/X

2.4.3 CASE III: CRYSALIS

[Engelmore and Terry 1979; known as SU/P in Nii and Feigenbaum 1978]



Task Description

The task of CRYSALIS program is to infer the three-dimensional structure of protein molecules. The protein model is derived from an interpretation of the electron density map of the protein, which in turn is derived from x-ray diffraction data of the crystallized protein. The data typically yield a poorly resolved distribution of the electron density distribution within the protein molecule, and the locations of individual atoms are generally not identifiable. Traditionally, the protein crystallographer represents his interpretation of the electron density map in a ball-and-stick molecular model fashioned from metal parts. These parts are strung together to form a model which conforms to the density map and is also consistent with protein chemistry and stereo-chemical constraints.

The initial CRYSALIS system attempted to "simulate" human model builders. Their strategy is to incrementally build from the most "obvious" regions of the electron density map, outward toward less certain regions, by using the amino acid sequence and stereo-chemical constraints.

Blackboard Structure

The initial attempt to form a hierarchical representation consisting of data and a target molecular model proved unsatisfactory. The primary reason for this was that the electron density data can form an abstraction hierarchy different from the hierarchy in the molecular model (see figure 2.6). In other words, one could form a data hierarchy using various signal processing methods that resulted in an abstraction hierarchy different from the hierarchy of the solution space containing atoms, amino acid residues, and stereo sub-structures. Thus, the hierarchical data structure in the blackboard was extended to contain two hierarchies, one that represented the molecular model, called the **model plane**

hierarchy, and another one representing the abstractions of the data, called the **density plane hierarchy**. With this organization on the blackboard, hypothesis formation activities can be divided into three broad categories: KSs that interpret the primary data into their abstract descriptions, KSs that reason within the model space, and KSs that map information in one plane to the other plane.

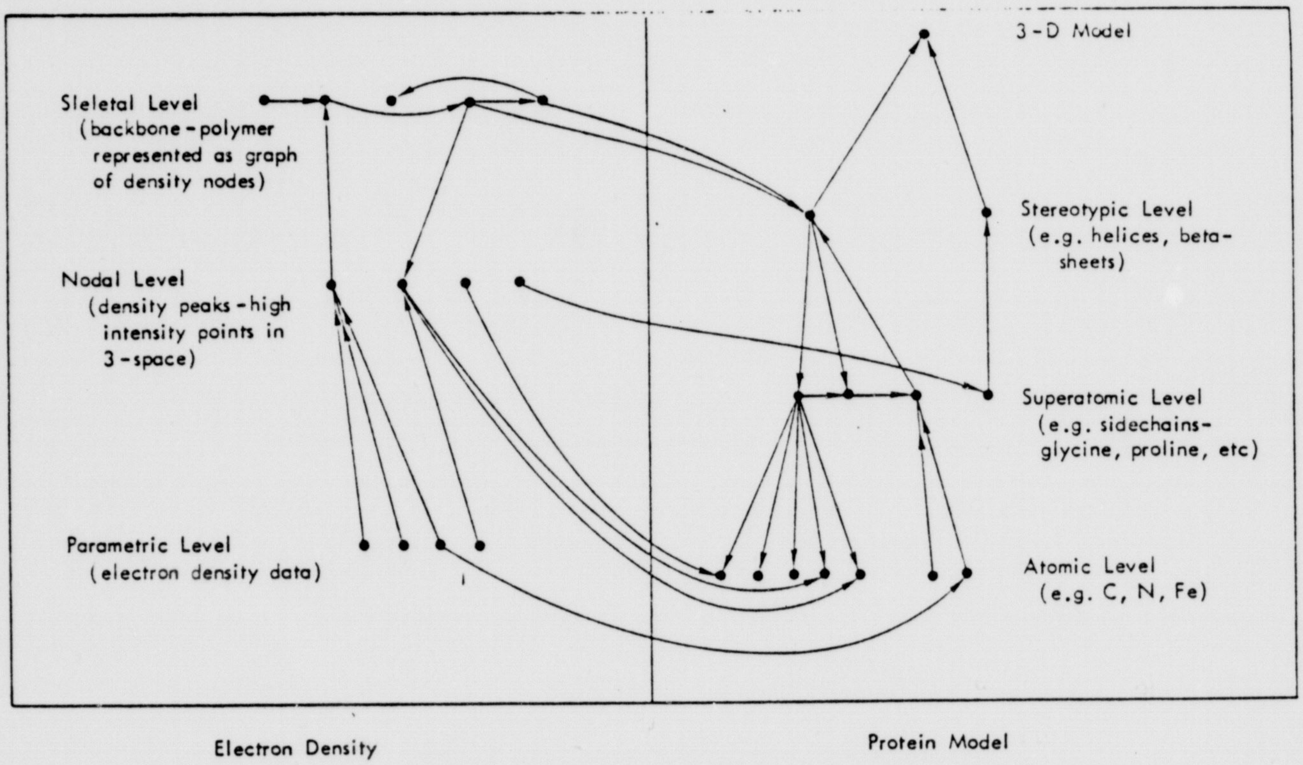

3-D Model
Sleletal Level (backbone-polymer represented as graph of density nodes)
Stereotypic Level (e.g. helices, beta-sheets)
Nodal Level (density peaks-high intensity points in 3-space)
Superatomic Level (e.g. sidechains-glycine, proline, etc)
Parametric Level (electron density data)
Atomic Level (e.g. C, N, Fe)
Electron Density
Protein Model

Figure 2.6 Multiple Hierarchies in CRYSALIS

<u>Knowledge Source Structure</u>

CRYSALIS uses mixed representation for its KSs--some are represented as sets of production rules similar to SU/X, and others requiring extensive numerical calculations are represented as procedures. <u>Control Structure</u>

CRYSALIS uses a three-tiered control structure; all control information is uniformly represented as production rules. The overall control of the system is assigned to the Strategy level. A set of strategy rules governs the choice of task to perform <u>and</u> a region of the hypothesis in which to work. An example strategy rule is:

If (there is a toehold A) and

```
        (A has a certainty > 400) and
        (the direction of the backbone at A is known)
    then (do Expansion.Trace task).
```

At the **Task level**, decisions are made on how to best accomplish a specific task, i.e. which KS or combination of KSs should be used to achieve a goal. This knowledge is contained in the task rules whose left-hand-sides operate on the Event-list (similar to the one used in SU/X) and whose right-hand-sides activate specific KSs. The detailed knowledge about the domain is contained in the KSs on the **KS-level**.

The overall control is **event-driven**. An event is a description of the changes made to the hypotheses posited by KSs in the event-list. Each task is completed (i.e. there is an inner loop of task rules to/from KSs) before the control passes back to the Strategy level where focus of attention is shifted to another region.

Miscellaneous Notes

1. The decision to use multiple hierarchies to represent the problem-space clarified the kinds of KSs that were being used by human experts.

2. In SU/X the control loop was basically:

   focus on hypothesis element,
   select relevant KSs,
   fire the KSs.

   This method entails analysis of events at each cycle to select a focus point in the hypothesis. In CRYSALIS, the control loop is:

   focus on region of the hypothesis,
   set up a goal oriented task,
   fire KSs.

   This method allows for focussing on larger regions on the strategy level, and a more detailed focussing on the more task oriented level.

3. CRYSALIS combines symbolic processing with numerical computation.

4. Pattern recognition of "visual" objects requires complex computation on the LHS of rules. This means that the LHS of rules must be flexible enough to allow searches and loops.

2.4.4 CASE IV: Modeling Planning

[Hayes-Roth and Hayes-Roth 1979]

Task Description

This planning project has two basic objectives--to write a computer program that formulates plans and to insure that the planning processes are psychologically reasonable. We will deal only with the former, a program that does errand planning. The planner program begins with a list of desired errands. The errands differ implicitly in importance and the amount of time required to perform them. The planner also has prescribed starting and finishing times and locations. Ordinarily, the available time does not permit performance of all of the errands. Given these requirements, the planner decides which errands to perform, how much time to allocate for each errand, in what order to perform them, and by what route to travel between successive errands.

Blackboard Structure

The blackboard is partitioned into five planes containing conceptually different categories of decisions. Each plane contains several levels of abstraction of the planning space. The five planes are (refer to figure 2.7):

1. **meta-plan:** Decisions on this plane indicate what the planner intends to do during the planning process. For example, on the policies level the KS specifies general criteria to impose on the problem solution, such as, "the plan must be efficient," or "minimize certain risks."

2. **Plan:** Decisions on this plane indicate actions that the planner actually intends to take. Decisions at each level within this plane specify a more refined plan than those at the next higher level. For example, the Outcomes level indicates what the planner intends to accomplish by executing the final plan, whereas the Procedures level specifies specific sequences of actions (errands).

3. **Plan Abstraction:** Decisions on this plane characterize desired attributes of potential plans. These abstract decisions serve as heuristic aids to the planning process suggesting potentially useful qualities of planned actions.

4. **Knowledge base:** This plane records observations and computations about relationships in the world which the planner generates while planning. This knowledge supports two types of planning functions--the analysis of the current state of affairs, and the analysis of the likely consequences of hypothesized actions. For example, at the errand level, the planner might compute the time required to perform all of the currently intended errands to evaluate the plan's gross feasibility.

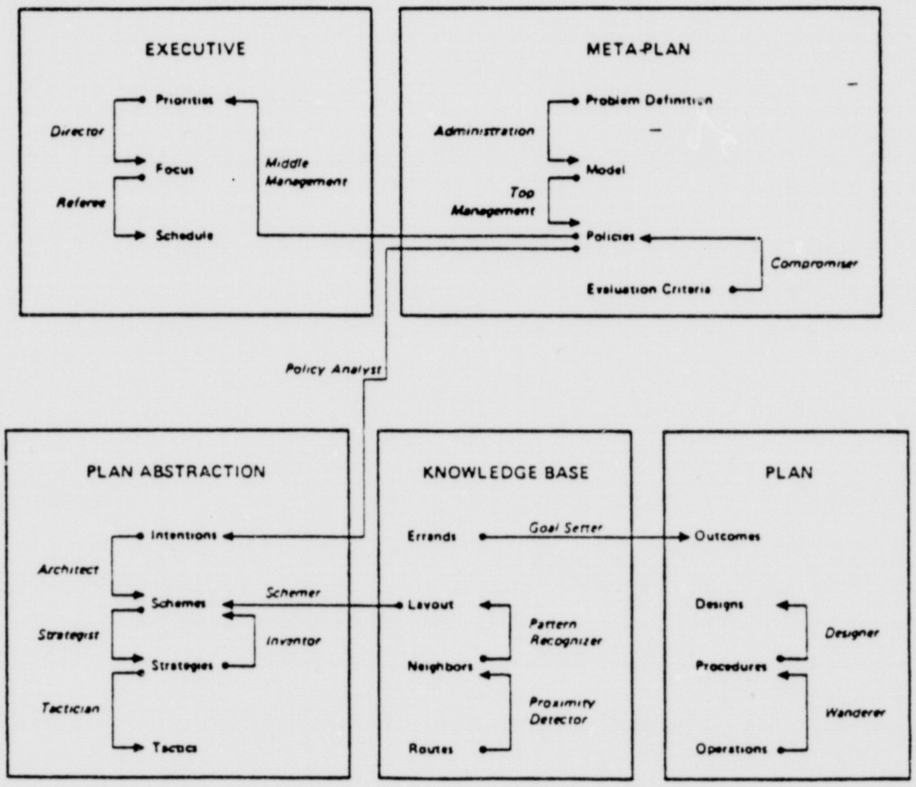


Figure 2.7 Multiple hierarchies and Specialists

<u>Knowledge Source Structure</u> (KSs are called specialists)

Each specialist is structured as "condition -> action" module. The condition component characterizes decisions whose occurrence on the blackboard warrant a response by the specialist. It has two parts, a "trigger" and a "test". The trigger provides a preliminary test of the specialists' relevance by looking at the focus node. The "focus node" is defined as the node most recently added or modified. The test specifies all other prerequisites of applicability.

The action component defines the modification it makes to the blackboard when executed. Most specialists produce new nodes with particular attributes; a few modify the attributes of existing nodes.

<u>Control Structure</u>

The system consists of four global data structures: the map on which the errands occur; the blackboard; the agenda, a list of invoked specialists that need to be scheduled; and the event-list, a process history for tracing and debugging.

On each cycle, the current focus node triggers some number of specialists which are put on the agenda. The pending specialists on the agenda are processed in three phases:

1. The invocation phase evaluates the test part of the specialists on the agenda. Specialists whose tests are satisfied are invoked. If there are no invoked specialists, the program terminates.

2. In the scheduling phase one of the invoked specialists is recommended for execution. The bases of the recommendation are recency of invocation and current focus decision, the node most recently modified. In other words, the program recommends executing the most recently tested specialist whose actions will occur in the region most recently changed.

3. In the execution phase the scheduled specialist is executed. The program immediately evaluates the trigger of all specialists against the focus node (the one just changed), and adds those specialists whose triggers are satisfied on the agenda list.

Miscellaneous Notes

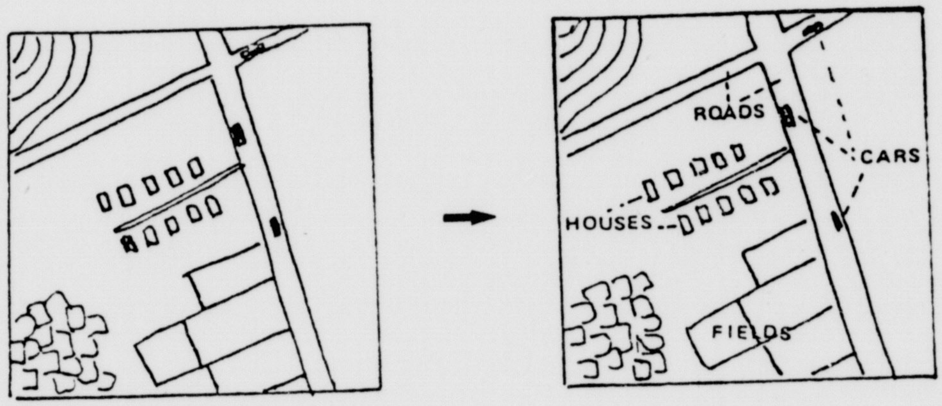
1. All control related modules are on one plane as in SU/X, although the processes are slightly different.

2. The two-pass evaluation of the preconditions is a modification to the H-II precondition evaluation.

3. Although the control is event-driven, it emphasizes recency for reasons that has to do with a psychological model. 4. There are many more KSs (specialists), each of which is smaller than in other applications. In H-II there were approximately ten powerful KSs; in this program there are about 50 simpler KSs.

2.4.5 CASE V: Structured Analysis of Complex Aerial Photographs

[Nagao, et.al. 1979]



Task Description

Given a large aerial photograph of a complex suburban area taken at low altitude, the task of the program is to identify and label objects in the photograph. The variety of objects includes, among other things, cars, houses, rivers, roads, crop fields, etc.

Blackboard Structure

The hierarchic structure on the blackboard is organized as follows (refer to figure 2.8): the lowest level contains regions segmented according to multi-spectral properties called "elementary regions". The attributes of each elementary region is its average grey level, size, location, and basic shape features, together with pointers to the digitized picture indicating its relative position. A combination of elementary regions forms an object on the "characteristic regions" level. On this level, seven characteristic features are extracted: large homogeneous regions, elongated regions, shadow regions, shadow-making regions, water, vegetation, and high-contrast-texture regions. These levels are further abstracted into "object" and "object category" levels.

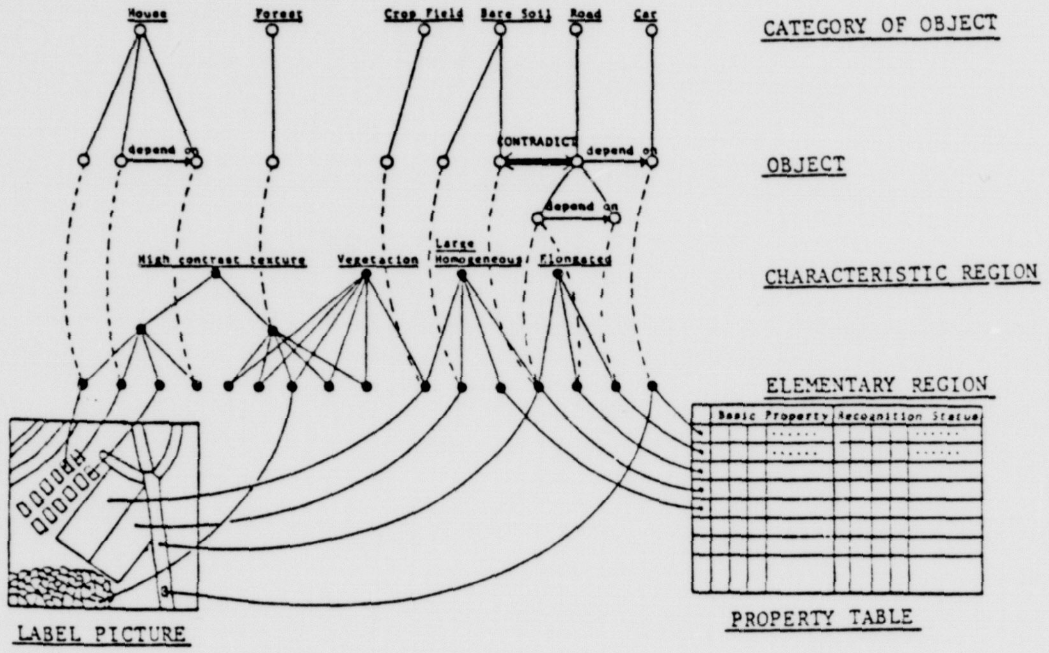


Figure 2.7 Structure of the Blackboard

Knowledge Source and Control Structure

Each object category in the system is a priori specified and has associated with it KSs that can recognize or reject objects in that category. For example, there is a "house expert" that, when give an object, can recognize it as a house or reject the possibility of it being a house. These KSs are further specialized into data-driven KSs and model-driven KSs. Data-driven KSs are capable of combining objects on the characteristic-regions level into identifiable "objects". The model-driven KSs pick up objects in characteristic-regions level based on already identified objects, e.g. identify cars once a road has been recognized.

Miscellaneous Notes

1. This is a working program that performs well, but the details are hard to decipher from the technical paper.

2. The notion of semantically driven vision programs has existed for some time. Projects headed by Riesman, Nagao, and Kanade all have programs in varying stages of development that use the blackboard model to integrate symbolic information with signal analysis programs.

3 The Blackboard Framework in AGE

In AGE the Blackboard-based program design consists of three major components. They are:

1. **The blackboard**: The blackboard contains hierarchical data structures that represent the task domain as a hierarchy of analysis levels of the task.

2. **The knowledge base**: The knowledge of the task domain can be represented in two different ways. The description and relations among the objects in the domain can be represented in an object-centered, representation scheme as implemented in the Units package [see Stefik 79]. The knowledge that uses these facts and/or information on the blackboard to perform a task is represented in production rules. A set of rules that "belong together" are called Knowledge Sources.

3. **The control**: The selection and invocation of the KSs, and the selection of items on the blackboard for focus of attention, are independent of the knowledge base. They constitute the control mechanisms to be specified by the user.

The combined process of KS selection and incremental changes to the blackboard is viewed as a general process of hypothesis formation. Consequently, the data structure in the blackboard is referred to as hypothesis structure.

3.1 The Hypothesis Structure

The hypothesis structure is hierarchically organized. There may be more than one hierarchy (generally known as blackboard planes--see Cases III and IV) that defines the hypothesis. Each hierarchy consists of **hypothesis elements** integrated by **links** that represent support from above and support from below; the links are called **expectation-links** (or model-derived links) and **reduction-links**, respectively.

A hypothesis element is a named node (object) in the hypothesis that represents an aggregation (summary, interpretation, abstraction, etc.) of lower level hypothesis elements. Each element contains a description in the form of **attribute-value-weight** triplets that are meaningful at that particular level of the hypothesis. All nodes on the same level have element names of the form <level-name n>, N >= 1; or a name assigned by the user. Each name is unique within the hypothesis structure.

A link in the hypothesis structure represents a relationship between elements in the hypothesis, and indirectly represents the rule(s) that created the link. A link can point to a hypothesis element from any other element, including one from its own level. (Note that by doing this we are allowing for data structure that is not strictly hierarchical.) A link is represented as an attribute of the element it is rooted in; the name of the link is the value of the link-attribute.

The hypothesis is built incrementally by rules that add or modify hypothesis element(s), or relationship(s) between hypothesis elements directly, by:
1. interpreting data (support from below),

2. specializing or instantiating a more inclusive hypothesis element (support from above); or
3. indirectly, by generating expectation(s) from:
   a. a model ("theoretical" support), or
   b. a more inclusive hypothesis element that must be verified by data.

As in SU/X, the blackboard contains, in addition to the hypothesis structure, information that is accessible to the KSs and that can be viewed as a part of the hypothesis:

1. Event-list: A list of the changes made to the hypothesis elements; the types, the names of originating KSs, and the specifics of the changes.

2. Expectation-list: A list of all the expectations that still need to be met, together with the changes to be made to the hypothesis elements, if and when the expectations are met.

3. Goal-list: A list of all the goals that still need to be achieved, together with the changes to be made to the hypothesis elements, if and when the goals are met.

4. History-list: A detailed history of all the changes made to the blackboard items. This list is essentially a list of the right-hand-sides of the rules that fired, together with their predecessor and successor rules.

3.2 The Knowledge Source Structure

The knowledge necessary to accomplish the goals of the program is represented in production rules. These rules are organized into one or more sets of rules called the Knowledge Sources (KSs). The program may contain one set of homogeneous rules; these may be chained to represent a line of reasoning in the style of the MYCIN program [Shortliffe], or left loosely organized. Or the object program may contain many KSs to be used as hypothesis generators and validators as in all the Cases mentioned above.

A KS is a labeled set of production rules which are a priori deemed to belong together. For example, a model-like KS may contain rules that are organized around some object or concept; a data-oriented KS may contain rules that generate hypotheses from particular data elements. A KS may be thought of as a mega-chunk of knowledge. It has associated with it a precondition for its invocation. A precondition is a pattern generated by the RHSs of rules in some other KS. Local contexts applicable to the KS can be established before the rules are actually evaluated; these include a rule evaluation strategy (single hit, multiple hit, once-only), links to be generated by the rules, and local variable bindings.

The KSs may be manipulated by other higher level knowledge sources that are part of the control structure. These high level KSs can be used to represent problem solving strategies, task definitions, and other control mechanisms (see Cases II and III).

Rules

Each rule consists of a left-hand-side (LHS) and a right-hand-side (RHS). The LHS specifies a set of conditions or patterns for the applicability of the rule (see Davis 1977 for comprehensive discussion of production rule formalism). The term "applicability" can mean that all of the specified conditions must be true or only some need be true. Because of the wide range of possibilities for evaluating the LHS, AGE asks the user the define "applicability" in the form of a function to serve as the **LHS evaluator**. Some simple LHS-evaluators, such as "all conditions must be true," are provided.

The RHS represents the implication to be drawn under the situation specified by the LHS. These implications are in the form of changes that are to be made to the hypothesis structure. Currently there are three kinds of changes that the RHS can make:

1. actual changes to the hypothesis elements or the links, by using a PROPOSE argument; these changes are called "events" and are posted on the Event-list <u>after</u> the changes are made.

2. desired changes can be posted on the Goal-list by using an ACHIEVE argument; no actual change is made to the hypothesis until the desired changes have been made.

3. expected changes can be posted on the expectation-list by using an EXPECT argument; no actual change is made until the expected changes occur in the hypothesis or appear as input data.

The implication can have associated with it a probability (or some informal weight) that reflects the confidence in the implication for the particular situation. This probability will be reflected in the attribute-value-weight triplet of the hypothesis element when the rule is executed. The integration of weights in the rules and the weights already assigned in the hypothesis can be interpreted in many ways, and can be computed in many ways. For this reason, AGE allows the user to specify the computation in the form of a function to serve as a weight adjuster. There are some predefined weight-adjuster functions available in AGE, as well as default "passive match" when no function is defined.

3.3 The Control Structure

There are several functional components grouped under the heading of Control. They are:

1. Input: specifying the format and names of the input; and getting or asking for input data from a file or from a terminal.

2. Initialization: preprocessing of data, if necessary; and indicating the first KS to be invoked after initialization.

3. Kernel control: This component actually drives the Blackboard control and will be discussed below in more detail.

4. Termination: There is no uniform way in which Blackboard-based programs terminate. The user must specify the condition under which the program is to terminate; for example, occurrence of some event, nothing left to do, etc.

5. Postprocessing: postprocessing; for example, printing the hypothesis or generating an "explanation" using the hypothesis and the history list.

Kernel Control

The primary function of the kernel control is to select an item on the blackboard to process and invoke the appropriate KS(s). This activity consists of two major steps (see figure 3.1):

1. Inference generation: Within the invoked KS the rules either (1) PROPOSE a change in the hypothesis, (2) indicate that some change in the hypothesis is EXPECTed to occur, or (3) indicate that the KS desires a particular value or state to be ACHIEVEd. Each of these actions taken by the RHS is called a step.

2. Focus of attention: First, select a step type (an event, an expectation, or a goal) to process next. Form the selected event-list, expectation-list, or goal-list, choose a specific step. This has the affect of choosing an hypothesis element for focus. Then, choose a relevant KS, and invoke that KS. KS selection may result in complex processing associated with each step type, for example, backward-chaining of rules to achieve a goal.

To allow flexibility, both in the selection of focus-of-attention item and in the invocation of KSs, the kernel control is subdivided into smaller modules to be specified by the user.

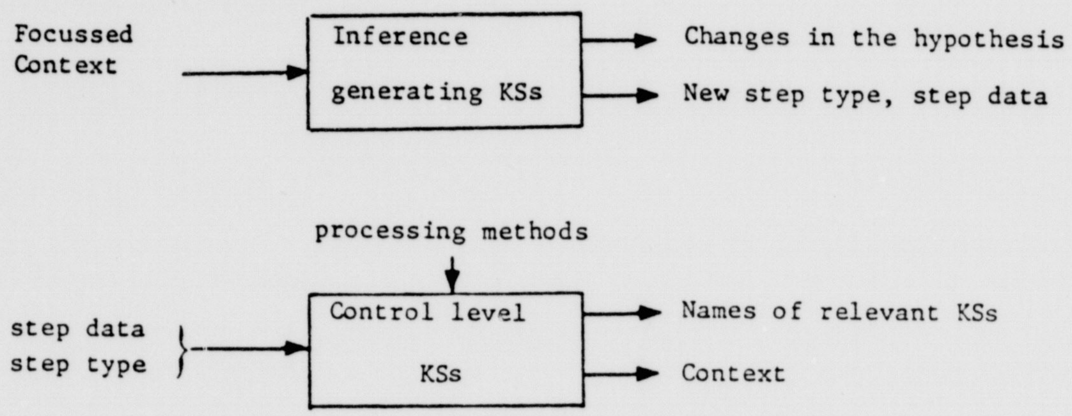


Figure 3.1 Functional Separation of the KSs

Control Macros

Because the control mechanisms are steeped in detail that is confusing to novice users, we have prepackaged two rather simple control structures. They are useful for event-driven control and for expectation-driven control.

Event-driven Control Macro: Event-driven hypothesis formation is characterized by incremental formation of hypothesis elements from evidence found in data or in lower level hypothesis elements. The elements can be processed by rules either on the basis of first-in-first-out, first-in-last-out, or best-first. These correspond roughly to breadth-first, depth-first, and best-first processing of the hypothesis space. When an element is chosen to be processed, it is termed "focussed". The event-type associated with the focussed element determines which KS is to be invoked next.

An event-driven process has the following general structure:

1. <rule> modifies the hypothesis and causes
   an event, with associated event-type.
2. If <event-type> = <KS precondition> then invoke
   the KS. [go to 1]

Expectation-driven Control Macro: Within an expectation-driven system, expected states of the hypothesis elements are generated by the domain rules. The expectation can be based on previously generated hypothesis elements. These elements can be created by using the event-driven technique. The rules that generate the expectation are normally grouped around objects, much like the way in which schemata or frames are organized. These groups usually form models of objects from which properties of objects can be inferred or expected to occur. For example, a rule of the form:

```
if (isa disease oad) and (isa severity severe)
  then (expect RV > 200)
     (expect RV/TLC.RATIO >20)
```

has a schema-like flavor. Of course the model, in this case a model of OAD disease, can be represented in Units. There are both access and storage interfaces to information represented in the Units package in the AGE rules.

In order to determine if a specified expectation has been met, or can be met, the user must provide an expectation evaluation function. AGE will always check to see if the expected situation has occurred by monitoring the incoming data (if data flows in over time) and the Event-list.

The expectation-driven control has the following general structure:

1. <rule> generates expectation(s).
2. If an expectation is met, then modify the hypothesis as specified. This action generates an event with an associated event-type.
3. If <event-type> = <KS precondition> then invoke the KS. [go to 1]

Miscellaneous Notes

1. Currently there is no capability to set up a control mechanism similar in affect to the one in H-II. This is due partly to the fact that the preconditions in AGE are expressions consisting only of atoms that are names of event-types. If we allow general Lisp expressions as preconditions, a part of the differences can be eliminated.

2. The current loop within the control structure is too time consuming. A control structure similar to CRYSALIS may be more practical.

4 Rules for Knowledge Engineers

If at first you don't succeed, try again.

"Trying to find the solution, we may repeatedly change our point of view, our way of looking at the problem. We have to shift our position again and again. Our conception of the problem is likely to be rather incomplete when we start the work; our outlook is different when we have made some progress; it is again different when we have almost obtained the solution..." [Polya].

One of the difficulties of writing knowledge-based programs is that at least two parties are constantly shifting their points of view: the domain expert and the knowledge engineer. As the knowledge in the program accumulates and the problem becomes clearer, the knowledge engineer may find better ways to represent and process the knowledge. The resulting behavior of the program may inspire the expert to shift his view of the problem, creating for the knowledge engineer further problems to be solved. Development of expert programs involves a process of finding a workable relationship between experts and programmers and slowly evolving a program structure that will work.

To develop a viable product, plan to throw one (or more) away.

Even in a well defined problem, the first system built to solve the problem is barely useable as a finished product. The first system has many patches made to meet unexpected problems, is too slow and too large, and is awkward from having deviated from the original design. It needs to be redesigned and rebuilt, incorporating lessons learnt in the first system. In working with ill-defined, ill-structured problems, a knowledge engineer should be aware that it may take several iteration on system design and implementation before one satisfactory one would emerge. It should be noted, however, that once knowledge has been extracted from an expert the knowledge extraction process need not be repeated. The knowledge representation may need to be changed, but that is the knowledge engineer's problem, not the expert's.

To have a successful project, choose the "right" problem.

Because AI is a young and emerging field, the concepts and techniques are few, and fewer still have been tested in "real-world" applications. The knowledge engineer must carefully choose those problems for which there are tools, conceptual or otherwise, or for which tools can be easily built. Some heuristics for choosing the "right", or at least a viable, application are:

1) Choose a problem which is well bounded, one in which large amounts of specialized knowledge may be needed but which does not require general knowledge of the world.

2) Choose a domain in which experts have a common language or representation with which they communicate.

3) Choose an application for which there is a general interest and need.

4) Choose a problem for which there are recognized experts.

5) Choose a non-trivial problem, but one that can be solved within a few hours by a human. If a problem takes days to be solved by a person, it's probably too ill-structured to be solved using current AI techniques.

If you don't have an enthusiastic expert and if he isn't willing to spend a lot of time, then forget it.

A knowledge engineer is a conduit through which an expert's knowledge get codified, stored, and used by a computer. If the expert is not willing to spend the necessary time to expose the knowledge, then no amount of fancy programming will result in a knowledge-based program. Also the expert, as a potential user, could play an important role in the transition phase from an experimental program to a useful product.

If you want to do any serious application, you need to meet the expert more than half way and learn the basics of his field of expertise--don't expect the expert to meet you half way.

If your expert has no scientific background or has never been exposed to the world of computers, your job is harder.

The concept of using informal knowledge is a difficult one to explain even to experienced programmers and scientists; but to get a non-programmer to think in terms of domain primitives, heuristics, inference, etc., is almost an impossible task.

Use carefully chosen test cases for the orderly development of your program.

In building knowledge-based programs, testing serves several purposes, in addition to ensuring correctness. First, it is a method by which the attention of the expert can be focussed on specific problems. Second, by doing so, highly specific "expert" solutions can be obtained. Third, by iterating on similar cases, these solutions can be generalized. Finally, the performance level of the program can be improved in an orderly manner by incrementally accumulating and testing the knowledge base.

In dealing with problems with noisy data, choose test data carefully.

In order to improve the problem solving capability of the program in an orderly manner, starting with the simplest problem and progressing gradually to more difficult ones, the content of the data must be carefully controlled. That is, at every instance the content of the data must be precisely known and furthermore, must be ranked in order of difficulty. In signal analysis problems one obvious strategy is to use simulated data. But, simulated data have some drawbacks: (1) When the real data is noisy, it is impossible to build a simulator that can also simulate realistic noise. (2) Real data tend to be much richer than most simulated data. By using simulated data, much of the detailed knowledge needed to process real data is missed. An ideal strategy for data selection appears to be to alternate between real data and simulated data: first, use the simulated data to find solutions to obvious problems, and then use real data to refine the knowledge to deal with subtleties that are not found in simulated data.

Choose a "classic" case as the first example problem to be attacked by the program.

In order for the domain expert to explain "what" knowledge he needs and "how" he uses this knowledge, he needs a set of examples or cases as a vehicle. Every field has "classic" examples--representative, non-trivial problems that are well understood and agreed-upon among experts. One of the major advantages of a classic example is that it exposes most of the essential kinds of knowledge and reasoning involved in the rest of the problem.

To get an expert hooked on the program, have a running program immediately, even if it does only simple things.

Although an expert may know thousands of facts and rules for solving a problem, it is extremely difficult for him to state them in a vacuum. Once a program is running using some of his knowledge, the program serves as a lubricator for the flow of his "hidden" expertise. It triggers other knowledge to be encoded and other test problems to be tried.

If you can't find an appropriate AI paradigm for the problem, build a program that "thinks" the way the expert does.

Polya says, "Modern heuristics endeavors to understand the process of solving problems, especially the mental operation typically useful in this process." Trying to understand and to mimic the processes by which the expert solves his problem is an important part of the knowledge engineering activity. It is important not only for scientific reasons, but also to sustain the interest of the expert who needs to identify with a program that "thinks" like him. Short of this goal, the program needs to at least construct lines of reasoning that the expert can easily understand.

If the program doesn't explain, who will?

Explanation of the program's behavior plays three important roles in knowledge-based programs:

1) For a domain expert, the process of testing the program's performance is a very difficult one. What humans consider to be simple problems may prove quite difficult for a machine. This is especially true when dealing with man's "intelligent eyes". A person can detect objects and signal features more accurately and quickly than can a program with the current state of technology. Conversely, there are problems that are easier for a program to perform, like keeping track of many things at once. One effective method to help the expert is to have the program explain its path of reasoning in a language natural to him. This allows the expert to interact with and understand the program without going into the details of program code.

2) In addition to the "bugs" which occur in ordinary programs, knowledge-based programs have "bugs" in their knowledge bases. These "bugs" arise from clerical errors, incomplete knowledge, inappropriate use of knowledge, or "errors" of disagreement among experts. In order for the expert to correct the knowledge base he needs to know the context in which the errors occurred--what knowledge was used, or not used, and why.

3) In programs that contain uncertain knowledge, a user <u>cannot</u> blindly accept the results without considering the line of reasoning that led to the result.

These "explanations" need to be in a language that is "natural" in the domain. Sometimes, it may be English; for chemical problems, it may be chemical graphs.

If you are dealing with anything but facts, you must face "uncertainty".

For understandable reasons, experts are reluctant to have heuristic knowledge treated in the same manner as facts. Knowledge-based programs need facilities that allow the expert to express statements like, "I strongly believe that...", "I'm definite that....", and "The evidence suggests that..." Concurrent to representing the less then definite information, programs must have some reasonable way of cumulating and computing the believability of the solutions based on the uncertain knowledge.

If you eventually want a high-performance program OR want the expert to take over the program, the knowledge must be easily accessible and modifiable.

In expert programs, the knowledge bases need to be frequently modified and augmented. It is important that the representation and implementation of knowledge be carefully chosen, and appropriate facilities provided to make the knowledge easily accessible and modifiable. Decision-tree programs, often mistaken for programs written using production rules, lack this essential ingredient.

If you, as a knowledge-engineer, are also the expert, then you will probably face some difficulties. Although there have been successful programs in which the knowledge engineer was also the expert, generally, it is almost impossible for a person to objectively assess what he knows, and doesn't know, and at the same time analyze how he is using his knowledge to solve a problem. He also tends to bring a biased view of the task or programming methodology, which a "disinterested" second party could detect and correct.

Although we have found the above rules useful in our work, they are only heuristics. To put them, and this document in general, in proper perspective, we would like to quote Polya again: "'To apply a rule to the letter, rigidly, unquestioningly, in cases where it fits and in cases where it does not fit, is pedantry. Some pedants are poor fools; they never did understand the rule which they apply so conscientiously and so indiscriminately. Some pedants are quite successful; they understand their rule, at least in the beginning (before they become pedants), and chose a good one that fits in many cases and fails only occasionally. To apply a rule with natural ease, with judgment, noticing the cases where it fits, and without ever letting the words of the rule obscure the purpose of the action or the opportunities of the situation, is mastery."

4.1 What do I do about....? Some Often Asked Questions about Blackboard Model

We have listed below some of the questions that are asked often by people in the early stages of designing knowledge-based programs. We have made no attempt to answer these questions at this point. We hope that some of the examples we presented will help you in answering these questions.

Given the knowledge needed to achieve the goals of the program, how does one decide what knowledge goes into what knowledge sources?

How do I go about constructing the appropriate hierarchical structure?

When should information be represented as levels, nodes, attributes, or links?

When should I use the frame-like Units representation? production rule representation?

How much information should go into one rule? i.e. What is the appropriate grain size of a rule in application programs?

What kind of knowledge utilization method should I use (i.e. what control mechanisms are best for my problem)?

When does expectation-driven processing pay off?

How important is it to be able to express rules in English?

Do we really need to represent inexactness in the knowledge? How do I get the necessary knowledge from an expert? Should I take protocols?

REFERENCES

Brooks, Frederick P., Jr., The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, 1975.

Davis, R. and King J., *An Overview of Production Systems*, Machine Intelligence 8: Machine Representation of Knowledge, Elcock, E.W. and Michie, D. (eds.), John Wiley, 1977.

Engelmore, R. and Terry, A., *Structure and Function of the CRYSALIS System*, Proc. of IJCAI 6, 1979, vol. 1, pp. 251-256.

Feigenbaum, Edward A., *The Art of Artificial Intelligence: I. Themes and Case Studies of Knowledge Engineering*, Proc. IJCAI 5, 1977, pp.1014-1029.

Hayes-Roth, B. and Hayes-Roth, F., *Modelling Planning as an Incremental, Opportunistic Process*, Proc. IJCAI 6, 1979, vol. 1, pp 375-383.

Hayes-Roth, F. and Lesser, V.R., *Focus of Attention in the HEARSAY-II Speech Understanding System*, Proc. IJCAI 5, 1977. pp.27-35.

Lesser, V.R. and Erman, L.D., *A Retrospective View of the HEARSAY-II Architecture*, Proc. IJCAI 5, 1977, pp. 790-800.

Nagao, M., Matsuyama T., and Mori, H., *Structured Analysis of Complex Photographs*, Porc. of IJCAI 6, 1979, vol. 2, pp. 610-616.

Newell, Allen, *Heuristic Programming: Ill-structured Problems*, Progress in Operations Research , vol. iii, John Wiley, 1969.

Newell, A., Barnett, J., Green, C., Klatt, D., Licklider, J.C.R., Munson, J., Reddy, R., and Woods, W., Speech Understanding System: A Final Report of a Study Group,, North-Holland, 1973.

Nii, H. P. and Feigenbaum, E.A.,"*Rule-based Understanding of Signals*, in Pattern-Directed Inference Systems, D.A. Waterman and R. Hayes-Roth (eds.), Academic Press, 1978.

Nii, H.P. and Aiello, N., *AGE: a knowledge-based program for building knowledge-based programs*, Proc. IJCAI 6, 1979, vol. 2, pp. 645-655.

Polya, George, How to Solve It, Doubleday Anchor Books, New York, 1957.

Shortliffe, Edward H., Computer-Based Medical Consultation: MYCIN, American Elsevier, New York, 1976.

Stefik, Mark, *An Examination of a Frame-structured Representation System*, Proc. IJCAI-6, vol. 2, pp. 845-852.

Waterman, D.A. and Hayes-Roth, R. (eds.) Pattern Directed Inference Systems, Academic Press, 1978.

Weinberg, Gerald, M., The Psychology of Computer Programming, Van Nostrand Reinhold, New York, 1971.