The Organization of Expert Systems*, A Tutorial

Mark Stefik, Jan Aikins, Robert Balzer, John Benoit, Lawrence Birnbaum, Frederick Hayes-Roth, Earl Sacerdoti**

Xerox Palo Alto Research Center, Palo Alto, CA 94304, U.S.A.

Recommended by Daniel G. Bobrow

ABSTRACT

This is a tutorial about the organization of expert problem-solving programs. We begin with a restricted class of problems that admits a very simple organization. To make this organization feasible it is required that the input data be static and reliable and that the solution space be small enough to search exhaustively. These assumptions are then relaxed, one at a time, in case study of ten more sophisticated organizational prescriptions. The first cases give techniques for dealing with unreliable data and time-varying data. Other cases show techniques for creating and reasoning with abstract solution spaces and using multiple lines of reasoning. The prescriptions are compared for their coverage and illustrated by examples from recent expert systems.

1. Introduction

Twenty years ago, Newell [29] surveyed several organizational alternatives for problem solvers. He was concerned with how one should go about designing problem-solving systems. Many techniques have been developed in artificial intelligence (henceforth AI) research since then and many examples of *expert systems* have been built. Expert systems are problem-solving programs that

* There is currently much interest and activity in expert systems both for research and applications. A forthcoming book edited by Hayes-Roth, Waterman, and Lenat [21] provides a broad introduction to the creation and validation of expert systems for a general computer science audience. An extended version of this tutorial, which introduces concepts and vocabulary for an audience without an AI background, will appear as a chapter in the book.

** Additional affiliations: J. Aikins, Hewlett-Packard, Palo Alto, CA; R. Balzer, USC/Information Sciences Institute, Marina del Rey, CA; J. Benoit, The MITRE Corporation, McLean, VA; L. Birnbaum, Yale University, New Haven, CT; F. Hayes-Roth, Teknowledge, Palo Alto, CA; E. Sacerdoti, Machine Intelligence Corp., Palo Alto, CA.

Artificial Intelligence **18** (1982) 135–173

0004-3702/82/0000–0000/$02.75

solve substantial problems generally conceded as being difficult and requiring expertise. They are called *knowledge based* because their performance depends critically on the use of facts and heuristics used by experts. Expert systems have been used as a vehicle for AI research under the rationale that they provide a forcing function for research in problem solving and a reality test.

Recently some textbooks have appeared that organize principles of AI (e.g., [30]) and give examples of advanced programming techniques (e.g., [6]). However, there is no guidebook for an expert systems designer to the issues and choices in designing a system. Furthermore, an unguided sampling of expert systems from the literature can be quite confusing. Examples are scattered in various journals, conference proceedings, and technical reports. Systems with seemingly similar tasks sometimes have radically different organizations, and seemingly different tasks are sometimes performed with only minor variations on a single organization. The variations reflect the immaturity of the field in which most of the systems are experimental. From the diversity of experiments one would like to extract alternatives and principles to guide a designer.

This tutorial is organized as follows: Section 2 is a catalog of some generic expert tasks. For each task there is a checklist of requirements and key problems associated with expert performance of the task. The purpose of this section is to extract a set of architecturally relevant issues that cover a variety of problem-solving tasks. Section 3 presents the substance of the organizational ideas. It addresses the issues from Section 2 and makes prescriptions.

2. A Characterization of Expert Tasks

In this section we will consider several generic tasks that experts perform. Examining these tasks will help us to understand what makes expert reasoning difficult. The difficulties provide a guide to architectural relevance; they enable us to focus on issues that relate to critical steps in reasoning.

Interpretation

Interpretation is the analysis of data to determine their meaning.

Example. Interpretation of mass spectrometer data [2]. In this case, the data are measurements of the masses of molecular fragments and interpretation means the determination of one or more chemical structures.

Requirements. Find consistent and correct interpretations of the data. It is often important that analysis systems be rigorously complete, that is, that they consider the possible interpretations systematically and discard candidates only when there is enough evidence to rule them out.

Key problems. Data are often noisy and errorful, that is, data values may be missing, erroneous, or extraneous.

(1) This means that interpreters must cope with partial information.

(2) For any given problem, the data may seem contradictory. The interpreter must be able to hypothesize which data are believable.

(3) When the data are unreliable, the interpretation will also be unreliable. For credibility it is important to identify where information is uncertain or incomplete and where assumptions have been made.

(4) Reasoning chains can be long and complicated. It is helpful to be able to explain how the interpretation is supported by the evidence.

Diagnosis

Diagnosis is the process of fault-finding in a system (or determination of a disease state in a living system) based on interpretation of potentially noisy data.

Example. Diagnosis of infectious diseases [34].

Requirements. Requirements include those of interpretation. A diagnostician must understand the system organization (i.e., its anatomy) and the relationships and interactions between subsystems.

Key problems. (1) Faults can sometimes be masked by the symptoms of other faults. Some diagnostic systems ignore this problem by making a *single fault assumption.*

(2) Faults can be intermittent. A diagnostician sometimes has to stress a system in order to reveal faults.

(3) Diagnostic equipment can itself fail. A diagnostician has to do his best with faulty sensors.

(4) Some data about a system are inaccessible, expensive, or dangerous to retrieve. A diagnostician must decide which measurements to take.

(5) The anatomy of natural systems such as the human body is not fully understood. A diagnostician may need to combine several (somewhat inconsistent) partial models.

Monitoring

Monitoring means to continuously interpret signals and to set off alarms when intervention is required.

Example. Monitoring a patient using a mechanical breathing device after surgery [17].

Requirements. A monitoring system is a partial diagnostic system with the requirement that the recognition of alarm conditions be carried out in real time. For credibility, it must avoid false alarms.

Key problems. What constitutes an alarm condition is often context-dependent. To account for this, monitoring systems have to vary signal expectations with time and situation.

Prediction

Prediction means to forecast the course of the future from a model of the past and present.

Example. Predicting the effects of a change in economic policy. (Some planning programs have a predictive component. There is currently an opportunity to develop expert prediction programs in a variety of areas.)

Requirements. Prediction requires reasoning about time. Predictors must be able to refer to things that change over time and to events that are ordered in time. They must have adequate models of the ways that various actions change the state of the modeled environment over time.

Key problems. (1) Prediction requires the integration of incomplete information. When information is complete, prediction is not an AI problem (e.g., where will Jupiter be two years from next Thursday).

(2) Predictions should account for multiple possible futures (hypothetical reasoning), and should indicate sensitivity to variations in the input data.

(3) Predictors must be able to make use of diverse data, since indicators of the future can be found in many places.

(4) The predictive theory may need to be contingent; the likelihood of distant futures may depend on nearer but unpredictable events.

Planning

A plan is a program of actions that can be carried out to achieve goals. Planning means to create plans.

Example. Experiment planning in molecular genetics [36].

Requirements. A planner must construct a plan that achieves goals without consuming excessive resources or violating constraints. If goals conflict, a planner establishes priorities. If planning requirements or decision data are not fully known or change with time, then a planner must be flexible and opportunistic. Since planning always involves a certain amount of prediction, it has the requirements of that task as well.

Key problems. (1) Planning problems are sufficiently large and complicated that a planner does not immediately understand all of the consequences of his actions. This means that the planner must be able to act tentatively, so as to explore possible plans.

(2) If the details are overwhelming, he must be able to focus on the most important considerations.

(3) In large complex problems, there often are interactions between plans for different subgoals. A planner must attend to these relationships and cope with goal interactions.

(4) Often the planning context is only approximately known, so that a planner must operate in the face of uncertainty. This requires preparing for contingencies.

(5) If the plan is to be carried out by multiple actors, coordination (e.g. choreography) is required.

Design

Design is the making of specifications to create objects that satisfy particular requirements.

Example. Designing a digital circuit. (This is an area of increased interest and activity in expert systems.)

Requirements. Design has many of the same requirements as planning.

Key problems. (1) In large problems, a designer cannot immediately assess the consequences of design decisions. He must be able to explore design possibilities tentatively.

(2) Constraints on a design come from many sources. Usually there is no comprehensive theory that integrates constraints with design choices.

(3) In very large systems, a designer must cope with the system complexity by factoring the design into subproblems. He must also cope with interactions between the subproblems, since they are seldom independent.

(4) When a design is large, it is easy to forget the reasons for some design decisions and hard to assess the impact of a change to part of a design. This suggests that a design system should record justifications for design decisions and be able to use these justifications to explain decisions later. This is especially apparent when subsystems are designed by different designers.

(5) When designs are being modified, it is important to be able to reconsider the design possibilities. During redesign, designers need to be able to see the 'big picture' in order to escape from points in the design space that are only *locally optimal.*

(6) Many design problems require reasoning about spatial relationships. Reasoning about distance, shapes, and contours demands considerable computational resources. We do not yet have good ways to reason approximately or qualitatively about shape and spatial relationships.

Several issues appear repeatedly across this catalog of expert tasks:

Large solution spaces. In interpretation problems like the mass spectrometery example [2], some problems require millions of possible chemical structures to be considered. In planning and design tasks, the number of reasonable solutions is usually a very small fraction of a very large number of possible solutions. In each of these tasks, the size and characterization of the solution space is an important organizational parameter.

Tentative reasoning. Many diagnostic procedures profitably employ assumptions about the number of faults or about the reliability of sensors. Part way

through diagnosis, it may be discovered that these assumptions are unwarranted. This places a premimum on the ability to undo the effects of the assumptions. Similarly, in design and planning tasks it is often appropriate because of scale to employ simplifying assumptions (e.g., abstractions). In any given design, some of the assumptions will fail in a design, so there is an incentive to employ methods that facilitate the reworking of assumptions and trade-offs during iterations of the design process.

Time-varying data. Patient monitoring and diagnosis tasks are concerned with situations that evolve over time—as diseases follow their natural course or as treatments are administered.

Noisy data. Sensors often yield noisy data. This is a factor for any task involving reasoning from measurements such as interpretation, diagnostic, and monitoring tasks.

The next section considers organizational prescriptions that deal with each of these issues.

3. Knowledge Engineering Prescriptions

Feigenbaum [18] defines the activity of *knowledge engineering* as follows.

> "The knowledge engineer practices the art of bringing the principles and tools of AI research to bear on difficult applications problems requiring experts' knowledge for their solution. The technical issues of acquiring this knowledge, representing it, and using it appropriately to construct and explain lines-of-reasoning, are important problems in the design of knowledge-based systems. ... The art of constructing intelligent agents is both part of and an extension of the programming art. It is the art of building complex computer programs that represent and reason with knowledge of the world." [18, pp. 1014–1016]

This section is intended as a prescriptive guide to building expert systems. To illustrate the strengths and limitations of organizational alternatives we will cite a number of contemporary systems. In presenting these examples we adopt a level of detail that is adequate for making the ideas clear yet avoiding the particulars of the task and the programming implementation. The reader seeking a more detailed discussion of implementation is encouraged to consult a textbook on AI programming (e.g., [6]).

One of the most variable characteristics of expert systems is the way that they search for solutions. The choice of search method is affected by many characteristics of a domain, such as the size of the solution space, errors in the data, and the availability of abstractions. Inference is at the heart of a reasoning system and failure to organize it properly can result in problem-solvers that are hopelessly inefficient, naive, or unreliable. As a consequence of this, search is one of the most studied topics in artificial intelligence.

Our pedagogical style is to start with a very restricted class of problems that admits a very simple search process. We will articulate the domain restrictions under which this organization is applicable, and thereby expose its limitations. Then we will relax the requirements on the task domain and introduce ameliorating techniques as architectural prescriptions. Fig. 1 shows a chart of

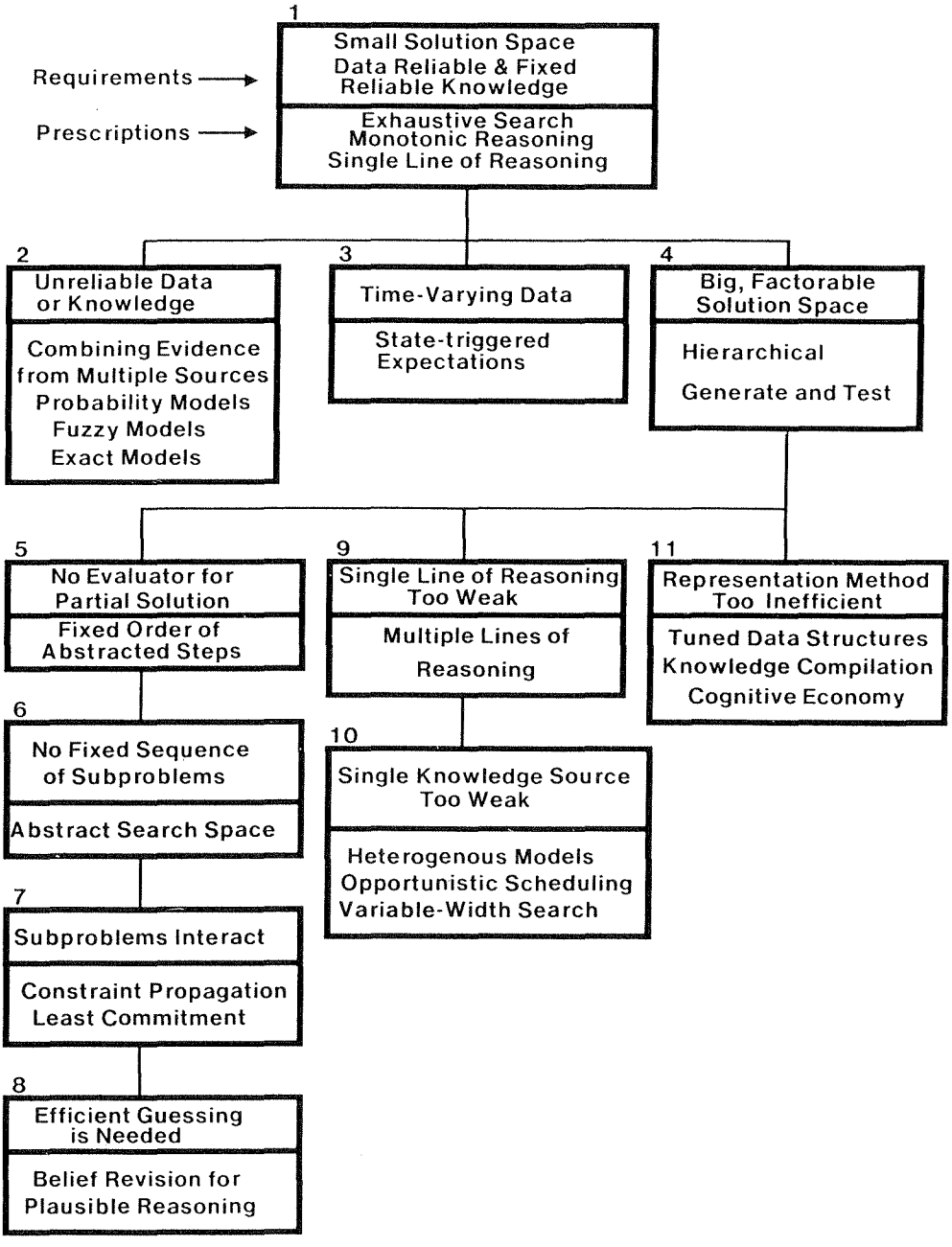


FIG. 1. Case 1 begins with a restricted class of problems that admits a very simple organization. These assumptions are relaxed one at a time in the other cases.

the cases that we will consider. Each box in the figure corresponds to one of the cases and the numbering indicates the order in which the cases are discussed. The lines connecting the boxes organize the cases into a tree structure such that a sequence of cases along a branch corresponds to increasingly elaborate considerations of a basic idea. The first three branches consider the complications of unreliable data or knowledge, time-varying data, and a large search space. Any given problem may require combining ideas from any of these topics. The problem of a large search space is then considered along three major branches. The first branch (cases 5 through 8) considers organizations for abstracting a search space. The second branch focuses on methods for incomplete search. The third branch considers only ways to make the knowledge base itself more efficient. This breakdown is mainly pedagogical. Real systems may combine these ideas.

3.1. Case 1—Small search space with reliable knowledge and data

Systems for complex tasks are generally more complicated than systems for simple tasks. In this section we will consider a very simple architecture which has been used for relatively simple applications. We begin by listing the requirements for task simplicity:

(1) The data and knowledge should be reliable.
(2) The data and knowledge should be static.
(3) The space of possible solutions should be small.

On the surface these requirements may seem quite mild. Indeed, there is a widely held belief among people who have not looked closely into problem solving that most problems satisfy these requirements. After all, for many problems the facts seem straightforward and there are not *that* many solutions to consider. Under closer examination, however, most real tasks fail to meet these requirements including the examples of expert tasks listed in Section 2.

The first requirement is that the data and knowledge be reliable. Reliable data are not noisy or errorful. There can be no extraneous signals and no missed signals (e.g., due to sampling). In real applications few sources of data meet these requirements. In addition to data reliability, the knowledge must be reliable. Reliable knowledge is applicable without concern about consistency or correctness. Systematic application of reliable knowledge should not lead to false, approximate, or tentative conclusions. The main advantage of reliability for both data and knowledge is the monotonicity of the system. In the simplest architecture, the memory is a monotonic data base to which conclusions are added by the reasoning system as they are inferred. No provisions need to be made for retraction of facts pending new information. It is enough to develop a single line of reasoning; that is, there is no need to develop multiple arguments to support potential conclusions. If more than one inference rule is applicable at a given time, the order in which they are applied is unimportant.

The second requirement is intended to avoid the problem of reasoning with time-dependent data. This means that the system need not be concerned with invalidating facts as time passes.

The requirement that the search space should be small implies that no provisions need to be made to cope with the limitations of computational resources. There need be no concern about computationally efficient data structures or for avoiding combinatorial explosions. It doesn't matter whether the search is for one solution or all possible solutions as long as the space is small. If the search is exhaustive, the maximum size of the search space depends on the time it takes to consider a single solution. A useful number to keep in mind for this maximum is ten factorial (10!). If 25 milliseconds are required to consider a solution, then 10! solutions can be considered sequentially during a full twenty-four hour day. This is often a practical upper limit for exhaustive search. The surprise is that the ceiling is so low.

An organization for solving these problems has two main parts: a memory and an inference method. The simplest organization of the memory would be a list of inferred facts (i.e., beliefs). For many problems, the beliefs can be represented in the predicate calculus[1] such as

(On Block1 Block2),
(NOT (On Block2 Table-1)).

Some systems attempt to optimize the storage format of the data. For example, in frame systems (see [3]) the indexing of facts is organized to make the most common access paths more efficient. Data which are used together are stored together in frames.

In the following sections we explore some more sophisticated organizations that will enable us to relax these restrictions on the problems.

3.2. Case 2—Unreliable data or knowledge

Experts sometimes make judgments under pressure of time. All the data may not be available; some of the data may be suspect; some of the knowledge for interpreting the data may be unreliable. These difficulties are part of the normal state of affairs in many interpretation and diagnostic tasks. The general problem of drawing inferences from uncertain or incomplete data has invited a variety of technical approaches.

One of the earliest and simplest approaches to reasoning with uncertainty was incorporated in the MYCIN expert system [7, 34] for selecting antibiotic therapy for bacteremia. One of the requirements for MYCIN was to represent judgmental reasoning such as "*A suggests B*" or "*C* and *D tend to* rule out *E*". To this end, MYCIN introduced a model of approximate implication using

[1] In our examples we use the prefix form of the notation because of its obvious similarity to list notations as in LISP.

numbers called *certainty factors* to indicate the strength of a heuristic rule. The following is an example of a rule from MYCIN's knowledge base.

> If (1) the infection is primary-bacteremia and
> (2) the site of the culture is one of the sterile sites and
> (3) the suspected portal of entry of the organism is the gastro-intestinal tract,
> then there is suggestive evidence (.7) that the identity of the organism is bacteroides.

The number '.7' in this rule indicates that the evidence is strongly indicative (0.7 out of 1) but not absolutely certain. Evidence confirming a hypothesis is collected separately from that which disconfirms it, and the 'truth' of the hypothesis at any time is the algebraic sum of the evidence. This admits the combination of evidence in favor and against the same hypothesis.

The introduction of these numbers is a departure from the *exactness* of predicate calculus. In MYCIN, things are not just true or false; reasoning is inexact and that inexactness is numerically characterized in the rules by an expert physician. Facts about the world are represented as 4-tuples corresponding to an atomic formula with a numeric truth value. For example,

(IDENTITY ORGANISM-2 KLEBSIELLA .25)

is interpreted as 'The identify of organism-2 is *Klebsiella* with certainty 0.25'. In predicate calculus, the rules of inference tell us how to combine wffs and truth values. MYCIN has its own way to combine formulas. When the premise of a rule is evaluated, each predicate returns a number between 1 and -1 (-1 means 'definitely false'). MYCIN's version of AND performs a minimization of its arguments; OR performs a maximization of its arguments. This results in a numerical value between -1 and 1 for the premise of a rule. For rules whose premise values surpass an empirical threshold of 0.2, the rule's conclusion is made with a certainty that is the product of the premise value and the certainty factor of the rule. These rules of combination can be shown to have certain properties—such as insensitivity to the order in which the rules are applied. MYCIN's certainty factors are derived from probabilities but have some distinct differences (see [34]).

A reasonable question about such approaches is whether they are unnecessarily ad hoc. A commonly voiced criticism is that MYCIN introduces its own formalism for reasoning with uncertainty when there are thoroughly-studied probabilistic approaches available. For example, Bayes' Rule could be used to calculate the probability (e.g., of a disease) in light of specified evidence, from the a priori probability of the disease and the conditional probabilities relating the observations to the diseases. The main difficulty with Bayes' Rule is the large amount of data that are required to determine the conditional probabilities needed in the formula. This amount of data is so unwieldy that the

conditional independence of observations is often assumed. It can be argued that such independence assumptions undermine the rigorous statistical model. A middle ground which replaces observations with subjective estimates of prior probabilities has been proposed by Duda, Hart, and Nilsson [14] and analyzed for its limitations by Pednault, Zucker, and Muresan [31].

Another approach to inexact reasoning that diverges from classical logic is *fuzzy logic* as discussed by Zadeh [40] and others. In fuzzy logic, a statement like 'X is a large number' is interpreted as having an imprecise denotation characterized by a *fuzzy set.* A fuzzy set is a set of values with corresponding *possibility values* as follows.

Fuzzy Proposition:
X is a large number.

Corresponding fuzzy set:
$(X \in [0,10], .1)$,
$(X \in [10,1000], .2)$,
$(\{X > 1000\}, .7)$.

The interpretation of the proposition 'X is large' is that 'X might be less than 10' with possibility 0.1, or between 10 and 1000 with possibility 0.2, and so on. The fuzzy values are intended to characterize an imprecise denotation of the proposition.

Fuzzy logic deals with the laws of inference for fuzzy sets. Its utility in reasoning about unreliable data depends on the appropriateness of interpreting *soft data* (see Zadeh [41]) as fuzzy propositions. There is little agreement among AI researchers on the utility of these *modified logics* for intelligent systems, or even on their advantages for reasoning with incomplete data.

The pseudo-probability and fuzzy approaches for reasoning with partial and unreliable data depart from the predicate calculus by introducing a notion of inexactness. Other approaches are possible. For example, the use of *exact* inference methods on unreliable data in an expert system is illustrated by the GA1 program reported by Stefik [37]. GA1 is a data interpretation system which copes with errorful data. It exploits the redundancy of experimental data in order to correct errors that it may contain.

GA1 infers DNA structures from segmentation data. GA1's task is to assemble models of complete DNA structures given data about pieces (called segments) of the structures. The segment data are produced by chemical processes which break DNA apart in predictable ways. In a typical problem, several independent breaking processes called digestions are performed and the resulting segments are measured. These digestions give independent measurements of the DNA molecule. For example, independent estimates of the molecular weight can be computed by summing the weights of the segments in any of the 'complete' digestions. (A digest is called complete if all of the molecules have been cleaved in all possible places by the enzymes.)

An example of a rule for correcting missing data is:

> If a segment appears in a complete digestion for an enzyme that
> fails to appear in the incomplete digestion for that enzyme,
> it may be added to the list of segments for the incomplete digestion.

This rule is based on the observation that segments are easier to overlook in incomplete digestions than in complete digestions. This rule places more confidence in data from complete digestions than from incomplete ones. Other data correction rules incorporate more elaborate reasoning by modeling predictable instrument error such as failure to resolve measurements which are too close together. Such rules enable GA1 to look to the data for evidence of instrument failure.

In summary, several methods for reasoning with unreliable data and knowledge have been proposed. The probability-related and possibility methods use modified logics to handle approximations. They use numerical measures for combining evidence. In contrast, data correction rules can reason with partial information without compromising the exactness of predicate calculus. All of these methods depend on the formalization of extra *meta-knowledge* in order to correct the data, take back assumptions, or combine evidence. The availability of this meta-knowledge is a critical factor in the viability of these approaches to particular applications.

A special method for contending with both fallible knowledge and limited computational resources will be considered in Section 3.10.

3.3. Case 3—Time-varying data

Some expert tasks involve reasoning about situations that change over time. One of the earliest approaches in AI to take this into account was the situational calculus introduced by McCarthy and Hayes for representing sequences of actions and their effects [25]. The central idea is to include 'situations' along with the other objects modeled in the domain. For example, the formula

> (On Block-1 Table-2 Situation-2)

could represent the fact that in Situation-2, Block-1 is on Table-2. A key feature of this formulation is that situations are discrete. This discreteness reflects the intended use of this calculus in robot planning problems. A robot starts in an initial situation and performs a sequence of actions. After each action, the state of the robot's world is modeled by another situation. In this representation, a situation variable can take situations as values. In some implementations, the actual situation variable is usually left implicit by indexing the formulas according to situations.

Actions in the situational calculus are represented by functions whose

domains and ranges are situations. For each action, a set of frame axioms characterizes the set of assertions (i.e., 'the frame') that remain fixed while an action takes place within it. In a robot planning task, an example of an action would be the *Move* action for moving an object to a new location. A frame axiom for *Move* would be that all objects not explicitly moved are left in their original location.

While many issues can be raised about this approach to representing changing situations, many AI systems have used it for a variety of tasks with only minor variations. Sometimes the changes of situation are signalled by time-varying data, rather than by the autonomous actions of a robot. An example of this is shown by the VM system reported by Fagan [16, 17]. VM (for Ventilator Manager) is a program that interprets the clinical significance of patient data from a physiological monitoring system. VM monitors the post-surgical progress of a patient requiring mechanical breathing assistance. A device called a mechanical ventilator provides breathing assistance for seriously ill patients. The type and settings of the ventilator are adjusted to match the patient's needs. As the patient's status improves, various adjustments and changes are made, such as replacing the mechanical ventilator with a 'T-piece' to supply oxygen to the patient.

In VM's application, a patient's situation is affected by the progression of disease and the response to therapeutic interventions. For such applications, the model of clinical reasoning must account for information received from tests and observations over time.

VM illustrates knowledge suitable for coping with time-varying data. This knowledge in VM is organized in terms of several kinds of rules: transition rules, initialization rules, status rules, and therapy rules. Periodically VM receives a new set of instrument measurements and then it reruns all of its rules. Transition rules are used to detect when the patient's state has changed, e.g., when the patient starts to breathe on the T-piece. The following is an example of a transition rule.

> If (1) the current context is 'Assist' and
> (2) respiration rate has been stable for 20 minutes and
> (3) I/E ratio has been stable for 20 minutes,
> then the patient is on 'CMV' (controlled mandatory ventilation).

This rule governs the transition between an 'Assist' context and a 'CMV' context. When the premise of a transition rule is satisfied in VM, a new 'context' is entered. These contexts correspond to specific states or situations. When a context is changed, VM uses initialization rules to update its information for the new context (e.g., expectations and unacceptable limits for the measurements). These rules refer to the recent history of the patient to establish new expectations and information for the new context. Part of one

such rule follows:

> If (1) the patient transitioned from 'Assist' to 'T-piece' or
> (2) the patient transitioned from 'CMV' to 'T-piece'
> then expect the following:

			[--Acceptable--]			
	Very low	Low	[-Ideal-] Min	Max	High	Very high
SYS		110			150	
DIA		60			95	
MAP	60	75			110	120
Pulse rate		60			120	
ECO2	22	28	30	40	45	50

VM's reasoning about time is limited to adjacent time intervals. It is concerned only with the previous state and the next state. Its mechanisms for dealing with this are (1) state-triggered expectations and (2) rules for dynamic belief revision. The transition rules in VM govern changes of context. Data arrives periodically, but context is changed only when it is adequately supported by the evidence. The initialization rules are essentially like the frame axioms—establishing what changes and what stays the same in the new context. Once a context is set, the expectations are used to govern VM's behavior until the context is changed again.

Programs which need to reason about more distant events require more elaborate representations of events and time. For example, planning and prediction tasks require reasoning about possible futures. For these applications, the situational calculus must be extended to allow for multiple possible futures with undetermined operations, unordered sets of possible future events, and the possible actions of uncontrolled multiple actors. While AI systems capable of such sorts of reasoning seem within reach, their construction is still a research enterprise.

3.4. Case 4—Large but factorable solution space

In the restricted class of problems that we started with in Section 3.1, it was stipulated that:

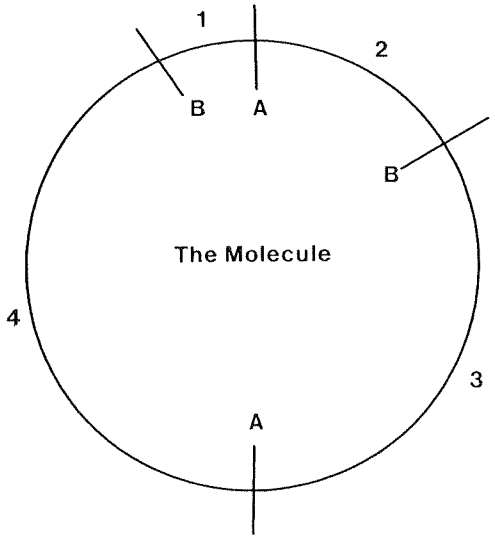
(1) The data and knowledge must be reliable.
(2) The data must be static.
(3) The search space must be small.

We have already discussed some techniques for relaxing the first two requirements. In this section we will begin the consideration of techniques for coping with very large search spaces.

In many data analysis tasks, it is not enough to find just one interpretation of the data. It is often desirable to find every interpretation that is consistent with the data. This conservative attitude is standard in high risk applications such as the analysis of poisonous substances or medical diagnosis. A systematic approach would be to consider all possible cases, and to rule out those that are inconsistent with the data. This approach is called *reasoning by elimination* and has been familiar to philosophers for years, but it has often been regarded as impractical. The difficulty is that there is often no practical way to consider all of the possible solutions.

The DENDRAL program [2] is probably the best known AI program that reasons by elimination (using generate and test). The key to making it work is to incorporate early pruning into the generate and test cycle. This section illustrates some of the characteristics of problems on which this approach will work, and gives examples of the kinds of knowledge that are needed. Since the problem area for the DENDRAL program is rather complicated for tutorial purposes, we will consider a simpler expert system, GA1 [37], that was already mentioned in Section 3.2.

Like DENDRAL, GA1 is a data interpretation program that infers a complete molecular structure from measurements of molecular pieces. Fig. 2 shows a



Complete A Digest: 5 5
Complete B Digest: 3 7
Complete A&B Digest: 1 2 3 4

FIG. 2. Enzyme *A* cleaves the circular molecule at the points labeled *A*. Enzyme *B* cleaves it at the points labeled *B*. The table lists the fragments that would be observed under ideal digestion experiments.

simple example of the kind of data that GA1 would have about a molecule. The top part of the figure shows that a molecule is made up of segments of measureable length. The lines labeled *A* and *B* that cut across the circle indicate the sites where the molecule is cleaved by enzymes (named *A* and *B*, respectively). All of the molecules that GA1 is concerned with are linear or circular. This means that all of the molecular segments are linear pieces that can be arranged end to end. When a sample of molecules is completely digested by an enzyme, pieces are released whose sizes can be measured. The goal is to infer the structure of the original molecule from the digest data. Sometimes more than one molecular structure is consistent with the available data.

A primary task in problems like this is to create a workable generator of all of the possible solutions (i.e., all of the possible molecules). In GA1, the first step is to apply data correction rules as shown in Section 3.2. Then GA1 determines an initial set of generator constraints—a set of segments and enzyme sites for building candidate molecules. The rules for deriving the list will not be elaborated here, but they make conservative use of molecular weight estimates and redundant data from several digests. The generation process then begins by combining these segments and sites, and testing whether the combinations fit the evidence. For example, the following lists (among others) correspond to the complete molecule in Fig. 2.

(1 *A* 2 *B* 3 *A* 4 *B*),
(2 *B* 3 *A* 4 *B* 1 *A*),
(1 *B* 4 *A* 3 *B* 2 *A*).

These equivalent representations can be generated by starting with any of the four segments in the picture of the molecule in Fig. 2, and reading off the sites and segments around the circle either clockwise or counterclockwise. This provides us with eight equivalent representations for the same molecule. A generator is said to be nonredundant if it produces exactly one of the equivalent representations of a solution (the *canonical form*) during the generation process. GA1 does this by incorporating rules for pruning non-canonical structures during the generation process. An example of such a rule follows:

> If circular structures are being generated, only the smallest segment in the list of initial segments should be used for the first segment.

The key to effective use of generate and test is to prune classes of inconsistent candidates as early as possible. For example, consider the following structures from the generation process for the sample problem:

(1 *B* 2 *B*).

GA1 treats this as a description of all the molecules that match the pattern

(1 *B* 2 *B* ⟨Segment⟩ ⟨Site⟩ ⟨Segment⟩ ⟨Site⟩)

where any of the remaining segments and sites may be filled in the template. It is easy to see that no molecule matching this template is consistent with the data in Fig. 2—because all such molecules would yield a segment of length 2 in the complete digest for *B*. Other pruning considerations are more global. When such pruning rules exist, the solution space is said to be factorable.

GA1 has been run on problems where the number of possible candidates is several billion. However, the pruning rules are so effective at eliminating classes of solutions that most problems require only a few seconds of computer time.

After the generator is finished, usually twenty or thirty candidates make it through all of the pruning rules. Only one or two of these will be consistent with all of the data. In principle, it is possible to add more pruning rules to GA1 so that only the consistent solutions remain. However, the rules needed to do this become increasingly complex and specialized. It becomes difficult to prove the correctness of such rules (so that solutions will not be missed) and to ensure that the rules are faithfully represented in the program. There is also a point of diminishing returns when each new specialized rule covers a smaller number of cases. In GA1 this problem is addressed by applying a digestion process model to the final candidates and comparing its predictions with the observed data. A simple scoring function then penalizes candidates that predict extra or missing segments. Because the number of disagreements between the idealized digests of two different molecules diverges rapidly for small molecular differences, it was not necessary to tune the scoring function to recognize wrong solutions.

In summary, generate-and-test is an appropriate method to consider when it is important to find all of the solutions to a problem. For the method to be workable, the generator must partition the solution space in ways that allow for early pruning. These criteria are often associated with data interpretation and diagnostic problems.

3.5. Case 5—No evaluator for partial solutions

There are many problems involving large search spaces for which generate and test is a method of last resort. The most common difficulty is that no generator of solutions can be found for which early pruning is viable. Design and planning problems are of this nature. One usually cannot tell from a fragment of a plan or design whether that fragment is part of a complete solution; there is no reliable evaluator of partial solutions expressed as solution fragments.

In this section we will consider the first of several approaches to problem solving without early pruning. These approaches have in common the idea of abstracting the search space but differ in their assumptions about the nature of that space. Abstraction emphasizes the important considerations of a problem and enables its partitioning into subproblems. In the simplest case there is a

fixed partitioning in the abstract space which is appropriate for all of the problems in an application.

This case is illustrated by the R1 program reported by McDermott [27]. R1's area of expertise is the configuring of Digital Equipment Corporation's VAX computer systems. Its input is a customer's order and its output is a set of diagrams displaying the spatial relationships among the components on the order. This task includes a substantial element of design. In order to determine whether a customer's order is satisfactory, R1 must determine a spatial configuration for the components and add any necessary components that are missing.

The configuration task can be viewed as a hierarchy of subtasks with strong temporal interdependencies. R1 partitions the configuration task into six ordered subtasks as follows.

(1) Determine whether there is anything grossly wrong with the customer's purchase order (e.g., mismatched items, major prerequisites missing).
(2) Put the appropriate components in the cpu and cpu expansion cabinets.
(3) Put boxes in the unibus expansion cabinet and put the appropriate components in those boxes.
(4) Put panels in the unibus expansion cabinets.
(5) Lay out the system on the floor.
(6) Do the cabling.

The actions within each subtask are highly variable; they depend on the particular combination of components in an order and on the way these components have been configured so far. Associated with each subtask in R1 is a set of rules for carrying out the subtask. An example of a rule for the third subtask follows:

If the most current active context is assigning a power supply
  and a unibus adaptor has been put in a cabinet
  and the position it occupies in the cabinet (its nexus) is known
  and there is space available in the cabinet for a power supply for that nexus
  and there is an available power supply
  and there is no H7101 regulator available,
then add an H7101 regulator to the order.

R1 has about 800 rules about configuring VAX systems. Most of the rules are like the example above. They define situations in which some partial configuration should be extended in particular ways. These rules enable R1 to combine partial configurations to form an acceptable configuration. They indicate what components can (or must) be combined and what constraints

must be satisfied in order for the combinations to be acceptable. They make use of a data base describing properties of about 400 VAX components. Other rules describe the temporal relationships between subtasks by determining their ordering. (These rules are analogous to the transition rules described for VM in Section 3.3 except that the rules monitor the state of R1's problem solving instead of data from external sensors.)

The approach that R1 uses is called *Match*. It is one of Newell's *weak methods* for search [28]. Match enables R1 to explore the space of possible configurations with the basic operations of creating the extending partial configurations. Match explores this space by starting in an initial state, going through intermediate states, and stopping in a final state without any backtracking. Each state in the space is a partially instantiated configuration. R1 proceeds through its six major tasks in the same order for each problem; it never varies the order and it never backs up in any problem. The benefit of the abstraction space is that R1 needs to do very little search.

The conditions that make Match viable are both its source of power and its weakness. The key requirement is that there can be no backtracking. This means that at any intermediate state, R1 must be able to determine whether the state is on a solution path. This requires that there must exist a partial ordering on decisions for the task such that the consequences of applying an operator bear only on 'later' parts of the solution.

It is interesting that Match is in fact insufficient for the complete task in R1. The subtask of placing modules on the unibus is formulated essentially as a bin-packing problem—namely how to find an optimal sequence that fits within spatial and power constraints. No way of solving this problem without search is known. Consequently R1 uses a different method for this part of the problem.

In summary, the use of abstractions should be considered for applications where there is a large search space but no method for early pruning. R1 is an example of a system which uses a fixed abstract solution. Within this framework, it uses the Match weak method to search for a solution. Whether Match is practical for an application depends on how difficult it is to order the intermediate states.

3.6. Case 6—No fixed partitioning of subproblems

When every example problem in an application can be usefully partitioned into the same subproblems, then the organization described in the previous section should be considered. In applications with more variety to the problems, no fixed set of subproblems can provide a useful abstraction. For example, planning domains such as errand running (see [22]) require plans rich with structure. To be useful, abstractions must embody the variable structure of the plans.

In this section we will consider an approach called *top-down refinement* that tailors an abstraction to fit each problem. The following aspects of the approach are important.

(1) Abstractions for each problem are composed from terms (selected from a space of terms) to fit the structure of the problem.

(2) During the problem-solving process, these concepts represent partial solutions that are combined and evaluated.

(3) The concepts are assigned fixed and predetermined abstraction levels.

(4) The problem solution proceeds *top down*, that is, from the most abstract to the most specific.

(5) Solutions to the problem are completed at one level before moving down to the next more specific level.

(6) Within each level, subproblems are solved in a problem-independent order. (This creates a *partial* ordering on the intermediate abstract states.)

The best known example of a program using this approach is the ABSTRIPS program reported by Sacerdoti [33]. ABSTRIPS was an early robot planning program. It made plans for a robot to move objects (e.g., boxes) between rooms. A design goal for ABSTRIPS was to provide abstractions sufficiently different from the detailed 'ground' space to achieve a significant improvement in problem-solving efficiency, but sufficiently similar so that the mapping down from abstractions would not be time-consuming. This led to an interesting and simple approach for representing abstractions.

Abstractions in ABSTRIPS are plans. They differ from ground level plans only in the level of detail used to specify the preconditions of operators. This level of detail is indicated by associating a number (termed a *criticality value*) with all of the literals used in preconditions. For example, Sacerdoti suggested the following criticality assignments in a robot planning domain:

Type and Color	4
InRoom	3
PluggedIn and Unplugged	2
NextTo	1

In this example, the predicates for *Type* and *Color* of objects are given high criticalities, since the robot has no operator for changing them. These predicates together with the set of robot actions are combined to form plans for solving particular problems; the space of possible plans is the set of all of the plans that can be built up from these pieces. The most abstract plans are the ones that include only the higher criticality concepts.

Planning in ABSTRIPS starts by setting criticality to a maximum. In it, planning within each level proceeds backwards from goals. Preconditions whose criticality is below the current level are invisible to the planner, since it is presumed that they will be accounted for during a later pass. After a plan is completed at one level, the criticality level is decremented and planning is started on the

lower level. The previous abstract version of the plan is used to guide the creation of the next level. For example, an early version of a plan may determine the route that the robot takes through the rooms. In more detailed versions, steps for opening and closing doors are included. In this way, the abstract plans converge to the specific plan. The sequence of abstract plans is created differently for each problem.

ABSTRIPS was a great advance over its predecessor STRIPS, which lacked the hierarchical planning ability. Generally, when hierarchical and non-hierarchical approaches have been systematically compared, the former have dominated. ABSTRIPS was substantially more efficient than STRIPS, and the effect increased dramatically as longer plans were tried. Since then, many other hierarchical planning programs have been created. In most of the later programs, the abstraction concepts have simply been arranged in a hierarchy, without actually assigning them criticality numbers.

In summary, the interesting feature of top-down refinement is the flexibility of the abstractions. Abstraction states are individually constructed to fit each problem in the domain. In contrast to Match, top-down refinement places only a *partial* ordering on the intermediate states of the problem-solver. Still, there are some important conditions about problem solving in the domain of application that are inherent in the method. The basic assumption is that a small fixed amount of problem-solving knowledge about criticality levels and top-down generation is sufficient. Furthermore, it must be possible to assign a partial criticality ordering to the domain concepts. What is important for one problem must be important for all of the problems. The next section suggests some ways to relax these requirements.

3.7. Case 7—Interacting subproblems

One basic difficulty with top-down refinement is the lack of feedback from the problem-solving process. It is presumed that the same kinds of decisions should be made at the same point (i.e., criticality level) for each problem in the domain. In this section, we will explore an approach that is based on a different principle for guiding the reasoning process called the *least-commitment* principle. The basic idea is that decisions should not be made arbitrarily or prematurely. They should be postponed until there is enough information.

Reasoning based on the least-commitment principle requires the following abilities.

(1) The ability to know when there is enough information to make a decision.

(2) The ability to suspend problem-solving activity on a subproblem when the information is not available.

(3) The ability to move between subproblems, restarting work as information becomes available.

(4) The ability to combine information from different subproblems.

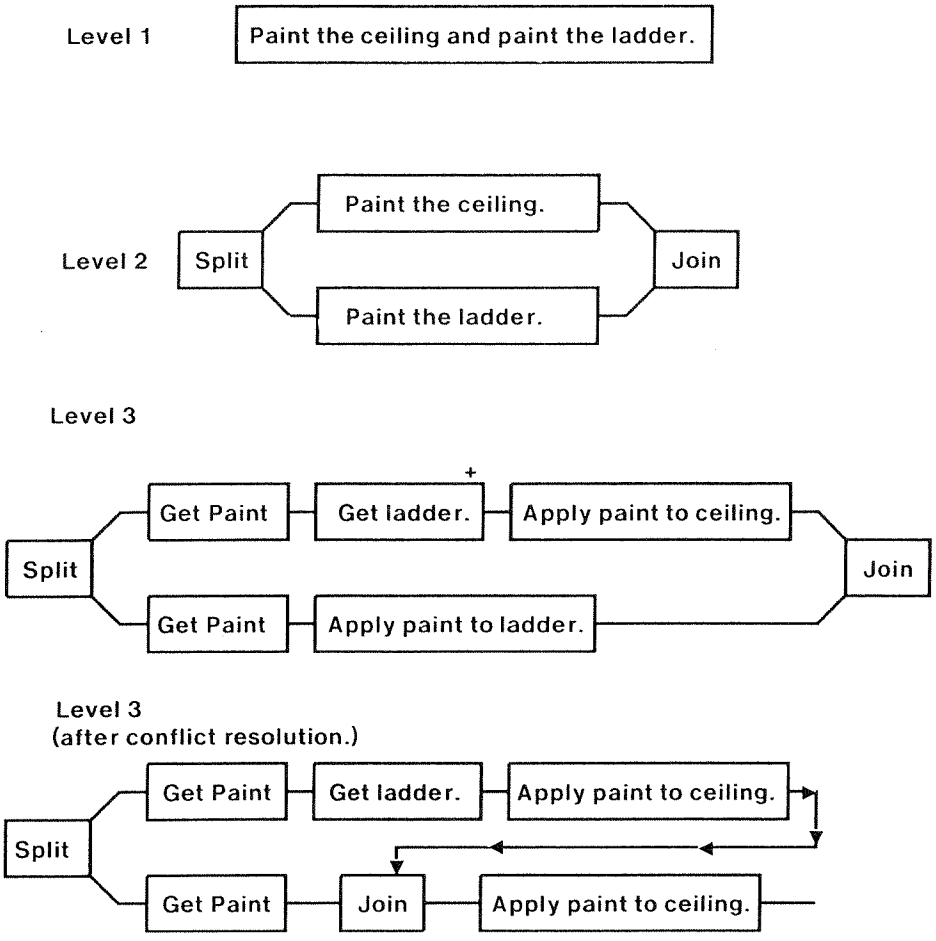


FIG. 3. Example of planning in NOAH. NOAH analyzes the interactions between the steps in order to assign them an ordering in time. In this example, the 'painting the ladder' is seen to be in conflict with using it. To complete both goals, NOAH decides to paint the ceiling first. Later processing will factor out common subplans like 'get paint'.

Fig. 3 shows an example of this style of reasoning from the NOAH system reported by Sacerdoti [32]. NOAH was a robot planning system that used a least-commitment approach to assign a time-ordering to operators in a plan. Earlier planning programs inserted operators into a plan as they worked backwards from goals. In contrast, NOAH assigned the operators only a *partial* ordering and added specifications for a complete ordering of the operators only as required.

In Fig. 3, NOAH starts with two subgoals: paint the ceiling and paint the ladder. Plans for the two subgoals are expanded in parallel and a conflict is found. If the ladder is painted first, it will be wet and we won't be able to paint the ceiling. In other words, the step to paint the ladder violates a precondition (that the ladder be usable) for the step to paint the ceiling. This interference

between the subgoals provides NOAH with enough information to order the tasks. If it had arbitrarily ordered the steps and painted the ladder first, it would have had to plan around the wet ladder, perhaps waiting for it to dry. The resulting plan would not have been optimal.

Another example of this idea was given in MOLGEN reported by Stefik [36]. MOLGEN is an expert system that used this style of reasoning for designing molecular genetics experiments. MOLGEN's organization involved the following features:

(1) Interactions between nearly independent subproblems are represented as constraints.

(2) Interactions between subproblems are discovered via constraint propagation.

(3) MOLGEN uses explicit *problem-solving operators* (as opposed to domain-specific operations) to reason with constraints.

(4) MOLGEN alternates between least-commitment and heuristic strategies in problem-solving.

In the least commitment strategy, MOLGEN makes a choice only when its available constraints sufficiently narrow its alternatives. Its problem-solving operators are capable of being suspended so that a decision could be post-poned. Constraint propagation is the mechanism for moving information between subproblems. It enables MOLGEN to exploit the synergy between decisions in different subproblems. In contrast with ABSTRIPS, strict backward expansion of plans within levels, MOLGEN expands plans opportunistically in response to the propagation of constraints.

The fourth feature illustrates an interesting limitation of the least commit-ment principle. Every problem-solver has only partial knowledge about solving problems in a domain. With only the least-commitment principle, the solution process must come to a halt whenever there are choices to be made, but no compelling reason for deciding any of them. We call this situation a *least-commitment deadlock*. When MOLGEN recognizes this situation, it switches to its heuristic approach and makes a guess. In many cases, a guess will be workable, and the solution process can continue to completion. In other cases, a bad guess can lead to conflicts. The number of conflicts caused by (inaccurate) guessing is a measure of the incompleteness of the problem-solving knowledge. Conflicts can also arise from the least-commitment process in cases where the goals are fundamentally unattainable.

In summary, the least-commitment principle coordinates decision-making with the availability of information and moves the focus of problem-solving activity among the available subproblems. The least-commitment principle is of no help when there are many options and no compelling reasons for choices. In these cases, some form of plausible reasoning is necessary. In general, this approach uses more information to control the problem-solving process than the top-down refinement approach.

3.8. Case 8—Guessing is needed

Guessing or *plausible reasoning* is an inherent part of heuristic search. For example, the generator in a generate and test system guesses about partial solutions so that they can be tested. A more subtle example is a problem-solver based on top-down refinement, which implicitly assumes that it will be able to refine its higher abstractions to specific solutions. Some examples follow of generic situations in reasoning where guessing is important:

(1) Many problem-solvers need to cope with incomplete knowledge and may be unable to determine the best choice at some stage in its problem solving. In such cases, a problem-solver is unable to finish without making a guess. Examples of this are assumptions introduced as a first step in hypothetical reasoning and assumptions introduced to break a least-commitment deadlock as discussed in the previous section.

(2) A search space may be quite dense in solutions. If solutions are plentiful and equally desirable and there is no need to get them all, guessing is efficient.

(3) Sometimes there is an effective way to converge to solutions by systematically improving approximations. (Top-down refinement is an example of this.) In cases where convergence is rapid, it may be appropriate to guess even when solutions are rare.

The difficulty with guessing is in identifying wrong guesses and recovering from them efficiently. This section considers how plausible reasoning can benefit from particular architectural features.

One of the best known systems with architectural provisions for guessing is the EL system for circuit analysis reported by Stallman and Sussman [35]. EL analyzes analog electrical circuits. It has two main methods that are described below—forward reasoning and the method of 'assumed states'. The assumed states provide the examples of guessing.

Forward reasoning with electrical laws is used to compute electrical parameters (e.g., voltage or current) at one node of a circuit from parameters at other nodes. EL uses only a few laws, such as Ohm's Law which defines a linear relationship between voltage and current for a resistor, and Kirchhoff's current law which states that the current flowing out of a node equals the current flowing into it. Much of EL's power derives from two things: (1) the exhaustive application of these laws and (2) the ability to reason with these laws symbolically as shown in Fig. 4.

This figure illustrates a circuit in which resistors are connected in a ladder arrangement. The analysis task is to determine the voltages and resistances at all of the nodes of the circuit. The interesting aspect of this is that symbolic reasoning about the circuit is much simpler than writing and solving equations for the series and parallel resistor network. Analysis begins with the introduction of a variable e to represent the unknown node voltage at the end of the ladder. This yields a current $e/5$ through resistor R_6. Then by Kirchhoff's Law we

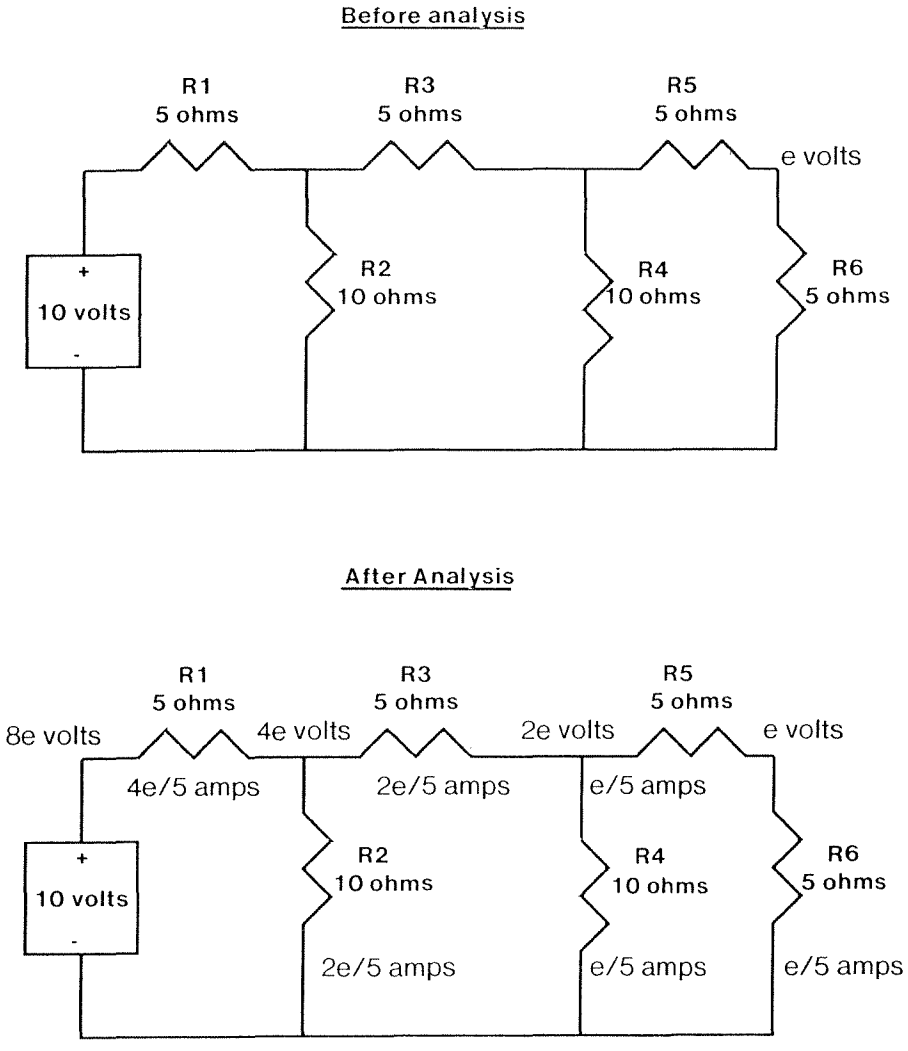


FIG. 4. Symbolic propagation of electrical parameters. Analysis begins by assigning the symbol e to the unknown voltage at the upper right corner of the ladder. Other values are derived by stepwise application of Ohm's and Kirchhoff's laws.

have the same current through R_5 which gives us a voltage $2e$ on the left of the resistor. This voltage across R_4 allows us to compute the current through it using Ohm's Law. The application of electrical laws in terms of symbolic unknowns continues until all of the voltages are defined in terms of e. At that time we have

$$8e = 10 \text{ volts}, \qquad e = 5/4 \text{ volts}.$$

Sometimes circuit analysis requires the introduction of more than one variable to represent unknown circuit parameters. In general, the analysis

involves two main processes: 1-step deductions and coincidence. The 1-step deductions are direct applications (sometimes symbolic) of the electrical laws. A coincidence occurs when a 1-step deduction is made which assigns a value to a circuit parameter that already has a value (symbolic or numeric). At the time of the coincidence, it is often possible to solve the resulting equation for one variable in terms of the others. This allows EL to eliminate unknowns.

The propagation method can be extended to any devices where the electrical laws are invertible and where the algebra required for the symbolic reasoning is tractable. Unfortunately, many simple and important electrical devices, such as inverters and transistors, are too complicated for this approach. For example, a diode is approximately represented by exponential equations. Electrical engineering has an approach for these devices called the *method of assumed states*. This is where guessing enters into EL's problem solving.

The method of assumed states uses a piecewise linear approximation for complicated devices. The method requires making an assumption about which linear region a device is operating in. EL has two possible states for diodes (*on* or *off*) and three states for transistors (*active*, *cutoff*, and *saturated*). Once a state is assumed, EL can use tractable linear expressions for the propagation analysis as before.

After making an assumption, EL must check whether the assumed states are consistent with the voltages and currents predicted for the devices. Incorrect assumptions are detected by means of a contradiction, which is the event in which chosen assumptions are seen to be inconsistent. When this happens, then some of the assumptions need to be changed. Intelligent processing of contradictions involves determining which assumptions to revise. Implementing this idea in the problem-solver led to the following important architectural features:

(1) queue-based control;

(2) dependency-directed backtracking.

The operators in EL that perform the propagation analysis are called demons. They are placed in queues and run sequentially by a scheduler. When demons run, they make assertions in the data base and then return to the scheduler. This data base activity causes other demons whose triggers match the assertions to be added to the queue.

EL has three queues for DC analysis with different priorities. The lowest priority queue is used for device-state assumptions. These demons are given a low priority so that the immediate consequences of an assumption will be explored before more assumptions are made. The middle queue is used for most of the electrical laws. The high priority queue is used for demons that detect contradictions. These demons are given a high priority so that invalid assumptions will be detected before too much computational work is done. These demons trigger the dependency-directed backtracking.

EL keeps dependency records of all of its deductions and assumptions. In EL,

an assertion is believed (or *in*) if it has well-founded support from atomic assumptions. An assertion without such support is said to be *out*. If an *out* assertion returns to favor, it is said to be *unouted*. Fig. 5 shows an example of this process in a data base. $A1$, $B1$, and $C1$ are atomic data that are currently in. Suppose that $A1$ and $A2$ are mutually exclusive device-state assumptions. The top of Fig. 5 shows which facts are in when $A1$ is in. Arrows are used to indicate support. (Assertions following from $A2$ are shown in dotted lines to show that they are in the data base, but that they are out.) The bottom figure shows what happens if $A1$ is outed and $A2$ is unouted.

An important aspect of EL's problem-solving is its ability to recover from tentative assumptions. The details of the implementation and knowledge will not be covered here, since there are many ways to approach this problem. The main points are:

(1) In the event of a contradiction, EL needs to decide what to withdraw. It is

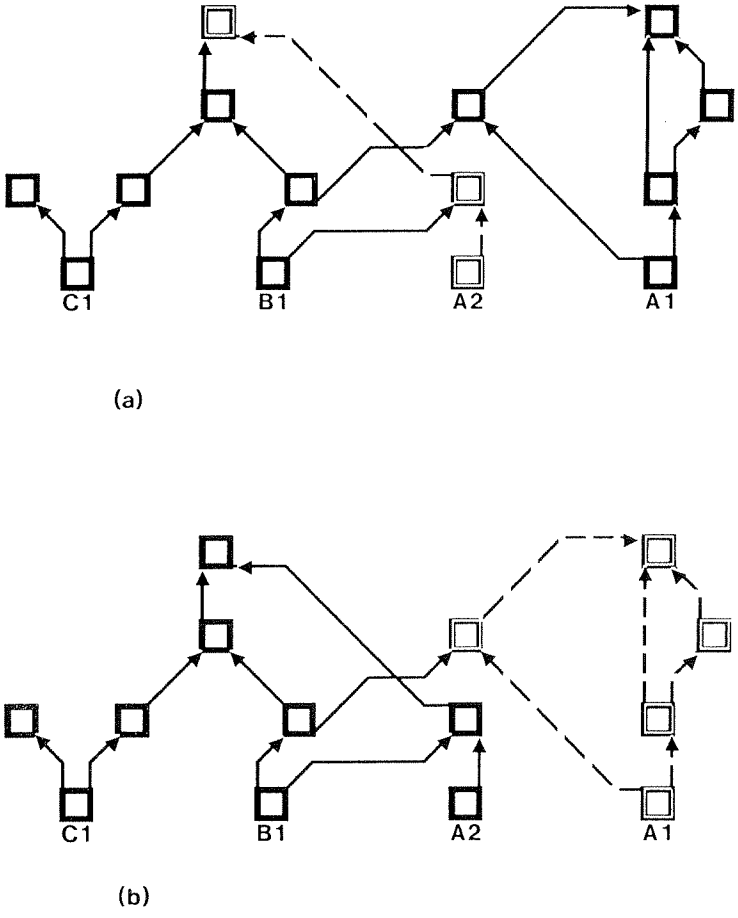


FIG. 5. Example of belief revision in EL. The dark boxes are in and the lighter ones are out. In (a), $A1$ is in and so all of its consequences are also in. In (b), $A1$ is out but $A2$ is in.

not effective to simply withdraw all of the assumptions that are antecedants of the contradictory assertion. EL must decide which of the assumptions are most unlikely to change and this requires domain-specific knowledge.

(2) EL must redo some of the propagation analysis. Sometimes it is possible to salvage some of the symbolic manipulation (e.g., variable elimination) that has been done. EL has special demons that decide carefully what to forget.

(3) Contradictions are remembered so that choice combinations that are found to be inconsistent are not tried again.

These ideas were the intellectual precursors to work on belief revision systems.[2] Belief revision can be used for reasoning with assumptions or defaults. For a problem solver to revise its beliefs in response to new knowledge, it must reason about dependencies among its current set of beliefs. New beliefs can be the consequences of new information received or derived. A critical issue in this style of reasoning is well-founded support and there are some pitfalls for the unwary involving cyclical support structures. An important question is "what mechanisms should be used to resolve ambiguities when there are several possible revisions?" It is clear this choice needs to be controlled, but the details for making the decision remain to be worked out. In the examples above, we used knowledge about justifications to reason about choices. Doyle [13] has proposed a style of dialectical argumentation where the primary step is to argue about the kinds of support for beliefs. In such system the complexity of knowledge about belief revision would itself require a substantial knowledge base. Every approach depends critically on the kinds of dependency records that are created and saved. This work is at the frontier of current AI research.

In summary, EL is an example of a program with organizational provisions for plausible reasoning. It uses symbolic forward reasoning for analyzing circuits. To analyze complicated devices EL has to assume linear operating regions. It uses dependency-directed backtracking so that it can recover from incorrect assumptions. It uses a priority-oriented queue to schedule tasks so that contradictions will be found quickly and so that the immediate consequences of assumptions will be considered before further assumptions are made.

3.9. Case 9—Single line of reasoning is too weak

When we explain to someone how we solved a problem, we often invoke 20-20 hindsight and leave out the mistakes that we made along the way. Our explanation makes it appear that we followed a very direct and reasonable route from beginning to end. For developing intuitions about problem-solving behavior, this gives a misleading impression that problem-solving is the pursuit

[2] A bibliography of recent papers on these ideas was published by Doyle and London [12]. Basic algorithms for revising beliefs have been reported by Doyle [11] and McAllester [26].

of a single line of reasoning. Actually, there are important and somewhat subtle reasons for being able to use multiple lines of reasoning in problem solving and several of the systems described above gain power from this ability. These systems use multiple lines of reasoning for two major purposes as explained below:

(1) To broaden the coverage of an incomplete search, or

(2) to combine the strengths of separate models.

The HEARSAY-II system [15] provides the best example of the first purpose. (It is described in the next section.) In coping with the conflicting demands of searching a large space with limited computational resources, HEARSAY-II performs a heuristic and incomplete search. In general, programs that have fallible evaluators can decrease the chances of discarding a good solution from weak evidence by carrying a limited number of solutions in parallel.

A good example of combining the strengths of multiple models is given by the SYN program reported by Sussman, Steele, and de Kleer [10, 38]. SYN is a program for circuit synthesis, that is, for determining values for components in electrical circuits. The EL program described in the previous section determined circuit parameters such as voltage given fully specified components in a circuit. SYN determines values for the components (e.g., the resistance of resistors) given the form of the circuit and some constraints on its behavior.

SYN uses many of the propagation analysis ideas developed for EL. The interesting new organizational idea in SYN is the idea of *slices* or multiple views of a circuit. This corresponds to the idea of equivalent circuits in electrical engineering practice. A simple example of a slice is the idea that a voltage divider made from two resistors in series can be *viewed* as a single resistor; one slice of the circuit describes it as two resistors and another slice describes it as one. To analyze the voltage divider, SYN uses the second slice to compute the current through the divider. Then by reverting back to the first slice, SYN can compute the voltage at the midpoint. In general, the idea is to switch to equivalent representations of circuits to overcome blockages in the propagation of constraints. The power of slices is that they provide redundant paths for information to travel in propagation analysis.

By exploiting electrically equivalent forms of circuits involving resistors, capacitors, and inductances, SYN is capable of analyzing rather complex circuits without extensive algebraic manipulation. The idea of slices is not limited to electrical circuits. For example, algebraic transformations of equations can be viewed as means for shifting perspectives. Sussman and Steele also give an example of understanding a mechanical watch by using structural and functional decompositions.

In summary, slices are used to combine the strengths of different models. When they are combined with forward reasoning, they provide redundant paths for information to propagate. A problem-solver based on this idea must know how to create and combine multiple views.

3.10. Case 10—Single source of knowledge is too weak

An important adjunct to the use of multiple reasons in problem solving is the use of multiple sources of knowledge. In this section we will consider the HEARSAY-II system, which coordinates diverse sources of knowledge using an opportunistic scheduler. HEARSAY-II is a system for speech understanding reported by Erman et al. [15]. It recognizes spoken requests for information from a data base. Production of speech involves a series of transformations starting with the speaker's intentions, through choice of semantic and syntactic structures, and ending with sound generation. To understand speech HEARSAY-II must reverse this process.

In HEARSAY-II the knowledge for understanding speech is organized as a set of interacting modules (called knowledge sources or KSs) as shown by the arrows in Fig. 6. The KSs cooperate in searching a multi-level space of partial

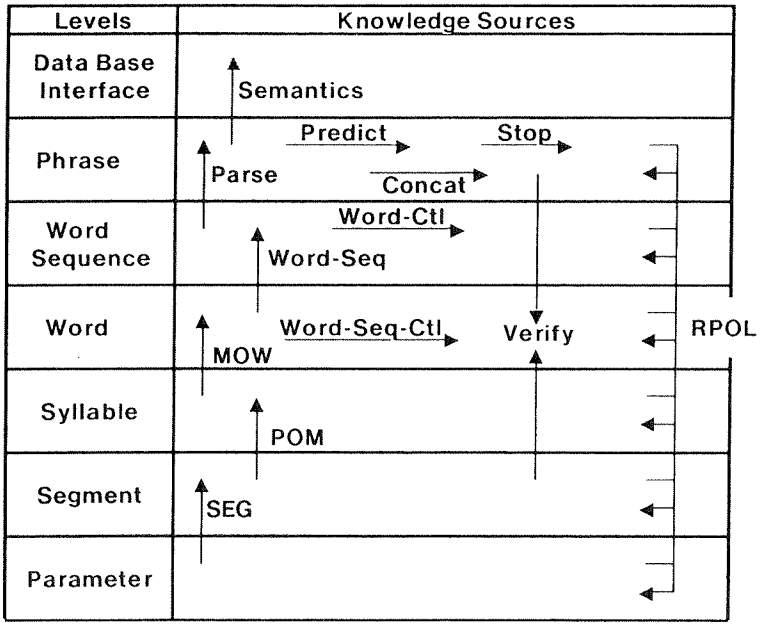


FIG. 6. (Levels and knowledge sources in HERSAY-II). The knowledge sources are as follows:
Semantics: generates interpretation for the information retrieval system.
SEG: digitizes the signal, measures parameters, produces labeled segmentation.
POM: creates syllable-class hypotheses from segments.
MOW: creates word hypotheses from syllable classes.
Word-Ctl: controls the number of hypotheses that MOW makes.
Word-Seq: creates word-sequence hypotheses for potential phrases.
Word-Seq-Ctl: controls the number of hypotheses that Word-Seq makes.
Predict: predicts words that follow phrases.
Verify: rates consistency between segment hypotheses and contiguous word-phrase pairs.
Concat: creates a phrase hypothesis from a verified contiguous word-phrase pair.
RPOL: rates the credibility of hypotheses.

solutions. They extract acoustic parameters, classify acoustic segments into phonetic classes, recognize words, parse phrases, and generate and evaluate hypotheses about undetected words and syllables. The KSs communicate through a global data base called a *blackboard* with seven information levels as shown in the figure. These levels are HEARSAY-II's heterogeneous abstraction spaces. The primary relationship between levels is compositional: word sequences are composed of words, words are composed of syllables, and so on. Entities on the blackboard are hypotheses. When KSs are activated, they create and modify hypotheses on the blackboard, record the evidential support between levels, and assign credibility ratings. For example, a sequence of acoustic segments can be evidence for identifying a syllable in a specific interval of the utterance; the identification of a word as an adjective can be evidence that the following word will be an adjective or noun.

HEARSAY-II's use of abstraction differs from the systems considered in cases 5 through 8. Those systems all use uniform abstraction spaces. The abstractions are uniform in that they use the same vocabulary as the final solutions and diper only in the amount of detail. For example in the planning systems, abstract plans have the same structure and vocabulary as final plans. In HEARSAY-II, the diversity of knowledge needed to solve problems justifies the use of heterogeneous abstraction spaces.

A computational system for understanding speech is caught between three conflicting requirements: a large space of possible messages to understand, variability in the signal, and the need to finish in a limited amount of time. The number of possible *ideal messages* is a function of the vocabulary, language constraints, and the semantics of the application. The number of actual messages that a system encounters is much larger than this because speech is affected by many sources of variability and noise. At the semantic level, errors correspond to peculiarities of conceptualization. At the syntactic level errors correspond to peculiarities of grammar. At the lexical and phonemic levels the variance is in word choice and articulation. Errors at the lower levels compound difficulties at the high levels. For example, the inability to distinguish between the four phrases

till Bob rings,
tell Bob rings,
till Bob brings,
tell Bob brings,

may derive from ambiguities in the acoustic levels. HEARSAY-II copes with this by getting the KSs at different levels to cooperate in the solution process. This has led to the following interesting architectural features:

(1) HEARSAY-II combines both top-down and bottom-up processing.

(2) HEARSAY-II reasons about resource allocation with a process called opportunistic scheduling.

An example of top-down processing is the reduction of a general sentential concept into alternate sentence forms, each sentence form into specific alternative word sequences, specific words into alternative phonic sequences and so on until a best interpretation is identified. Bottom-up processing tries to synthesize interpretations from the data. For example, one might combine temporally adjacent word hypotheses into syntactic or conceptual units. In HEARSAY-II, some KSs use top-down processing and other KSs use bottom-up processing.

All KSs compete to be scheduled and HEARSAY-II tries to choose the most promising KSs at any given moment using opportunistic scheduling. Opportunistic scheduling combines the idea of least commitment with strategies for managing limited computational resources. The opportunistic scheduler adapts automatically to changing conditions of uncertainty by changing the breadth of search. The basic mechanism for this is the interaction between KS-assigned credibility ratings on hypotheses and scheduler-assigned priorities of pending KS activations. When hypotheses have been rated equally, KS activations for their extension are scheduled together. In this way, ambiguity between competing hypotheses causes HEARSAY-II to search with more breadth, and to delay the choice among competing hypotheses until more information is brought to bear.

HEARSAY-II's approach to data interpretation differs from that of GA1 discussed in Section 3.4. Both programs contend with very large search spaces. Both programs need to have effective ways to rule out large classes of solutions. GA1 does this with early pruning. In the absence of constraints, it would expand every solution in the space. HEARSAY-II constructs a complete solution by extending and combining partial candidates. Because of its opportunistic scheduler, it heuristically selects a limited number of partial candidates to pursue. To avoid missing solutions, HEARSAY-II must not focus the search too narrowly on the most 'promising' subspaces.

In summary, HEARSAY-II provides an example of an architecture created to meet several conflicting requirements. Multiple levels provide the necessary abstractions for searching a large space. The levels are heterogeneous to match the diversity of the interpretation knowledge. Opportunistic scheduling combines the least-commitment idea with the ability to manage computational resources by varying the breadth of search and by combining top-down and bottom-up processing.

3.11. Case 11—General representation methods are too inefficient

Research on expert systems has benefited from the simplicity of using uniform representation systems. However, as knowledge bases get larger, the efficiency penalty incurred by using declarative and uniform representations can become significant. Attention to these matters will become increasingly important in

ambitiously conceived future expert systems with increasingly large knowledge bases.

This section considers architectural approaches for tuning the performance of expert systems by making changes to the representation of knowledge. Three main ideas will be considered:

(1) Use of specialized data structures;
(2) Knowledge compilation;
(3) Knowledge transformations for cognitive economy.

The organization of data structures has consequences for the efficiency of retrieving information. The selection and creation of efficient data structures is a principal part of most computer science curriculums. Consequently, several of the programs discussed in the previous cases (e.g., DENDRAL, GA1, and HEARSAY-II) use specialized data structures. In general, these data structures are designed so that facts that are used together are stored together and facts that are used frequently can be retrieved efficiently.

Choice of data structure depends on assumptions about how the data will be used. A common assumption about special data structures is that they are complete with regard to specific relationships. From example, in GA1's data structure for molecular hypotheses, molecular segments are connected if and only if they are linked in the data structure. This sort of assumption is commonplace in representations like maps, which are assumed to show all of the streets and street intersections. Representations whose structure in the medium is analogous to the structure of the world being modeled are sometimes called analogical representations.

A less understood issue is the use and selection of data structures in systems where many kinds of information are used. There is not much experience with systems that mix a variety of different representations. One step in this direction is to tag relations with information describing the chosen representation so that specialized information can be accessed and manipulated using uniform mechanisms. Some ideas along this line appeared in Davis' thesis [8], where schemata were used to describe some formatting and computational choices, but the work has not been extended to describe general dimensions of representation (see [4]) for use by a problem solver.

A second important idea for knowledge bases is the idea of *compiling* knowledge. By compilation, we mean any process which transforms one representation of knowledge into another representation which can be used more efficiently. This transformation process can include optimization as well as the tailoring of representations for particular instruction sets. Space does not permit a detailed discussion of techniques here, but some examples are listed below to suggest the breadth of the idea.

Example 1. Burton reported a system for taking ATN grammars and compiling them into executable code [5]. The compiled grammars could be executed to

parse sentences much more rapidly than previous interpreter-based approaches.

Example 2. Production system languages have been studied and experimented with for several years (e.g., see [9]). A basic difficulty with many production systems is that large production system programs execute more slowly than small ones. The extra instructions do not need to execute to slow down the system; their mere presence interferes with the matching process that selects productions to run. An example of one such language is OPS2 reported by Forgy and McDermott [19]. Forgy conducted a study of ways to make such production systems more efficient by compiling them into a network of 'node programs' [20]. The compiler exploits two properties of production systems: structural similarity and temporal redundancy. Structural similarity refers to the fact that many productions contain similar conditions; temporal redundancy refers to the fact that individual productions change only a few facts in the memory so that most of the data is unchanged from cycle to cycle. Forgy's RETE matching process exploits this by looking only at the *changes* in the memory. Forgy's analysis shows how several orders of magnitude of speed up can be achieved by compiling the productions and by making some simple changes to computing hardware.

Example 3. Another system that compiles a knowledge base of production rules, EMYCIN, was reported by van Melle [39]. EMYCIN is not considered a 'pure' production system since it is not strictly data-driven; the order that the rules are tried in EMYCIN is controlled by the indexing of parameters. This means that EMYCIN's interpreter does not repeatedly check elements in the working memory so that some of the optimizations used by Forgy would provide much less of an improvement for EMYCIN. EMYCIN's rule compiler concentrates on eliminating redundancy in the testing of similar patterns in rules and compiles them into decision trees represented as LISP code.

Example 4. The HARPY system for speech recognition reported by Lowerre [24] illustrates several issues about compilation. HARPY represents the knowledge for recognizing speech in a unified data structure (context-free production rules) which represents the set of all possible utterances in HARPY's domain. This data structure represents essentially the same information that was used in HEARSAY-II except for the parameterization and segmentation information. HARPY's knowledge compiler combines the syntax, lexical, and juncture knowledge into a single large transition network. First, it compiles the grammer into a word network, then it replaces each word with a copy of its pronunciation graph and inserts word-juncture rules at the word boundaries. In the final network, each path from a start node to an end node represents a sequence of segments for some sentence. With the knowledge in its compiled form, HARPY is capable of performing a rapid search process that attempts to find the best match between

the utterance and the set of interpretations. The major concerns about the extensibility of this idea are (1) that the highly stylized form of the input that HARPY can accept makes it difficult to add new knowledge and (2) the compilation is expensive for a large knowledge base (13 hours of PDP-10 time for a thousand word, simple grammar).

The promise of knowledge compilation ideas is to make it possible to use very general means for representing knowledge while an expert system is being built and debugged. Then a compiler can be applied to make the knowledge base efficient enough to compete with hand coding. Given a compiler, there is no need to sacrifice flexibility for efficiency: the knowledge base can be changed at any time and recompiled as needed. In addition, the compiler can be modified to re-represent the knowledge efficiently as hardware is changed, or as the trade offs of representation become better understood. The techniques for doing this are just beginning to be explored and will probably become increasingly important in the next few years.

So far in our discussion of efficiency we have assumed that it is necessary for the designers of a knowledge base to anticipate how knowledge will be used and to arrange for it to be represented efficiently. Lenat, Hayes-Roth, and Klahr [23] coined the term cognitive economy to refer to systems which automatically improve their performance by changing representations, changing access (e.g., caching), and compiling knowledge bases. Systems like this need to be able to predict how representations should be changed, perhaps by measurements on representative problems. The ideas of cognitive economy and knowledge compilation are more speculative than the other ideas we have considered and there are many theoretical and pragmatic issues to be resolved before they can be widely used. They are included here at the end in the hope that they will receive more attention in artificial intelligence research.

4. Summary

Our pedagogical tour of cases began with the consideration of a very simple architecture. It required that problems in an application have a small search space and that data and knowledge be reliable and constant.

In the second case we considered ways to cope with unreliable data or knowledge. Probabilistic, fuzzy, and exact methods were discussed. All of these methods are based on the idea of increasing reliability by combining evidence. Each method requires the use of meta-knowledge about how to combine evidence. The probabilistic (and pseudo-probabilistic) approaches use various a priori and conditional probability estimates; the fuzzy approaches use fuzzy set descriptions; the exact approaches use non-monotonic data correction rules. Errorful data and knowledge seem to be ubiquitous in real applications. In the HEARSAY-II example of case 10, we saw how an opportunistic scheduler was used to cope with the conflicting requirements of errorful data, limited com-

putational resources, and a large search space by varying the breadth of a heuristic search.

In the third case we considered ways to work with time-varying data. We started with the situational calculus and then considered a program that used transition rules to trigger expectations in a monitoring task. More sophisticated ways to reason with time seem to require more research.

The remaining cases dealt with ways to cope with large search spaces. We started with the hierarchical generate and test approach in the fourth case which requires a search space to be factorable in order to allow early pruning. This approach explores the space of solutions systematically and can be quite effective for returning all consistent solutions. We considered the issues of canonical forms and completeness in using this approach.

Generate and test is a weak method, applicable only when early pruning is feasible. It requires the ability to generate candidates in a way that large classes can be pruned from very sparse partial solutions. In many applications, solutions must be instantiated in substantial detail before they can be ruled out. Fortunately, it is not necessary in all applications to consider all possible solutions and to choose the best. The next few cases describe reasoning with abstractions to reduce the combinatorics without early pruning. The methods presented are usually applicable in satisficing tasks.

In case five, we considered a form of reasoning based on fixed abstractions. This approach requires that all of the information needed for testing partial solutions be available (or be able to be generated) before a subproblem is generated. This requirement exposes the weakness of this approach: some problems cannot be solved from the available information without backtracking. While the abstractions make the problem solver efficient, their use is too rigid for some applications.

The next approach to abstraction is more flexible. Abstractions are composed from a set of concepts in a hierarchical space. This method is called top-down refinement. The simplest version of top-down refinement (case six) uses a fixed criticality ordering of the concepts and a fixed partial order for solving subproblems.

Top-down refinement does not allow for variability in the readiness to make decisions. In the seventh case, we introduced the least commitment principle which says that decisions should be postponed until there is enough information. This approach tends to exploit the synergy of interactions between subproblems. It requires the ability to suspend activity in subproblems, move information between them, and then restart them as information becomes available. In this case, the problem-solving knowledge is much richer than in the previous methods. It was suggested that principles like least commitment should be incorporated as part of meta-level problem solving.

An inherent difficulty with pure least commitment approaches is the phenomenon of a least commitment deadlock. When a problem solver runs out

of decisions that it knows it can make correctly, it must guess to continue. We suggested that the amount of guessing is a measure of its missing knowledge, but that knowledge bases will always be incomplete. This theme was continued in case eight where dependency-directed backtracking was discussed as an architecture to support efficient retraction of beliefs in plausible reasoning.

Cases nine and ten illustrated: (1) the use of multiple lines of reasoning to enhance the power of a problem solver, (2) the use of heterogeneous abstraction models to capture the variety of knowledge in some applications, and (3) the use of an opportunistic scheduler to use knowledge sources as soon as they become applicable (either top-down or bottom-up) and to control the breadth of search.

Finally, we considered some methods for speeding up processing and information retrieval: specialized data structures and knowledge compilers. These techniques do not attack the basic combinatorics of search, but they do reduce the *cycle time* of problem solvers and will become increasingly important in future expert systems with large knowledge bases.

In articulating these ideas about expert systems, we were forced to decide what was essential and important about expert systems. In some cases, other choices of expert systems could have been made to make the same points. Our view is that the recent critical work in expert systems has focused on the mechanisms of problem-solving, and research in this area has been the most fruitful when it has been directed towards substantial applications. The applications and the research reported here are both empirical: knowledge bases are tried, tested, and revised; the architectural research follows a similar, but slower cycle.

An expert system is always a product of its time. System builders operate against a background of competing ideas and controversies and must also confront the limitations of their resources. To summarize experiments and create a simplified theory, we must necessarily step outside of this rich historical context. Inescapably, we are bound to our current vantage point. As this paper is written, there is a ferment of activity in expert system research. The theory of building intelligent systems is far from complete and the ideas expressed here are by no means universally accepted in the AI community. To wait for the ideas to settle and survive the test of history would preclude creating a timely guide to current thinking.

ACKNOWLEDGMENT

In August 1980, the National Science Foundation and the Defense Advance Research Projects Agency co-sponsored a workshop on expert systems in San Diego. The conference organizers were Rick Hayes-Roth, Don Waterman, and Douglas Lenat. The purpose of this workshop was to bring together researchers from many different institutions to write a definitive book [21] on expert systems. The participants were organized in groups corresponding to book chapters.

The ideas in this paper are the product of the 'architecture' group, of which Mark Stefik was the appointed chairman at the conference. The other members of the group were the other co-authors of this paper. During the conference we struggled intensively for three days to organize these ideas. The task of writing this tutorial fell to the chairman and was carried out over the next several months. Credit, however, belongs to all of the members of the group for actively and generously pulling together to define the scope of this tutorial, sharpen distinctions, organize the ideas, and later critique versions of the text.

Thanks also to Daniel Bobrow, John Seeley Brown, Johan de Kleer, Richard Fikes, Adele Goldberg, Richard Lyon, John McDermott, Allen Newell, and Chris Tong for reading early versions of this text and providing many helpful suggestions. Thanks to Lynn Conway for encouraging this work, and to the Xerox Corporation for providing the intellectual and computing environments in which it could be done.

REFERENCES

1. Barr, A. and Feigenbaum, E.A., The handbook of artificial intelligence, Computer Science Department, Stanford University, Stanford, CA, 1980.
2. Buchanan, B.G. and Feigenbaum, E.A., DENDRAL and Meta-DENDRAL: Their applications dimension, *Artificial Intelligence* **11** (1978) 5–24.
3. Bobrow, D.G. and Winograd, T., An overview of KRL, a knowledge representation language, *Cognitive Sci.* **1**(1) (1977) 3–46.
4. Bobrow, D.G., Dimensions of representation, in: D.G. Bobrow and A. Collins, Eds., *Representation and Understanding* (Academic Press, New York, 1975).
5. Burton, R.R., Semantic grammar: an engineering technique for constructing natural language understanding systems, Bolt Beranek and Newman Rept. No. 3453 (December 1976).
6. Charniak, E., Riesbeck C.K., and McDermott, D.V., *Artificial Intelligence Programming* (Erlbaum, Hillsdale, NJ, 1980).
7. Davis, R., Buchanan, B.G. and Shortliffe, E., Production rules as a representation for a knowledge-based consultation program, *Artificial Intelligence* **8** (1977) 15–45.
8. Davis, R., Applications of meta-level knowledge to the construction, maintenance and use of large knowledge bases, Ph.D. Thesis, Stanford University, AI Lab Memo AIM-283 (July 1976).
9. Davis, R. and King, J., An overview of production systems, in: E.W. Elcock and D. Michie, Eds., *Machine Intelligence* (Wiley, New York, 1976) 300–332.
10. de Kleer, J. and Sussman, G.J., Propagation of constraints applied to circuit synthesis, *Circuit Theory Appl.* **8** (1980) 127–144.
11. Doyle, J., A truth maintenance system, *Artificial Intelligence* **12** (1979) 231–272.
12. Doyle J. and London P., A selected descriptor-indexed bibliography to the literature on belief revision, *Sigart Newsletter* **71** (1980) 7–23.
13. Doyle, J., A model for deliberation, action, and introspection, AI TR 581, Artificial Intelligence Laboratory, MIT, Cambridge, MA (May 1980).
14. Duda, R.O., Hart, P.E. and Nilsson, N.J., Subjective Bayesian methods for rule-based inference systems, SRI International, Artificial Intelligence Center Technical Note 124 (January 1976).
15. Erman L.D., Hayes-Roth, F., Lesser, V.R. and Reddy, D.R., The HEARSAY-II speech-understanding system: integrating knowledge to resolve uncertainty, *ACM Comput. Surveys* **12**(2) (1980) 213–253.
16. Fagan, L.M., Kunz, J.C., Feigenbaum, E.A. and Osborn, J.J., Representation of dynamic clinical knowledge: measurement interpretation in the intensive care unit, *Proc. Sixth Internat. Joint Conf. Artificial Intelligence*, August 1979.
17. Fagan, J.M., VM: Representing time-dependent relations in a medical setting, Doctoral Dissertation, Computer Science Department, Stanford University, Stanford, CA, June 1980.

18. Feigenbaum, E.A., The art of artificial intelligence: I. Themes and case studies of knowledge engineering, *Proc. Fifth Internat. Joint Conf. Artificial Intelligence* (1977) 1014–1029.
19. Forgy, C. and McDermott, J., OPS: A domain-independent production system, *Proc. Fifth Internat. Joint Conf. Artificial Intelligence* (1977) 933–939.
20. Forgy, C.A., On the efficient implementation of production systems, Doctoral Dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, February 1979.
21. Hayes-Roth, F., Waterman, D. and Lenat, D., Eds., *Building Expert Systems* (in preparation).
22. Hayes-Roth, B. and Hayes-Roth, F., A cognitive model of planning, *Cognitive Sci.* **3** (1979) 275–310.
23. Lenat, D.B., Hayes-Roth, F. and Klahr, P., Cognitive economy, Computer Science Department, Stanford University, Heuristic Programming Project Rept. HPP-79-15 (June 1979).
24. Lowerre, B.T., The HARPY speech recognition system, Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1976.
25. McCarthy, J., and Hayes, P.J., Some philosophical problems from the standpoint of artificial intelligence, in: B. Meltzer and D. Michie, Eds., *Machine Intelligence* **9** (Wiley, New York, 1979).
26. McAllester, D.A., An outlook on truth maintenance, MIT AI Memo No. 551, April 1980.
27. McDermott, J., R1: A rule-based configurer of computer systems, Department of Computer Science, Carnegie-Mellon University, Rept. CMU-CS-80-119 (April 1980).
28. Newell, A., Artificial intelligence and the concept of mind, in: R.C. Schank and K.M. Colby, Eds., *Computer Models of Thought and Language* (Freeman, San Francisco, 1973).
29. Newell, A., Some problems of basic organization in problem-solving programs, in: M.C. Yovits, G.T. Jacobi and G.D. Goldstein, Eds., *Proc. Second Conf. on Self-Organizing Systems* (Spartan Books, Washington DC, 1962).
30. Nilsson, N.J., *Principles of Artificial Intelligence* (Tioga, Palo Alto, 1980).
31. Pednault, E.P.D., Zucker, S.W., and Muresan, L.V., On the independence assumption underlying subjective Bayesian updating, *Artificial Intelligence* **16** (1981) 213–222.
32. Sacerdoti, E.D., *A Structure for Plans and Behavior* (Elsevier, New York, 1977).
33. Sacerdoti, E.D., Planning in a hierarchy of abstraction spaces, *Artificial Intelligence* **5**(2) (1974) 115–135.
34. Shortliffe, E.H., *Computer-Based Medical Consultations*: MYCIN (Elsevier, New York, 1976).
35. Stallman, R.M. and Sussman, G.J., Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence* **9** (1977) 135–196.
36. Stefik, M.J., Planning with constraints, *Artificial Intelligence* **16**(2) (1981) 111–140.
37. Stefik, M., Inferring DNA structures from segmentation data, *Artificial Intelligence* **11** (1978) 85–114.
38. Sussman, G.J. and Steele, G.L., CONSTRAINTS—A language for expressing almost-hierarchical descriptions, *Artificial Intelligence* **14** (1980) 1–39.
39. van Melle, W., A domain-independent system that aids in constructing knowledge-based consultation programs, Doctoral Dissertation, Computer Science Department, Stanford University, Rept. No. STAN-CS-80-820 (June 1980).
40. Zadeh, L.A., A theory of approximate reasoning, in: J.E. Hayes, D. Michie and L.I. Mikulich, Eds., *Machine Intelligence* **9** (Wiley, New York, 1979).
41. Zadeh, L.A., Possibility theory and soft data analysis, University of California at Berkeley, Electronics Research Laboratory, Memo No. UCB/ERL M79/66 (August 1979).

Received August 1981