Artificial Intelligence in Medicine
I. De Lotto and M. Stefanelli (Editors)
© Elsevier Science Publishers B.V. (North-Holland) 1985

Software Tools for Developing Expert Systems

William J. Clancey
Stanford Knowledge Systems Laboratory
(incorporating the Heuristic Programming Project)
701 Welch Road, Building C
Palo Alto, CA 94304

"Expert systems" are programs with knowledge and inference procedures encoded in high-level computer languages. Ideally, these languages are structured to separate the knowledge of a domain from how it should be used, so that principles of fact and procedures for judgmental reasoning are stated in general form, rather than being implicitly and redundantly encoded. Such a design provides an opportunity and need for new software tools--programs that can reason about a knowledge base to partially automate the task of knowledge base design, debugging, and modification. Unfortunately, few existing languages are adequately developed and packaged for routine use. The situation has been worsened by our inadequate understanding of how to appropriately match available tools to problems. This paper provides two views of software tools for building expert systems, combining an appraisal of our slowly growing understanding of what tools must be able to do, with a survey of what well-developed systems, particularly EMYCIN, are capable of doing.

1. What is a knowledge engineering tool?

Over the past decade a variety of medical artificial intelligence (AI) programs have been developed (Szolovits, 1982, Clancey and Shortliffe, 1984). As for all *expert systems*, the methodology for writing these programs departs signficantly from traditional programming. Ideally, expert systems separate out *what is true about the world* from *how to use knowledge to solve problems.* The effect is that knowledge about a problem area, such as medicine, is written in a modular, *declarative* way--in the form of saying what is true. Procedures for using this knowledge, for example to carry on a diagnostic consultation, are written separately, so that the medical knowledge is *interpreted* by the procedure. Figure 1 illustrates this idea in a simple way.

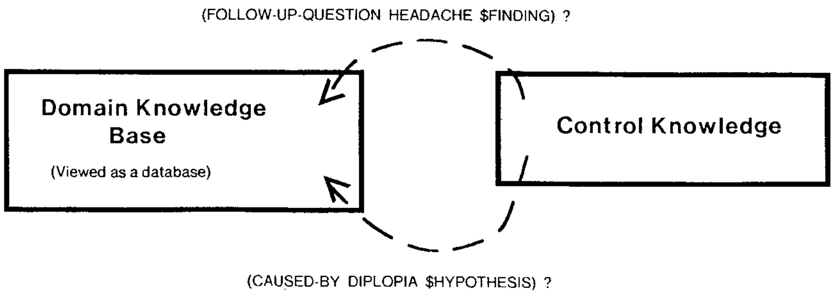

(FOLLOW-UP-QUESTION HEADACHE $FINDING) ?
Domain Knowledge Base
(Viewed as a database)
Control Knowledge
(CAUSED-BY DIPLOPIA $HYPOTHESIS) ?

Figure 1-1: Separation of medical knowledge and inference procedure

We refer to the process of defining a problem, identifying the medical facts and procedures, and representing this knowledge in a program as *knowledge engineering*. The term "engineering" in this context is somewhat strange, for the only physical artifacts knowledge engineers build are programs, and the term "programming" describes that process just fine. However, building an expert system involves a good deal of *organization and synthesis of knowledge* that is articulated by a domain expert in often a piecemeal, case-specific fashion. The main task of the knowledge engineer is to structure this knowledge and formulate principles for its use. Thus, the formulation of disease hierarchies and causal networks that can be used to practically simulate an expert's problem solving behavior involves "engineering" the knowledge so that it all works together.

Knowledge engineering tools are programs that the knowledge engineer uses to build an expert system. Again, in the traditional parlance, these are just software packages or *shells* that the knowledge engineer configures and augments to build programs for solving particular problems. Because expert systems are just programs written in computer programming languages, one could write an expert system in just the way ordinary programs are written, using screen editors and normal debugging techniques. However, as a result of the well-structured, modular nature of a knowledge base, it is possible to develop special programs for helping the knowledge engineer. These new programs are akin to drafting or computer-aided design software systems. They help the knowledge engineer write down knowledge in the program's language, test it for consistency, completeness, and correctness, and easily modify it. Thus, software tools for developing expert systems are special kinds of editors, printers, bookkeepers, and reporting programs.

Expert systems tools may be very complex, capitalizing on the latest methods in graphics, data abstraction, and program specification. For example, part of a knowledge base may be specified by drawing a network of disease states and symptoms. Thus, expert systems software tools derive their power from well-structured, high-level languages that enable the knowledge engineer to state facts in a direct way. The messy algorithmic details of ordinary programming are hidden in the knowledge interpreter, and not part of the normal concern of writing a new program. This is the advantage of using a tool to develop a new expert system.

A second, perhaps more significant, distinction between expert systems tools and ordinary software packages is that the tools themselves can reason about the knowledge in the program and help the knowledge engineer debug it, detect gaps, or understand how it is being used to solve problems. In this sense, the tools are analogous to the program that interprets the knowledge base to solve problems; they interpret the knowledge base for the purpose of building and evaluating it.

We can summarize the difference between expert systems tools and traditional software packages as follows:

- Expert systems, like traditional programs use variables for generality, but the data being interpreted are *symbolic facts*, not numbers.

- An expert system building tool is a software package, but it does more than solve single problems, it helps the knowledge engineer *construct new programs*, specifically, new expert systems.

- Separation of facts from procedures in expert systems is analogous to data base architectures, but (ideally) in expert systems the *relations are defined* so they can be reasoned about.

- Knowledge and procedures in expert systems follow the principles of structured programming to allow multiple use of knowledge (interpretation by different programs); but explanation, teaching, and learning specifically require that the *rationale for the design* be explicit.

In this paper, we will consider several software tools that help the knowledge engineer build expert systems. The examples are all medical, and appropriately so, for most of the earliest expert systems are in the medical arena. However, it should be remembered that the vast majority of systems under development today are non-medical. Where appropriate, we will consider new techniques that might be applied to medicine.

In the sections that follow, we consider requirements for selecting an appropriate tool, knowledge acquistion and explanation capabilities offered by existing tools, and the needs that tools of the future will have to satisfy.

2. Requirements for selecting a tool

Even though it has been a number of years since the first expert system tool, EMYCIN (van Melle, 1979), was developed, there have not been many well-defined principles for deciding what kind of tool to use to solve a particular problem. Various studies have demonstrated advantages of using one representation language instead of another--for ease in specifying knowledge relationships, control of reasoning, and perspicuity for maintenance and explanation (Clancey, 1981, Swartout, 1981, Aikins, 1983, Clancey, 1983a). Other studies have characterized in low-level terms why a given problem might be inappropriate for a given language, for example, because data are time-varying or subproblems interact (Hayes-Roth et al., 1983). But attempts to describe *kinds of problems* that a tool is good for, in terms of language requirements, have not been entirely satisfactory: Applications-oriented descriptions like "diagnosis" are too general (e.g., a diagnostic shell might allow for a physiological model, but not all diagnostic expert systems require one), and technological terms like "rule-based" describe a language for encoding knowledge, not what kind of problem is being solved (Hayes, 1977, Hayes, 1979).

However, now that we can study existing expert systems and make generalizations, it is possible to classify different kinds of medical problems and different methods for solving them. The first important idea (Section 2.1) is to consider *problem solving methods* in terms of two very simple alternatives: *Selecting* an answer from a pre-enumerated list versus *constructing* an answer out of pre-enumerated pieces. The second idea (Section 2.2) is that the problem being solved can be described in terms of some *system* that is being reasoned about. We classify problems according to what is being done with this system, for example diagnosing it or modifying it. Different *system problems* make one *problem solving method* more suitable than another, and this makes one software tool more suitable than another (Section 2.3).

2.1. Studying problem solving methods

2.1.1. Simple classification

The simplest kind of classification problem is to identify some unknown object or phenomenon as a member of a known class of objects, events, or processes. Typically, these classes are stereotypes that are hierarchically organized, and the process of identification is one of matching observations of an unknown entity against features of known classes. A paradigmatic example is identification of a plant or animal, using a guidebook of features, such as coloration, structure, and size. MYCIN solves the problem of identifying an unknown organism from laboratory cultures by matching culture information against a hierarchy of bacteria (Figure 2-1). While we show these as trees for simplicity, they are in reality often tangled hierarchies with multiple parents.

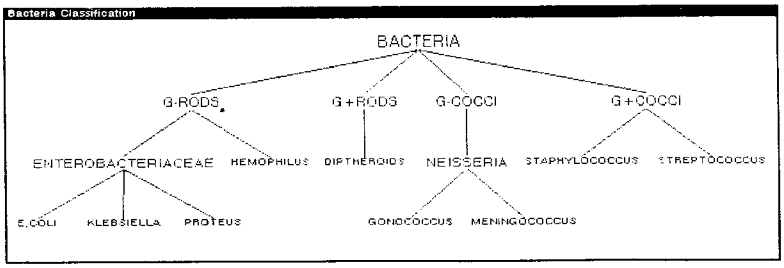


Figure 2-1: Mycin's classification of bacteria

The essential characteristic of classification is that the problem solver *selects* from a set of pre-enumerated solutions. This does not mean, of course, that the "right answer" is necessarily one of these solutions, just that the problem solver will only attempt to match the data against the known solutions, rather than construct a new one. Evidence can be uncertain and matches partial, so the output might be a ranked list of hypotheses.

Besides matching, there are several general *rules of inference* for making assertions about solutions. For example, evidence for a class is indirect evidence that one of its subtypes is present. We speak informally of inferring concepts by "matching," inferring a more general concept (moving upwards in a classification hierarchy) by "abstraction," and inferring a more specific concept by "refinement."

In the simplest problems, data are solution features, so the matching and refining process is direct. For example, an unknown organism in MYCIN can be classified directly given the supplied data of gram stain and morphology. The features "gram-negative" and "rod" match a class of organisms. The solution might be refined by getting information that allows subtypes to be discriminated.

For many problems, solution features are not supplied as data, but are inferred by *data abstraction.* There are three basic relations for abstracting data:

- *definitional abstraction* based on essential, necessary features of a concept ("if the method of drug administration is oral, then the drug is poorly absorbed");
- *qualitative abstraction,* a form of definition involving quantitative data, usually with respect to some normal or expected value ("if the patient is an adult and white blood count is less than 2500, then the white blood count is low"); and
- *generalization* in a type hierarchy ("if the organism is cryptococcus, then it is a fungus").

A good software tool will enable to the knowedge engineer to write down these facts directly, distinguishing explicitly among different kinds of abstraction relations.

2.1.2. Heuristic classification

In simple classification, data may directly match solution features or may match after being abstracted. In heuristic classification, solutions and solution features may also be matched *heuristically,* by direct, non-hierarchical association with some concept in another classification hierarchy. For example, MYCIN does more than identify an unknown organism in terms of visible features of an organism: MYCIN heuristically relates an abstract characterization of the patient to a classification of diseases. We show this *inference structure* schematically, followed by an example (Figure 2-2).

Basic observations about the patient are abstracted to patient categories, which are heuristically linked to diseases and disease categories. While only a subtype link with E.coli infection is shown here, evidence may actually derive from a combination of inferences. Some data might directly match E.coli features (an individual organism in the form of a rod, yielding a gram-negative stain is seen growing in a culture taken from the patient). Descriptions of laboratory cultures (describing location, method of collection, and incubation) can also be related to the classification of diseases. Note that the order in which these inferences are made is immaterial to our discussion.

The important link we have added is a heuristic association between a characterization of the patient ("compromised host") and categories of diseases ("gram-negative infection"). Unlike definitional and hierarchical inferences, this inference makes a great leap. A *heuristic relation* is uncertain, based on assumptions of typicality, and is sometimes just a poorly understood correlation.

Heuristics of this type reduce search by skipping over intermediate relations (this is why we don't call abstraction relations "heuristics"). These associations are usually uncertain because the intermediate relations may not hold in the specific case. Intermediate relations may be omitted because they are unobservable or poorly understood. In a medical diagnosis program, heuristics typically skip over the causal relations between symptoms and diseases.

To summarize, in heuristic classification an abstracted problem statement is associated with specific solutions or features that characterize a solution. This can be shown schematically in simple terms (Figure 2-3).

This diagram summarizes how a distinguished set of terms (data, data abstractions, solution abstractions, and solutions) are related systematically by *different* kinds of relations and rules of inference. This is the *structure of inference* in heuristic classification. As a *knowledge level description*, it should be contrasted with the implementation-level statement that "Mycin is a rule-based system with backward chaining." Helpful tools for building expert systems include a high-level language of knowledge distinctions, not just programming terminology.

2.1.3. Heuristic classification inference strategies

The arrows in inference structure diagrams indicate possible inference relations, connecting data to conclusions. However, the actual order in which assertions are made is often not strictly left to right, from data to conclusions. This process, most generally called *search*, includes a data-gathering strategy and an inference strategy for relating new data and hypotheses and to each other. Strategies are general principles for making choices.

The kinds of strategies that can be encoded in an expert system depend on the nature of the representation language. For example, most tools that implement a form of simple classification constrain reasoning to be hierarchically top-down or directly bottom up from known data. To enable heuristic classification reasoning, tools may allow "triggering rules." Search can then focus on intermediate hypotheses, so that it is *opportunistic* (combining data and hypothesis-directed inference in a flexible way).

Data- and hypothesis-directed search are not to be confused with the implementation terms "forward" or "backward chaining." R1, a computer-configuration expert systm, provides a superb example of how different implementation and knowledge level descriptions can be. Its rules are *interpreted* by forward-chaining, but it does a form of hypothesis-directed search, systematically setting up subproblems by a fixed procedure that focuses reasoning on spatial subcomponents of a solution (McDermott, 1982).

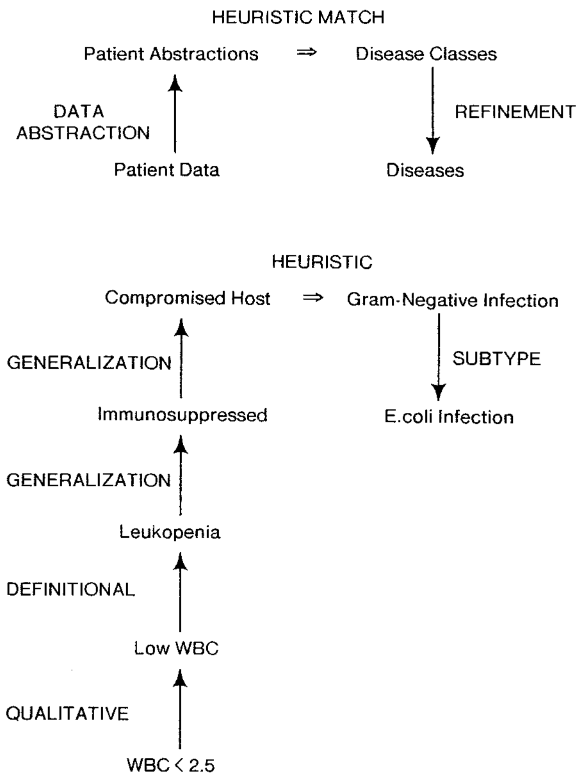


Figure 2-2: Inference structure of MYCIN

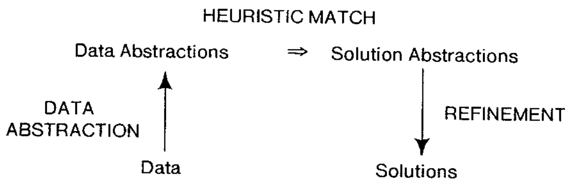


Figure 2-3: Inference structure of heuristic classification

The degree to which search is focused depends on the level of indexing in the implementation and how it is exploited. For example, MYCIN's "goals" are solution classes (e.g., types of bacterial meningitis), but selection of rules for specific solutions (e.g., E.coli meningitis) is unordered. Thus, MYCIN's search within each class is unfocused (Clancey, 1983b). Generalized heuristics, of the form "data class implies solution class" (e.g., "compromised host implies gram-negative rods") make it possible to focus search on useful heuristics in both directions (e.g., if attempting to classify a disease, recall that patient types are associated with diseases; if attempting to understand a patient's condition, recall that diseases are associated with patient types).

Opportunistic reasoning requires that at least some heuristics be indexed so that they can be applied in either direction, particularly so new data and hypothesized solutions can be related to previously considered data and solutions. The HERACLES program (Heuristic Classification Shell) cross-indexes data and solutions in several ways that were not available in EMYCIN. HERACLES' inference procedure consists 75 metarules comprising 40 reasoning tasks (Clancey, 1984). Focusing strategies include:

- working upwards in type hierarchies before gathering evidence for subtypes;
- discriminating hypotheses in terms of their process descriptions (location, change over time);
- making inferences that relate a new hypothesis to previously received data;
- seeking general classes of data before subclasses; and
- testing hypotheses by first seeking triggering and causal-enabling data.

In general, the rationale for an inference procedure might be very complex. A study of HERACLES' inference procedure reveals four broad categories of constraints:

- mathematical (e.g., efficiency advantages of hierarchical search)
- sociological (e.g., cost for acquiring data),
- cognitive (computational resources), and
- problem expectations (characteristics of typical problems).

These are discussed in some detail in (Clancey, 1984). Representing inference procedures so they can be explained and easily modified is currently an important research topic (e.g., see (Clancey, 1983a), (Genesereth, 1983), (Neches, et al., 1985)). Making the *assumptions* behind these procedures explicit so they can be reasoned about and dynamically modified is a challenging issue that AI is just beginning to consider.

The heuristic classification model emphasizes that domain concepts can be separated from an inference procedure. The "frame" paradigm fosters this point of view, leading to clear hierarchies of concepts. On the other hand, the "rule" paradigm demonstrates that much of the useful problem solving knowledge is in non-hierarchical associations, and that there are clear engineering benefits for procedures to be encoded explicitly, as conditional actions. HERACLES brings the advantages of both representations together, with domain concepts hierarchically related; domain rules for representing heuristic, non-hierarchical associations; and metarules for representing an inference procedure that interprets the domain knowledge, solving problems by heuristic classification. The architecture of HERACLES, with details about the encoding of metarules in MRS (Genesereth et al., 1981), a metarule compiler, and explanation program, is described in (Clancey, 1985).

2.1.4. Causal process classification

One form of heuristic classification, commonly used for solving diagnostic problems, is *causal process classification.* Data are generally observed malfunctions of some system, and solutions are abnormal processes causing the observed symptoms. We say that the inferred model of the system, the diagnosis, *explains* the symptoms. In general, there may be multiple causal explanations for a given set of symptoms, requiring an inference strategy that does not realize every possible association, but must reason about *alternative chains of inference.* In the worst case, even though diagnostic solutions are pre-enumerated (by definition), assertions may be taken back, so reasoning is *non-monotonic.* However, the most well-known programs that solve diagnostic problems by causal process classification are monotonic, dealing with alternative lines of reasoning by assigning weights to the paths. Indeed, many programs do not even compare alternative explanations, but simply list all solutions, rank-ordered.

In this section, we will compare SOPHIE (Brown, 1977), an electronic diagnostic program (which reasons non-monotonically, using assumption-based belief maintenance), to CASNET (Weiss, et al., 1978) (which compares alternative chains of inference without explaining contradictions), and NEOMYCIN, an augmentation and reconfiguration of MYCIN for teaching (which reasons exhaustively, using certainty factors to rank alternative inference chains). In these programs, solutions are pre-enumerated, but paths to them must be constructed. Our study serves several purposes: 1) To describe alternative heuristic classification inference strategies provided by different tools, 2) to distinguish between classification and constructive problem solving, and 3) to use the heuristic classification model to distinguish between electronic and medical diagnosis, to show again how a knowledge-level analysis allows problems and tools to be compared.

In SOPHIE, valid and abnormal device states are exhaustively enumerated, can be directly confirmed, and are causally related to component failures. None of this is generally possible in medical diagnosis, nor is diagnosis in terms of component failures alone sufficient for selecting therapy. Medical programs that deal with multiple disease processes (unlike MYCIN) do reason about abnormal states (called *pathophysiologic states,* e.g., "increased pressure in the brain"), directly analogous to the abnormal states in SOPHIE. But curing an illness generally involves determining the cause of the component failure. These "final causes" (called diseases, syndromes, etiologies) are processes that affect the normal functioning of the body (e.g., trauma, infection, toxic exposure, psychological disorder). Thus, medical diagnosis more closely resembles the task of computer system diagnosis in considering how the body relates to its environment (Lane, 1980). In short, there are two problems: First to explain symptoms in terms of abnormal internal states, and second to explain this behavior in terms of external influences (as well as congenital and degenerative component flaws). This is the inference structure of programs like CASNET and NEOMYCIN (Clancey, 1981) (Figure 2-4).

A network of causally related pathophysiologic states causally relates data to diseases. States are linked to states, which are then linked to diseases in a classification hierarchy. Diseases may also be non-hierarchically linked by heuristics ("X is a complication of Y" (Szolovits and Pauker, 1978)). The causal relations in these programs are heuristic because they assume certain physiologic structure and behavior, which is often poorly understood and not represented. In contrast with pathophysiologic states, diseases are abstractions of *processes*--causal stories with agents, locations, and sequences of events. Disease networks are organized by these process features (e.g., an organ system taxonomy organizes diseases by location). A more general term for disease is *disorder stereotype.*

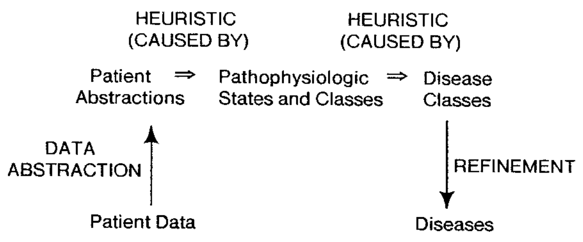


Figure 2-4: Inference structure of causal process classification

As mentioned above, programs differ in whether they treat pathophysiologic states as independent solutions (NEOMYCIN) or find the causal path that best accounts for the data (CASNET). If problem features interact, so that one datum causes another (D1 -> D2 in Figure 2-5), then paths of inference cannot be correctly considered independently. The second feature explains the first, so classifications (alternative explanations) of the former can be omitted; there is a "deeper cause" (C2 dominates C1). This presumes a single fault, an assumption common to programs that solve problems by heuristic classification.

$$\begin{tikzcd} & C1 & C2 \\ D0 \arrow[r] & D1 \arrow[u] \arrow[r] & D2 \arrow[u] \end{tikzcd}$$

Figure 2-5: Interacting data in classification

Moreover, a causal explanation of the data requires finding a state network, including normal states, that is internally *consistent on multiple levels of detail.* Combinatorial problems, as well as elegance, argue against pre-enumerating such networks, so solutions must be constructed, as in ABEL (Patil, 1981a). In ABEL, each diagnostic hypothesis is a separately constructed *case-specific model.* SOPHIE deals with many state interactions at the component level; others are captured in the exhaustive hierarchy of module behaviors. To do this, SOPHIE uses belief maintenance to detect faults by propagating constraints: recording assumptions (about correct behavior of components and modules) upon which inferences are based, explaining contradictory inferences in terms of violated assumptions, and making measurements to narrow down the set of possibly violated assumptions to a single fault.

In the simplest rule-based systems, such as MYCIN, search is exhaustive and alternative reasoning paths are compared by numerical belief propagation (e.g., certainty factors). A more complicated approach involves detecting that one reasoning path is subsumed by another, such as the conflict resolution strategy of ordering production rules according to specificity. HERACLES/NEOMYCIN uses such a strategy in not evoking a hypothesis for "non-specific" findings already explained by "red-flag" findings. CASNET uses a more comprehensive approach of finding the path with the greatest number of confirmed states and no denied states. The path describes a causal process, with attached findings explained by the process and a starting state at the head, the etiology or selected diagnosis.

Structure/function models are often touted as being more general by doing away with "ad hoc symptom-fault rules" (Genesereth, 1984). But programs that make a single fault assumption, such as DART, select a diagnosis from a pre-enumerated list, in this case negations of device descriptions, e.g., (NOT (XORG X1)), "X1 is not an exclusive-or gate." However, a structure/function model makes it possible to *construct tests.* Note that it is not generally possible to construct structure/function models for the human body, and is currently even impractical for the circuit SOPHIE diagnoses (IP-28 power supply).

To summarize, a knowledge level analysis reveals that medical and electronic diagnosis programs are not all trying to solve the same kind of problem. Examining the nature of solutions, we see that in a electronic circuit diagnosis program like SOPHIE solutions are component flaws, "parts that have gone bad." Medical diagnosis programs like CASNET attempt a second step, *causal process classification,* which is to explain abnormal states and flaws in terms of processes external to the device or developmental processes affecting its structure. It is this experiential knowledge--what can affect the device in the world--that is captured in disease stereotypes. This knowledge can't simply be replaced by a model of device structure and function, which is concerned with a different level of analysis.

The heuristic classification model and our specific study of causal process classification programs significantly clarifies what NEOMYCIN knows and does, and how it might be improved:

1. Diseases are processes affecting device structure and function;

2. Diseases are stereotypes;

3. Device history stereotypes (classes of patients) are distinct from diseases;

4. Pathophysiologic states are malfunctioning module behaviors.

Furthermore, it is now clear that the original (bacteremia) MYCIN system matches features of organisms (gram stain, morphology, etc.), using simple rules for filtering out contaminants. This knowledge has nothing to do with either malfunctioning parts or external processes. MYCIN simply has to classify an object that is growing in a culture. The meningitis knowledge base is more complex because it can infer the organism class heuristically, from patient abstractions, without having a culture result. NEOMYCIN goes a step further by dealing with different processes (infectious, trauma, psychogenic, etc.) and reasoning backwards from internal descriptions of current system processes to determine original causes (etiologies).

Probably the most important idea here is that medical diagnostic programs should separate descriptions of people from descriptions of diseases heuristically associated with them. Triggers should suggest patient types, just as they select diseases. Thus, medical diagnostic reasoning, when it takes the form of heuristic classification, is analogous to the problem solving stages of GRUNDY, the expert librarian that relates people stereotypes to books (Rich, 1982).

2.1.5. Selecting vs. constructing solutions

A common misconception is that there is a kind of problem called a "classification problem," opposing, for example, classification problems with design problems (Sowa, 1984). Indeed, some tasks, such as identifying bacteria, are inherently solved by *simple classification.* However, heuristic classification is *a discription of how a particular problem is solved by a particular problem solver.* If the problem solver has a priori knowledge of solutions and can relate them to the problem description by data abstraction, heuristic association, and refinement, then the problem can be solved by classification. For example, if it were practical to enumerate all of therapies that MYCIN might select, or if the solutions were restricted to predetermined sets of drugs, the program could be reconfigured to solve its problem by classification.

Furthermore, as illustrated by ABEL, it is incorrect to say that medical diagnosis is a "classification problem." In ABEL, a solution is a hierarchical description of a disease process, constructed by operators that group, refine, and causally relate disorder states on different levels of detail. As Pople concluded in analyzing the inadequacies of INTERNIST, only *routine* medical diagnosis problems can be solved by classification (Pople, 1982). When there are multiple, interacting diseases, there are too many possible combinations for the problem solver to have considered them all before. Just as ABEL reasons about interacting states, the physician must construct a consistent network of interacting diseases to explain the symptoms. The problem solver *formulates a solution*; he doesn't just make yes-no decisions from a set of fixed alternatives. For this reason, Pople calls non-routine medical diagnosis an ill-structured problem (Simon, 1973) (though it may be more appropriate to reserve this term for the theory formation task of the physician-scientist who is defining new diseases).

In summary, a useful distinction is whether a solution is selected or constructed. To *select* a solution, the problem solver needs experiential ("expert") knowledge in the form of *patterns of problems and solutions* and heuristics relating them. To *construct* a solution, the problem solver applies models of structure and behavior, by which objects can be assembled, diagnosed, or employed in some plan.

Of course, the question arises, how is constructing paths to solutions in NEOMYCIN (exhaustive inference), CASNET (comparative propagation of weights), or SOPHIE (assumption-based belief maintenance) different from constructing diseases described as networks in ABEL or CADUCEUS? If all programs construct inference paths, aren't they all solving problems by construction of solutions? At issue is a point of view about what is a solution. In NEOMYCIN, CASNET, and SOPHIE, the solutions are single faults, pre-enumerated. The inference paths and reasoning about them is a mechanism for selecting these solutions. In ABEL and CADUCEUS, solutions are descriptions of disease processes, constructed by multiple operators, not just propagation of belief (what we commonly call "implication"). The constructed solution is not simply an inference path from data to solution, but a *configuration* of primitive solution components. These programs *configure disease descriptions*, they do not select them directly.

We can relate the process of selecting a canned solution to theorem proving. Solving problems by construction, common to configuration design, requires *model-building operators that can piece together solution components into a coherent description of a system.* In fact, defining structure-building operators for describing disease processes has been of central concern in CADUCEUS and ABEL. Whether a solution can be selected or must be pieced together has important computational implications and is a pivotal consideration for choosing or designing a tool.

2.2. Describing problems in terms of systems

Besides the consideration of how solutions are computed, the heuristic classification model suggests that we classify problems according to what is being selected or constructed (diagnoses, patient stereotypes, therapies, etc.). A study of existing expert systems suggests a taxonomy of *kinds of problems* that an expert might solve. This in turn corresponds to a classification for *generic tools.*

Figures 2-6 and 2-7 summarize hierarchically what we can do to or with a system. A *system* is any set of parts that achieve some behavior through their interaction. Examples are: the respiratory system, the human body as a whole, and a hospital. We group operations in terms of those that *construct* a system and those that *interpret* a system, corresponding to what is generally called *synthesis* and *analysis.* Common synonyms appear in parentheses below the generic operations. In what follows, our new terms appear in upper case.

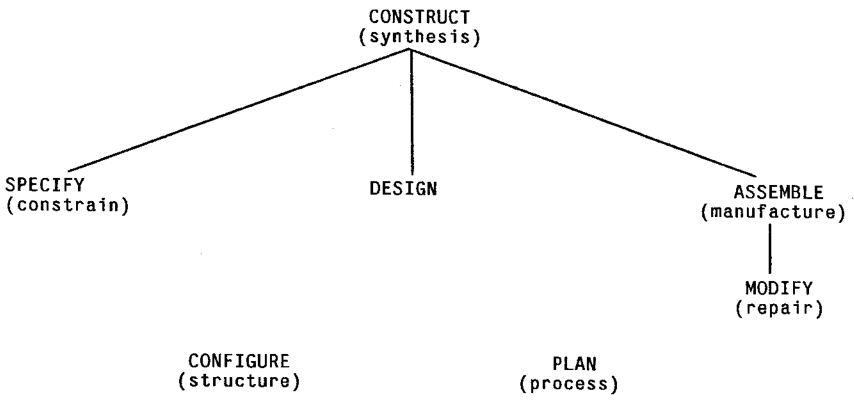


Figure 2-6: Generic operations for synthesizing a system

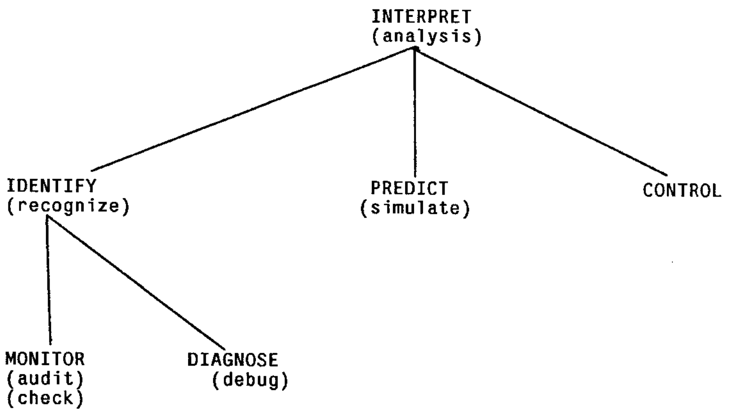


Figure 2-7: Generic operations for analyzing a system

INTERPRET operations concern a working system in some environment. In particular, IDENTIFY is different from DESIGN in that it takes I/O behavior and maps it onto a system. If the system has not been described before, then this is equivalent to (perhaps only partial) design from I/O pairs. PREDICT is the inverse, taking a known system and describing output behavior for given inputs. CONTROL, not often associated with expert systems, takes a known system and determines inputs to generate prescribed outputs (Vemuri, 1978). Thus, these three operations, IDENTIFY, PREDICT, and CONTROL, logically cover the possibilities of problems in which one factor of the set {input, output, system} is unknown.

Both MONITOR and DIAGNOSE presuppose a pre-existing system design against which the behavior of an actual, "running" system is compared. Thus, one identifies the system with respect to its deviation from a standard. In the case of MONITOR, one *detects* discrepancies in behavior (or simply characterizes the current state of the system). In the case of DIAGNOSE, one *explains* monitored behavior in terms of discrepancies between the actual (inferred) design and the standard system.

Given the above categorization of construction and interpretation problems, it is striking that expert systems tend to solve a sequence of problems pertaining to a given system in the world. Perhaps the most commonly occurring sequence in medicine is the *Maintenance Cycle:* {MONITOR + SIMULATE +} DIAGNOSE + MODIFY. The sequence of MONITOR and SIMULATE is commonly called *test*. MODIFY is also called *therapy*. MODIFY can be characterized as transforming a system to effect a redesign, usually prompted by a diagnostic description. MODIFY operations are those that change the *structure* of the system, for example, using drugs (or surgery) to change a living organism. Thus, MODIFY is a form of "reassembly" given a required design modification. MYCIN might be described as:

> MONITOR (patient state) + DIAGNOSE (disease category) +
> IDENTIFY (bacteria) + MODIFY (body system or organism)

When a problem solver uses heuristic classification for multiple steps, we say that the problem solving method is *sequential heuristic classification*. Solutions for a given classification (e.g., stereotypes of people) become data for the next classification step. Note that MYCIN does not strictly do sequential classification because it does not have a well-developed classification of patient types, though this is a plausible model of how human physicians reason. However, it seems fair to say that MYCIN does perform a monitoring operation in that it requests specific information about a patient; this is clearer in NEOMYCIN and CASNET where there are many explicit, intermediate patient state descriptions. On the other hand, SOPHIE provides a better example of monitoring because it interprets global behavior, attempting to detect faulty components by comparison with a standard, correct model (discrepancies are violated assumptions).

Heuristic classification is particularly well-suited for problems of interpretation involving a system that is known to the problem solver. In this case, the problem solver can select from a set of systems he knows about (IDENTIFY), known system states (MONITOR), known system faults (DIAGNOSE), or known system behaviors (SIMULATE/CONTROL). The heuristic classification method relies on experiential knowledge of systems and their behaviors. In contrast, constructing a new system may produce a new system, by definition requiring construction of new structures (new materials or new organizations of materials). Nevertheless, we intuitively believe that experienced problem solvers construct new systems by modifying known systems. This confluence of classification and constructive problem solving is an important area for research.

2.3. Relating methods and tasks to tools

What have we learned that enables us to match problems to tools? Given a task, such as medical diagnosis, we might have previously asked, "Is this tool good for diagnosis?" Now, we insert an intermediate question about *computational requirements*: Is it possible or acceptable to pre-enumerate solutions? Is it possible or acceptable to rank order solutions? Rather than matching tasks to tools directly, we interpose some questions about the method for computing solutions. The basic choice of classification versus construction is the missing link that takes us below implementation terminology ("rules," "blackboard," "units") to the level of conceptual structures and search.

In summary, we suggest the following sequence of questions for matching problems to tools:

1. Describe the problem in terms of a sequence of operations relating to systems. If the problem concerns ASSEMBLY or construction of a perceptual system, seek a specialist in another area of AI. If the problem concerns numerical SIMULATION or CONTROL, it might be solved by traditional systems analysis techniques.

2. Can solutions at each stage be pre-enumerated? If so, use heuristic classification. If not, is there a grammatical description that can be used to generate possible solutions? Are there well-defined solution construction operators that are constrained enough to allow an incremental (state-space) search?

3. Are there many uncertain choices that need to be made? If a few, exhaustive generation with a simple certainty-weighing model may be sufficient; if many, some form of lookahead or assumption/justification-based inference mechanism will be necessary.

Confusion about the nature of expert system problem-solving methods is pronounced in some recent commercial tools on the market that superficially appear to be designed to do heuristic classification. Close inspection reveals that they are capable of only simple classification, lacking structures for data abstraction, as well as a means to separate definitional features from heuristic associations between concepts (Harmon and King, 1985).

The relation of tool features to methods can be summarized as:

- *Classification*
  Useful knowledge representation features include: Rules, class hierarchy, uncertainty calculus. Inference methods include: inheritance (inferring specifics from generalizations), forward and backchaining deduction, top-down refinement, and opportunistic data and hypothesis-directed search.

- *Construction*
  Useful knowledge representation features include: All of the above, plus constraints, global work space ("blackboard" or partitioned database), and belief maintenance. Inference methods include least commitment (avoiding decisions that constrain future choices), multiple "hypothetical" worlds (branching on choices to construct alternative solutions), variable propagation or relaxation (systematic refinement), backtracking, version space search (bounding a solution using variables and constraints), and debugging (fixing an unsatisfactory solution).

While many of the conceptual structures and inference mechanisms a heuristic classification or construction program might need have now been identified, no knowledge engineering tool today combines these capabilities in a complete package. Neveretheless, a number of programs (all available as prototype software tools) provide some of the necessary features.

For example, the blackboard model of CRYSALIS (Engelmore and Terry, 1979) clearly separates data and solution abstractions. The EXPERT rule language (Weiss, 1979) similarly distinguishes between "findings" and a taxonomy of hypotheses. In PROSPECTOR (Hart, 1977), rules are expressed in terms of relations in a semantic network, making explicit the relations among clauses. In CENTAUR (Aikins, 1983), a variant of MYCIN, solutions are explicitly a hierarchy of disease *prototypes*. Chandrasekaran and his associates have been strong proponents of the classification model: "The normal problem-solving activity of the physician... (is) a process of classifying the case as an element of a disease taxonomy" (Chandrasekaran and Mittal, 1983). Recently, Chandrasekaran and Weiss and Kulikowski have generalized the classification schemes used by their programs (MDX and EXPERT) to characterize problems solved by other expert systems (Chandrasekaran, 1984, Weiss and Kulikowski, 1984).

In general, rule-based research in AI emphasizes the importance of heuristic association; frame systems emphasize the centrality of concepts, schema, and hierarchical inference. A series of knowledge representation languages beginning with KRL have identified structured abstraction and matching as a central part of problem solving (Bobrow and Winograd, 1979). These ideas are well-developed in KL-ONE, whose structures are explicitly designed for classification (Schmolze and Lipkis, 1983). Boose's ETS knowledge acquisition program (Boose, 1984) makes good use of a psychological theory of concept associations, called *personal construct theory*. However, ETS elicits only simple classifications from the expert, does not exploit distinctions between hierarchical, definition, and heuristic relations, and has no provision for data abstraction.

3. Knowledge acquisition tools

The issues of knowledge representation and inference aside, there are basic facilities that a tool can provide to make system development easier. In the course of developing a medical expert system, programmers naturally build tools that will help them construct and maintain the knowledge base. Most of these tools have a direct purpose and are useful, but some are experimental, motivated by what is logically possible. In this section we consider knowledge acquisition tools. Because most are obvious or easy to understand, they are only described briefly. Most of the examples are drawn from the EMYCIN system.

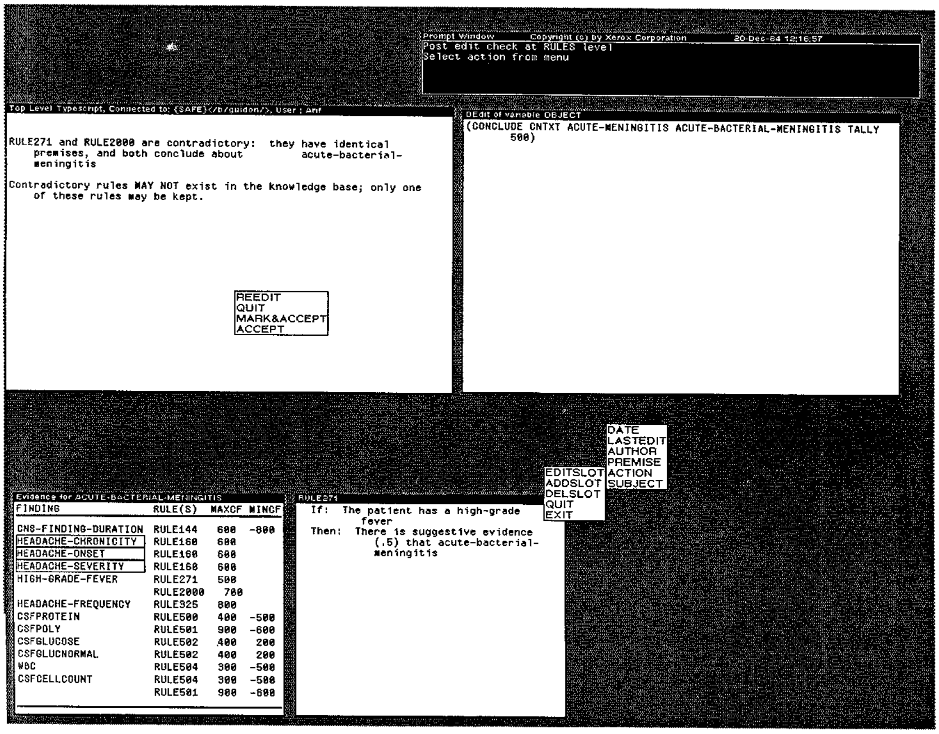
3.1. Knowledge base editors

The simplest knowledge acquisition tools are editors that prompt the knowledge engineer for portions of the knowledge base to be added, modified, or deleted. In what follows, an *object* is any knowledge base entity or unit, such as a rule, concept, or attribute. A *slot* is some description of an object, such as a name, author, or connection to other objects, such as a rule premise. An *expression* is the value associated with a slot. Useful features include:

- a "like" or "copy" command to copy an expression from one object to another;
- automatic prompting of required slots when acquiring a new object;
- renaming of objects (and automatic changes to all pointers within the system);
- a "semantic grammar" for type checking expressions (e.g., the expression must be a list of attribute names);
- representational metastatements that allow a program to enforce consistency (e.g., an indication that CAUSES is the inverse of CAUSED-BY);
- initialization for setting up a knowledge base from scratch (prompting the knowledge engineer for all necessary objects and helping him fill in necessary slots, as in EMYCIN);
- automatic prompting (and perhaps keeping an agenda) for describing objects that have been mentioned, but not defined.

With the easy availability of graphics, new editors allow the maintainer to point to objects and links to be modified. Figure 3-1 illustrates such a capability in HERACLES, in which browsing and editing are cleanly integrated by making a display serve as a menu.

Typically, editors check knowledge base changes for syntactic correctness and detect inconsistencies and redundancies. Changes may be marked for storage in a temporary file or made immediately effective through the means of random access (hashed) databases. In programs like EMYCIN, with many system developers making changes concurrently, programs merge "changes files" and report overlapping or inconsistent changes.



```
Prompt Window          Copyright (c) by Xerox Corporation          20-Dec-84 12:16:57
Post edit check at RULES level
Select action from menu
```

```
Top Level Typescript, Connected to: {SAFE}</b/guidon/>, User : Anf

RULE271 and RULE2000 are contradictory:  they have identical
    premises, and both conclude about         acute-bacterial-
    meningitis

Contradictory rules MAY NOT exist in the knowledge base; only one
    of these rules may be kept.
```

```
DEdit of variable OBJECT
(CONCLUDE CNTXT ACUTE-MENINGITIS ACUTE-BACTERIAL-MENINGITIS TALLY
       500)
```

```
Evidence for ACUTE-BACTERIAL-MENINGITIS
FINDING                 RULE(S)   MAXCF  MINCF

CNS-FINDING-DURATION    RULE144    600   -800
HEADACHE-CHRONICITY     RULE160    600
HEADACHE-ONSET          RULE160    600
HEADACHE-SEVERITY       RULE160    600
HIGH-GRADE-FEVER        RULE271    500
                        RULE2000    700
HEADACHE-FREQUENCY      RULE325    800
CSFPROTEIN              RULE500    400   -500
CSFPOLY                 RULE501    900   -600
CSFGLUCOSE              RULE502    400    200
CSFGLUCNORMAL           RULE502    400    200
WBC                     RULE504    300   -500
CSFCELLCOUNT            RULE504    300   -500
                        RULE501    900   -600
```

```
RULE271
 If:   The patient has a high-grade
          fever
 Then:   There is suggestive evidence
            (.5) that acute-bacterial-
            meningitis
```

Figure 3-1: Graphic editing in HERACLES

Most editors do standard bookkeeping regarding who changed the knowledge base and when. System maintainers are slowly realizing that it is critically important to document every change, particularly as knowledge bases become large and maintained for several years by different people. This documentation is accomplished online in EMYCIN by entering a "comments" subproperty for every object.

3.2. Seletively listing knowledge bases

While most editing occurs online, interactively, it is useful to have hardcopy listings of the portions or all of the knowledge base. These may be sorted in various ways: alphabetically, by internal codes (e.g., rules in numeric order), or by how knowledge is used or asserted. Probably the most useful way to list rules is by the attribute that they conclude about, e.g., rules in NEOMYCIN that conclude about the identity of the organism are grouped by name of the organism. Cross indices are also useful, e.g., in EMYCIN one cross index associates rules in numeric order with the attribute they conclude about.

The ability to selectively print the knowledge base online is useful for editing. These *browsing* facilities allow selective retrieval, treating the knowledge base just as a data base. It is also possible in EMYCIN to compute a list of rule or attribute names by selective retrieval and hand this list over to the editor, which will then prompt the maintainer for modification. For example, this is useful for examining and updating all rules that mention a particular attribute. Similarly, one can inform the editor that a particular subproperty (e.g., the natural language translation) of a given set of objects will be modified, and the editor will print and then prompt for changes to this subproperty.

Again, with the use of graphics, listing, and browsing can be made much easier. A means for pointing to objects on the screen and the use of menus obviates the need to remember internal program names and commands. In HERACLES, we have gone to great length to make the knowledge structures of the program inspectable. An example is shown in Figure 3-2. The graphics package of HERACLES is described in detail in (Richer and Clancey, 1985).

3.3. Dynamic tracing

An important tool for knowledge acquisition is the capability to inspect the program's reasoning while it is solving a problem. At first, knowledge engineers used the conventional method of printing out at routine intervals what the program is doing. This could be done easily by modifying the inference engine to indicate when a rule was being tried, when a conclusion was made, etc. However, for complex programs, such extensive traces can be voluminous. An obvious partial solution is to have different levels of tracing detail. Thus, in HERACLES setting a flag to different values will reveal (cumulatively):

1. Conclusions made by the program.
2. When a task is attempted and when a rule succeeds.
3. When a rule fails.
4. When a domain goal is pursued.

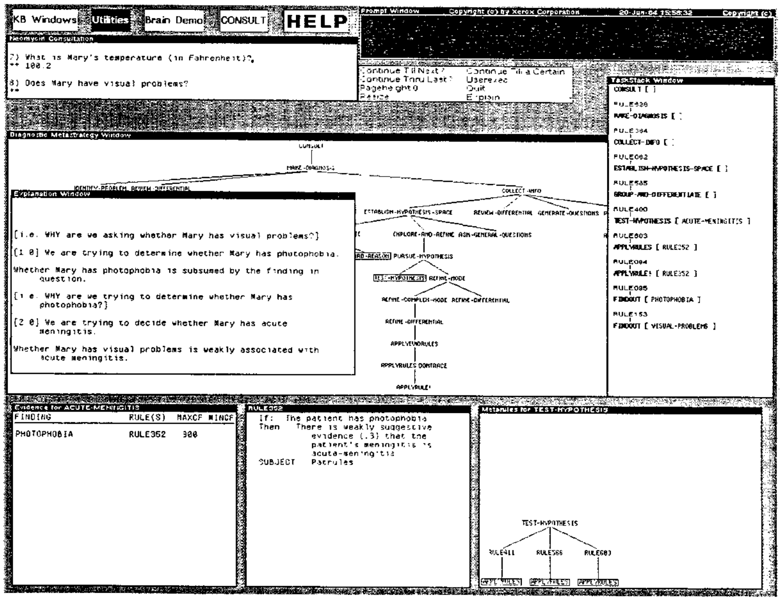
This is not satisfactory either, for there is no means to go back and get more details. The obvious solution is to keep a history and to allow selective retrieval at any time. With the use of graphics, retrieval is convenient and easy. Figure 3-2 shows some of the information available in GUIDON-WATCH, an experimental system for making all of NEOMYCIN's reasoning accessible to a student or system maintainer.

There are many complex issues in constructing a graphics tracing facility of this kind, including how to make the interface transparent, how to manage windows on the screen, and when to update the display. These issues are discussed in some detail in (Richer and Clancey, 1985).

The next more sophisticated approach is an *explanation program*, which reasons about which part of the history and knowledge base to mention to the maintainer. This is described in a separate section below. We note in passing that it is also easy to build "break packages" for expert systems such as to cause a trap or interrupt to occur when certain knowledge is used or concluded about. However, these are not universally used, perhaps because the natural form of a consultation program causes the program to stop at each question, allowing arbitrary inspection and modification.

3.4. Case libraries

Given that the only direct means of assuring knowledge base quality for most problems is to test the system on a variety of cases, it becomes important to have a facility for saving, retrieving, and rerunning a library of test cases. For example, in developing the meningitis expertise of MYCIN, a library of 100 cases were collected and systematically rerun with every major system modification.



Neomycin Consultation

```
7) What is Mary's temperature (in Fahrenheit)?
** 100.2

8) Does Mary have visual problems?
**
```

Explanation Window

```
[i.e. WHY are we asking whether Mary has visual problems?]

[1.0] We are trying to determine whether Mary has photophobia.

Whether Mary has photophobia is subsumed by the finding in
     question.

[i.e. WHY are we trying to determine whether Mary has
     photophobia?]

[2.0] We are trying to decide whether Mary has acute
     meningitis.

Whether Mary has visual problems is weakly associated with
     acute meningitis.
```

TaskStack Window

```
CONSULT [ ]
RULE526
MAKE-DIAGNOSIS [ ]
RULE384
COLLECT-INFO [ ]
RULE062
ESTABLISH-HYPOTHESIS-SPACE [ ]
RULE565
GROUP-AND-DIFFERENTIATE [ ]
RULE400
TEST-HYPOTHESIS [ ACUTE-MENINGITIS ]
RULE603
APPLYRULES [ RULE352 ]
RULE084
APPLYRULE! [ RULE352 ]
RULE085
FINDOUT [ PHOTOPHOBIA ]
RULE153
FINDOUT [ VISUAL-PROBLEMS ]
```

Evidence for ACUTE-MENINGITIS

FINDING	RULE(S)	MAXCF	MINCF
PHOTOPHOBIA	RULE352	300	

RULE352

```
If:   The patient has photophobia
Then    There is weakly suggestive
          evidence (.3) that the
          patient's meningitis is
          acute-meningitis
SUBJECT   Patrules
```

Figure 3-2: Task stack in NEOMYCIN

When run in "batch mode," EMYCIN will flag new questions (and automatically answer "unknown" if no information is in the case record), also it summarizes how the diagnosis and therapy have changed since the last time the case was solved by the program. The batch program automatically invokes the explanation program to indicate why the new answers were concluded. All of this is printed in files in the form of messages that can be selectively inspected, sorted, and disposed of.

The case library facility of EMYCIN includes text summaries of each case (written by the person who entered the case), besides a record of the "correct" and last answers. It is possible to read these summaries online, but no facility was developed for indexing cases on the basis of case data or diagnoses, e.g., "What cases mention that the patient has a fever?" However, with continued expansion of the case library (and its use for teaching), such a facility would be useful.

3.5. Semi-automated learning

The well-structured form of a knowledge base makes it conceivable to construct tools that reason about possible improvements to the system. Probably the best example of this is SEEK (Politakis and Weiss, 1984), a system which analyzes performance of the program over a range of cases and proposes and carries out experiments for modifying the knowledge base. However, such programs are always limited by the extent to which they understand the *design* of the knowledge base. For example, it makes little sense to suggest reordering or deleting rule clauses if the program doesn't have any understanding of the role each clause plays and their relation to one another. Thus, automatic learning places a premium on making knowledge explicit at different levels of detail.

One intriguing method for semi-automatically constructing a knowledge base is to use some method of pattern recognition to analyze a case library and construct rules that will discriminate among the cases (given a correct diagnosis). This method has been tried experimentally in the ODYSSEUS system, using MYCIN's case library (Wilkins, et. al, 1984). The analysis helps us understand more about the space of all possible cases and what case data are most significant for making a diagnosis. In contrast, when we construct an ordinary knowledge base as a collection of rules, we may have no idea which could be deleted and which are most important for making a correct diagnosis.

In this respect, two other forms of analysis are possible. First, one can simply keep statistical records of how knowledge is used by the program, and detect outliers, such as rules that succeed every time or are never tried. (This was done by Aikins for EMYCIN.) Second, one can perform a sensitivity analysis, either analytically, or by systematically modifying the program to see how it affects performance. Such an experiment was performed for MYCIN, and it revealed how the coverage of drugs for broad classes of organisms make the program relatively insensitive to changes in rule certainty factors. In the PROSPECTOR system, sensitivity analysis is used as a routine means of testing and developing the knowledge base.

3.6. Generic systems

With the formalization of architectures for problem solving, specialized for some combination of methods and tasks, generic knowledge acquisition programs can be built. An early example is ROGET (Bennett, 1983), a program that guides the knowledge engineer in the early design of his EMYCIN knowledge base. The program has built-in conceptual structures that describe different kinds of attributes and how they are typically related in diagnostic programs. The program interviews the knowledge engineer and helps him decide whether EMYCIN is appropriate for his task and how it should be configured. These conceptual structures are developed significantly further in HERACLES, but a knowledge acquisition interview facility has not been constructed for this program.

4. Explanation and debugging tools

While explanation and debugging tools are an intricate part of knowledge acquisition, we treat them separately because they are complex, major areas of research.

The simplest form of "explanation" is a natural language translation of the audit trail, a kind of glorified trace-printing routine. An example is the well-known HOW and WHY facility of EMYCIN. The program shows one slice of the reasoning, the goal stack, but is exhaustive in what it prints. The main advantage is that the line of reasoning is clearly laid out and the user or maintainer may skip through the history at will. This facility was greatly extended in TEIRESIAS, in which the program tracks back through the reasoning to determine what rule modifications would change the final answers (fixing omissions or incorrect diagnoses). This can be a very convenient facility for the naive user, but in practice most debugging was done in MYCIN by programmers who were intimately familiar with knowledge base details. For them, the debugging dialog was tedious and unnecessary. However, as knowledge bases grow in complexity we should expect that systems like TEIRESIAS will not only be convenient, but necessary.

Below we consider some advanced forms of explanation that are useful for constructing expert systems.

4.1. Explaining "why not"

Even a simple explanation facility can be difficult to write if it is to explain why the program does not behave in a different manner. EMYCIN has the capability to explain why a question wasn't asked, a rule was not applied (or did not succeed), and why a conclusion was not made. Again, we expect that such a facility will be useful for people who are not already familiar with the knowledge base.

4.2. Comparing alternatives

Explaining why the program didn't do something differently is particularly valuable because when people do not understand something they often have an expectation of what should have happened. Thus, a question of "Why did you do X?" often is based on an implicit, "I thought that Y was the right thing to do." A natural way to ask the question is therefore, "Why did you do X instead of Y?" Rather than responding to either X or Y independently, the program then tries to find some direct way of comparing the alternatives. This capability is nicely illustrated by the explanations provided by MYCIN's therapy program (Clancey, 1977).

4.3. Condensation and user models

The aim of most researchers who develop explanation facilities is to incorporate user models and sophisticated text generation, allowing explanations to be more appropriate and concise. While no readily available software tool for developing expert systems goes beyond an audit trail explanation, knowledge representation research progress suggests that future systems will be able to generate text more flexibly (Swartout, 1981, Patil, 1981b, Clancey, 1985).

5. Future needs

Software tools for developing expert systems only exist as prototypes today; many more capabilities are possible and will be necessary in the future. Considerations for the future include:

- the use of higher-level languages, with more generic, built-in structures,
- compilers for runtime efficiency, and
- interface languages for integrating knowledge-based systems with:
    - graphics and report-generating programs
    - databases,
    - word processors,
    - signal-detection and measuring devices, and
    - spreadsheet programs.

Indeed, it is likely that rather than being stand-alone systems as they are today, often restricted to specialized hardware, the expert systems of the future will exist as software packages, or perhaps as "outboard engines" for making inferences. Of considerable practical concern is that expert systems communicate with traditional software tools (like graphics and statistical packages), rather than requiring these programs to be recoded in special AI languages.

Finally, recent research indicates that it might be difficult or practically impossible to design a language for conceptual structures that can be unambiguously and consistently used by knowledge engineers (Brachman, 1983). Just as the rule notation was "abused" in MYCIN by ordering rule clauses to encode hierarchical and procedural knowledge, users of KL-ONE implicitly encode knowledge in structural properties of concept hierarchies, relying on the effect of interpreter to make correct inferences. Brachman proposes a model of *knowledge representation competence*, in which a program is told what is true and what it should do, and left to encode the knowledge according to its own conventions to bring about the correct reasoning performance.

6. Conclusions

In this paper we considered two aspects of software tool design for expert systems--the requirements for expressing knowledge and software engineering facilities. While treated separately, the two are not unrelated. For example, tools can provide specialized aids for particular kinds of applications or solution methods. Yet for the most part, generic systems are still in the future. Our appraisal of existing systems is that they have been poorly described in terms of what problems they are solving and the methods they use. Therefore, the software tools built upon them are limited to editors, browsers, and other syntactic forms of construction and debugging aids.

In our consideration of expert system tools, begin with the ideal that an expert system should know explicitly whatever people can articulate about its design, so that the program can automatically explain and modify itself. We observe patterns in existing knowledge bases that indicate a missing "knowledge level" of description about problem solving methods and tasks. Making these patterns explicit is of value scientifically, to develop theories of problem solving, as well as providing a basis for software engineering--tools tuned to patterns of use and the problems they are applied to solve. Consideration of existing tools reveals that none fully incorporate the range of features that one or more individually have demonstrated to be useful.

Over the next few decades, the main problems of AI will probably remain focused on issues of representation and inference. Meanwhile, we can expect the traditional areas of computer science, especially database inference, to integrate AI techniques in their methodologies. Differences between these areas of computer science and current-day AI may blur. Significantly, the methodology of software engineering itself may become transformed, as tools for building expert systems themselves become intelligent.

Acknowledgments

I would like to thank the former members of the MYCIN project (Jan Aikins, Jim Bennett, Randy Davis, Larry Fagan, Carli Scott, Ted Shortliffe, Bill van Melle), my colleagues at the Stanford Knowledge Systems Laboratory (Bruce Buchanan, Diane Hasling, Arif Merchant, Mark Richer, Tim Thompson) and my associates at Teknowledge, Inc. (Denny Brown, Lee Erman, Bob London, Phil London, Steve Hardy, Rick Hayes-Roth, Earl Sacerdoti) who have constructed so many good examples of software tools for building expert systems.

This research has been supported in part by ONR and ARI Contract N00014-79C-0302, and by the Josiah Jr. Macy Foundation. Computational resources have been provided by the SUMEX-AIM facility (NIH grant RR00785). Graphic output was all produced on a XEROX-1108 running INTERLISP-D.

References

Aikins J. S. Prototypical knowledge for expert systems. *Artificial Intelligence*, 1983, *20(2)*, 163-210.

Bennett, J. *ROGET: A knowledge-based consultant for acquiring the conceptual structure of an expert system.* HPP Memo 83-24, Stanford University, October 1983.

Bobrow, D. and Winograd, T. KRL: Another perspective. *Cognitive Science*, 1979, *3*, 29-42.

Boose, J. *Personal construct theory and the transfer of human expertise*, in *Proceedings of the National Conference on AI*, pages 27-33, Austin, TX, August, 1984.

Brachman, R. J., Fikes, R. E. , and Levesque, H. J. A functionally-specified representation system. (FLAIR report).

Brown, J. S. *Remarks on building expert systems (Reports of panel on applications of artificial intelligence)*, in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 994-1005, 1977.

Chandrasekaran, B. Expert systems: Matching techniques to tasks. In W. Reitman (editor), *AI Applications for Business*, pages 116-132. Ablex Publishing Corp., 1984.

Chandrasekaran, B. and Mittal, S. Conceptual representation of medical knowledge. In M. Yovits (editor), *Advances in Computers*, pages 217-293. Academic Press, New York, 1983.

Clancey, W. J. *An antibiotic therapy selector which provides for explanations*, in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 858, August, 1977. (Also in Buchanan and Shortliffe (editors), *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, 1984).

Clancey, W. J. and Letsinger, R. *NEOMYCIN: Reconfiguring a rule-based expert system for application to teaching*, in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 829-836, August, 1981. (Revised version in Clancey and Shortliffe (editors), *Readings in medical artificial intelligence: The first decade*, Addison-Wesley, 1984).

Clancey, W. J. *The advantages of abstract control knowledge in expert system design*, in *Proceedings of the National Conference on AI*, pages 74-78, Washington, D.C., August, 1983.

Clancey, W. J. The epistemology of a rule-based expert system: A framework for explanation. *Artificial Intelligence*, 1983, *20(3)*, 215-251.

Clancey, W. J. *Acquiring, representing, and evaluating a competence model of diagnosis.* HPP Memo 84-2, Stanford University, February 1984. (To appear in Chi, Glaser, and Farr (Eds.), *Contributions to the Nature of Expertise*, in preparation.).

Clancey, W. J. Representing control knowledge as abstract tasks and metarules. (To appear in *Computer Expert Systems*, eds. M. J. Coombs and L. Bolc, Springer-Verlag, in preparation).

Clancey, W. J. and Shortliffe, E. H. *Readings in medical artificial intelligence: The first decade.* Reading: Addison-Wesley 1984.

Engelmore, R. and Terry, A. *Structure and function of the CRYSALIS system*, in *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 250-256, August, 1979.

Genesereth, M. R. *An overview of meta-level architecture*, in *Proceedings of The National Conference on Artificial Intelligence*, pages 119-124, August, 1983.

Genesereth, M. R. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 1984, *24(1-3)*, 411-436.

Genesereth, M.R., Greiner, R., Smith, D.E. *MRS Manual.* Heuristic Programming Project Memo HPP-80-24, Stanford University, December 1981.

Harmon, P. and King D. *Expert Systems: Artificial Intelligence in Business.* New York: John Wiley & Sons 1985.

Hart, P. E. *Observations on the development of expert knowledge-based systems,* in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* pages 1001-1003, August, 1977.

Hayes, P.J. *In defence of logic,* in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* pages 559-565, August, 1977.

Hayes, P. The logic of frames. In D. Metzing (editor), *Frame Conceptions and Text Understanding,* pages 45-61. de Gruyter, 1979.

Hayes-Roth, F., Waterman, D., and Lenat, D. (eds.). *Building expert systems.* New York: Addison-Wesley 1983.

Lane, W. G. Input/output processing. In Stone, H. S. (editor), *Introduction to Computer Architecture, 2nd Edition,* chapter 6. Science Research Associates, Inc., Chicago, 1980.

McDermott, J. R1: A rule-based configurer of computer systems. *Artificial Intelligence,* 1982, *19(1),* 39-88.

Neches, R., Swartout, W. R., and Moore, J. Explainable (and maintainable) expert systems. .

Patil, R. S., Szolovits, P., and Schwartz, W. B. *Causal understanding of patient illness in medical diagnosis,* in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence,* pages 893-899, August, 1981.

Patil, R. S. *Causal representation of patient illness for electrolyte and acid-base diagnosis.* MIT/LCR/TR 267, Massachusetts Institute of Technology, October 1981.

Politakis, P. and Weiss, S. M. Using empirical analysis to refine expert system knowledge bases. *Artificial Intelligence,* 1984, *22(1),* 23-48.

Pople, H. Heuristic methods for imposing structure on ill-structured problems: the structuring of medical diagnostics. In P. Szolovits (editor), *Artificial Intelligence in Medicine,* pages 119-190. Westview Press, 1982.

Rich, C. *Knowledge Representation Languages and Predicate Calculus,* in *Proceedings of the National Conference on Artificial Intelligence,* pages 193-196, AAAI, 1982.

Richer, M. and Clancey, W. J. GUIDON WATCH: A graphic interface for browsing and viewing a knowledge-based system. (Submitted to *IEEE Computer Graphics and Applications*).

Schmolze, J. G. and Lipkis, T. A. *Classification in the KL-ONE knowledge representation system,* in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* pages 330-332, August, 1983.

Simon, H. A. The structure of ill structured problems. *Artificial Intelligence,* 1973, *4,* 181-201.

Sowa, J. F. *Conceptual Structures.* Reading, MA: Addison-Wesley 1984.

Swartout W. R. *Explaining and justifying in expert consulting programs,* in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence,* pages 815-823, August, 1981.

Szolovits, P. (editor). *Artificial Intelligence in Medicine.* Boulder: Westview Press, Inc. 1982.

Szolovits, P. and Pauker, S. G. Categorical and probabilistic reasoning in medical diagnosis. *Artificial Intelligence,* 1978, *11,* 115-144.

van Melle, W. *A domain-independent production rule system for consultation programs,* in *Proceedings of the Sixth International Joint Conference on Artificial Intelligence,* pages 923-925, August, 1979.

Vemuri, V. *Modeling of Complex Systems: An Introduction.* New York: Academic Press 1978.

Weiss, S. M. and Kulikowski, C. A. *EXPERT: A system for developing consultation models*, in *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 942-947, August, 1979.

Weiss, S. M. and Kulikowski, C. A. *A Practical Guide to Designing Expert Systems.* Totowa, NJ: Rowman and Allanheld 1984.

Weiss, S. M., Kulikowski, C. A., Amarel, S., and Safir, A. A model-based method for computer-aided medical decision making. *Artificial Intelligence*, 1978, *11*, 145-172.

Wilkins, D., Buchanan, B. G., Clancey, W. J. *Inferring an expert's reasoning by watching*, in *Proceedings of the 1984 Conference on Intelligent Systems and Machines*, 1984.