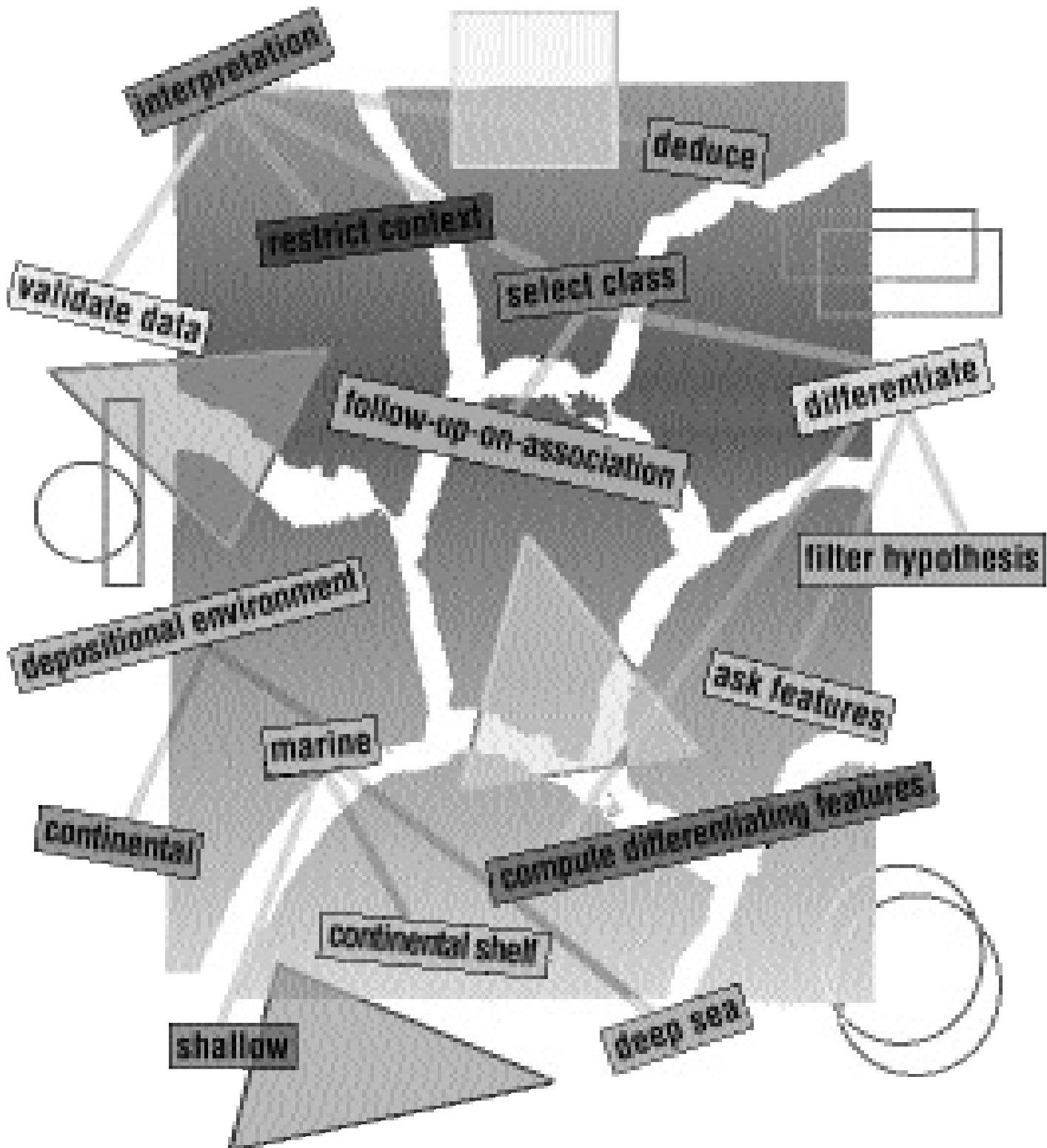


Components of Expertise

Luc Steels



Over the past decade, it has become clear that one should go beyond the level of formalisms and programming constructs to understand and analyze expert systems. In this article, I first review some of the existing proposals. I discuss the idea of inference structures

such as heuristic classification (Clancey 1985), the distinction between deep and surface knowledge (Steels 1984), the notion of problem-solving methods and domain knowledge filling roles required by the methods (McDermott 1988), and the idea of generic tasks and task-specific architectures (Chandrasekaran 1983). These various proposals are obviously related to each other, which makes it desirable to construct a synthesis that combines their strengths. Such a synthesis is presented here in the form of a componential framework. The framework stresses modularity and consideration of the pragmatic constraints of the domain.

Motivation

A major question with knowledge engineering is (or should be) that given a particular task, how do we go about solving it using expert system techniques. The standard answer to this question used to be computational in nature. Textbooks talk about different computational formalisms such as rules, frames, and logic programming. They assume that knowledge can be translated more or less directly into computational structures from observations of the expert's problem solving or from verbal reports about this knowledge. It is true that at some point in the process of developing a working application, we have to face decisions on which implementation medium to use; however, the computational answer is only partly satisfactory. The gap between the implementation level and the knowledge and problem solving that we observe in the human expert is too wide. What is needed is another level of discourse that talks about knowledge and problem solving independent of their implementation.

A key to the identification of this level is contained in Newell's (1982) landmark article entitled "The Knowledge Level." The article covers the capabilities of a cognitive system and how its contents are independent of the

This article discusses frameworks for studying expertise at the knowledge level and knowledge-use level. It reviews existing approaches such as inference structures, the distinction between deep and surface knowledge, problem-solving methods, and generic tasks. A new synthesis is put forward in the form of a componential framework that stresses modularity and an analysis of the pragmatic constraints on the task. The analysis of a rule from an existing expert system (the Dipmeter Advisor) is used to illustrate the framework.

implementation: "Knowledge is to be characterized entirely functionally, in terms of what it does, not structurally, in terms of physical objects with particular properties and relations" (p. 105).

The adoption of a knowledge-level perspective is clearly a

step in the right direction because it focuses the analysis of expertise on issues such as what the abstract task features are, what knowledge the task requires, and what kind of model the expert makes of the domain. It also helps to explicitly focus on how to go from the knowledge level to the symbol or program level. I call this in-between level the *knowledge-use level*. At the knowledge-use level, we focus on issues such as how the overall task will be decomposed into manageable subtasks, what ordering will be imposed on the tasks, what kind of access to knowledge will be needed (and, consequently, what representations must be chosen), and how pragmatic constraints such as limitations of time and space or limited observability can be overcome. It is only when these issues have been worked out that the program itself can be constructed. Given the current state of AI programming technology, this last step does not pose any major difficulty.

When we go through the expert system literature of the past decade, we see that several ideas have emerged that go in the direction of a knowledge level and a knowledge-use-level analysis of expertise: the concept of an inference structure, the distinction between deep and surface knowledge, the decomposition of expertise into problem-solving methods and domain knowledge filling the roles of these methods, and the notion of generic task. All these ideas have led to new architectures, new explanation capabilities, and new approaches to knowledge acquisition. They have led to a focus on the knowledge, as opposed to the information-processing aspects of a system, and a new breed of so-called second-generation expert systems.

The new architectures have attempted to be a foundation for systems that are less brittle, in the sense that they combine more principled knowledge of the domain with the heuristic knowledge that formed the bread and butter of first-generation expert systems.

The explanations given by first-generation

... one should go beyond the level of formalisms and programming constructs to understand and analyze expert systems.

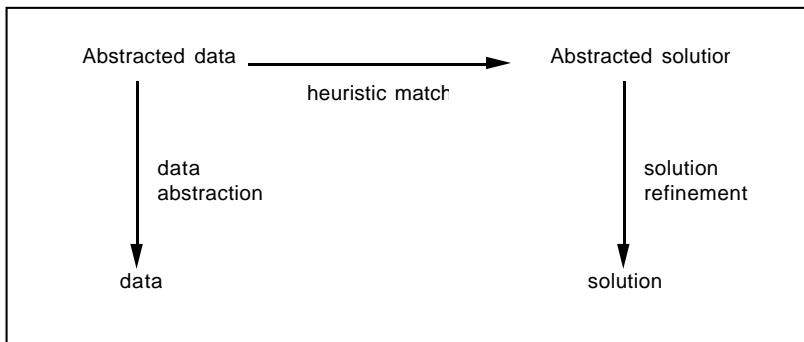


Figure 1. Heuristic classification inference structure.

expert systems (responses to why and how questions) were somewhat unsatisfactory because they were a simple replay of the rules that were used to arrive at a conclusion. This approach to explanation is computational. When more of the knowledge level and knowledge-use-level decisions that go into the system design are explicitly represented, it is possible to formulate much richer explanations and justifications.

However, it is probably in the area of knowledge acquisition that the new perspective has had the largest impact. Early on, expert system developers approached their expert with the question, What are your rules? In other words, they asked questions in terms of the computational formalism they happened to be using. We now have a whole arsenal of new questions to ask the expert, such as What kind of model do you make? What are the main features of the task? and Does the task decomposition follow an established pattern? This insight has led to new knowledge-acquisition tools that almost directly interact with the human problem solver (Marcus 1988a).

In the first part of this article, these developments are briefly reviewed. Then, a synthesis is presented in the form of a componential framework. This framework is a tool for studying expertise at the knowledge level and knowledge-use level. The implications of the framework are discussed in the final part of the article.

Existing Approaches

In this section, I survey four ideas for understanding expert systems at the knowledge level and knowledge-use level: inference structures, deep versus surface knowledge, problem-solving methods, and generic tasks. Each of these ideas focuses on a different aspect of expertise: the pattern of inference,

the domain model, the problem-solving method, and the task.

Inference Structures

One of the first steps beyond the program level was based on the notion of inference structure. An *inference structure* describes the pattern of inferences found in a particular expert system. Heuristic classification is the most widely studied class of inference structure (Clancey 1985; Van de Velde 1987). Heuristic classification assumes three major inference types: those making abstraction of the data, those matching the data with an (abstract) solution class, and those refining this solution to the actual solution. How these inferences are made (for example, by one rule or many or by other kinds of inference mechanisms) is not at issue, although the analysis includes a characterization of the kind of relation that is used to perform the inference. The different types of inferences are sequentially ordered, as shown in figure 1. Consider the following Mycin rule:

IF
 1. A complete blood count is available
 2. The white blood count is less than 2500
 THEN
 The following bacteria might be causing
 the infection:
 E.coli (.75)
 Pseudomonas-aeruginosa (.5)
 Klebsiella-pneumoniae (.5)

This rule can be analyzed in terms of heuristic classification, as shown in figure 2 (adapted from Clancey 1985, p. 296). Figure 2 not only displays the inference structure but also depicts what domain relation is used for each type of inference step. For example, abstraction makes use of qualitative, definitional, and generalization relations. Heuristic match is based on a causal relation. Refinement is based on a subtype relation. Other applications might use different domain relations with the same roles or might decompose some steps (such as heuristic match) into several intermediate steps.

There are three advantages to an analysis in terms of inference structure. First, it shows that there is a lot of structure underlying the rules in expert systems, including many hidden assumptions. Making this underlying structure explicit is important for maintaining the rule base, explaining it to others, or querying the expert during knowledge acquisition.

Second, the categorization of inference structures shows relationships between rules that go beyond the syntactically based rule

The distinction between deep and surface knowledge . . . is at the knowledge level and not the implementation level.

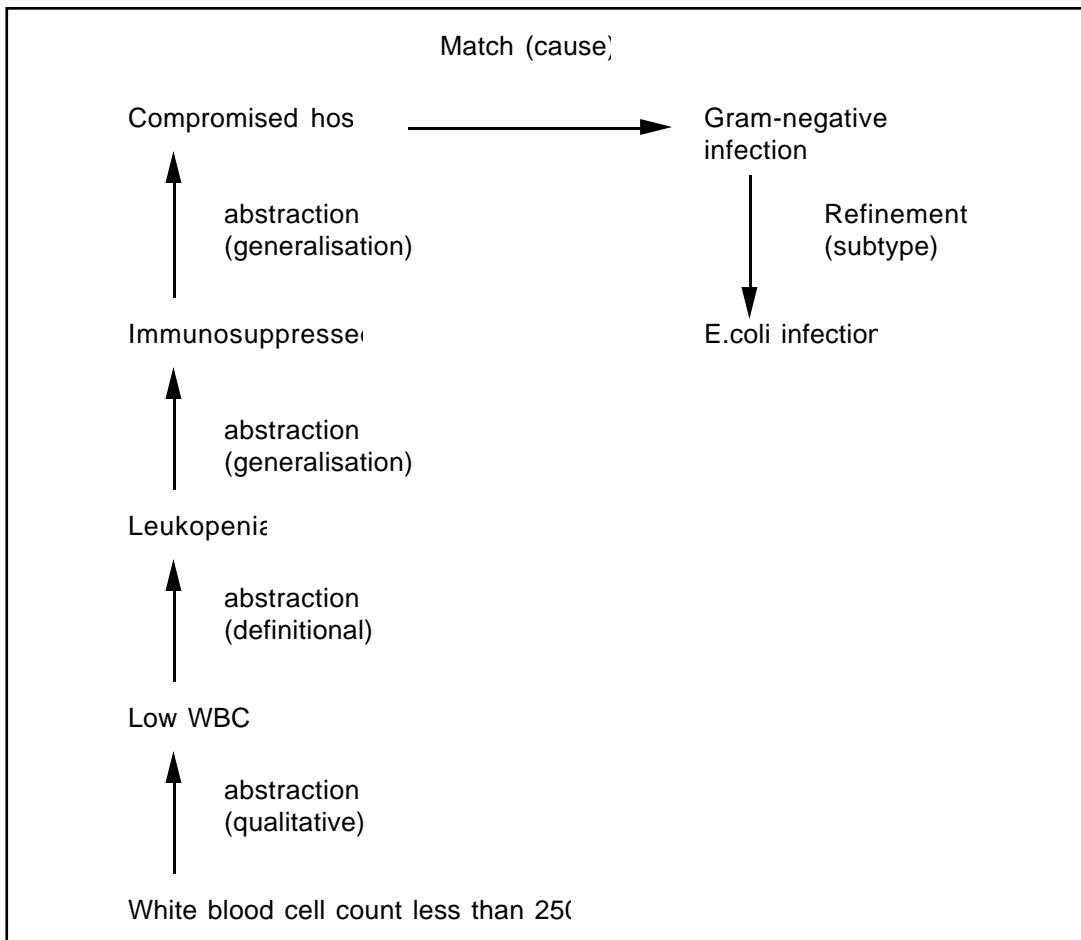


Figure 2. Inference structure underlying MYCIN rule.

generalizations, such as those used in Tereisias (Davis 1982). In particular, it makes abstraction from specific domain-dependent details and focuses on the role of domain knowledge in the overall inference pattern. Again, this guideline is important for knowledge acquisition or maintenance.

Third, it shows the similarities between expert systems constructed for apparently widely diverse domains and tasks. Clancey (1985) analyzed the inference structure underlying a series of expert systems, showing how the heuristic classification framework applies to each of them. He thus illustrated that the inference structure is one of the ways that can be used to analyze a task beforehand and detect its underlying solution method.

Although the concept of heuristic classification has been instrumental in focusing expert system research on a knowledge-use level, it is not yet completely satisfactory, mainly because the classification inference structure appears to be so broad that almost any expert system can be analyzed in terms of it. This sit-

uation is, of course, both good and bad. It is good because it shows that the framework is a significant empirical generalization. It is bad because the framework omits important distinctions.

For example, the first clause in the Mycin rule discussed earlier (a complete blood count is available) is a *screening clause* (Clancey 1982), a clause that prevents unnecessary questions from being asked, thus controlling the data gathering. The notion of a screening clause does not form part of the heuristic classification inference structure, and it is not clear how it should be incorporated. Given that careful data gathering is one of the important features of expertise, this oversight seems to be major, particularly because screening clauses are a reoccurring inference pattern that can help achieve careful data gathering.

As another illustration, a clear distinction can be made between different classification methods. Top-down refinement and weighted evidence combination are only two examples.

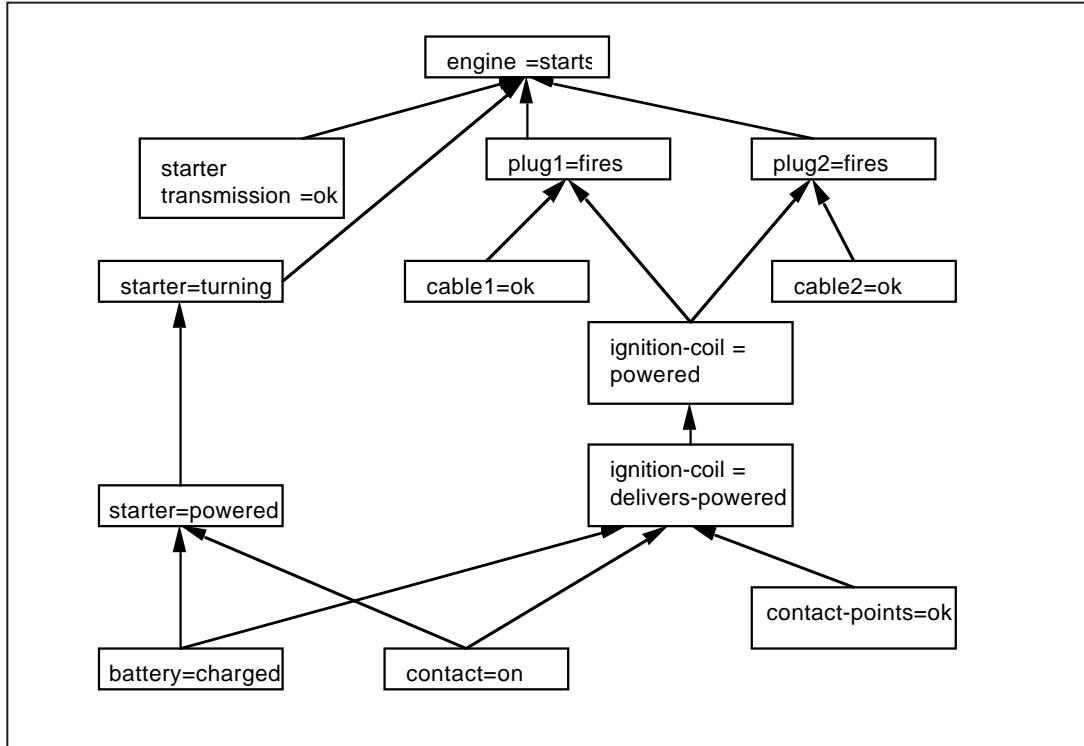


Figure 3. Example of deep model in the form of a causal network.

Top-down refinement performs classification by starting with the most general class and refining this class to a more specific one. *Weighted evidence combination* determines all the features of each class, combines the weights, and then takes the class with the highest value as the best matching class. This distinction between different classification methods disappears in the heuristic classification inference structure. Therefore, we cannot explain why there is more than one conclusion in the rule (*E.coli*, *Pseudomonas-aeruginosa*, and *Klebsiella-pneumoniae*), what the function of the weights is, why these specific weights are appropriate, or how these weights interact with the weights given in the other rules.

Finally, heuristic classification only describes the pattern of inference. It does not say anything about the kind of modeling that takes place by the expert. On the contrary, it blurs the distinction between different domain relations (for example, causal, structural) by categorizing them according to their contribution to the abstract, match, or refine phase.

Deep versus Surface Knowledge

Even before heuristic classification and other types of inference structure were widely discussed, another type of analysis was proposed

based on a distinction between deep and surface knowledge (Hart 1984; Steels 1984). This distinction focused not on the pattern of inference but on the domain models underlying expertise. *Deep knowledge* makes explicit the models of the domain and the inference calculus that operates over these models. A typical example of a domain model for diagnosis is a causal model linking properties of components through cause-effect relations. An inference calculus operating over this model could take the form of a set of axioms that prescribe valid inferences over the causal network, for example, inferences showing that a certain cause possibly explains a specific set of symptoms. *Surface knowledge* contains selected portions of the deep knowledge, in particular, those portions that are relevant for the class of problems that is likely to be encountered. It also contains additional heuristics and optimizations, for example, shortcuts in the search space or decisions based on the most probable situation.

Figure 3 contains an example of a deep causal model (Steels and Van de Velde 1985). A rule that represents a useful shortcut based on this model is as follows:

```

IF
  Plug-1 does not fire
  Contact-points are okay
  
```

THEN

Check whether cable-1 is okay

The rule focuses on one malfunction, makes one observation, and directly suggests a test. Many such rules could be extracted from the same network.

The distinction between deep and surface knowledge has nothing to do with the formalism that is chosen to implement it. Deep knowledge can be implemented with rules, surface knowledge with frame-based representations, or vice versa. The distinction is at the knowledge level and not the implementation level.

Traditional expert systems only code the surface knowledge. This limitation is clearly seen in the Mycin rule quoted earlier, which contained just enough information to have the effect of the required inference (low white blood cell count raises evidence for E.coli infection) but none of the underlying domain knowledge, such as the causal relation between compromised host and gram-negative infection, or the organism-type hierarchy supporting refinement to E.coli infection. The fact that only surface knowledge is represented explains why traditional expert systems are efficient and effective in problem solving. However, because first-generation expert systems only code surface knowledge, they have important drawbacks, such as brittleness, weak explanation, and unclear boundaries during knowledge acquisition. Deep expert systems are intended to overcome these limitations.

Deep expert systems contain two components: One implements the deep knowledge of the domain, that is, the domain model and an inference calculus operating over it, and the other implements the surface knowledge. The second component typically has the form of a collection of (heuristic) rules.

Two types of problem solving are possible. First, reasoning can take place using the rules from the surface component. This approach is similar to a traditional expert system. Second, reasoning can fall back on the deep knowledge component if rules are missing to cover a specific case or if it is difficult to explicitly represent all possible rules. Some deep expert systems also have a learning component, so that problem solving using deep knowledge can also be turned into more efficient surface knowledge (Steels and Van de Velde 1985). There are several motivations behind this design:

Making deep knowledge explicitly available allows a system to fall back on underlying models and (weaker) problem-solving methods if no explicit surface knowledge is avail-



The importance of the deep expert system movement is that it introduced a focus on the underlying domain models.

able to efficiently solve a problem. Although deep problem solving is, in principle, less efficient, it typically covers a wider class of problems. Deep expert systems are, therefore, less brittle.

Because deep knowledge is supposed to be less biased toward use, it is hoped that knowledge becomes, to some extent, reusable. For example, a causal network is used in both design and diagnosis.

Deep knowledge can be the source of better explanations because the domain knowledge that went into an inference step can also be reported. An example of this approach is Explain (Swartout 1981).

Making deep knowledge explicit is assumed to be a step forward, toward more methodical knowledge acquisition, because it is more systematic and closer to the theoretical knowledge of a domain that is typically analytic and explicit. For example, if we know all possible classes for a classification task in advance, we can ask the expert why there are no rules for a particular class or point out that there is insufficient discrimination between two classes. Moreover, the deep knowledge provides the background for justifying the inferences that are part of the surface knowledge.

On occasion, surface knowledge has been equated with associations and deep knowledge with more principled knowledge of the domain. From this perspective, first-generation expert systems have been criticized because they supposedly only contain associational knowledge (DeKleer 1984). This criticism of first-generation systems is not valid and is not intended here, in part because an analysis of expert systems such as Mycin reveals that they also construct and use models (Clancey 1988) and in part because the knowledge in first-generation systems can seldom be characterized as purely associational. Consider, for example, the following Mycin rule (VanMelle 1981, p. 25):

IF

It is not known whether there are factors
that interfere with the patient's
normal bleeding

THEN

It is definite that there are not factors
that interfere with the patient's
normal bleeding

This rule implements a knowledge-based approach to incompleteness (as opposed to a weak domain-independent approach such as circumscription or default logic). It is based on the principle that if a particular event is so important that it would have been noticed if it occurred, then ignorance about its occurrence indicates that it did not happen. Clearly, this association is not simply empirical; it is also an ingenious piece of heuristic knowledge.

Another example comes from R1, an expert system for configuring computer systems (McDermott 1982):

Verify-SBI-and-MB-device-adequacy-3

IF

The most current active context is verifying SBI and massbus device adequacy and there are more than two memory controllers on the order

THEN

Mark the extra controllers as unsupported (that is, not to be configured) and make a note to the salesperson that only two memory controllers are permitted per system

This rule notices a constraint violation (there should be no more than two memory controllers) and proposes a fix of this constraint by eliminating excessive controllers and making a note to the salesperson. The constraint fix uses strong domain-dependent knowledge as opposed to a weak domain-independent constraint-relaxation algorithm. To classify this association as empirical completely misses the point of this rule.

These two examples indicate that the common characterization of early expert systems as just a collection of shallow if-then rules is inappropriate. It is probably better to think of them as compiling lots of deep knowledge into a form that can immediately and efficiently be used. The point about the distinction between deep and surface is that surface systems contain only the compiled form, and everything that is not needed for efficient performance is omitted.

Although the distinction between deep and surface is clearly another step in the right direction, and many expert systems have recently been built that work along these lines, it also raises other questions. First, early work on deep expert systems seemed to imply that there was one underlying deep model that was more principled and more abstract than the surface knowledge. Indeed, further work along this line gave rise to the so-called *model-based approach*, which postulates that expert system building should start with an

encoding of the first principles of a domain as deep knowledge (for example, qualitative or quantitative models of the behavior of a device to be diagnosed) (Davis 1984; DeKleer 1984). However, for most tasks, it is possible to think of a variety of models, each focusing on different aspects of the problem domain. For example, for diagnosis, we might have a structural model describing part-whole relationships between components and subsystems, a causal model representing the cause-effect relationships between properties of components, a geometric model representing the spatial relations between the components, a functional or behavioral model representing how the function of the whole follows from the functioning of the parts, a fault model representing possible faults and components for each function that might be responsible for the fault, and an associational model relating observed properties with states of the system. Which of these models would be the one used for deep knowledge? For some researchers, this model would be the causal one. For others, it would be the functional one. The associational model tends to be associated with surface knowledge. However, clearly, all these models are useful and, in some sense, at the same level. Simmons (1988) describes, for example, a system that translates causal models into associational models and shows how they have complementary utility in problem solving. Therefore, the distinction between deep and surface cannot be in terms of the type of model.

Another point is that principled models are not effective in problem solving except when unrealistic assumptions are made (for example, all observations are assumed to be correct and have no associated cost, combinatorial explosions are ignored, it is assumed that an exact domain theory is known.). To avoid ineffective problem solving, expert system builders exploring deep models use knowledge that is less principled than purely theoretical domain knowledge but not as sketchy and as optimized as pure associations. Hart (1984) talks in this context about second-level knowledge, and Chandrasekaran and Mittal (1983) seek a level "between the extremes of a data base of patterns on one hand and representations of deep knowledge (in whatever form) on the other." However, the introduction of more levels leaves the question about what kind of model is most appropriate and how the in-between models are related to the first-principled theoretical knowledge on the one hand and the performance system on the other.

The importance of the deep expert system movement is that it introduced a focus on the underlying domain models. However, some questions remain unanswered, particularly those associated with what type of model is appropriate in certain circumstances or how effective problem-solving performance is obtained with deep models.

Problem-Solving Methods

Around 1985, McDermott (1988) and his collaborators began developing a series of knowledge-acquisition tools that emphasized the problem-solving method (and not the inference pattern or the domain model) as the central key in understanding and building an application. A *problem-solving method* is a knowledge-use-level characterization of how a problem might be solved. For example, diagnosis might be done using the cover-and-differentiate method: First, find possible explanations covering most symptoms, and then, differentiate between the remaining explanations (Eshelman 1988). Construction might be done using the propose-and-revise method: First, propose a partial solution, then revise this solution by resolving violated constraints (Marcus 1988). Each problem-solving method contains certain roles that need to be filled by domain models. For example, the cover-and-differentiate method requires knowing relationships between explanations and the symptoms they cover and knowing additional observations or tests that will further differentiate. The propose-and-revise method implies two roles: one to be filled by knowledge that is proposing solutions, another to be filled by knowledge that revises these solutions.

The advantages of focusing on the problem-solving method are as follows: The roles that need to be filled by a problem-solving method prescribe what domain knowledge is expected from the expert. The domain knowledge is, therefore, no longer something that is acquired and represented independent of how it will be used in concrete problem solving, as is the case with deep expert systems. However, it is still explicitly and separately represented, giving the advantages of maintainability and systematicness in knowledge acquisition. The knowledge-acquisition tool Mole (Eshelman 1988) is an example. The domain knowledge of Mole is represented as a fault model that explicitly represents relations between symptoms (such as excessive engine noise, lack of power) and faults that act as explanations (such as worn cylinders, worn crankshaft bearings). The relations are further qualified

by states of the system that can increase or decrease their possibility or plausibility (for example, if the air conditioner is not turned on, the fault "worn crankshaft bearings" for the symptom "excessive engine noise" is ruled out). This domain model is definitely not a model based on first principles but is not that different from what we find in deep expert systems. What is different is that there is a clear prior idea of how this domain model will be used in the final performance system and that only those parts of the model that are needed in driving Mole's problem-solving method are acquired.

Focusing on the problem-solving method leads to a streamlined methodology for doing knowledge acquisition and might even lead to a set of tools that can be directly used by experts, provided the problem-solving method has been clearly identified in advance. Several such tools have been built and successfully used in a variety of real-world applications (for example, Salt [Marcus 1988b] and Sizzle [Offutt 1988]).

The problem-solving method can be a major source for the kind of explanations that go beyond a pure recall of the steps that were used. For example, when the reason for a query is asked, it can be in terms of which phase of the problem-solving method is being executed and what domain knowledge plays a role in this step.

The approach of focusing on the problem-solving method appears to be an important step forward, judging from the many performance systems that have recently been generated using this approach. However, there are some questions here.

One of the major problems in knowledge acquisition is knowing when to stop; that is, When is the performance system sufficiently complete to be put into actual use? The idea of deep expert systems is to delineate—at least in principle—the conditions under which a system would be complete (for example, all possible observations and all possible causes are represented in the causal network). The surface system selects those parts that are deemed important in practical problem solving. However, if knowledge acquisition is completely driven by the problem-solving method, we no longer have any clear sense of how much additional knowledge is needed.

A second problem is that the problem-solving methods proposed to date handle a complete problem and, therefore, make a lot of assumptions about how each subtask will be handled. For example, a propose-test-refine method might perform some classification during the propose phase. This classification

is typically built into a specific method. This approach leads to a proliferation of methods and points to the need for a more modular decomposition of expertise than assumed to date. This feature is a major part of the synthesis that I propose in the second part of this article.

Generic Tasks

Another line of research focuses on task features and, thus, directly addresses the problem of developing an engineering methodology to build expert systems based on a task analysis. The analysis of expertise in terms of tasks (and, particularly, the ordering of the tasks, that is, the control structure imposed on the task structure) used to be a completely domain-dependent matter, enforcing the view that expert system development does not show generalizations across domains of expertise and, therefore, is doomed to start new, in an ad hoc fashion, for every new application being tackled. However, several researchers have observed that tasks fall into major classes. These tasks are called *generic tasks* (Chandrasekaran 1983). In specific fields of expertise, tasks are instances of these generic tasks. Typical generic tasks are classification, interpretation, diagnosis, and construction (including planning and design). The idea of generic tasks is not that interesting in itself until we realize that the way in which generic tasks are executed shows many similarities across application domains: In the diagnosis of circuits, cars, power plants, or diseases, significant elements are in common, specifically, the same problem-solving methods and the same type of domain models.

Same Problem-Solving Methods. The same decomposition of the task into simpler subtasks is found across domains. For example, construction tasks are often decomposed into the following subtasks: (1) generate a partial solution, possibly by retrieving it from a list of earlier solved cases; (2) test this solution to see whether it satisfies the constraints; and (3) adapt the partial solution so that the constraints are no longer violated. It is significant that this decomposition was found independently by Marcus, Stout, and McDermott (1987) for the domain of elevator configuration; Simmons (1988) in the domain of geologic plan formulation; and Mittal, Dym, and Morjaria (1986) in the domain of equipment design. Similarly, common decompositions are found when other generic tasks are studied. For example, interpretation often starts with an attempt to restrict the context.

Diagnosis often starts by following up on associations between symptoms and malfunctions.

Same Type of Domain Models. The same type of domain models and, therefore, the same sort of inferences operating over these domain models are found when tasks of the same generic nature are studied. For example, classification typically makes use of a domain model in the form of a catalog of hierarchically organized prototypes. Such a domain model supports generic subtasks of classification such as context restriction and selection of the class from a list of remaining classes. Similarly, diagnosis typically uses causal models and subtasks such as "try to find a deeper cause" or "try to eliminate part of the network by making an appropriate observation." Construction typically uses constraints as domain models and uses subtasks such as "adapt the partial solution by varying one of the elements that play a role in a violated constraint."

The focus on tasks and task decomposition is important. From a theoretical point of view, it gives us a way to build a theory of expertise that makes significant empirical generalizations. This theory should identify a set of generic tasks and list, for each one, what kind of problem-solving methods and what kind of domain models are to be expected. To develop such a theory is a matter of empirical investigation of expert knowledge.

Once such a theory exists, we have strong models for interpreting knowledge-acquisition data (that is, protocols, interviews from experts, and so on). For example, if a task is a design task, we could interpret knowledge-acquisition data from the perspective that the expert's decomposition of the task is generate-test-adapt. This approach is pursued by several researchers concerned with knowledge acquisition (Bennett 1983; Breuker and Wielinga 1984).

Generic tasks and their associated solution methods could be implemented at a generic level and made available as software modules that are to be instantiated for specific applications. This approach was followed by researchers such as Chandrasekaran and his collaborators, who developed such modules for generic tasks such as diagnosis or classification (Bylander and Mittal 1986).

Although the generic task approach has been received with great enthusiasm, it raises other issues. It is generally agreed that diagnosis and classification are common generic tasks; however, there is less agreement with other tasks. For example, is interpretation a

generic task, or is it a special case of classification? These questions point to a need to better define the criteria for classifying a task as generic. Should these criteria be in terms of the problem to be solved? Should they be in terms of the problem-solving methods or domain models used?

Tasks of the same generic nature (for example, diagnosis) can be done in many different ways. For example, diagnosis can be done by selecting from a list of possible fault models the one that is appropriate in a given situation or selecting in a causal network the cause that best explains the observed symptoms. However, diagnosis can also be done by constructing a model of the system and then of its deviations in functionality from the normal model. This diversity indicates that diagnosis is not the right way (or at least not a sufficient way) to categorize and select problem-solving methods and domain models.

Moreover, as in the case of the problem-solving methods proposed by McDermott et al., there seems to be a problem of grain size or something similar. The existing task-specific architectures solve the complete task and, therefore, make concrete decisions about the methods and representations for each of the subtasks. A task of the same generic nature might require a slightly different approach for one of the subtasks. Rather than using a completely different architecture then, there should be more modularity in the task-specific architectures so that new architectures can be dynamically configured. This point was also recognized by researchers developing task-specific architectures (Brown and Chandrasekaran 1989).

Summary

In this section, we studied a variety of ways to describe aspects of expertise and expert systems at the level of knowledge and knowledge use. Each of these characterizations has led to specific theories on how knowledge acquisition should be done or what the architecture of the resulting expert system should look like. They all focused on different aspects of an application: the pattern of inference, the domain model, the problem-solving method, and the task.

Obviously, there are many interconnections between these viewpoints. For example, heuristic classification or other abstract inference structures can be seen as examples of problem-solving methods. In this view, heuristic classification implies roles for abstraction, matching, and refinement. Similarly, the focus on problem-solving methods for knowl-

edge acquisition and system construction implies explicit representations of the domain models, similar to the way deep expert systems explicitly represent the domain models, albeit only those parts that are really needed by the problem-solving method are made explicit. Finally, the identification of generic tasks is a way to start systematically cataloging domain models and problem-solving methods, although the task classification used to date might not be sufficiently fine grained. Given all these interdependencies, it should be possible to come to a synthesis that works out the different components of expertise in a top-down manner. I turn to such a synthesis in the next section.

. . . problem solving centers around the idea of modeling.

The Componential Framework

This section outlines a framework for knowledge and knowledge-use analysis that combines the different ideas briefly reviewed in the previous section. Apart from combining the ideas (and, with hope, their strengths), the framework also imposes a lot more modularity on the different components of expertise and emphasizes a consideration of the task's pragmatic constraints.

Task Analysis

The componential framework assumes first of all that there is a detailed analysis of the task. A real-world task is seldom concerned with handling a single, simple-to-describe problem. Instead, a conglomerate of mutually dependent tasks has to be dealt with, and tasks have internal structure: They can be decomposed into subtasks with input-output relations between them. Each task and subtask is analyzed from a conceptual and a pragmatic point of view.

From a conceptual viewpoint, a task is characterized in terms of the problem that needs to be solved, that is, diagnosis, interpretation, design, planning, and so on. This characterization is based on properties of the input, the expected output, and the nature of the operation taking place to map input to output. It runs along the lines of the generic tasks discussed earlier. For example, if the input consist of observed symptoms, and the output is an explanation of how the symptoms came about, then we characterize the task as diagnosis. If the input consist of observed data, and the output is a categorization of these data, then we characterize the task as interpretation. Design starts from specifications as input and produces an object that conforms to these specifications as output.

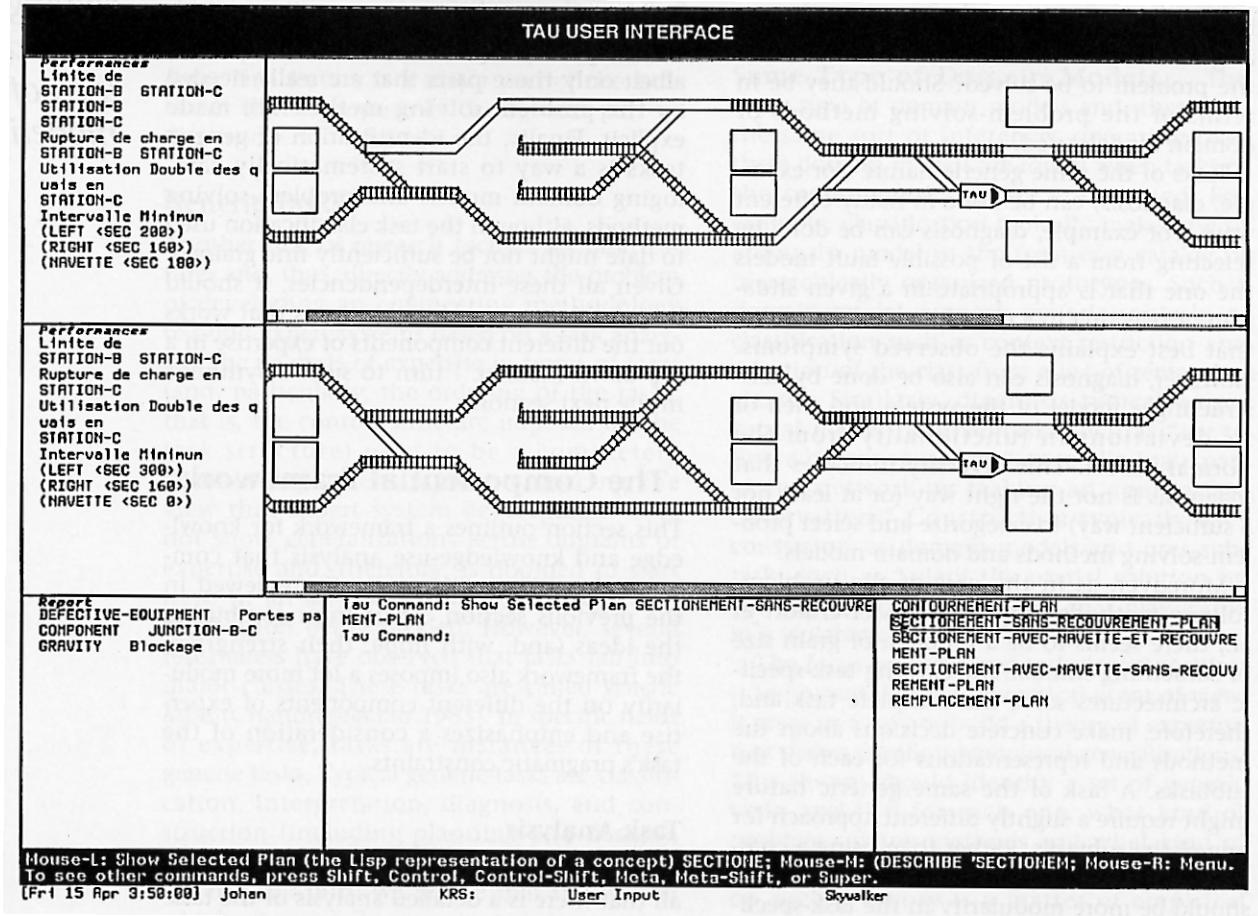


Figure 4. A topological model of a subway network.

The pragmatic viewpoint focuses on the constraints in the task that result from the environment in which the system will operate or from the (epistemological) limitations that humans (but also computers) have.

The first limitation is in time and space. Decisions have to be made in finite time (sometimes even in a small time interval), and available memory for storing structure is always finite. Therefore, models that require the exploration of large search spaces or that take too much time to compute cannot be used.

The second limitation is in observation. Observation must be done through sensors or human intermediaries that directly interact with the real world and, thus, are imprecise to some degree. Observations also differ in the amount of effort needed to make them. Therefore, the data necessary to use a model might not be available or might not have the required degree of precision.

The third limitation is in theory formation. Models must be inductively derived using

real-world interaction or communication with other humans. This approach often limits the models in their accuracy and scope of prediction.

From these epistemological limitations follow pragmatic constraints, such as the need to avoid search; work with weak inference rules or weakly defined concepts (in other words, concepts not defined in terms of necessary and sufficient conditions but typical features or exemplars); handle incomplete, inconsistent, and uncertain data; and handle explosions of information. The presence of these problems sets domains tackled by expert systems apart from domains for which other techniques, such as algorithms or numeric and logical modeling, are appropriate. Thus, if there is a complete quantitative theory, and all the data necessary to make effective computations based on the theory can easily be obtained, there is no point in building an expert system. This expert system is only relevant when the

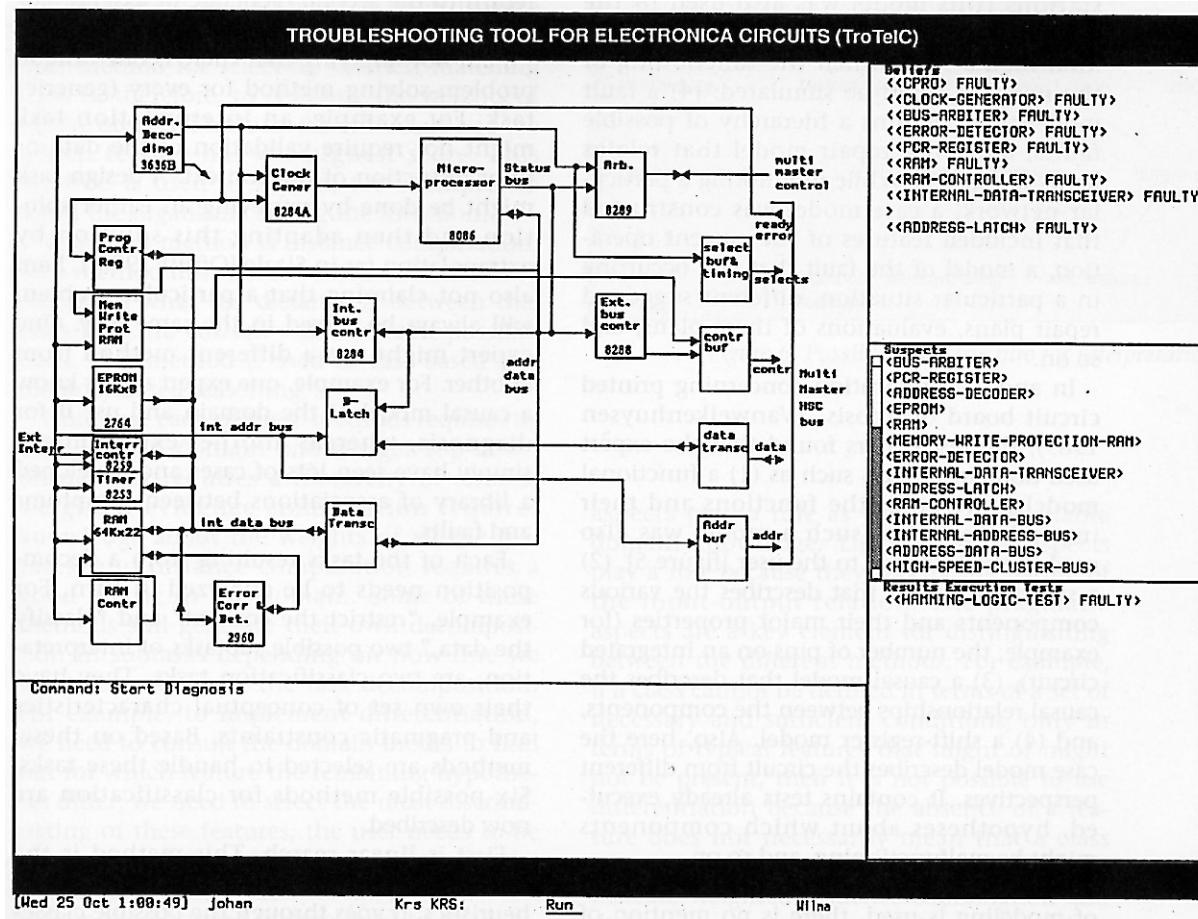


Figure 5. Functional model of a printed circuit board.

theory is not too certain; some data are missing or might be difficult to get; there are no effective computations, for example, because the equations cannot be solved, and so on. This notion of pragmatic constraints is related to the insight that an intelligent system will necessarily have bounded rationality (Simon 1982).

I believe that it is of particular importance to explicitly characterize the various pragmatic constraints of a task. The following are typical examples of such constraints: (1) The symptoms used as input to a diagnostic task might have an associated cost, and part of the problem might be to minimize the number and cost of observations. (2) The data to be interpreted might show errors or might be incomplete. (3) The categories into which the data must be classified might not be strict; that is, they might only be definable in terms of typical features and not in terms of necessary and sufficient conditions. (4) The specifications given for a design task might be

incomplete or inconsistent, or the number of possible combinations might be so large that exhaustive search is impossible.

Models

The perspective of deep expert systems has made us aware that problem solving centers around the idea of modeling. The problem solver constructs one or more models of the problem-solving situation (something I call the *case model*) and uses various more abstract *domain models* to expand the case model by inference or data gathering. The next step is, therefore, to find out what kind of case models are being constructed for each task and what domain knowledge is available to do so.

For example, in one application involving the monitoring of subway traffic (Steels and Vanwelkenhuysen, 1990), researchers at VUB AI Lab found that the following domain models were used: (1) a structural model containing the stations, the tracks, and so on; (2)

a topological model detailing the layout of stations (this model was also used in the interface to the system [figure 4]); (3) a behavioral model with which the functioning of the network could be simulated; (4) a fault model that contains a hierarchy of possible faults; and (5) a repair model that relates faults to repairs. While monitoring a particular network, a case model was constructed that included features of the current operation, a model of the fault that was occurring in a particular situation, different suggested repair plans, evaluations of these plans, and so on.

In another application concerning printed circuit board diagnosis (Vanwelkenhuysen 1989), the researchers found that the expert used domain models such as (1) a functional model that details the functions and their interrelationships (such a model was also used in the interface to the user [figure 5]), (2) a structural model that describes the various components and their major properties (for example, the number of pins on an integrated circuit), (3) a causal model that describes the causal relationships between the components, and (4) a shift-register model. Also, here the case model describes the circuit from different perspectives. It contains tests already executed, hypotheses about which components might be malfunctioning, and so on.

Note that during the analysis of what kind of modeling is used, there is no mention of any implementational constructs. Some of these models (such as the repair model that relates faults to repair plans) might be in the form of rules; others might be in the form of hierarchies or networks.

Problem-Solving Methods

The problem-solving methods are responsible for applying domain knowledge to a task. In general, they perform two functions: divide a task into a number of subtasks or directly solve a subtask. In either case, they can consult domain models; create or change intermediary problem-solving structures; perform actions to gather more data, for example, by querying the user or performing a measurement; and expand the case model by adding or changing facts.

Here are two examples of problem-solving methods that decompose a task into subtasks. For interpretation, validate the data, restrict the context, classify the data, and deduce additional features based on class membership. For design, test specifications for incompleteness or inconsistency, generate or extend partial solution, test adequacy of the solution,

refine by resolving violated constraints, and return to the second (generate or extend partial solution) until the design is complete.

I am not implying that there is one unique problem-solving method for every (generic) task. For example, an interpretation task might not require validation of the data or prior restriction of the context. A design task might be done by retrieving an earlier solution and then adapting this solution by extrapolation (as in Sizzle [Offutt 1988]). I am also not claiming that a particular problem will always be solved in the same way. One expert might use a different method from another. For example, one expert might know a causal model of the domain and use it for diagnosis, whereas another expert might simply have seen lots of cases and developed a library of associations between symptoms and faults.

Each of the tasks resulting from a decomposition needs to be analyzed in turn. For example, "restrict the context" and "classify the data," two possible subtasks of interpretation, are two classification tasks. They have their own set of conceptual characteristics and pragmatic constraints. Based on these, methods are selected to handle these tasks. Six possible methods for classification are now described.

First is linear search. This method is the simplest, requiring no domain-dependent heuristics: It goes through the possible classes each time, comparing all the features of a class with the features of a case. The class where all features match is selected.

The second method is top-down refinement. This method assumes that the classes have been organized in a hierarchy, with the general classes on top. It systematically searches from the top, each time establishing the most appropriate class at a particular level.

Third is association. This method follows up on the association between a feature and its class. It assumes a domain model that indicates what features are strongly associated with a class.

The fourth method is differentiation. If only a limited list of classes remains, the classes could be compared and their differentiating features computed. These differentiating features are good candidates for additional data gathering, so that discrimination between the classes can take place.

Fifth is weighted evidence combination. Each feature (or combination of features) associated with a class might be more or less essential to the class. For example, the color of a dog might be less essential than the shape (although we do not expect orange

dogs). Based on this information, we could calculate how much evidence each feature contributes to the recognition of the class. One method for selecting the best matching class is, therefore, to combine the weights of all the observed features for each class and see which feature has the highest score. This method is used in Mycin, for example, to select the organism causing the infection.

The sixth method is distance computation. This method introduces a distance metric and computes what the distance is between this class and the current case for each possible class. This method is used in case-based and memory-based reasoning.

Note that each of these methods requires its own type of domain knowledge. Top-down refinement requires a hierarchy of classes, weighted evidence combination requires knowledge about the weights of all the features, and distance computation requires a good metric in the domain. Some of these methods still generate their own decomposition in subtasks depending on how fine we want to modularize the task decomposition. For example, to implement differentiation, we need to consult the domain model to find out for which feature the remaining hypotheses differ; we need to select the most discriminating of these features; the user needs to be asked about one of these features; and if the user knows the answer, those hypotheses that violate the observed feature need to be eliminated. The list of current hypotheses (usually called the *differential*) and the list of differentiating features are typical examples of intermediate problem-solving structures.

The *task structure* is the backbone of the whole analysis. It describes the task decomposition at the different levels. An example of part of the task structure for an interpretation task is contained in figure 6. Not all these tasks are necessarily executed. Which tasks are executed, as well as the order in which they are executed, is determined at run time, as specified in the task decomposition method.

Relating Task Features to Solutions

I started this article by posing a question about how we could select expert system solutions based on task characteristics. The componential framework is designed to address this question because it allows us to make a mapping from conceptual features, pragmatic constraints of a task, and available knowledge to components such as problem-solving methods, domain models, and task structures.

In selecting the appropriate problem-solving method, both conceptual and pragmatic

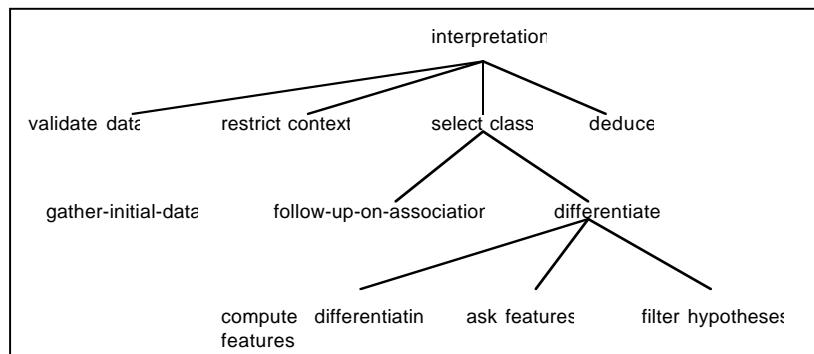


Figure 6. Possible task structure for interpretation task.

aspects play a role as well as the available domain knowledge. The conceptual aspects play a role because they specify the nature of the input-output relation. The pragmatic aspects are a key element for distinguishing between the different methods: For example, if a class cannot be defined in terms of a set of necessary and sufficient conditions, only in terms of typical features that might or might not be present, then it is not possible to use differentiation because the absence of a feature does not necessarily mean that a class can be eliminated from consideration. Instead, weighted evidence combination and distance computation are the appropriate methods because they allow recognition of a class even if some of the features are not present. However, if the cost of observation plays an important role, then weighted evidence combination is less appropriate because it requires asking for a significant number of features to make a balanced decision. Here, differentiation is more effective because, often, one observation, if selected carefully, can significantly reduce the number of remaining hypotheses.

The selection and acquisition of the domain model are related to the selection of the problem-solving method, but there is usually still a choice in terms of the depth at which knowledge is represented. There are two viewpoints. The first viewpoint is associated with the problem-solving-method school, the second with the deep expert system school.

A problem-solving method has a set of roles that need to be filled by specific domain knowledge. For example, differentiation requires that we know the differentiating features between classes. We can choose to represent only those parts of the domain knowledge that are necessary to drive the problem-solving method. The differentiation

method is apparent in Mole (Eshelman 1988), a diagnostic system. Mole acquires just enough knowledge about the domain to find an initial set of hypotheses that cover the symptoms and differentiate between the remaining hypotheses. Mole has no representation of the causal relationships between symptoms and explanations, the functioning of the underlying device, or any other domain theory. Note that Mole still has an explicit representation of the domain model in the form of a network of symptoms and explaining causes and, thus, differs from first-generation expert systems where this model would be embedded in rules.

Alternatively, we can look at domain models from the viewpoint of domain theories. Domain theories are the (more) principled theories underlying problem solving in a domain and are sought by deep expert systems. A problem-solving method selects certain parts of these theories and, possibly, requires additional heuristic knowledge. An example is the system discussed in Steels and Van de Velde (1985); as with Mole, this system also uses (implicitly) a cover-and-differentiate method, but the association between symptoms and covering causes, as well as further observations that should be done for differentiation, is derived from the causal model. The causal model is represented explicitly and exhaustively. The expert is queried for a causal model, not for knowledge to implement the cover-and-differentiate method. Similarly, More (Kahn 1988) contains a domain theory in the form of an event model, which is accessed and used in a selective way by More's problem-solving methods.

These two viewpoints are, of course, complementary and which approach is followed depends on what is available. If a complete and explicit domain theory is available or derivable, there is no reason why it should not be used. In this case, the problem-solving method implies a particular way of accessing this domain theory. The domain theory, moreover, plays an important role in validating the domain model. However, if a domain theory is not available or cannot be used because of pragmatic constraints in the domain, then it might be better to acquire and represent only those parts that are really relevant to drive the problem solving. Thus, we find a synthesis between the problem-solving-method approach and the deep expert system approach.

The previous discussion concerned the conceptual aspects of domain knowledge: features for doing classification, causal relations

between symptoms and states of devices, temporal relations between events, and so on. However, there are also pragmatic aspects of domain knowledge, that is, aspects that explicitly address the pragmatic constraints in the domain, such as weights for weighted evidence combination or additional heuristics for overcoming missing observations. Here, we can adopt the two viewpoints previously introduced: The pragmatic aspects could be expressed independent of use in problem solving—in which case, the problem-solving method would have to select the appropriate parts—or they could only be expressed and acquired as needed for driving the problem solving.

Thus, More (Kahn 1988) uses Bayesian probability theory, in particular, the concept of expected likelihood and a theory on the relation between reliability and the significance of evidence to determine the diagnostic significance of an observed symptom. To make use of these theories, the system needs additional information about the domain, such as perceived reliability or conditions under which a cause is more likely to occur. More explicitly queries the expert for this additional information. This pragmatic domain knowledge is, therefore, not derived from a deeper pragmatic domain theory but is acquired and represented as needed by the problem-solving methods. Similarly, weighted evidence combination, possibly using the Buchanan and Shortliffe (1984) uncertainty calculus, requires knowing what the contribution is of a particular constellation of observations for the belief in a particular hypothesis. This contribution could possibly be derived through statistical reasoning if sufficient amounts of data are available (as demonstrated in Adams [1984]), or alternatively (as was the case for Mycin), the expert could be queried to fill in these weights. Note that the statistical derivation acts as a justification for the weights in the same way a conceptual domain theory can act as a way of validating the domain model.

In the work at the VUB AI Lab, researchers have found it useful to separately represent those parts of the domain model that are specifically put in to deal with the pragmatic constraints in the domain and the conceptual aspects of the domain model. These parts are the *heuristic annotations*. For example, the weights needed for using weighted evidence combination are implemented as heuristic annotations of the list of classes and their associated features. This approach makes it possible to separately develop the (conceptual) domain theory and its heuristic annotations.

Implications

Several implications of the framework were sketched in the previous section: for design and knowledge acquisition, the architecture of expert systems, and the analysis and understanding of existing expert systems to better maintain or justify them. I briefly cover the first two points, then close the article with a careful analysis of rules from existing expert systems.

Implications for Design and Knowledge Acquisition

The componential framework suggests a structured methodology for doing knowledge acquisition. The researchers at the VUB AI Lab have successfully tried this methodology in a number of industrial expert system projects. The process moves in a top-down systematic fashion, as follows: Starting from the complete application task, first characterize the task identifying the major features—the description, the input and output, the generic task, the pragmatic constraints, the case model, and the available domain knowledge.

Second, investigate the set of possible methods and associated domain models that either decompose the task into subtasks or directly solve the task. Some of the tasks can be done by existing software components or human intervention, which indicates that the framework can also be used to build integrated expert systems.

Third, given a decomposition, instantiate the tasks implied by the method. Reiterate for each of the subtasks from the first task (characterize the task) until tasks have been reached that are directly solved by the application of domain knowledge.

For example, the task of instructing a student might be analyzed as follows:

Student Instruction

Description: Instructing a student in algebra.

Input: Observations of student behavior.

Output: Tailored explanations of course material, exercises, reports on progress.

Generic task: Instruction.

Domain knowledge: Knowledge about algebra, detection of errors in student knowledge, tutoring.

Case model: Model of student.

Pragmatic problems: Large set of possible errors, partial unpredictability of student behavior, complexity of algebra.

Method: Divide and conquer. Decompose instruction into diagnosis, repair planning, and monitoring.

Subtasks: Diagnose student, plan repair, monitor.

Each of the resulting subtasks is further analyzed. Here, for example, is the diagnosis subtask:

Diagnose Student

Description: Identify the malfunction of the student.

Input: Student behavior.

Output: Indicate what knowledge the student is missing or what knowledge is in error.

Generic task: Diagnosis.

Domain knowledge: Knowledge about possible student errors.

Case model: Model of student behavior.

Pragmatic problems: Missing parts of the observations, large amount of possible errors.

Method: Cover and differentiate.

Subtasks: Find possible error classes covering the current case. Differentiate between remaining possibilities.

Once this task analysis is available, the details are worked out. For example, a domain model is selected for the student errors that is compatible with the chosen method (cover and differentiate). For the cover part, this domain model might consist of associations between symptoms and a catalog of possible behaviors and misbehaviors. A possible representation might be in terms of a set of rules, where the if part gives the associations and the then part the suggested behaviors. For the differentiate part, the domain model might be a list of behaviors and associated features, so that the most discriminating feature can be computed and observed in the student model.

The selection of a domain model makes the acquisition of the contents of this model easy because it is clear what is expected from the expert, and there is a lot of systematicity and, thus, opportunities for validation in the domain model.

Implications for Architecture

Apart from a systematic investigation of problem-solving methods, domain models, and their associated task features, it is also possible to develop shells that closely model the componential framework discussed in this article. Researchers at the VUB AI Lab are developing such a shell, called Kan (see Rademaekers and VanWelkenhuysen 1990 for related work). With this shell, a componential analysis can be directly translated into a

The componential framework suggests a structured methodology for doing knowledge acquisition.

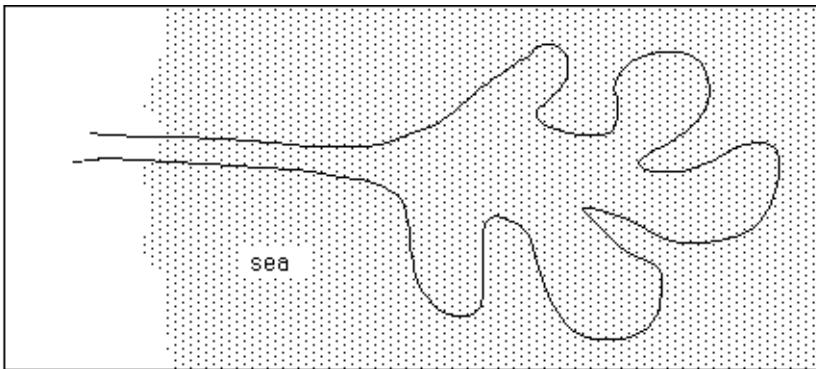


Figure 7. Distributary system of channels and fans.

working system. The shell supports the explicit representation of case models, domain models (of all types, that is, not just rules but also hierarchies, networks, or constraint systems), and problem-solving methods. For example, the heuristic annotation needed by weighted evidence combination is represented in a model that looks like the following:

```
(define (weighted-element weighted-finches-model) red-eared-firetail
  (weighted-feature (species red-eared-firetail) 1)
  (weighted-feature (beak red) 0.5)
  (weighted-feature (lores insignificant-uniform) 0.1)
  (weighted-feature (eyebrow insignificant-uniform) 0.1)
  (weighted-feature (ear-patch red) 0.8)
  (weighted-feature (crown insignificant-uniform) 0.1)
  (weighted-feature (back-and-wings insignificant-uniform) 0.05)
  (weighted-feature (rump red) 0.2)
  (weighted-feature (tail brown) 0.8)
  (weighted-feature (throat insignificant-uniform) 0.05)
  (weighted-feature (breast spotted) 0.2)
  (weighted-feature (belly spotted) 0.1)
  (weighted-feature (flanks spotted) 0.1)
  (weighted-feature (legs brown) 0.1))
```

This example concerns the classification of different species of the Australian finch. Red-eared-firetail is one species. One of the features is (ear-patch red). Weights of different features are 0.5 or 0.8. The problem-solving methods are also explicitly described. For example, the frame for weighted evidence combination looks like the following:

```
(define classificatory-method weighted-evidence-combination
  (domain-model-type collection)
  (solution-method solve-by-selection)
  (method-cliche classify-cliche)
  (fact-finding environment-driven)
  (purpose goal-based)
  (activation data-driven)
  (reason-about case-model)
  (element-status class)
  (feature-status prototypical)
  (domain-complexity small)
  (cost cost-not-relevant)
  (uncertainty uncertain)
  (incompleteness incomplete))
```

These descriptions are used to perform the task of selecting expert system solutions once an analysis of the task has been performed. They talk about how data gathering will be performed (does the problem solver wait for the environment to supply descriptions [environment-driven], or does it actively seek them [system-driven]), how knowledge will be accessed (data-driven versus goal-driven), what the complexity of the classification task is (small or large), what the status of the features (necessary and sufficient versus prototypical) is, and so on. Code is associated with the problem-solving method that implements the inference engine. This code is constructed by filling in and instantiating a cliche (the classify-cliche).

Each task in the task structure is mapped onto a *problem solver*, which is a structure that contains goals, case models, problem-solving methods, and domain models. The problem solver can decompose its goal into subgoals (and pass it on to new problem solvers) or directly solve a goal by applying the domain knowledge to problem solvers. Using Kan, the researchers have also explored the use of knowledge compilation techniques to transform knowledge from one form (for example, a list of classes) into another (for example, a decision tree).

Implications for Analyzing Existing Systems

A lot more space is needed to fully elaborate on the design methodology and the Kan shell. Thus, I end this article with an analysis of a rule from an existing expert system to illustrate the utility of the framework for analyzing existing systems. I take a rule from the Dipmeter Advisor, a geologic expert system that interprets a fine-grained measurement of varying resistivity in the rocks alongside a bore. The interpretation is in terms of the tilt-

ing of the subsurface layers and the geologic structures that caused these patterns. Consider now the following rule from the Dipmeter Advisor system (Smith 1984):

IF

- 1) Delta-dominated marine zone
- 2) Continental shelf marine zone
- 3) Sand zone intersecting marine zone
- 4) Blue pattern in intersection

THEN

Distributary fan with

- Top of fan equal to top of blue pattern
- Bottom of fan equal to bottom of blue pattern
- Direction of flow equal to azimuth of blue pattern.

When someone is confronted for the first time with a rule such as this one, it appears to be pure magic. How can the expert conclude from the presence of a particular pattern on the dipmeter log the presence and direction of a distributary fan? These first impressions have enforced the view that experts use incomprehensible heuristics and rules of thumb that defy rational explanation. One of the advantages of a knowledge level and knowledge-use-level analysis of expertise is that you come to understand rules in rational ways.

Task Structure. The task structure of the Dipmeter Advisor contains the following major tasks (Smith 1984):

Validity check: See whether the logs are of good quality. Identify areas where there is evidence of tool malfunction or incorrect processing.

Structural analysis: Find the large-scale structural features that result from folding, faulting, uplifting, and so on.

Lithological analysis: Identify the zones with constant lithology (for example, sand zones).

Depositional environment analysis: Identify what the depositional environment is underlying a specific zone, and infer properties about this environment.

Stratigraphic analysis: Find out which geologic system (river channel, dune, reef, bar) was the source of the structures seen by the data.

The rule is part of the final task, stratigraphic analysis. It decomposes into the following three subtasks:

Consider the depositional environment: The above rule concerns delta-dominated,

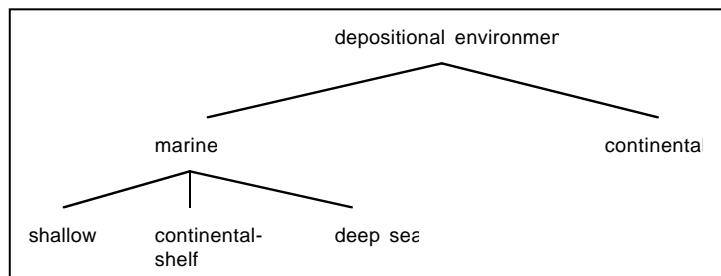


Figure 8. Tree of more specific geological contexts.

continental shelf marine zones.

Identify the depositional system: The rule recognizes a distributary fan.

Deduce further properties: The rule deduces the top and bottom of the fan and the direction of flow.

Task Typology. The task decomposition for stratigraphic analysis can be interpreted in terms of the generic tasks and subtasks previously discussed. Specifically, the generic task underlying the distributary fan rule is interpretation. Its decomposition into three generic subtasks is typical for interpretation: context restriction (consider depositional environment), selection (select depositional system), and deduction (deduce further properties).

I now investigate the problem-solving methods and domain models that solve these tasks.

Context Restriction (Consideration of Depositional Environment). This task is done in clauses 1 and 2 of the rule. It uses the following geologic facts:

First, there are such elements as marine environments, that is, environments covered by the sea.

Second, some marine environments are dominated by deltas. A *delta* (such as the Nile delta or the Mississippi River delta) is a system where a river ends at the sea. The river has an impact deep into the sea.

Third, marine environments are divided into different subenvironments, depending on how far they are from the coast. One subenvironment is the continental shelf.

Fourth, once inside a continental shelf marine environment, only one significant geologic structure occurs, namely, the channels formed by the river delta and the fans at the end of these channels. This system of channels and fans is called a *distributary system* (because it distributes material from

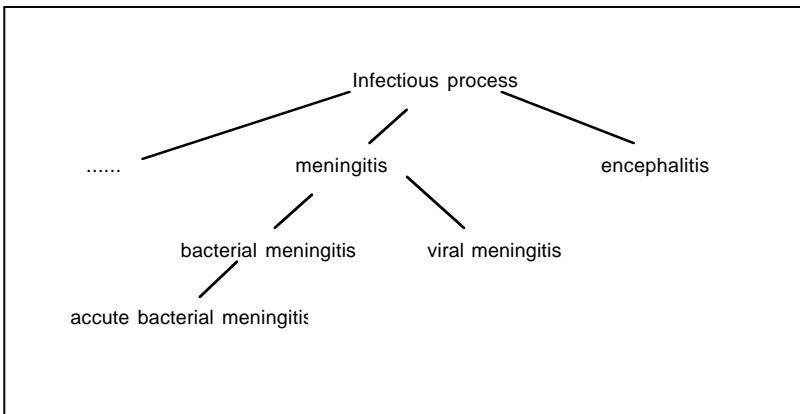


Figure 9. Context tree of more specific infections

the river into the sea). The channels and fans that are part of the system are called *distributary channels* and *distributary fans* (figure 7).

This domain model can easily be represented in a hierarchy with lower nodes containing more specific contexts (figure 8). This context tree is the domain model underlying the context restriction task.

Although there are different uses for this domain model, in this particular case, it is used to narrow the set of interpretations to those that occur in a particular geologic environment. Here, the context is narrowed from a marine environment to a delta-dominated marine environment to a continental shelf zone and, finally, to a distributary system as the only significant geologic structure. Within this system, only three contexts remain: distributary channel, deltaic plain, and distributary fan. The rule itself does not explain how each of these contexts is established, but it does specify in which order contexts need to be considered and refined. Specifically, it says that one should use top-down refinement: Start from the most general context, and narrow this context to the most specific one. Top-down refinement is the problem-solving method used in conjunction with the context tree.

Consider the following rule from Mycin (Buchanan and Shortliffe 1984, p. 554) as another illustration of the same generic subtask:

IF

- 1) The infection which requires therapy is meningitis
- 2) The type of meningitis is bacterial
- 3) Only circumstantial evidence is available for this case
- 4) The age of the patient is greater than 17 years
- 5) The patient is an alcoholic

THEN

There is evidence that the organisms which might be causing the infection are diplococcus-pneumoniae (.3) or E.coli (.2)

The first part of this rule (clauses 1 and 2) is again a case of context restriction using top-down refinement. (Because Mycin uses backward chaining, the ordering of the clauses in this rule is really a way to formulate in which order the questions raised by each clause are to be considered.) The underlying context tree is depicted in figure 9. The context is refined from infection to meningitis and then to bacterial meningitis. Then, only a limited group of organisms remains, and circumstantial evidence (namely, alcoholism) is considered to identify them.

These examples illustrate the major characteristics of the componential analysis of expertise previously outlined: tasks, domain models, and problem-solving methods. Tasks are partial steps in the solution of the problem. The generic task considered here is interpretation, with context restriction as the first generic subtask. Domain models represent facts about the domain. An example is a context tree. Problem-solving methods are ways of realizing a particular task, through either further decomposition or concrete actions. An example is top-down refinement.

A specific rule compiles elements of these components for immediate and efficient use. Thus, in its first two clauses, the distributary fan rule compiles the task of context restriction using top-down refinement based on a tree of geologic contexts.

The same types of domain models, task types, and problem-solving methods can be found across domains. The Dipmeter Advisor, as well as Mycin, uses the same realizations of the same generic tasks. However, although the type of these components is domain independent, their application is domain specific. The expert knows that in his domain, it is a good idea to first restrict the context.

Selection of Depositional System. I now return to the distributary fan rule. After context restriction, which is done by clauses 1 and 2, three possibilities remain: distributary fan, deltaic plain, and distributary channel. Each of these possibilities has an associated prototype: The distributary fan can be found on continental shelf, delta-dominated marine zones; is made up of sand; has layers decreasing in dip (blue pattern); and has a direction of flow that is the azimuth of decrease. The deltaic plain can be found on continental shelf, delta-dominated marine zones; is made

up of shale; has a constant grain size; and has layers constant in dip (no pattern). The distributary channel can be found on continental shelf, delta-dominated marine zones; is made up of sand; has layers with decreasing dip (blue pattern); has a grain size that becomes finer as it goes up; has top layers with increasing dip; and moves in a perpendicular direction (red pattern).

Clauses 3 and 4 of the distributary fan rule take care of selecting a prototype from this catalog:

IF

- 1) Delta-dominated marine zone
- 2) Continental shelf marine zone
- 3) Sand zone intersecting marine zone
- 4) Blue pattern in intersection

THEN

Distributary fan with

- Top of fan equal to top of blue pattern
- Bottom of fan equal to bottom of blue pattern
- Direction of flow equal to azimuth of blue pattern

These two clauses implement the task of selecting the prototype that best fits the current situation from the catalog. The problem-solving method by which the Dipmeter Advisor finds selection is differentiation. The system looks at distinguishing features and decides on the basis of these features that a distributary fan is present. The features are lithology (sand versus shale), which differentiates between distributary channel or fan on the one hand and deltaic plain on the other, and pattern of dip (blue versus red over blue), which differentiates between fan and channel. Other features could have been chosen, such as the evolution in the grain size, but were not, presumably because they are not as easily observed on the dipmeter log.

The same selection task is found in other applications. The Mycin system contains a subsystem for identifying the organism causing the infection, using rules such as the following (Buchanan and Shortliffe 1984, p. 92):

IF

- 1) The stain of the organism is gramos, and
- 2) The morphology of the organism is coccus, and
- 3) The growth conformation of the organism is chains

THEN

There is suggestive evidence (.7) that the identity of the organism is streptococcus

Again, there is a catalog of prototypes for organisms. Each prototype describes one organism with its various features (identity,

stain, morphology, aerobicity, growth conformation, and so on). One member of this catalog follows:

Streptococcus

- member of G+cocci
- grampos stain
- coccus morphology
- growth conformation in the form of stains

This rule is concerned with identifying the prototype. Note that the rule does not perform context restriction but immediately focuses on the selection itself. The problem-solving method used for selection in this case is not differentiation but weighted evidence combination: Combinations of observed features raise the evidence for the presence of a particular prototype from the catalog. In the previous rule, the combination of three features adds more plausibility (namely .7) to the Streptococcus hypothesis. Other rules add strength to other members of the catalog. In general, all rules are tried, and at the end, the prototype with the highest strength wins out. In this case, the heuristic annotation specifies what the import of each feature or combination of features is for the strength of a specific prototype.

Deduction. The analysis of the distributary fan rule is not yet finished. We have seen how clauses 1 and 2 restrict the context and how clauses 3 and 4 select a prototype from a limited remaining catalog of prototypes. We still have to explain aspects of the then part of the rule. It is already clear that the conclusion we have is a distributary fan. However, what about the other conclusions, that is, top of fan equal to top of blue pattern, bottom of fan equal to bottom of blue pattern, and direction of flow equal to azimuth of blue pattern.

Clearly, the generic task carried out in this part of the rule is deduction. The domain model that contains the knowledge needed to make these inferences is an example of a constraint system. A *constraint system* ties together a number of domain properties using equality relations, possibly augmented with numeric or logical operators. The following is the constraint system underlying the then part of the distributary fan rule:

Top of fan = top of blue pattern

Bottom of fan = bottom of blue pattern

Direction of flow = azimuth of blue pattern

The problem-solving method used here is calculation. The constraints are translated into a formula that can be used in computation. In this case, the calculation is trivial. In other applications, it can be much more com-

A constraint system ties together a number of domain properties using equality relations . . .

plex. For example, Mycin uses complex equations to compute the recommended dosage using factors such as weight of the patient and age.

This analysis shows that the framework sketched in the previous section can be used to analyze rules in existing rule-based systems. This analysis clarifies the rationale behind a rule and shows that it is not a simple empirical association but a carefully crafted combination of task decomposition, problem-solving method, domain model, and heuristic annotations. It also shows that the view that surface knowledge consists of shallow associations is not justified, at least not if we describe the Dipmeter Advisor as an example of an expert system with only surface knowledge. Instead, the Dipmeter Advisor appears to have as much deep knowledge as we might want; it is only represented in a different form, compiled for efficient problem solving.

Conclusions

This article analyzed different ideas that have recently emerged for the description and design of expert systems. These ideas all attempt to go beyond the implementation level, which focuses on formalisms and implementation constructs. They try to capture what has become known as the knowledge level and the knowledge-use level. Four key ideas were studied: inference structures, deep expert systems, problem-solving methods, and generic tasks. Each of these ideas focuses on one aspect of expertise and problem solving: the inference pattern, the domain models, the problem-solving methods, and the task features.

The article also proposed a componential framework that attempts to synthesize the

different viewpoints. It is based on an analysis of expertise broken down into components: the tasks and subtasks, the domain models, the case models, and the problem-solving methods. The implications of this framework for the methodology of expert system design, architectures, and analysis were briefly outlined.

The perspective put forward in this article allows for a vast amount of work concerned with cataloging the various components of expertise found in different application domains and systematically studying the relationships between task features (both conceptual features and pragmatic constraints) and choices for each of the components. At some point, we should end up with a knowledge engineering handbook, similar to handbooks for other engineering fields, that relates task features with expert system solutions.

Acknowledgment

This work was supported in part by the Belgian IWONL Program on Expert Systems.

The ideas expressed in this article have been growing for a long period of time and have involved discussions with many people: first with my colleagues at Schlumberger—in particular, Howard Austin, Bud Frawley, and Anatole Gershman—and later at the VUB AI Lab—in particular, Walter Van de Velde. The following people at the VUB AI Lab also contributed to realizing and debugging the ideas in this article: Danny Bogemans, Rudy deDonder, Filip Rademaekers, Yves Tassenoy, Peter Strickx, Kris Van Marcke, and Johan Vanwelkenhuysen. Comments from Pattie Maes and John McDermott improved the current version of this article.

Bibliography

- Adams, J. B. 1984. Probabilistic Reasoning and Certainty Factors. In *Rule-Based Expert Systems*, eds. B. Buchanan and E. Shortliffe, 263–272. Reading, Mass.: Addison-Wesley.
- Bennett, J. S. 1983. ROGET: A Knowledge-Based Consultant for Acquiring the Conceptual Structure of an Expert System, Memorandum, HPP-83-24, Stanford Univ.
- Breuker, J., and Wielinga, B. 1984. Interpretation of Verbal Data for Knowledge Acquisition. In Proceedings of ECAI-84, 41–50. Amsterdam: North-Holland.
- Brown, D., and Chandrasekaran, B. 1989. *Design Problem Solving: Knowledge Structures and Control Strategies*. London: Pitman.
- Buchanan, B., and Shortliffe, E. H., eds. 1984. *Rule-Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project*. Reading, Mass.: Addison-Wesley.

- Bylander, T., and Mittal, S. 1986. CRSI: A Language for Classificatory Problem Solving and Uncertainty Handling. *AI Magazine* 7(3): 66–77.
- Chandrasekaran, B. 1983. Towards a Taxonomy of Problem-Solving Types. *AI Magazine* 4(1): 9–17.
- Chandrasekaran, B., and Mittal, S. 1983. Deep versus Compiled Approaches to Diagnostic Problem Solving. *International Journal of Man-Machine Studies* 19:425–436.
- Clancey, W. J. 1988. Viewing Knowledge Bases as Qualitative Models. *IEEE Expert* 4(2): 9–23.
- Clancey, W. J. 1985. Heuristic Classification. *Artificial Intelligence* 27(3): 289–350.
- Clancey, W. J. 1984. Extensions to Rules for Explanations and Tutoring. In *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, eds. B. Buchanan and E. Shortliffe, 531–571. Reading, Mass.: Addison-Wesley.
- Clancey, W. J. 1982. The Epistemology of Rule-Based Expert Systems: A Framework for Explanation. *Artificial Intelligence* 20(3): 215–251.
- Davis, R. 1984. Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence* 24(1–3): 347–411.
- Davis, R. 1982. Tereisias: Applications of Meta-Level Knowledge. In *Knowledge-Based Systems in Artificial Intelligence*, eds. R. Davis and D. Lenat, 229–485. New York: McGraw-Hill.
- de Kleer, J. 1984. How Circuits Work. *Artificial Intelligence* 24(1–3): 205–281.
- Eshelman, L. 1988. Mole: A Knowledge Acquisition Tool for Cover-and-Differentiate Systems. In *Automating Knowledge Acquisition for Expert Systems*, ed. S. Marcus, 37–79. Boston: Kluwer.
- Hart, P. 1984. Artificial Intelligence in Transition. In *Knowledge-Based Problem Solving*, ed. J. Kowalik, 296–311. Englewood Cliffs, N.J.: Prentice-Hall.
- Kahn, G. 1988. More: From Observing Knowledge Engineers to Automating Knowledge Acquisition. In *Automating Knowledge Acquisition for Expert Systems*, ed. S. Marcus, 7–37. Boston: Kluwer.
- McDermott, J. 1988. A Taxonomy of Problem-Solving Methods. In *Automating Knowledge Acquisition for Expert Systems*, ed. S. Marcus, 225–256. Boston: Kluwer.
- McDermott, J. 1982. R1: A Rule-Based Configurer of Computer Systems. *Artificial Intelligence* 19(1): 39–88.
- Marcus, S. 1988a. *Automating Knowledge Acquisition for Expert Systems*. Boston: Kluwer.
- Marcus, S. 1988b. SALT: A Knowledge-Acquisition Tool for Propose-and-Revise Systems. In *Automating Knowledge Acquisition for Expert Systems*, ed. S. Marcus, 81–124. Boston: Kluwer.
- Marcus, S.; Stout, J.; and McDermott, J. 1987. VT: An Expert Elevator Designer. *AI Magazine* 8(4): 39–56.
- Mittal, S.; Dym, C.; and Morjaria, M. 1986. PRID: An Expert System for the Design of Paper Handling Systems. *IEEE Computer* 19(7): 102–114.
- Newell, A. 1982. The Knowledge Level. *AI Journal* 19(2): 87–127.
- Offutt, D. 1988. Sizzle: A Knowledge-Acquisition Tool Specialized for the Sizing Task. In *Automating Knowledge Acquisition for Expert Systems*, ed. S. Marcus, 175–198. Boston: Kluwer.
- Rademaekers, P., and VanWelkenhuysen, J. 1990. Mapping the Knowledge Level into the Computational Level. In Proceedings of ECAI-90. London: Pitman. Forthcoming.
- Simmons, R. 1988. Generate, Test, and Debug: A Paradigm for Solving Interpretation and Planning Problems. Ph.D diss., AI Lab, Massachusetts Institute of Technology.
- Simon, H. A. 1982. *The Sciences of the Artificial*. Cambridge, Mass.: MIT Press.
- Smith, R. G. 1984. On the Development of Commercial Expert Systems. *AI Magazine* 5(3): 61–73.
- Stanfill, C., and Waltz, D. 1986. Toward Memory-Based Reasoning. *Communications of the ACM* 29(12): 1213–1228.
- Steels, L. 1987. The Deepening of Expert Systems. *AI Communications* 1:9–17.
- Steels, L. 1984. Second-Generation Expert Systems. Presented at the Conference on Future Generation Computer Systems, Rotterdam. Also in *Journal of Future Generation Computer Systems* (1)4: 213–237.
- Steels, L., and Van de Velde, W. 1985. Learning in Second-Generation Expert Systems. In *Knowledge-Based Problem Solving*, ed. J. Kowalik. Englewood Cliffs, N.J.: Prentice-Hall.
- Steels, L., and Vanwelkenhuysen, J. 1990. Monitoring Subway Traffic Using TAUMES: The Compositional Framework in Practice, VUB AI Memo 90-4.
- Swartout, W. 1981. Producing Explanations and Justifications of Expert Consulting Programs. Ph.D. diss., Massachusetts Institute of Technology.
- Van de Velde, W. 1987. Inference Structure as a Basis for Problem Solving. In Proceedings of ECAI-8, 202–208. London: Pitman.
- VanMelle, W. 1981. *System Aids in Constructing Consultation Programs*. Ann Arbor, Mich.: UMI Research Press.
- Vanwelkenhuysen, J. 1989. TroTelc: An Expert System Troubleshooting Printed Circuit Boards, VUB AI Memo 89-17.

Luc Steels is a professor of computer science and artificial intelligence at the Free University of Brussels (VUB) and director of the VUB AI Laboratory, Pleinlaan 2, 1050 Brussels, Belgium.