Edgar H. Sibley
Panel Editor

Based on a review of some actual expert-system projects, guidelines are proposed for choosing appropriate applications and managing the development process.

EXPERT SYSTEMS: PERILS AND PROMISE

DANIEL G. BOBROW, SANJAY MITTAL, and MARK J. STEFIK

By virtue of their flamboyant quality, the expressions *artificial intelligence* (*AI*) and *expert systems* have helped contribute to an expanding wave of activity and unrealistic expectations about the state of the art. Taking a long, hard look at those expectations, this article contrasts them with the results of some actual case studies and proposes both a more realistic view of the practice of building expert systems and some guidelines for choosing appropriate applications.

The term *expert system* refers to computer programs that apply substantial knowledge of specific areas of expertise to the problem-solving process. The term *expert* is intended to imply both narrow specialization and competence, where success of the system is often due to this narrow focus. Just as human experts have varying levels of expertise, so too do computer systems. Although, in general, expert systems have less breadth of scope and flexibility than human experts, and for this reason are sometimes criticized for creating unrealistic expectations, we find it more productive to ask about the level and range of expertise of a given program (i.e., how well does it do on a specific set of tasks), rather than struggling with the imprecise boundary of what constitutes "expert."

In some circles, the terms *knowledge-based systems* or *knowledge systems* are used instead of *expert system* [10] to focus attention on the knowledge the systems carry, rather than the question of whether or not such knowledge constitutes expertise. These terms also imply the use of technology for explicit representation of knowledge. But, once again, the boundary between explicit and implicit representation is imprecise in that knowledge can be represented explicitly to different degrees and can take different forms.

High-level performance can be achieved without explicit representation of knowledge as in an autopilot [29]; one might even ask whether a C-Compiler or a payroll program constitutes an expert system. Clearly they both embody knowledge: in the one case, of a language and computer and, in the other, of accounting and taxes. When constructed with conventional programming techniques, both would have a very limited range of capabilities. However, it is also possible to build either as a knowledge-based system. One might then ask the AI payroll system hypothetical questions, such as the net difference in taxes if you added two more deductions. In their usual embodiments, systems like these are generally designed with built-in commitments as to how the knowledge embedded in them is to be used, that is, to have "compiled out" (built into the programs) *lim-*

ited input/output behavior. To use this same knowledge to generate tax advice, in the case of the payroll system, would require recoding the system for that purpose. For this reason, the term *knowledge-based* is generally reserved for systems that have explicit knowledge bases and some flexibility in the use of that knowledge.

In this article, we examine primarily knowledge-based expert systems: systems that achieve expert-level performance using explicit representations of knowledge. For purposes of brevity, we will refer to them simply as expert systems.

APPROACHES TO BUILDING EXPERT SYSTEMS

Depending on the extent and depth of the explicit representation of knowledge, we can delineate three different approaches to expert-system development: the *low road*, the *middle road*, and the *high road* [6].

The low road involves direct symbolic programming, usually in the Lisp programming language. It takes advantage of new low-cost AI machines with flexible programming environments enhanced by user interfaces that exploit window systems on large displays. These environments support a style of program development called "exploratory programming" [28] in which there is incremental, parallel development of program specification and implementation—an appropriate style for applications where the primary concern is efficiency and the required knowledge base is small and does not need to be changed very frequently.

The low road was used for the early expert systems (e.g., Dendral [19]), which combined AI techniques for heuristic search with Lisp capabilities for symbolic manipulation. Dendral generated and tested hypotheses about chemical structures and spectroscopic data. It needed to be efficient because the search space of possible solutions is very large; moreover, programming it directly into procedures was practical since the knowledge used for interpreting spectral data is fairly static.

The high road, on the other hand, involves building a system that contains explicit representation of fairly complete knowledge of some subject matter, and can use that knowledge for more than one purpose. A system is called "deep" when its knowledge represents the principles and theories underlying the subject; a consequence of this depth is that such systems often require long chains of reasoning from first principles to practical results.

Sophie [11] is a high-road system that performs diagnostic reasoning and qualitative simulation, and can reason from first principles about how physical devices work. For many classes of devices, Sophie can determine the behavioral states that the devices will traverse for a given set of inputs. When the actual output of a device does not agree with the predicted output, the program uses the same fundamental knowledge to generate hypotheses about which parts of the device may be broken.

The ultimate goal of high-road systems is a knowledge processing capability that is general enough to span domains—an ability to apply general principles and commonsense reasoning to the articulation and testing of general facts and principles, and to experiment with processing and reasoning about these facts. Given current computers and compilation techniques, high-road systems are usually too slow for real-world (large-scale) applications, since they take only very small steps toward the solution of big problems; currently they are used only for research.

Middle-road systems, not surprisingly, fit squarely between these two extremes. They also involve explicit representation of knowledge, but though some direct programming may be used, most of the interesting behavior of the system is governed by knowledge that is articulated by experts and represented explicitly in a knowledge base. Canned problem-solving tactics rather than first principles are most often the rule.

Knowledge engineers working with experts often use knowledge-engineering tools and hybrid languages [15, 17] for this purpose. The technology facilitates the representation, reorganization, and debugging of programs expressed as interacting pieces of knowledge. A key characteristic of middle-road systems is that they are sharply focused on a single task and incorporate knowledge specialized for the task, but the explicit representations often do not specify the limitations of that knowledge.

A well-known example of a middle-road expert system is the Mycin system for medical diagnosis and prescription [7]. Mycin contains rules that associate symptoms with diseases, for which it can also evaluate and prescribe treatment. Mycin, like most expert systems, is called a *shallow* system because most of its reasoning chains are short. It has no physiological model of disease or health, no model of how diseases cause symptoms, and no model of how treatment can help cure diseases. For most applications, the middle road is the most effective approach now available for building expert systems.

Expert systems are no panacea for achieving the impossible or even the very difficult. A mere inclination to have an expert system is no guarantee that one can be built. Identifying a need—"We need an expert who can make money on the stock market" or "It would be great to have a program to transform a functional circuit specification to an optimal inte-

grated circuit layout"—does not suffice to determine an appropriate task for an expert system.

Instead, there are a number of fundamental issues and requirements that must be considered. In the balance of this article, we present three successful projects as case studies. We then generalize from these examples to present guidelines for choosing appropriate applications and developing successful systems.

CASE STUDIES

The R1 System for Configuring Vaxen

In 1978, the Digital Equipment Corporation and a group headed by John McDermott at Carnegie-Mellon University started a joint project to build an expert system that could aid in the configuration of VAX computers. As McDermott [21] describes it, the rule-based configurer R1 received a customer's purchase order and then determined what if any substitutions and additions were needed to make the order consistent and complete. It produced a number of diagrams showing the spatial and logical relationships among the 90 or so components that typically constitute a VAX-11 computer system.

R1 was the fourth attempt at automating the construction of complete configuration descriptions for DEC computers [20]. At least part of the reason the three earlier attempts failed was that their implementation technology did not allow knowledge to be expressed explicitly. They were built with standard programming technology, which made it hard to understand the interactions between pieces of the system and to change and augment the system as additional knowledge was gained.

Knowledge Representation. R1 is a rule-based system that uses the OPS-5 programming language [14] built in Lisp. In R1, a typical rule (rendered here in pseudo-English) might look like this:

```
IF: The most current active context is
putting the unibus modules in the
backplane in some box

AND it has been determined which
module to try to put in a backplane

AND ...

THEN: Enter the context of verifying
panel space for a multiplexer
```

The rule consists of a series of conditions to be tested (the `IF` parts), followed by actions to be taken provided the `IF` part is true. The conditions test such things as the state of the problem-solving process, availability of parts, and connectivity of the configuration.

The first prototype, tested in early 1980, contained roughly 750 such rules, which interacted with a database describing some 4000 parts that could be used in configuring a VAX system. This prototype was far from perfect, and over 50 person-years of effort were invested in developing R1 into the current DEC product known as XCON [2]. During that time, the system grew from 700 rules to about 3500. The completed system (which is still being changed and updated today) configures not only the VAX systems, which were originally chosen for the configuration problem, but the much more complicated PDP-11 family.

Evolution. In light of its unchallenged success today, it is perhaps hard to believe that R1 was almost canceled three times. The uncertainty among people within DEC was based on a misconception about the nature of expert-system technology. Basically, the term *expert system* encouraged very high expectations, while the R1 prototype continued to make mistakes: It was not an "instant expert." However, such an expectation is not appropriate for any system, or even a person new at a job. No matter how well trained a new employee is, new knowledge must be acquired and older knowledge restructured, and this takes time.

Ensuring proper development of an expert system after the prototype stage requires building the process in the organization that is to use it. In the case of R1, a group monitored the problems, which were categorized as follows:

- incorrect component description in the database,
- incorrect configuration knowledge,
- incomplete configuration knowledge,
- an error in the data input to the system, and
- a confusion by the person reporting the problem (essentially, a nonproblem).

For each of the first four problems, a specific person was assigned responsibility for obtaining the appropriate data and modifying the system. This might mean redesigning a set of rules when required by interactions with previous rules, or rewriting instructions, or changing the user interface in response to errors in the input data.

Of the 2500 rules added to the original system, somewhat less than 40 percent were used to make major refinements in the knowledge. Another 10 percent made minor refinements to improve output, while about 35 percent provided additional functionality to deal with new system types. A large core of configuration knowledge in the original system was specific to the VAX 780, while the current system (XCON) deals as well with the much more complicated PDP-11 family. Finally, about 15 per-

cent of the new rules were added to extend the definition of the configuration task to include things like laying out cables or floor layout of racks. (Layout was a new capability added to the system.) The moral of this story is, of course, that *expert systems must be extended and changed over time*—a task that is made simpler by using explicit knowledge programming.

The R1 team also had to define the delivery vehicle for the R1 system. Since the original OPS-5 implementation in Lisp was considered too slow to be used for production jobs, DEC reimplemented OPS-5 in Bliss, making it much faster. Nonetheless, the Lisp environment still provided much better tools for debugging, so that now changes to the system are reportedly done in the Lisp environment and then transferred to the Bliss production system. Today, with the high-performance, low-cost, personal Lisp machines currently available, such reengineering of the delivery engine might not have been necessary.

The current R1/XCON system configures over 97 percent of the orders received for VAX-11 systems. It has significantly reduced the time needed to configure an order and has produced more accurate configurations, both of which benefits have brought significant monetary advantage since customers pay for machines after they start working. Incorrectly configured orders cause significant delay if it is necessary to wait for additional parts. A further unexpected benefit for most orders is that configurations are created in a much more consistent style using R1 than when configured by the group technical editors. This has allowed fine-tuning the factory to handle a more stereotypical order.

In terms of the human side of this equation, R1/XCON has meant that the technical editors employed at DEC have a higher productivity. Given this tool to amplify their capabilities, they not only produce more, but are also more satisfied because they now concentrate on the interesting problems rather than the dreary, day-to-day repetitive tasks that consumed their time before. R1/XCON represents a real success story for expert-system technology, for the company, for its employees, and for customers.

What about maintaining the system? Although the encoding of XCON's knowledge in production rules was supposed to make it easier to manage the knowledge base, the cost of XCON's maintenance is still high, but very worthwhile given the economic benefits derived from it. Furthermore, the notion of declarative knowledge representations is open-ended, in the sense that it will accommodate changes to the knowledge base. Current research at Carnegie-Mellon is aimed at reducing maintenance costs still further through the creation of higher level "shells" that actively support the process of knowledge acquisition and testing. New prototype versions of the configuration system are now being reengineered using these shells.

The Pride System for Mechanical Design

Pride [25] is an expert system developed at Xerox to assist engineers designing paper transports inside copying and duplicating machines. It is a joint effort begun in early 1984 between the Xerox Palo Alto Research Center (PARC) and the Xerox Reprographics Business Group (RBG). The first prototype was ready for testing in early 1985. Since then the system has been tested extensively on past and current design cases, and a support group is being created within the RBG to gradually take over further development and testing.

Problem Domain. The first stage of the Pride project consisted of analyzing the problem-solving behavior of a group of experts working on the design of paper transports [23]. From this exercise, it became clear that knowledge was distributed among many different experts, and that these experts showed varying degrees of specialization based on the technology used (e.g., pinch roll versus belt versus vacuum transport technologies).

Other kinds of specializations were based on variations in design specifications (e.g., precisely registering paper or building a recirculating document handler). Depending on the volume of copies to be produced per month, there are differences based on trade-offs and cost considerations. Other specializations relate to different parts of the design task: material selection, jam clearance, cost analysis, or design and analysis. Some engineers generate perfectly plausible designs, but are unable to analyze the designs to determine whether the designs meet the requirements, or predict performance problems; others primarily analyze other engineers' designs.

Finally, there is a difference in the depth of knowledge: Some engineers know only standard approaches, whereas others can provide reasoned arguments about the trade-offs involved in different design decisions. The differences in expertise become more apparent in nonstandard designs or nonstandard problems.

It became apparent to technical managers that the existing engineering expertise from the engineering research division was often not used by the design engineers in the applied divisions. Moreover, design engineers often did not have adequate access to techniques for analyzing the designs they created, even though the techniques were often available either in research publications or as computer pro-

grams. Therefore, two chief purposes of Pride were to facilitate the necessary *knowledge transfer* and provide a single framework for creating and testing a design. Toward this latter end, the project has always had two consulting domain experts: one in the design of paper transports and the other in the analysis and simulation of mechanical systems.

Representation of Knowledge in Pride. A designer's assistant rather than an "automated designer," Pride reflects the reality that, at every stage of its development, some design knowledge and criteria will be missing from the knowledge base. On the other hand, the task of exploring a large design space—maintaining the dependencies between different parts of the design, systematically checking constraints and applying analyses, and maintaining alternate designs—is overwhelming in its detail. The design engineer and the Pride system are expected to work as a team: Together, they can explore a larger design space in a shorter time than either could working alone.

Pride's knowledge of design is organized in a generic design plan. The plan includes *goals*—what part of the design is being done (e.g., "design the paper path"); *methods*—how to make some design decision (e.g., "the width of a drive roll is typically 25 mm and can go up in increments of 1"); *constraints*—checks and requirements on the decisions made at a certain design goal (e.g., "the width of the idler must be ≥ 1.2 times the width of the driver"); and *calculations*—derivation of values from other design parameters. These primitives are represented as objects in Loops [4, 33]. For example, a simple constraint might be represented as the following:

type	SingleConstraint
description	"Idler width >1.2 times driver width"
applyWhen	(>(defRollerPair **idler** width) 50 mm) ;when to apply
mustSatisfy	YES ;could be an optimizer
paraConstrained	(defRollerPair **idler** width) ;focus on idler width
predicate	GreaterThan
testExpression	(TIMES 1.2 (defRollerPair **driver** width))
advice	(Increase(defRollerPair **idler** width)) (Decrease(defRollerPair **driver** width)

This simplified presentation of a single constraint indicates how the object groups different aspects of the constraint and makes them accessible. In addition to the textual comment and the parsed expression for the constraint, it provides advice about what to do if the constraint is not satisfied (e.g., try increasing the idler width, if possible, or if not, then try decreasing the driver width). The first advice might not be possible if rollers of that size are not available, or if a larger width would cause parts to rub together. The system applies advice in exploration of a large design space using a set of goal objects that represent a design plan for the transport. When constraints are violated and the advice does not lead to an acceptable modification of a design parameter, the system gets additional guidance from the designer.

Testing and Current Status. To operate successfully as a design assistant, Pride must support the process of exploration by an engineer: It must be able to both generate design alternatives and to verify that designs satisfy constraints. The system's ability to check constraints systematically is especially valuable when the design space is complex and when designs are optimized by different groups of engineers.

Pride's design verification ability has been tested on designs from ongoing projects where the system applies a set of analyses and constraints in the knowledge base to designs created by engineers. In one test, Pride found flaws in a design, helped pinpoint the source of the problems, and proposed a new design that avoided those problems. The knowledge base has now been successfully tested on many paper transport problems from past and current copier projects.

Although the current knowledge base contains only some of the copier technologies used by engineers, it has most of the knowledge for designing paper transports using those technologies. A development group has been created to acquire the knowledge for other technologies during the coming months. It is expected that Pride will begin to see operational use in 1986.

The Dipmeter Advisor

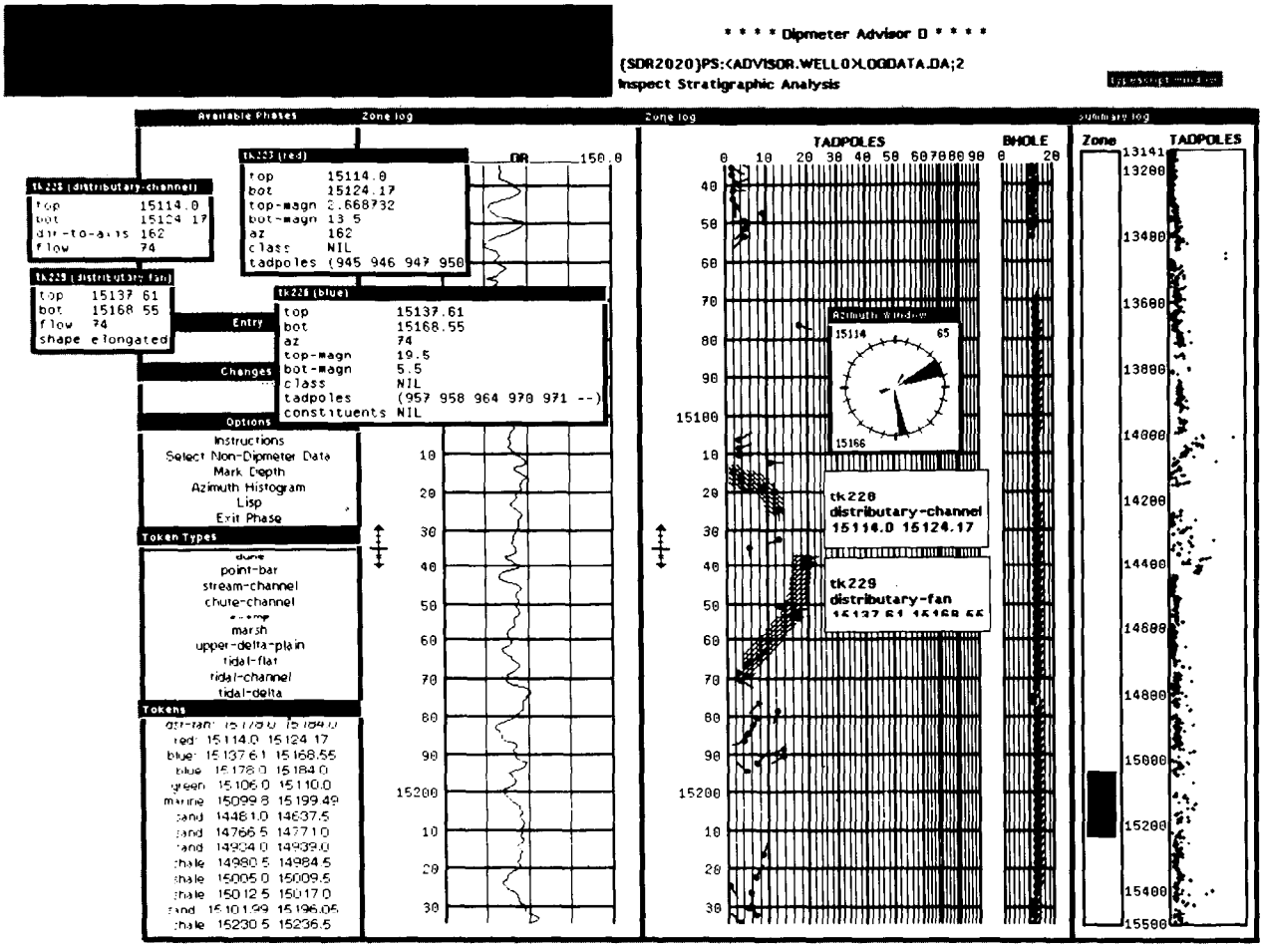
The history of the dipmeter advisor for petroleum exploration [31] is similar to that of the VAX configurer R1. A feasibility study in the Schlumberger research laboratories was started in 1978, and the initial prototype was ready for testing in 1980. It was built on the DEC 2020 in 245 kilobytes of Lisp code interacting with another program of 450 kilobytes written in VAX Fortran. This initial implementation proved too slow and unwieldy—a system that slows the expert doing his or her job has only negative value. A reimplementation was completed in 1983 using Interlisp-D on a Xerox 1100, a personal computer whose high-speed display allowed development of a user interface that was much more attuned to rapid interaction with the expert.

Problem Domain. To determine as early as possible whether an oil well will contain oil or be a dry hole, oil explorers lower specialized logging instruments into the borehole; these instruments provide information about the geology of the subsurface formations being pierced.

As consultants to the oil industry, Schlumberger provides instrumentation and data analysis (e.g., of the data logs taken at different depths). One such instrument, the "dipmeter," measures the inclination or tilt of the rock strata penetrated by a borehole. The dipmeter advisor developed by Schlumberger mimics the analysis of an expert using these data: Analysis programs process the signal, and identification and classification rules use this analysis in an expert-system environment.

Knowledge Representation. The dipmeter interpretation problem requires alternating stages of signal analysis and expert interpretation. After the signal analysis creates objects describing parts of the borehole, rules similar in structure to those for R1 are used to extend the description; the rules use both the results of signal analysis and other information, such as the geology of the undersea formation.

User Interface. Well-log analysts look at output from strip recorders showing the data progression for various depths in the well. The interface to the dipmeter system preserves the visual features of such displays, augmented by summary information, as shown in Figure 1. The multiple columns provide an overview of the data that is comparable, and proba-



This figure shows the user midway through a stratigraphic analysis. On the far right side is a log of the dipmeter data. The black "elevator box" to the left of that column shows the portion of the log expanded two columns to the left. Radical shifts in focus are obtained by moving the elevator. Other logs and summary information are shown on the left.

FIGURE 1. Dipmeter Interface

bly superior to, the long rolls of paper tacked up on the walls of experts' offices. A critical feature of the interface is the ease in moving from one portion of the data to another with a smooth scrolling operation: In fact, Schlumberger claims they can scroll the paper on the screen at 57 miles an hour—an unusual metric for a display system.

Much care was devoted to the user interface, as can be seen in the distribution of code given in Table I, which shows how important the interface is. For a knowledge-based system, one might have expected that most of the space would be devoted to the explicit knowledge and inference engine required to manipulate it. However, together, explicit knowledge and inference engine consume only about 30 percent of the total memory. Implicit knowledge for feature detection makes up another 13 percent, while by far the largest identifiable portion, 42 percent, is devoted to the interface. This is not atypical: In other systems we know, such as Pride, one-third to one-half of the code is devoted to the user interface.

CRITERIA AND STAGES FOR EXPERT SYSTEMS

The success of expert-system development depends to a very great measure on an appropriate selection of applications [27]. In this section, we consider some of the most important criteria for choosing projects and outline the major developmental stages once a project is under way.

Criteria for Problem Selection

Value of Solving the Problem. Choose the application carefully. The problem that is being approached must be worth solving. One way of confirming this is to determine whether management is willing to commit the necessary human and material resources. Since it may take tens of person-years to make a system real, as in the R1 example, the value of the solution must be substantial. We are told that R1/XCON now repays its development cost every few months.

Alternative Solutions. Non-AI solutions to the same problem are a real possibility. A vivid illustration of the value of considering alternative solutions comes from the Darn project at Xerox [24]—a prototype expert system built to aid in the diagnosis and repair of a computer disk subsystem. The disk diagnosis and repair process was a complicated one, requiring a large number of tests to take into account the many possible failure modes, including electronic failures of several controller boards. According to technicians' records, diagnosing and repairing disk problems for this unit were consuming 30–50 percent of the resources of the repair group.

TABLE I. Distribution of Memory in the Dipmeter Advisor

Inference engine	8%
Knowledge base	22%
Feature detection	13%
User interface	42%
Support environment	15%

After working three months on the prototype, external factors caused a three-month hiatus in the project. When it was resumed, the Darn system prototype was no longer of value because this particular disk and controller had been phased out—as a result of all the problems—and replaced by a new disk with a single circuit board. In the case of a suspected disk failure, the single board was easily replaced; if the replacement did not solve the problem, the disk itself would be sent back to the manufacturer. This new disk strategy had so reduced the time required to fix disk problems that aid from an expert system was no longer of significant value, making the knowledge base specific to the original disk obsolete.

Test Cases. Another critical resource in the successful development of an expert system is a good set of test cases by which to extract knowledge from the experts. By watching experts actually solve problems, rather than just having them *describe* how they do it, it is possible to understand the real process that goes on and the actual knowledge that is used. A suite of test cases can also be used for testing implementations as they reach different stages in development.

Asking potential users early about test cases also provides valuable clues as to the real feasibility and value of a proposed system. One does not easily get test cases for problems that occur only once every six months, and there probably would not be enough commonality between cases. Moreover, where solution methods are radically different for each problem, too much knowledge is probably required in the system. For example, after we were asked to consider producing an expert system to design device controllers and briefly exploring the domain, it became apparent to us that each device controller was unique in its requirements, and therefore the project was rejected. One might say that expert-system technology is suitable for automating tasks that are fairly routine and mundane, not exotic and rare.

Task Difficulty. There are other rules of thumb that should be applied to tasks being considered as expert-system domains. A suitable task is probably one that would take an expert an hour or two, not counting the time spent on mechanical tasks like

making sketches or filling in forms. Tasks that take only a few minutes of expert time can probably be solved by simpler technology, or may not be worth automating at all; tasks that require more than a few hours of expertise are probably too difficult and unbounded in terms of the knowledge they require. Unless the longer tasks are iterations of a much shorter one, done several times in the same session, the longer tasks are generally too complex.

Another warning sign is a predominance of commonsense knowledge, formalization of which is just now being explored in the AI community. Unless very specific forms of this knowledge can be isolated, this is a tar pit that should be avoided. In general, problems that are known to require English-language understanding, complicated geometric or spatial models, complex causal or temporal relations, or understanding of human intentions are not good candidates for the current state of the art in expert systems.

Expert Help. A good indicator as to whether a problem is appropriate for expert-system implementation is the availability of one or more human experts who deal with the problem routinely as part of their job and therefore have the knowledge and experience to understand what the real problems are and to choose appropriate test cases. These experts should be available to work on the project for a significant portion of their time and must be able to articulate what they are doing when solving a problem.

The choice of experts is critical. Since an expert-system project requires dedication and long-term commitment, an expert who is only mildly interested in the problem is not the best choice; rather, the expert should have a strong vested interest in obtaining a solution. The expert must also understand what the problem is and have actually solved it quite often. It is not enough to have somebody with a theory about how cases like this should be handled or some good ideas about a new way to do things, or even an "eager and bright" beginner who will learn. A final trap that knowledge engineers sometimes fall into is believing, after working with the experts for some time, that they have become experts in the problem area.

Knowledge Engineering. A knowledge engineer interviews the experts and develops both an appropriate framework for the system and the initial representation for the knowledge. Both interviewing and developing representations are arts that require training, and experts cannot be expected to be their own knowledge engineers.

Expert knowledge is usually articulated around specific cases, where the analysis of expert knowledge often deepens the experts' own understanding of what they do. Smith on the dipmeter advisor project reports that one of the most useful outputs of this endeavor was a comprehensive document containing the rules, figures, and justifications for the dipmeter advisor knowledge.

However, to make knowledge useful in a system, one cannot just "extract" the knowledge from the expert; one must structure it in such a way that it bears on the whole range of expected cases. Working with the expert, the knowledge engineer must define the depth of representation, the expected limits of the system's explicit knowledge, the conditions under which the knowledge becomes inapplicable, etc. Knowledge is an artifact, worthy of design.

Stages in the Development of Expert Systems

An expert system generally goes through a number of developmental stages: identification, conceptualization, prototyping, creating user interfaces, testing and redefinition, and knowledge-base maintenance. Each stage requires different sets of individual talents and resources. Although the process has many similarities to classical systems analysis, the emphasis in this case is on the use of AI technology to capture and apply the appropriate knowledge.

Identification. At this stage, a critical mass might consist of one or two knowledge engineers and a group of experts who can identify problems amenable to solution through expert-system technology. Five to 10 test cases should be collected for later use. When the expertise is distributed among several experts, the interviewing process should expose their relative specializations and also the degree of consensus in solution methods.

After determining that the availability of experts and test cases is sufficient to warrant further exploration, one must develop a clear understanding of what is meant by success: This means identifying the actual users of the proposed system, some detailed examples of the problems to be posed, and acceptable solutions that might be generated.

Conceptualization. Once the domain has been identified, the next step is conceptualizing and formalizing the knowledge. Initial knowledge acquisition sessions should start with a single expert who can demonstrate by working on several examples what it means to solve a particular problem. Having developed some sense of what the problem is, the knowledge engineers can begin to articulate in a semiformal language what they believe is going on in the problem-solving sessions.

A useful next step for knowledge engineers is simulating the solving of one or more of the test cases by following the semiformal prescription. The expert

will quickly notice steps that have been left out or are incorrectly formulated. At first, each case will reveal important things that have been left out. The contents of the semiformal knowledge base will evolve from this testing as it becomes clear that terminology must be more precise and as exceptions appear to the initially simple rules.

After several rounds of simulation by knowledge engineers and critiquing by the single expert, it is often useful to bring in other experts who can be observed solving one or more of the same problems from the suite of test cases. This will help identify any particular expert idiosyncrasies in the problem-solving process, and determine whether there is one style or a multiplicity of problem-solving styles. In the case of the dipmeter advisor, it was learned late in the development of the system that the expert for the initial knowledge base had a very different style than a number of other experts in the community. Using multiple experts in the early knowledge acquisition sessions can prevent this kind of surprise.

In the Pride project, the initial knowledge acquisition sessions led to the creation of a "design knowledge document." This document outlined the different stages in the design, the dependencies between the stages, and a detailed rendering of the various pieces of knowledge (rules, procedures, constraints, etc.). Before the first line of code was written, this document had evolved to over 20 closely typed pages with over 100 pages of appendixes containing detailed mathematical analyses, charts, tables, and figures.

This document played a crucial role in defining and verifying the knowledge that was eventually incorporated in the Pride system. It was circulated among the experts, and errors and omissions were corrected. It was then used experimentally by the knowledge engineers to solve new design problems by strictly following the document. This helped make explicit some of the knowledge that had been implicitly applied by the experts.

Prototyping. In an expert system, many problems are not revealed until actual implementation since, unlike classical software projects, the exact specifications of what can be done, and how, are not known. Recognizing this, one should build a prototype system fully expecting to throw away virtually all this code and start again. The prototype should be made to work on the core of the problem, using a detailed typical example as its focus, and should include experiments with user interfaces. One should expect the cycles of prototyping to alternate with the development of useful packages, tools, and interface ideas.

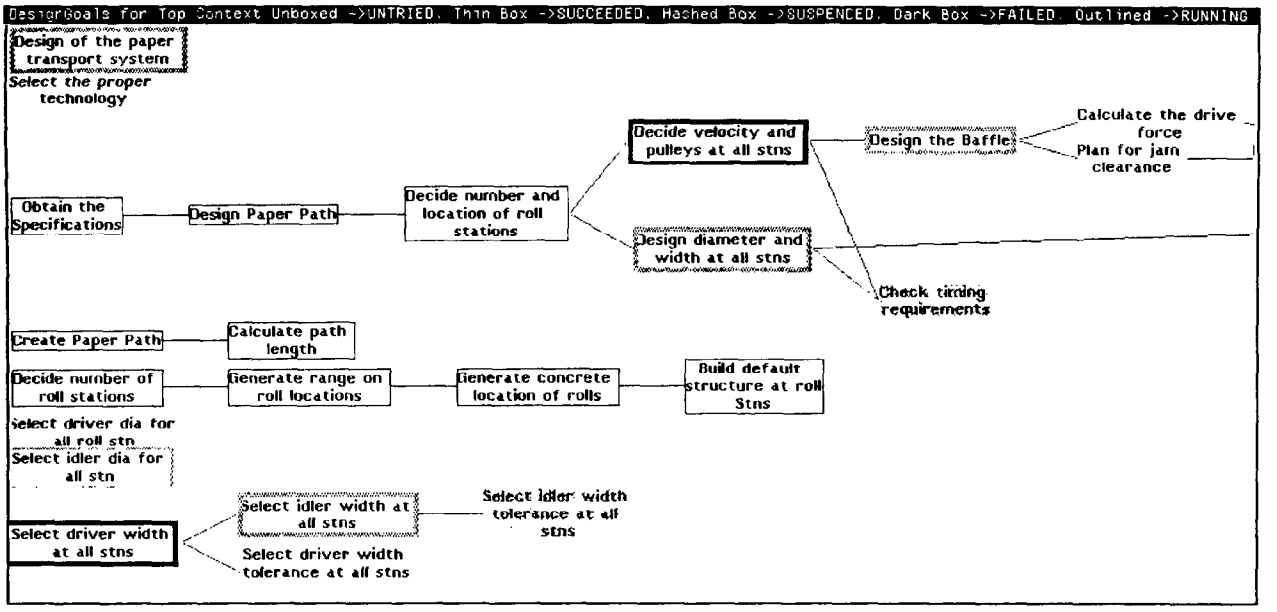
An implementation technology must be chosen for the prototype. There are fundamental choices to be made between off-the-shelf problem solvers (some of which are built around rule-oriented programming) and building a special-purpose system in a knowledge programming language that incorporates object-oriented programming, logic programming, or programming with frames. Other factors to be considered are the programming environment and the availability of consulting help with the tools. When using a commercially available implementation environment, it is appropriate to ask questions about the types of expert systems that have been built before and the levels of support that are available from the vendor for people building systems for the first time.

User Interfaces. One of the most important and time-consuming stages in the development of an expert system is the creation of a suitable user interface—particularly one that matches what users of the noncomputer system have been accustomed to.

A good example of a well-matched, well-defined user interface is that used by Oncocin, an expert system for cancer treatment therapy management developed at Stanford University [18, 30]. The Oncocin interface in effect replaced a paper form designed for use on the ward. The form contained sketches of the human figure for indicating body locations, and many lines for sequential entries related to each visit and treatment. The computer interface for Oncocin mimics many of the important features of the form so that doctors need not learn a whole new way of looking at their case data when using the system. It generates new lines for additional visits only on request (thereby shortening the form for inspection purposes) and allows specialized input techniques that make it easier for the doctor to enter data quickly and accurately. Together, these features helped considerably in promoting acceptance of the system.

The Pride system, on the other hand, presents two different (albeit related) user interfaces: The first is a goal browser (Figure 2) that lays out the design process as a network of different goals, and displays their status; the display is dynamically updated by the system as the exploration of the design space proceeds. The browser allows more detailed query of the goal structure through menu interaction on the screen: It allows the user to edit, undo, advise, and reexecute goals.

The second interface is the Active Design Document or "electronic design notebook" given in Figure 3, on page 890. It resembles in appearance the kind of report engineers typically prepare at the end



This browser shows the relationships between various goals—links to the right reflect explicitly known dependencies between goals. Boxes around the goals indicate whether a tried goal has succeeded, has failed, is suspended, or is running now.

FIGURE 2. Goal Browser

of a design cycle to record the major design decisions and analyses. However, the electronic version encapsulated in Pride also allows the user to view the design as it unfolds, to affect its course, and, in future versions, to obtain explanations and modify the design.

Testing and Redefinition. Once the prototype system has reached the stage where it is possible to go through the initial test problems from beginning to end, it becomes important to start testing the system with friendly users. This will sometimes reveal new problems that can cause a rethinking of what the entire project is supposed to accomplish.

In most cases, a second version of the prototype system will be built, and sometimes even a third version, since, in the process of building the system, the knowledge representation and inference procedures can be changed. Feedback from solving real problems often forces reimplementation—a cycle that is very characteristic of knowledge programming.

Knowledge-Base Maintenance. After friendly users have tried the system, a plan must be made for a *large* software project: The plan must provide for testing, development, transfer, and maintenance of the knowledge base. A process must be put in place at user locations to help tune the user interface and extend the knowledge base as new problems are found and easier ways to interact with the system are suggested. When this plan is complete, one can more easily evaluate the cost of the resources required versus the value of solving the problem.

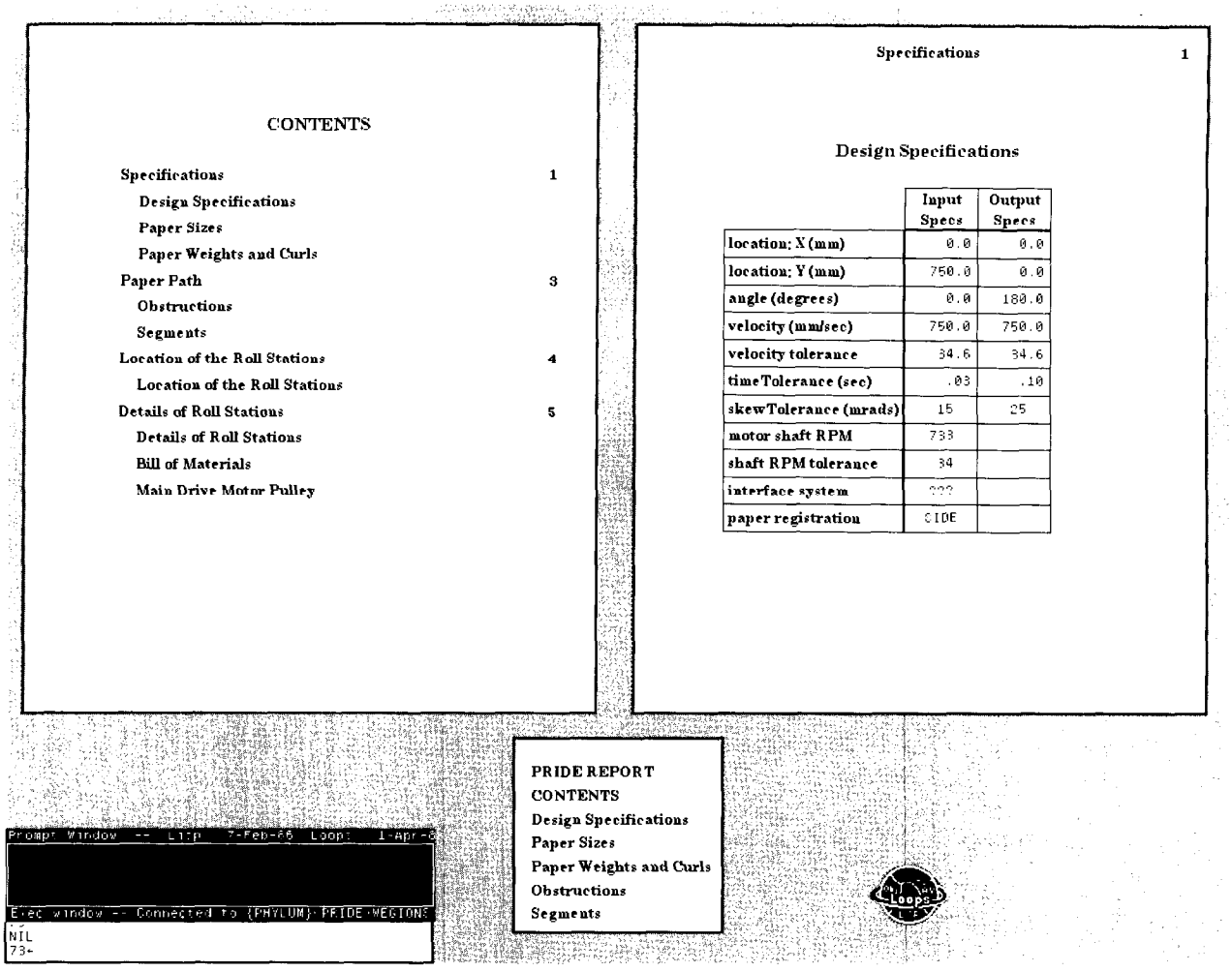
SOFTWARE TECHNIQUES AND THE KNOWLEDGE OF EXPERTS

In this section, we compare the AI programming environments—which have contributed to the flurry of activity in expert systems—with the techniques and languages of more conventional programming, showing how the differences derive from different assumptions about the kinds of problems for which they are intended.

The Fallibility of Expert Systems

Public opinion about AI is schizophrenic, ranging from "It will never work" to "It might cost me my job!" This range of attitude reflects a collective confusion about AI, where the very words themselves—artificial intelligence—raise questions about the nature of intelligence and the capabilities of machines [12].

Adding to the confusion are two well-known mathematical results that are popularly misunder-



This figure shows the table of contents and one page of the Pride electronic design notebook. Inputs from the user, and outputs from the design and analysis are both shown on these pages. By making this look like the hard-copy report, it is easy for designers to find their way around. By allowing input through the document, the users can obtain control the same way they see results.

FIGURE 3. Active Design Document

stood as implying that computers cannot achieve humanlike intelligence. In the 1930s, Kurt Godel's "incompleteness theorem" and Alan Turing's "incomputability" theorem [22] showed that there are well-defined problems that cannot be solved by any computational procedure. These theorems relate to the enumeration of mathematical sentences in self-referential systems and involve the demonstration that no program can infallibly determine whether all computer programs will eventually halt (i.e., be solved).

However, neither the halting problem nor the incomputability results go very far in demonstrating the limitations of computers vis-à-vis humans: We do not really know the limits of *people* for the same kinds of problems. In this sense, the popular misinterpretation of computing theories is very much like the popular misconception about theories from physics, where relativity and the uncertainty principles are often incorrectly interpreted as implying that everything is relative and nothing is certain.

In the theoretical work underlying AI, a major theme has been trying to understand how a computationally limited machine can behave intelligently at all. This has led to models of resource-limited reasoning (e.g., [34]) and to models of belief and knowledge (e.g., [13]). This work is generally regarded by AI researchers and cognitive psychologists as providing computational insights into the kinds of mental processes carried out by humans.

In pragmatic work on expert systems, it is commonly believed that *knowledge* is the key to intelligence. Indeed, the slogan "knowledge is power," attributed to Feigenbaum, is almost a guiding principle for work in knowledge engineering. It changes the emphasis from a search for general mechanisms of intelligence to the development of techniques for knowledge acquisition and representation.

Software Techniques

Broadly speaking, conventional programming practice offers two schools of thought [16] about techniques for developing reliable and predictable programs. One approach is to develop specification languages and validation procedures, where programs can be annotated with specifications like the type of input values and invariants that are intended to hold at different points in program execution. The task of the programmer or specification processor then becomes ensuring that the program satisfies the specifications.

Unfortunately, the challenge in most expert-system applications lies elsewhere, that is, in getting the specifications right in the first place, rather than simply checking the correspondence between specifications and programs. This brings us back to the phrase *exploratory programming*, which is often used to characterize AI programming environments. These environments are intended to shorten the time of the development cycle in which parts of a program are entered, tested, debugged, and modified. A systematic analysis of the costs and benefits of a prototyping approach is beyond the scope of this article, but results and criteria have been reported elsewhere (e.g., in [1]).

The notion of a development cycle brings us to a second school of thought in programming practice—software engineering, a discipline that grew out of the experience of software vendors trying to deliver products. Although there is no single dominating approach for managing the program "life cycle," the general principles include the process of obtaining feedback from clients at several different stages and respect for the notion that a program is seldom finished, meaning that it is necessary to plan for continued maintenance and service as requirements change.

A development cycle for building expert systems (e.g., [8]) is usually described in terms that are somewhat foreign to traditional software engineering practice: AI people, for example, talk about "knowledge acquisition" and "knowledge bases." Over the past two to three years, AI companies have gained considerable experience in delivering expert systems to customers. Teknowledge, one of the AI companies that delivers expert systems, used to include the following as a closing slide in their presentations about expert systems:

"Knowledge engineering is more than software engineering."

Recently, an additional line has been added:

"(But not much more.)"

This reflects, perhaps, a fresh convergence of thinking and experience.

Because tools are so important in the development of expert systems, providing tremendous amplification in the programming arena, many companies are now selling expert-system environments. However, it is important to remember that these tools themselves are not expert systems. Moreover, vendors sometimes overstate the power of the tools and underestimate the work required to create systems that are able to solve real problems. Despite the almost religious zeal that exists among sellers of various systems, it is important also to recognize that a single knowledge representation formalism is usually not sufficient for complete problems [3]. "Everything can be expressed in logic" or "Everything can be expressed in rules" is like saying "Everything can be expressed in English." While these statements may have a certain element of truth to them, this kind of statement obscures the practical issues of knowledge representation, such as "Can the knowledge be expressed concisely in symbols (or pictures)?" and "Can it be organized so that it can be changed easily?" A good environment will provide a number of different, complementary, and integrated formalisms to allow easy description of different kinds of knowledge.

The Knowledge of Experts

To better appreciate the limitations and enabling conditions for building expert systems, we describe here some basic intuitions guiding the field.

- Many problems can be solved by applying large amounts of appropriately structured knowledge.
- People become experts at something by starting with some basic knowledge. By applying that knowledge to solve problems, they create new knowledge structures more appropriate to the problems.
- Where task-specific knowledge structures exist, they can be documented by suitably questioning or otherwise testing the experts in relation to certain specific problems. Knowledge thus extracted can then be represented in a computer program and manipulated by some inference engine or

problem-solving framework to solve similar problems in much the same way as do the experts from whom the knowledge was obtained in the first place.

Given these intuitions, there are some basic rules of thumb that govern the acquisition of expert knowledge.

Expert Knowledge Is Expensive. A commitment of considerable time from knowledgeable people is essential to the development of expert systems. Usually, the people who make the best experts are the ones most highly valued by their own organizations and therefore in some respects the least accessible. In the case of Caduceus, a system for medical diagnosis developed by Harry Pople and Jack Meyers [26], one of the top diagnosticians in internal medicine in the United States, the success of the system is a direct result of the process put in place by Meyers to create and test a high-quality knowledge base. Meyers made use of his top medical students to gather the knowledge.

Among the experts behind Dendral (an early expert system described more fully on p. 881) were Joshua Lederberg, a Nobel-prize winning geneticist, and Carl Djerassi, a world-class expert on mass spectral analysis. For its day-to-day development, Dendral also consumed the talents of several professional chemists and computer scientists.

The point of these examples is that getting the attention of knowledgeable people for the period of time necessary to build expert systems represents a very tangible and not inconsiderable cost. However, the willingness and availability of experts to participate directly and strongly in the project are prerequisites for success.

Expert Knowledge Is (usually) Not in Textbooks. One trap awaiting the unwary is the expectation that textbook knowledge is the right stuff for incorporating into an expert system. Textbooks are not bad or incorrect; the problem is the great deal of practical material—obvious to an expert—that never finds its way into textbooks. Most textbook knowledge is too idealized: For applications of real interest, only an expert knows the messy but necessary details of real problems and the unpublished rules of thumb.

Furthermore, textbooks are often not designed to teach either problem-solving skills or knowledge structuring techniques, both of which seem to be essential to the expert. For example, prior to the development of Dendral, chemistry books gave examples of the interpretation of mass spectral data, but never said how to systematically enumerate the set of possible molecules. In fact, developing Dendral required the construction of some sophisticated mathematical theories not previously known.

Most textbooks try to provide the fundamentals of the subject matter. In the process of becoming an expert, people learn to use various elements of textbook knowledge to create knowledge structures that are more suitable for problem solving. Experts use these specialized knowledge structures in the common cases and tend to reason from fundamentals only in the difficult and unusual cases.

Expert Knowledge Has to Be Acquired Incrementally and Tested. Expert knowledge is not acquired all at once: The process of building an expert system spans several months and sometimes several years. In the course of this development, it is typical to expand and reformulate the knowledge base many times. In the beginning, this is because choosing the terminology and ways of factoring the knowledge base is subject to so much experimentation. In the middle phases, cases at the limits of the systems capabilities often expose the need to reconsider basic categories and organization. Approaches viable for a small knowledge base and simple test cases may prove impractical as larger problems are attempted.

The need to modify and change the knowledge base provides an enormous incentive to use and create knowledge programming tools. Most knowledge programming development systems provide a suite of tools for browsing and summarizing a knowledge base, tracing and explaining chains of reasoning, instrumenting program state, and graphing program structure. These tools make it simpler to both "put knowledge in its place" and to understand how changing a particular piece of knowledge will affect the overall behavior of a system. Tools for analyzing and reorganizing expert systems are essential for developing "middle-road" systems.

Even so, expert systems take time to build. Toy programs for a small demonstration can be built quickly—often in just a few months using current technology. However, for large-scale systems with knowledge spanning a wide domain, the time needed to develop a system that can be put in the field can be measured in years, not months, and in tens of worker-years, not worker-months.

Expert Knowledge Is Sometimes Distributed. Experience with systems that have survived the feasibility demonstration stage suggests that reliance on any single expert can either create blind spots in the knowledge base or result in a system that will not have users.

Problem solving often takes place in a community where many different experts pool their expertise. The medical domain, for example, has institutional-

ized specialties such as pathology, radiology, gynecology, and internal medicine, for which there are separate training and certification requirements. Furthermore, a protocol has developed for referring patients to another specialist when the problem reaches the limits of any particular expert's knowledge.

Collectively, these observations point to the need for creating community knowledge bases that integrate expertise from many different sources. The MDX system [9] was an early effort at integrating different kinds of expertise (diagnosis, pathology, radiology) in the same system. The Pride system integrates design and analysis knowledge. However, we are just beginning to understand how to cope with a single expert, and much more work is needed before community knowledge bases can be easily crafted.

Thoughts on the Futures of Expert Systems

One of the dreams of the expert-system community is to eventually have knowledge bases created and maintained by their users rather than by knowledge engineers—a dream that reflects certain financial realities. The maintenance cost for expert systems is substantial. Once the initial thrill of a prototype system and a fancy interface wears off, some projects come abruptly to an end as the expense of developing them further and maintaining them is assessed all too belatedly.

Although the dream of community knowledge bases is well beyond the current state of the art, work on expert-system shells is leading in that direction [32]. An expert-system shell is an environment designed to support applications of a very similar nature and represents an intermediate point between specific applications and general-purpose knowledge engineering environments. Shells could be built for such applications as diagnosis, design, planning, scheduling, and a variety of specialized office tasks. Shells contain several things that knowledge engineering tools do not: prepackaged representations for important concepts, inference and representation tools tuned for efficient and perspicuous use in the application, specialized user interfaces, generic knowledge for related applications, and specialized tools for acquiring and testing knowledge for the application.

A shell for a planning application would have representations that integrate time with possible worlds and beliefs; user interfaces for describing plans and alternatives; generic categories for things like time, tasks, and serially reusable resources; and some generic domain knowledge like the fact that an agent can be in only one place at a time. Shells provide the ability to share and standardize knowledge in larger communities than single expert-system projects.

Advanced shells are just beginning to appear in the research laboratories. They are a first step toward what may be the ultimate promise of expert systems: the creation of a medium for the widespread production, consumption, and distribution of human knowledge [32].

CONCLUSIONS

The expression *expert system* has been used as a buzzword for funding and a flag to wave for all sorts of projects. In this article, we have tried to capture some of the relevant experience with systems that clearly fit our model, rather than trying to define precisely what is or is not an expert system. The emphasis here has been on the explicit representation of knowledge in a system, rather than system performance. Therefore, no matter how "clever" a C-Compiler is, we would not call it an expert system if it were not built in this style (although one could imagine a C-Compiler built that way). Expert systems need not be rule based or contain a theorem prover (e.g., Pride) and can be built in any language (e.g., Pascal or C), although it is easier where language and environment readily support the conversion of user's intent into programs [5]. Since building an expert system is a large software project, standard software engineering analyses are useful in understanding the scope and nature of the job.

Expert systems are neither the answer to all questions nor the answer to none. One can build expert systems for appropriate problems—ones that are valued, bounded, routine, and knowledge intensive—provided experts are available who are articulate, patient, and committed to a project for at least the initial phases. One should also ensure that appropriate hardware and software are available to the developers. Finally, if the initial prototyping is successful, it is important to determine whether management is willing to commit the resources it will take to make the demonstration system into a useful system for everyday use.

Acknowledgments. We would like to thank Peter Denning and Reid Smith for comments on earlier drafts of this article.

REFERENCES

1. Alavi, M. An assessment of the prototyping approach to information systems development. *Commun. ACM* 27, 6 (June 1984), 556–563. A comparison between a life-cycle approach and a prototyping approach to system development.
2. Bachant, J., and McDermott, J. R1 revisited: Four years in the trenches. *AI Mag. 5*, 3 (Fall 1984), 21–32. A cogent summary of the building of one of the best-known expert systems.

3. Bobrow, D.G. If Prolog is the answer, what is the question?, or what it takes to support AI programming paradigms. *IEEE Trans. Softw. Eng. SE-11*, 11 (Nov. 1985), 1401–1408. A critique of different programming styles used to support AI programming.
4. Bobrow, D.G., and Stefik, M. *The Loops Manual.* AI Systems, Xerox Corporation, Palo Alto, Calif., 1984. A description of a multiparadigm system used in expert-system development.
5. Bobrow, D.G., and Stefik, M. Perspectives on artificial intelligence programming. *Science V231*, 4741 (Feb. 28, 1986), 951. An account of the programming styles and environments that are used to develop systems in the AI community.
6. Brown, J.S. The low road, the middle road, and the high road. In *The AI Business*, P.H. Winston and K. Prendergast, Eds. MIT Press, Cambridge, Mass., 1984. Alternative approaches to the development of intelligent systems: their advantages and pitfalls.
7. Buchanan, B., and Shortliffe, E.H. *Rule Based Expert Programs: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, Reading, Mass., 1984. An account of 10 years' experience developing rule-based expert systems, using the MYCIN medical diagnosis system as the primary example.
8. Buchanan, B., Barstow, D., Bechtel, R., Bennett, J., Clancey, W., Kulikowski, C., Mitchell, T., and Waterman, D.A. Constructing an expert system. In *Building Expert Systems*, F. Hayes-Roth, D.A. Waterman, and D.B. Lenat, Eds. Addison-Wesley, Reading, Mass., 1983. An account of the developmental stages of an expert system.
9. Chandrasekaran, B., and Mittal, S. Conceptual representation of medical knowledge for diagnosis by computer: MDX and related systems. In *Advances in Computers*, Vol. 22, M.C. Yovits, Ed. Academic Press, New York, 1983, pp. 218–295. An approach to partitioning medical knowledge, and how this approach was used in building a diagnostic system.
10. Davis, R. Knowledge based systems. *Science V231*, 4741 (Feb. 28, 1986), 957. A general review of the concepts and current status of building knowledge-based systems.
11. de Kleer, J. How circuits work. In *Qualitative Reasoning about Physical Systems*, D.G. Bobrow, Ed. MIT Press, Cambridge, Mass., 1985, pp. 205–280. A description of a qualitative reasoner that uses constraints to build models of how circuits do and do not work.
12. Denning, P.J. The science of computing. *Am. Sci. 74*, 1 (Jan. 1985), 18–20. Nontechnical overview intended to explain expert systems to general scientists.
13. Fagin, R., and Halpern, J.Y. Belief, awareness, and limited reasoning: Preliminary report. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. 1985, pp. 491–501. The theoretical foundations underlying the formalization of commonsense understanding of differences such as belief and knowledge.
14. Forgy, C.L. The OPS5 user's manual. Tech. Rep., Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1980. The manual describing one of the primary languages used in expert-system development (and used in the R1 system).
15. Friedland, P. Special section on architectures for knowledge-based systems. *Commun. ACM 28*, 9 (Sept. 1985), 902–903. Introduction to a special issue on programming techniques for building knowledge-based systems.
16. Goldberg, A. About this issue . . . (special issue on the software development process). *Comput. Surv. 14*, 3 (Sept. 1982), 319–320. An overview of the life cycle of program development.
17. Kunz, J., Kehler, T., and Williams, M. Applications development using a hybrid AI development system. *AI Mag. 5*, 3 (Fall 1984), 41–54. Describes a commercially available multiparadigm tool useful for expert-system development.
18. Lane, C.D., Walton, J.D., and Shortliffe, E.H. Graphical access to a medical expert system: II. Design of an interface for physicians (Memo KSL-85-15). *Methods Inf. Med. 25* (1986). Describes the development of the user interface for the Oncocin system.
19. Lindsay, R.K., Buchanan, B., Feigenbaum, E., and Lederberg, J. *Applications of Artificial Intelligence for Organic Chemistry.* Kaufman, Los Altos, Calif., 1985. The history of the Dendral project, one of the earliest AI expert systems.
20. McDermott, J. R1: The formative years. *AI Mag. V2*, 2 (Spring 1981), 21–29. An easy-to-read description of the trials and tribulations of moving an expert system from a university to business environment.
21. McDermott, J. R1: A rule-based configurer of computer systems. *Artif. Intell. 19*, 1 (Jan. 1982), 39–88. Technical description of the first commercially successful expert system for building configuration descriptions of computer systems.
22. Minsky, M.L. *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, N.J., 1967. A good basic text on the formal properties of different classes of machines and the theoretical limitations of computation.
23. Mittal, S., and Dym, C. Knowledge acquisition from multiple experts. *AI Mag. 7*, 2 (Summer 1985), 32–36. Describes the approach used in the Pride project for integrating expertise from multiple experts.
24. Mittal, S., Bobrow, D.G., and de Kleer, J. DARN: A community memory for a diagnosis and repair task. Xerox ISL Lab. Rep., Xerox Corporation, Palo Alto, Calif., 1985. An approach to building a knowledge base that will aid people doing hardware repair.
25. Mittal, S., Dym, C., and Morjaria, M. Pride: An expert system for the design of paper handling systems. In *Applications of Knowledge-Based Systems to Engineering Analysis and Design*, C.L. Dym, Ed. American Society of Mechanical Engineers, New York, 1985. A general description of Pride, an expert system that aids in the design of paper handling systems.
26. Pople, H.E. CADUCEUS: An experimental expert system for medical diagnosis. In *The AI Business*, P. Winston and K. Prendergast, Eds. MIT Press, Cambridge, Mass., 1984. Describes the genesis of a medical diagnostic system that is based on the knowledge of one of the best internal medicine specialists in the United States.
27. Prerau, D.S. Selection of an appropriate domain for an expert system. *AI Mag. 7*, 2 (Summer 1985), 26–30. An account of the criteria used to select expert-system tasks in GTE laboratories.
28. Sheil, B. Power tools for programmers. In *Interactive Programming Environments*, Barstow et al., Eds. McGraw-Hill, New York, 1984, pp. 19–30. A description of the need for tools that support the exploratory program process.
29. Shore, J. *The Sachertorte Algorithm.* Viking Press, New York, 1985. A nontechnical book describing programming and computers.
30. Shortliffe, E.H., Scott, A.C., Bischoff, M.B., Campbell, A.B., van Melle, W., and Jacobs, C.D. ONCOCIN: An expert system for oncology protocol management. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*. Menlo Park, 1981, pp. 876–881. A good example of how a well-designed user interface facilitates expert-system acceptance.
31. Smith, R. On the development of commercial expert systems. *AI Mag. V5*, 3 (Fall 1984), 21–34. A description of the development of the dipmeter advisor, an expert system for geologic analysis.
32. Stefik, M. The next knowledge medium. *AI Mag. 7*, 1 (Spring 1986), 34–46. An account of the limited impact of AI so far and the possibilities and hurdles involved in creating more widespread participation in a computer-based knowledge medium.
33. Stefik, M., and Bobrow, D.G. Object-oriented programming: Themes and variations. *AI Mag. 6*, 4 (Winter 1986), 40–62. A general overview of the different approaches to object-oriented programming.
34. Winograd, T. Extended inference modes in reasoning by computer systems. *Artif. Intell. 13* (Winter 1980), 5–26. A very readable account of the need to extend conventional approaches to reasoning to expand the capabilities of AI systems.

CR Categories and Subject Descriptors: D.2.1 **[Software Engineering]**: Requirements/Specifications—*languages, methodologies, tools*; D.2.2 **[Software Engineering]**: Tools and Techniques—*user interfaces*; I.2.1 **[Artificial Intelligence]**: Applications and Expert Systems; I.2.4 **[Artificial Intelligence]**: Knowledge Representation Formalisms and Methods; K.6.1 **[Management of Computing and Information Systems]**: Project and People Management—*life cycle, staffing, system analysis and design*

General Terms: Design, Languages, Management

Additional Key Words and Phrases: case studies in AI, knowledge-based systems, knowledge programming

Received 11/85; accepted 4/86

Authors' Present Address: Daniel G. Bobrow, Sanjay Mittal, and Mark J. Stefik, Intelligent Systems Laboratory, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.