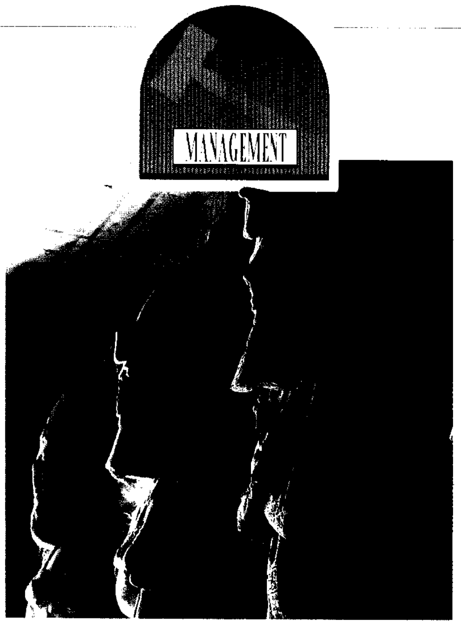

AI and Expert System Myths, Legends, and Facts

Mark S. Fox
Carnegie Mellon University

The "discovery" of new technologies often gives rise to misconceptions regarding their nature, use, and effectiveness. Such is the case with AI, which concerns the development of theories and techniques required for a computational engine to efficiently perceive, think, and act with intelligence in complex environments.

First, this definition incorporates the activities of receiving information, reasoning with it, and acting upon the world. Second, problems should be complex enough that they cannot be solved in a straightforward algorithmic way. Third, theories must be operational on a computing engine; that is, they must work efficiently on some type of machine.

While AI research has been underway for more than three decades, it is only in the past six years that AI's impact has been measurable. Many notable and well-publicized successes have occurred — for example, XCON,[1] Genaid,[2] and Opgen.[3] There have also been failures. The question is whether these failures are due to (1) technological limitations, or (2) the process by which technology has been applied. Without question, AI technology has limitations — it is no panacea. But have its failures been due to these limitations?

This article seeks to identify causes leading to ineffective AI applications. Views expressed herein are based on personal observations made over 10 years of building, deploying, and reviewing AI-based systems. Simply put, many failures with which I am familiar have been caused by one or both of the following: misconceptions regarding the nature of AI technology; or poor management skills in acquiring, nurturing, and applying that technology.

To manage and use AI, it is imperative that we understand its capabilities and limitations. My approach in identifying problems and misperceptions will be to examine questions and assertions that have been posed to me over the years. I have grouped these into three categories:

- **Myths** — Perceptions not based on any fact.
- **Legends** — Perceptions, once based on fact, that have been blown out of proportion.
- **Facts** — Perceptions that have a real basis in fact.

By identifying these perceptions and misperceptions, it is my hope that we can use AI technology selectively and successfully to increase the quality and productivity of decision making.

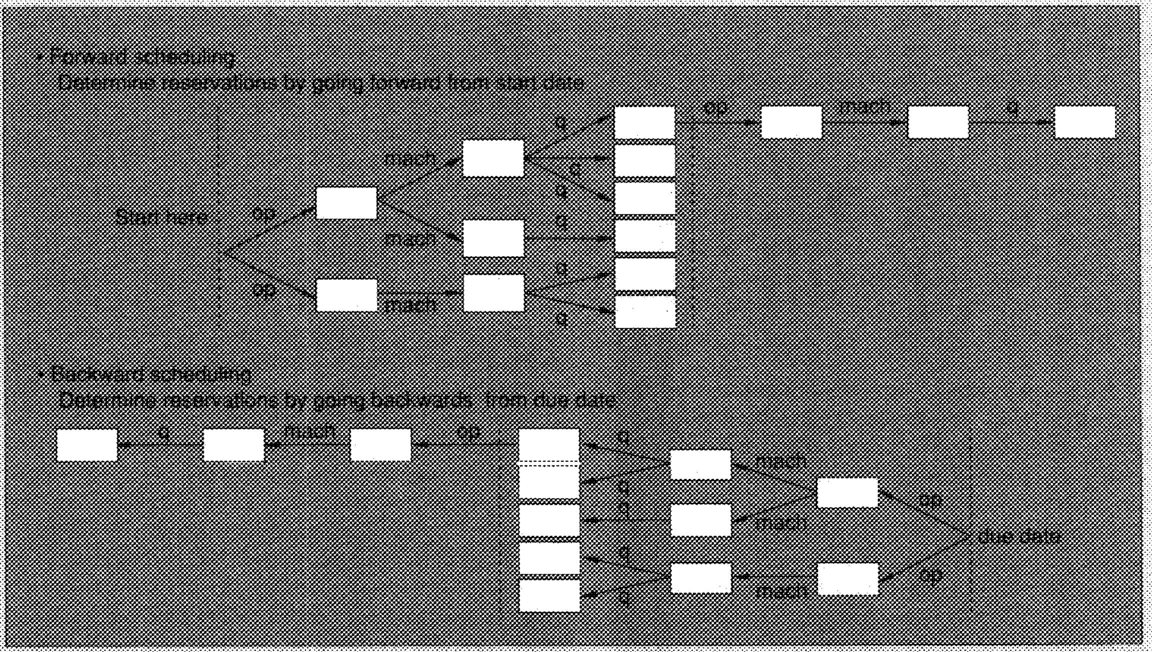
• Forward scheduling
Determine reservations by going forward from start date
Start here
op
mach
q
• Backward scheduling
Determine reservations by going backwards from due date
due date

Figure 1. The problem space for factory scheduling.

Core AI concepts

First, let's explore concepts at the core of AI.

Fact — Search is a core AI concept. Newell and Simon's physical-symbol system best articulates the key insight from which AI draws its strength.[4] A physical-symbol system — having the necessary and sufficient means for general intelligent action — is a machine that (1) "produces through time an evolving collection of symbol structures," and (2) can designate and interpret. From this we derive the heuristic-search hypothesis: Symbol structures represent solutions to problems. A physical-symbol system exercises its problem-solving intelligence by search — that is, by generating and progressively modifying symbol structures until it produces a solution structure.

The problem space, making the concept of a physical-symbol system operational, is composed of

- **States** — Collections of features that define some situation;
- **Operators** — Which transform one state into another; and
- **An evaluation function** — Which rates each state in the problem space.

Search begins at an initial state, and problems are solved when a path is found from the initial state to a goal state. Consider the factory-scheduling problem:

- **There are *n* jobs;**
- **Each job has a process graph, each node in the graph represents an operation, and each path through the graph is a possible process plan;**
- **Each operation requires zero or more resources, including machines, tooling, and material;**
- **Alternatives can be specified for each required resource;** and
- **Jobs enter the factory at random times and have varying due dates.**

Planning selects and sequences activities such that each activity achieves one or more goals and satisfies a set of domain constraints. Scheduling selects among alternative plans, and assigns resources and times for each activity that (1) obey temporal restrictions of activities and capacity limitations of a set of shared resources, (2) balance a conflicting set of preferential concerns (for example, reduce setups, prioritize activities, and the like), and (3) optimize a set of objectives (for example, minimize tardiness and work-in-process).

Schedules can be constructed incrementally (see Figure 1). Starting with a single order, an operator generates alternative first operations. Then, for each operation, it generates alternative machines on which to perform that operation — and, for each machine, it generates alternative queue positions (that is, times to perform the operation). Other alternatives may exist — including alternative shifts and substitute tooling and fixtures — that expand the tree into a larger network. Once the first operation is fully defined, search proceeds with the next operation. Search can be performed in a forward manner or in a backward manner, starting from the due date.

This is a very simple search example. But in a single factory having 85 orders, 10 operations, and only one substitutable machine, we could create over 10^{880} alternative schedules. Therefore, while this style of search is theoretically interesting, it is impractical because too many alternatives must be considered. Search must be smarter; in fact, search must be structured such that the search space is reduced from 10^{880} to something much smaller and more manageable.

Fact — AI reduces search combinatorics by applying situational knowledge. Feigenbaum provides an important AI insight[5] — namely, that we can reduce the combinatorics of moving through the problem space by the smarter selection of operators (that is, by utilizing domain knowledge). Domain knowledge can be represented as elaborations of operator conditions (that is, descriptions of situations in which a particular decision is to be made), which makes the application of operators more sensitive to the current context.

Operators are opportunistically applied; at each step in the search, the operator best matching the current situation is chosen to extend the search. Transition from one state to another in AI-program problem spaces is rational; at each step, the next step is opportunistically made (jumping from one island of certainty to another and from one decision process to another within the problem space).

Returning to our factory-scheduling example — rather than generate all possible alternative operations every time we schedule, expertise suggests that we consider alternatives only when capacity becomes scarce on machines used in the standard plan:

```
IF          Milling is to be scheduled
            AND Capacity is scarce
            AND Queue time is high
            AND Due date is tight
THEN        Consider
                  1. Grinding
                  2. Subcontracting
```

Many successful systems use expertise to reduce search combinatorics (for example, XCON and Opgen). But in the case of factory scheduling, the expert system approach presents two problems:

(1) Factory scheduling tends to be so complex that it is beyond the cognitive capabilities of human schedulers. Therefore, schedules produced by such schedulers are poor. Nobody wants to emulate their performance.

(2) Even if the problem is relatively simple, factory environments change often enough that expertise built up over time becomes obsolete.

Fact — AI reduces search combinatorics by reformulating problems. Expert systems seem appropriate only when problems are relatively small and stable or can be decomposed into such subproblems. When the scheduling problem cannot be solved using expert systems, we need more sophisticated search techniques. One approach reformulates the problem as a simpler task whose solution can be used to guide the original problem's solution.[6-8]

For example, the Isis-2 factory-scheduling system[8] separates search into four levels — job selection, capacity analysis, resource analysis, and resource assignment. Level 1 selects the next unscheduled job to be added to the existing shop schedule, according to a prioritizing algorithm that considers job type and requested due dates. The selected job is passed to Level 2 for scheduling.

Level 2 — representing the simpler reformulation of the original problem — simplifies the problem by removing resources and constraints from consideration, and performs a dynamic programming analysis of the plant (based on current capacity constraints). Level 2 determines the earliest start time and latest finish time for each selected operation, as bounded by the order's start and due dates. Times generated at this level are codified as operation-time-interval constraints, which influence search at the next level.

Level 3 solves the original scheduling problem. Beam search is performed using a set of search operators. The search space to be explored is composed of states representing partial schedules. Poor alternatives are pruned, based on local constraints and those generated at Level 2. Level 3 selects a specific routing for the job, and assigns reservation time intervals to resources required to produce that job. Level 3 outputs reservation time intervals for each resource required for operations in the chosen schedule.

Level 4 establishes actual reservations for resources required by selected operations that minimize the job's work-in-process time. This approach works well when the factory provides adequate capacity. But in situations where contention for resources is high, system performance is no better than that of human schedulers.

Fact — AI enhances search through the use of opportunism. When resources are highly contended for, experience shows that optimizing resource allocation by scheduling operations incident with the resource produces better results than does scheduling jobs one at a time. Opportunistic search,[9] as employed in scheduling,[10,11] extends hierarchical search by first analyzing capacity requirements for all jobs (to identify whether a high degree of resource contention exists). If contention does exist for a resource, a resource-centered scheduler is chosen to schedule operations incident with the resource. The job-centered scheduler schedules jobs out from the resource. Opportunism arises from the system's ability to (1) dynamically determine (at any point during schedule

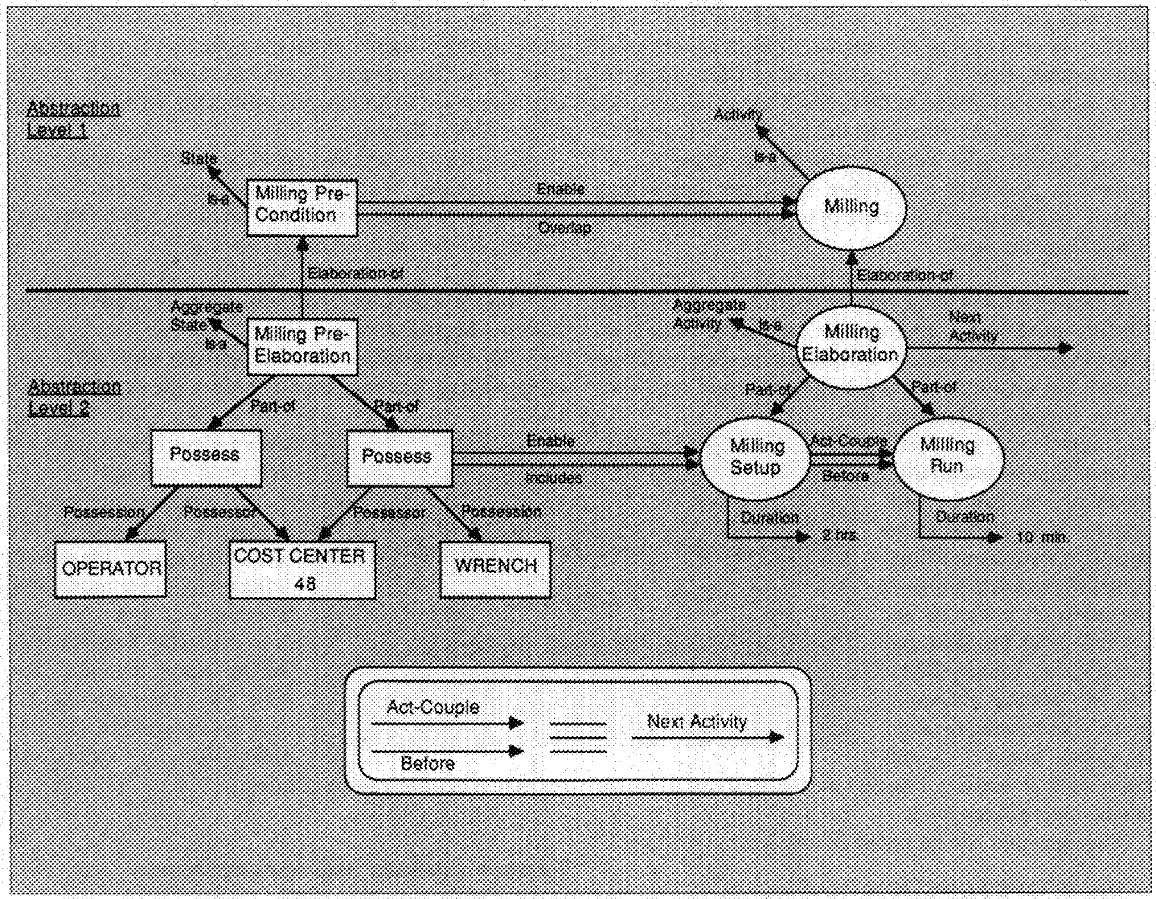

Figure 2. An activity semantic network.

construction) primary and secondary bottlenecks, and (2) schedule them rather than pursuing a strictly job-centered methodology.

Experiments with this approach show that it outperforms hierarchical search. In addition, it can solve typically complex factory-scheduling problems efficiently.

Fact — Knowledge representation is a core AI concept. AI's problem-solving power derives from its ability to perform search and from the richness of patterns that it can reason with and about. Pattern representation is a major research focus known as knowledge representation. Knowledge representation research focuses on developing ontologies and semantics that

- **Span the set of concepts that are required to solve a problem;**
- **Represent concepts precisely and unambiguously at all granularity levels;**
- **Provide a single representation understood and used by more than one application (other, more efficient representations may be constructed from this representation as required by an application);** and
- **Can be easily understood by the people who construct them.**

Consider the following factory activity description: The milling operation precedes the drilling operation and is composed of two steps, setup and run. Setup takes one hour. Runtime is 10 minutes. Two resources are required, a five-pound wrench and an operator. The wrench is required only during setup. The operation is performed in cost center 48.

Figure 2 shows how an AI knowledge representation represents the contents of the preceding paragraph. It is relational in form, and divides knowledge into two types — activity and state. A state describes a snapshot of the world before an activity is performed. For example "cost center 48 possesses a wrench" is a state description. It must be true for milling activity to occur. Causal relations link states and activities. A state describes what must be true of the world to enable an activity to occur. In addition, activities can be defined at multiple abstraction levels. Milling is refined into two subactivities, machine setup and running. Lastly, we must represent time. Setup, in time, occurs before the milling run. Time is relative, not absolute. When describing the factory floor, it is atypical to use absolute time periods; instead, activities are described as preceding each other. Once the time of one activity is determined, the time of all activities related to it can be determined.

Fact — AI knowledge representation extends quantitative models by abstraction and differentiation. Knowledge representations are qualitative abstractions of underlying quantitative models; they answer questions that an underlying quantitative model may not. Let's assume that a quantitative model of time has, for each activity to be performed, a specified start and end time. With this model, the following question could be answered: Find the start and end time of Activity 2 that has the longest duration, given start and end times of Activities 1 and 3 — and knowing that Activity 1 occurs during Activity 2, which occurs during Activity 3.

Experts solve problems by searching a problem space.

This question can be stated as follows:

OBJECTIVE FUNCTION: MAX $et_2 - st_2$
CONSTRAINTS:

$$\begin{array}{lll} st_1 < et_1 & st_1 >= st_2 & st_2 >= st_3 \\ st_2 < et_2 & st_1 <= et_2 & st_2 <= et_3 \\ st_3 < et_3 & et_1 <= et_2 & et_2 <= et_3 \end{array}$$

WHERE
st_1, et_1, st_3, et_3 are known
ALGORITHM: Simplex

We can view Allen's relational model of time as an abstraction of a quantitative model.[12] In the relational model, the temporal relation *during* denotes that one activity is performed during the time that another activity is performed. This relationship can be asserted without knowing the actual times of each activity. If we did not know any of the start times or end times for each activity — but knew only that Activity 1 occurs during Activity 2, which occurs during Activity 3 — we could still answer the question: Does Activity 1 occur during Activity 3 (stated as follows)? —

GIVEN: A_1, A_2, A_3 are activities, and

$$A_1 \xrightarrow{d} A_2 \xrightarrow{d} A_3$$

where d is the *during* relation and is transitive,
ALGORITHM: Hypothesis Introduction

An abstraction is a partial model of an underlying and more complete model. In fact, we can construct a continuum of partial models. As demonstrated, each partial model enables us to answer a different set of questions more or less efficiently.

AI representations also enable us to differentiate concepts contained in a quantitative representation. For example, many types of temporal relations can be defined between activities in addition to *during*:

- ***Before*** **specifies that one activity ends before another begins;** and
- ***Overlap*** **specifies that one activity ends after another begins, but that it ends before the other does.**

Allen identifies 13 types of temporal relation.[12]

Differentiating among concepts contained within (but not easily identified in) a quantitative model is a requirement when we are constructing knowledge-based search; the more precise the representation is, the more specific is the definition of situations for which actions can be defined.

Expert system misconceptions

Now, let's examine some common myths and legends surrounding expert systems, their building, and their maintenance.

Myth — Expert systems differ from AI-based systems. What is an expert system? The term has never really been defined.

Is it determined by a technology such as rules? Clearly, we would not consider a simple, rule-implemented accounting program as an expert system. If it were one, then any program written in a rule-based language would be an expert system.

Is it a level of performance — that is, are systems that operate at an expert level expert systems? If so, then a linear programming algorithm would be an expert system, since it can find an optimal solution to many linear constraint systems.

Does it refer to a quantity of knowledge? If it did, we could view any program accessing a large database as an expert system.

I define an expert system to be a computer program that emulates the search behavior of human experts in solving a problem. The important point is that experts solve problems by searching a problem space. What distinguishes expert search behavior from naive search behavior is the rich set of operators from which experts choose when solving a problem. Operators are rich in the sense that their patterns or conditions describe specific situations to which the operator applies.

These conditions represent an expert's years of experience, which create the ability to distinguish one problem from another and to reach corresponding solutions. The most commonly used programming methodology is rule-based, in which rules are used to implement an expert's pattern-directed search behavior.

Myth — If we have an expert, then we can create an expert system. This is the most common mistake made in selecting expert system applications. Criteria for selecting expert system applications include the following:

- **Inputs must be well defined. For many management problems, this condition may not be satisfied.**
- **Outputs must be well defined. Again, for some innovative design problems, this condition may not be satisfied.**
- **For obvious reasons, an expert is required.**

Once these conditions have been satisfied, we must determine the nature of expertise; if it regurgitates prior experience, knowledge can probably be extracted and put into an expert system. But if expertise requires the synthesis of new solutions, that expertise probably cannot be captured.

Myth — All expert systems are expert systems. Contrary to popular belief, there are few "pure" expert systems. Once an expert's knowledge has been extracted, AI engineers usually identify problem aspects that can be better solved through other problem-solving techniques. As a result, the final system tends to combine expertise (search guided by the expert's knowledge) with other forms of search (unrelated to how the expert solves the problem, but using a large amount of domain knowledge). Systems that use domain knowledge to guide search, in ways that differ from an expert's, are known as knowledge-based systems.

Myth — Expert systems do not make mistakes. If an expert system emulates a human expert's problem-solving methods, it will tend to make the same mistakes the expert makes. Such mistakes may arise in two ways: First, the expert's decisions may be wrong (not all experts are equally expert). Second, there may be gaps in what the expert knows or has communicated.

These errors can be reduced (but not eliminated) if we use multiple experts to build a knowledge base. If all possible inputs and outputs can be enumerated, and if the system is tested across these inputs, then we can validate the system's performance. For most problems, however, such enumeration cannot be done.

Knowledge-based systems provide another approach for increasing the validity of decisions. By combining strong methods (expert decision rules) and weak methods (more general search techniques), weak methods can fill in where strong methods fail. Such systems can move smoothly between available expertise and other problem-solving methods. But to do this successfully requires great skill on the part of AI engineers designing the system.

If an algorithm optimizes the solution, use it.

One approach towards minimizing the occurrence of mistakes is to learn from those mistakes (usually called machine learning). Interest in machine learning has occurred in cycles over the last 30 years. Lately, there has been a resurgence of interest.[13] Many results are just beginning to take shape, but have had little impact on the commercial development of expert or knowledge-based systems.

Some expert system tools contain algorithms that infer the conditions of rules from data. Such systems lack any provision for learning from their mistakes, and their algorithms do little more than infer Boolean functions from truth tables.

Myth — AI replaces conventional approaches. Some algorithms generate optimal solutions for some problem classes; for example, systems of linear constraints that can be solved using linear programming. If an algorithm exists that optimizes the solution, use it. On the other hand, no optimizing algorithms exist for many problems (for example, large integer problems). For such problems, AI provides a "satisficing" approach; it does not guarantee an optimal solution. Consequently, AI techniques are useful whenever an optimizing algorithm does not exist.

Legend — AI systems are easy to build. The ease of building an application depends on the amount of work required to map the problem into software, which can vary from easy to difficult. Let's examine four cases:

Case 1. The problem to be solved involves automobile engine troubleshooting, where operating problems are

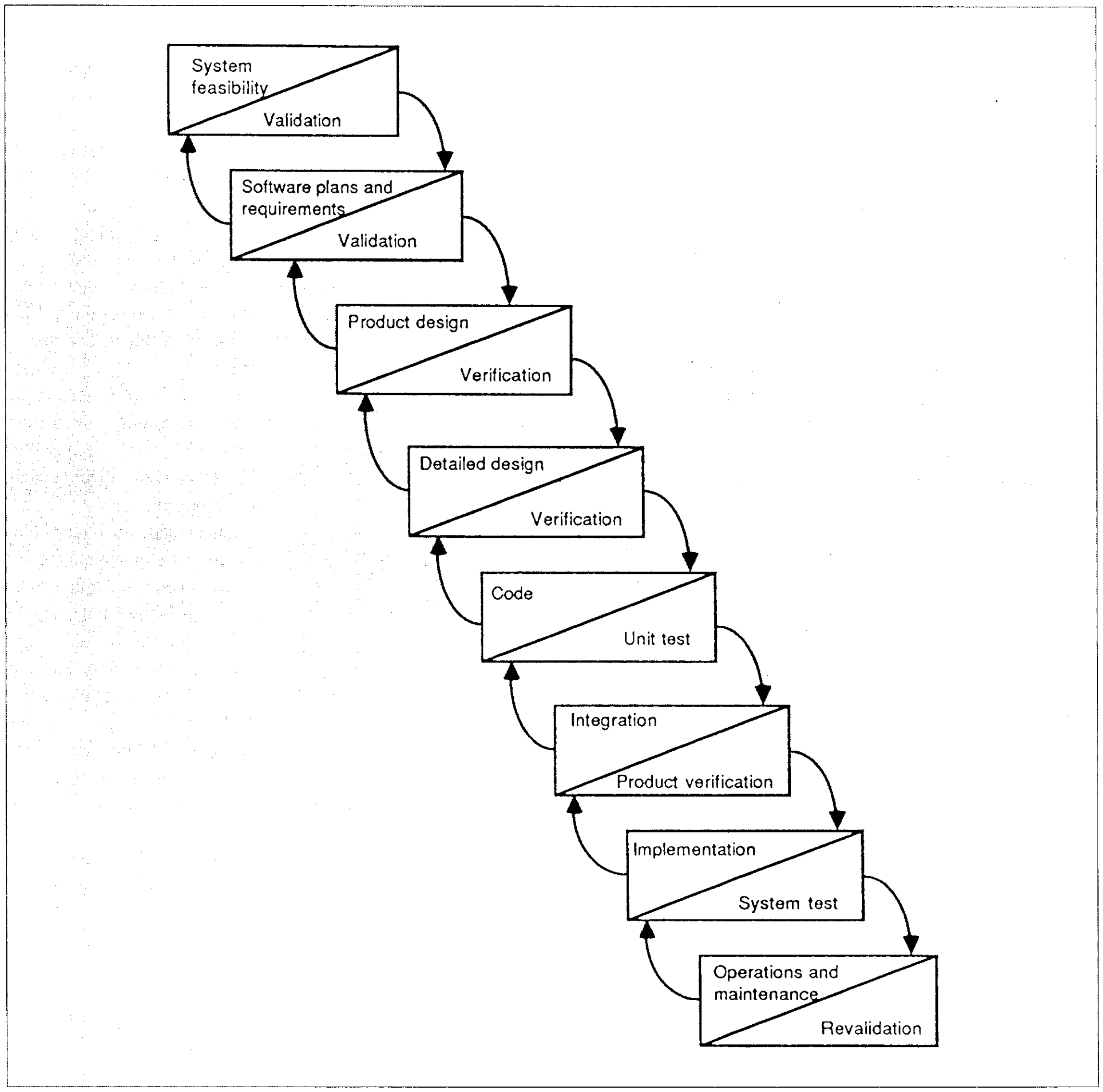

Figure 3. A waterfall model of the software process (from Boehm[18]).

successively traced back along causal chains of failure.[14] Testbench — an application shell produced by Carnegie Group and Texas Instruments — is well suited to this problem and does not require a knowledge engineer; experts can put their knowledge directly into the system. Consequently, mapping expertise onto the knowledge base is straightforward, since knowledge is represented in the same manner that experts conceptualize it.

Case 2. The problem to be solved exemplifies classificatory problem solving — that is, selecting a product from an existing product line, where problem characteristics are heuristically mapped directly onto solutions.[15] MYCIN, for medical diagnosis, was the first expert system of this type.[16] TI's Personal Consultant embodies MYCIN's problem-solving method and provides an interface to support mapping. A knowledge engineer is required to map expert knowledge onto the representation.

Case 3. The problem to be solved involves configuration — that is, configuring a product from more basic parts. R1/XCON, a computer configuration system, was the first expert system of this type.[17] No software product supports this problem-solving method directly, but a rule-based programming language (OPS5, for example) provides an environment for building this type of application.

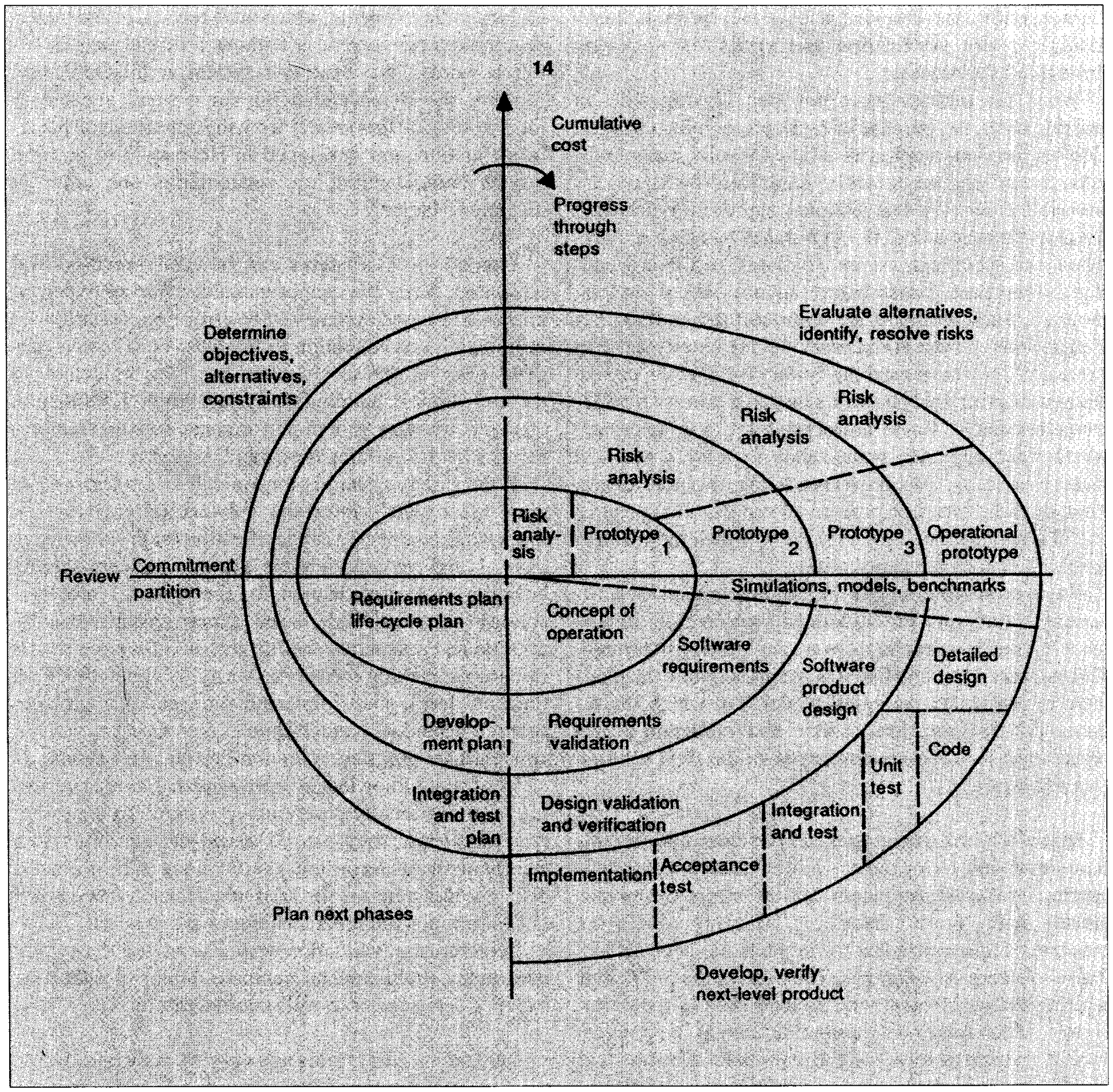

Figure 4. A spiral model of the software process (from Boehm[18]).

Consequently, an AI engineer must design a knowledge representation and devise steps in the problem-solving methodology extracted from an expert (other techniques available for configuration focus on assemblies and constraint satisfaction).

Case 4. The problem to be solved involves factory scheduling, in which the operations of multiple jobs are scheduled on available machines. The solution to this problem depends upon factory characteristics. No software product solves this problem. Consequently, an AI engineer must design the representation and problem-solving methodology.

The bottom line is that — to the extent that software represents directly how users think about the problem — mapping will be easy. But if users are required to design the representation or problem-solving methodology, the system will become more complex to build.

Legend — Rapid prototyping leads more quickly to final solutions. Rapid prototyping has frequently been touted as an approach for constructing solutions more quickly than conventional approaches can. And this is partially true. If the problem fits an application shell (for example, Testbench), knowledge gathered from experts can be put into the system quickly and then tested.

Consequently, the knowledge base can be built incrementally, with verification and validation occurring throughout the process.

When the problem does not map directly onto an existing shell, the purpose of rapid prototyping changes. Conventional approaches have failed to solve many problems that AI engineers tackle. As defined by Figure 3's waterfall model,[18] a conventional approach to software development sequences through stages in which requirements and specifications are developed before any coding; a long time passes before anyone sees a working program. Such problems are decision tasks that are not well defined — system end users will not know what they need until they have used the system for a while, and no amount of prior analysis will result in a correct requirements document. Rapid prototyping seeks to determine whether the approach being taken to solve a problem meets user needs. "Meets user needs" has two senses here: First, does the interface provide all needed functions in a useful form? Second, do the means by which the problem is solved match user expectations?

Rapid prototyping elicits the requirements and specifications of software for ill-defined problems; its importance has not been lost on software engineering in general. This approach is embodied in a recent software development model called the spiral model,[18] in which the requirement, specification, coding, and validation cycles repeat until the software converges on the right solution (see Figure 4).

Myth — Small prototypes can be scaled up into full-scale solutions. Using today's powerful knowledge engineering tools, AI engineers can construct prototypes quickly with "pretty" interfaces. This not only gives managers a false sense that the problem has been solved, it also misleads AI engineers. The heart of the problem is whether the problem-solving method used in the prototype — which solves only a small portion of the problem — will scale up to solve the entire problem. Consider the classificatory problem-solving methodology mentioned earlier; the method has been employed in currently used systems, and limitations have become clear. The method seems to apply when the number of rules is relatively small — 50 to 300, as in PUFF (a pulmonary diagnosis system)[19] and Cooker (a soup-making process control system used at Campbell Soups) — or when diagnostic knowledge for each problem/malfunction can be decomposed into independent subsets, as in Genaid.[2]

When knowledge is considerable and highly interdependent, classificatory problem-solving methods do not scale up. For example, the family of systems developed for diagnosis in internal medicine at the University of Pittsburgh (Internist, Caduceus, and MedQuick) use a different problem-solving methodology to handle complexities.[20]

We find another well-documented example in speech-understanding systems, which must recognize and understand complete sentences without artificial pauses between words. The Hearsay-I system architecture, performing speech understanding for a small vocabulary (100 words), differs greatly from the opportunistic blackboard architecture developed in Hearsay-II to perform speech understanding for vocabularies one order of magnitude larger.[9]

Legend — AI systems can be easily verified and validated. Since we cannot guarantee that an expert or knowledge-based system will be error free, we still need verification and validation techniques. Verification concerns whether the specification has been implemented correctly (that is, building the system "right"). Validation concerns whether system performance satisfies requirements (that is, building the "right" system).

If the expert system is implemented as rules, we can view the expert's inference network (elicited by the knowledge engineer) as a specification for the knowledge base — and we can view the expert's problem-solving behavior as a specification for the system's problem-solving behavior. Rule-based expert systems usually provide an explanation facility that is little more than a rule tracer, making it easier to verify by simulation both the knowledge map implementation and the problem solver's behavior — much simpler than trying to verify a conventional program written in Fortran, for example.

Verifying a knowledge-based system is akin to verifying a conventionally programmed system. To the extent that rules exist, they can be traced during simulation. Recent work in recording justifications with decisions also provides a means for verifying decision processes.[21] Otherwise, conventional techniques are required.

In either case, validation occurs as we test the system over many problems and compare output to either the expert's decisions or to well-known solutions.

Legend — AI systems are easy to maintain. Using rules as a programming language provides programmers with a high degree of program decomposability; that is, rules are separate knowledge chunks that uniquely define the context of their applicability. To the extent that we use them in this manner, we can add or remove rules independently of other rules in the system, thereby simplifying maintenance.

Building rule-based systems differs from this ideal.[22] Various problem-solving methods (including iteration) require that rules implementing these methods have knowledge of other rules, which breaks the independence assumption and makes the rule base harder to maintain. The much-heralded XCON system[1,17] has reached its maintainability limit (about 10,000 rules). The complexity of rule interactions at this level exceeds maintainer abilities. To reduce the complexity of interactions, attempts have been made to modularize the rule base.

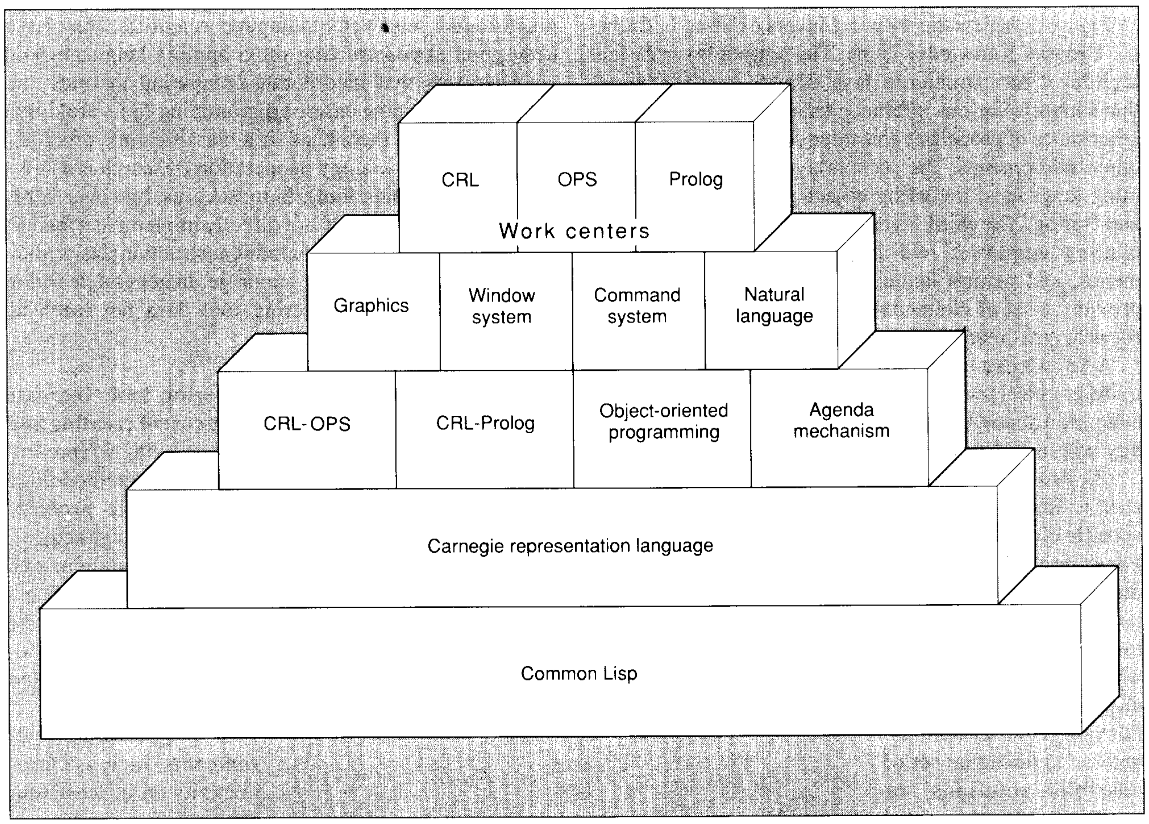

Figure 5. Knowledge craft modules.

However, an "up" side exists. We should view an AI programming language as a high-level programming language. In a rule-based programming language — if one rule is equivalent to 50 lines of Fortran code (a guess) — then maintaining a 1000-rule system should be much easier than maintaining a 50,000-line Fortran program.

Myth — Managing AI systems differs from conventional project management. Due in part to academic ignorance of requirements for building production-level systems, an incorrect belief prevails that managing AI system engineering should differ from managing conventional system engineering.

Rapid prototyping is an important means for acquiring problem requirements and specifications, and for eliciting and verifying knowledge maps. This is not meant to circumvent the need for creating requirements, specification, test code, and the like. The success of rapid prototyping indicates that the waterfall model of software development is inappropriate for many software development projects. Instead, the spiral model is the most suitable means for managing the construction of conventional and AI-based systems alike.

Myth — All AI tools are the same. There are many AI-software packages available in the marketplace. They fall into at least four categories:

(1) Programming languages — An AI programming language provides a basic knowledge representation and problem-solving methodology — basic in the sense that it can be applied to any problem (if not equally well). Both OPS5[22,23] and Prolog[24,25] fit into this category.

OPS5 provides a pattern-directed, forward chaining problem-solving strategy in which we represent knowledge in the form of lists and rules. Prolog provides a goal-directed, backward chaining problem-solving strategy in which axioms and assertions are represented in the form of lists.

Programming languages require AI engineers to design and construct a specific representation — plus a problem-solving strategy on top of it — to solve a problem.

(2) Programming environments — An AI programming environment contains one or more programming languages, interface modules, and interactive programming environments for the construction, debugging, and maintenance of programs.

Figure 5 depicts the software layers existing in Carnegie Group's Knowledge Craft: The bottom layer defines high-level data structures. It is a frame-based language that supports various inference techniques, including the inheritance of properties and values across taxonomic and other relationships. The second layer provides programming languages, including object programming, OPS5, and Prolog. The third layer provides an interface set including windows, two-dimensional color graphics, menus, and natural language. Finally, the fourth layer provides a set of interactive programming environments for program creation, debugging, and maintenance.

A knowledge engineering tool is designed to be used by AI engineers who will define and construct the knowledge representation and the problem-solving methodology within and on top of it.

(3) Problem-solving shells — A problem-solving shell is designed to solve a specific class of problems. For example, Texas Instruments' Personal Consultant is applied in classificatory problem solving, which uses heuristics to map problem characteristics (signs and symptoms) onto an enumerated set of possible solutions or explanations (diseases).[15] Personal Consultant's knowledge representation and inference strategy are engineered solely for that purpose. We can use classificatory problem solving in many different domains.

AI and expert systems are neither panaceas nor research curiosities.

A problem-solving shell is meant to be used by knowledge engineers trained in the use of that shell. Their job is to extract expert knowledge and map it onto the shell's knowledge representation, which may require some minor modification of the shell's problem-solving strategy.

(4) Application shells — An application shell is a narrower version of a problem-solving shell, and is specialized for a given domain so that knowledge representation is specific to that domain — thereby making knowledge acquisition simpler — and relieves users of having to worry about how the problem solver uses the representation. For example, Testbench troubleshoots electrical and mechanical machines and devices.

An application shell is provided with interfaces that eliminate the need for knowledge engineers, which enables experts to map their knowledge directly into the shell.

Myth — Learning an AI knowledge engineering tool is all we need to know about AI. AI is composed of many subfields (the expert system subfield is but one). Each contains many theories and techniques. Because AI practitioners tend to be computer scientists, they have been good at constructing programming languages and environments that incorporate computing theories and techniques. In some cases, programming tools are direct embodiments of theory, as in a classificatory problem-solving tool (the analogy in operations research is using a linear-programming tool). In most cases, however, tools like Knowledge Craft do not directly implement a theory; instead, they make AI-program construction easier than using a more conventional language. In general, learning an AI knowledge engineering tool does not teach us much about AI theory.

Fact — AI knowledge engineering tools increase productivity, thereby reducing the cost of creating and maintaining software. Shells provide a single representation of knowledge and problem-solving methodology for a specific problem class; knowledge acquisition, representation, and utilization are precise and efficient in shells because of the direct mapping between the problem and system.

AI programming environments increase productivity in at least four ways:

(1) Higher level data structures reduce the form and amount of data to be represented;

(2) Higher level languages decrease the amount of code that must be generated to accomplish a task;

(3) Interface modules reduce the number of modules that must be created, by providing many of them as "standard equipment"; and

(4) Programming environments reduce the time required to build programs, by providing a completely interactive environment for creating and debugging programs. Some of these interfaces enable users to follow program execution, and to display the knowledge base's structure graphically.

Myth — AI knowledge engineering tools are good for only AI applications. The belief exists that we can use AI tools only to reason with symbolic, qualitative information; consequently, quantitative algorithmic processing is not supported. In reality, AI software supports qualitative and quantitative reasoning equally well. The bottom line is that we can use AI programming environments to build any application.

Such environments provide AI problem-solving methodologies integrated with more conventional programming techniques found in the underlying implementation language.

How AI systems are being used

I am frequently asked whether any AI-based systems are really "in use." Currently, about 3000 such systems are estimated to be in use daily around the world. Feigenbaum, McCorduck, and Nii document numerous systems in production use.[26] For example,

- **Ace** — A telephone cable maintenance advisory system developed by Bell Labs for Southern Bell — has been in use for over two years;[27]
- **XCON** — A computer configuration system developed at Carnegie Mellon University for DEC — has been in use for over five years, and has configured over 100,000 orders;[1,17] and
- **Dispatcher** — A printed-wire, board assembly, work-dispatching system developed by Carnegie Group for DEC — increased throughput by 100 percent within 24 hours of its installation, and has saved DEC over $12 million per year.[28]

Other successful applications include APES (electronic design), CDS (configuration-dependent part sourcing), National Dispatcher (transportation sourcing and routing), XFL (floor layout assistance), and XSEL (sales assistance) — all five from Digital Equipment Corporation; a system for diagnosing robots and a system for wave-solder-machine diagnosis — both from Ford Motor Company; Compass (network management, from GTE); Cooker (food-processing control, from Campbell Soups); ESP (facility analysis, from Northrop); Ocean (computer configuration, from NCR); Opgen (PWB process planning, from Hazeltine); Trinity Mills Scheduler (FMS scheduling, from Texas Instruments); and VT (elevator configuration, from Westinghouse Electric).

The second wave of technology transfer embeds AI technology in applications that do not require users to learn that technology (nor even to be aware that AI is embedded). The following products use AI techniques partially or totally: CDS (FMS cell control, Hitachi); Genaid (steam turbine and generator diagnosis, Westinghouse Electric); Intellect (natural language database interfacing, AI Corporation); Mudman (drilling mud analysis, N.L. Baroid); Testbench (described earlier); and Telestream (telemarketing assistance, Transcom).

I have addressed questions and misconceptions regarding AI and expert systems. But the bottom line is that *both are here to stay*. They are not simply laboratory curiosities. Nor are they panaceas for many problems we face today. Instead, they are viable technologies that provide a fresh approach to solving many decision problems. Only by removing myths, laying open legends, and recognizing facts can we develop the technical and managerial skills required to apply AI successfully.

Acknowledgments

My research has been supported, in part, by the Defense Advanced Research Projects Agency under contract #F30602-88-C-0001. This article was inspired by questions posed to me over many years, and by a paper by Liebowitz from which one of the myths was drawn.[29]

References

1. J. Bachant and J. McDermott, "R1 Revisited: Four Years in the Trenches," *AI Magazine*, Vol. 5, No. 3, 1984.

2. R.L. Osborne, "Online, Artificial Intelligence-Based Turbine Generator Diagnostics," *AI Magazine*, Vol. 7, No. 4, 1986, pp. 97-103.

3. R.S. Freedman and R.P. Frail, "Opgen: The Evolution of an Expert System for Process Planning," *AI Magazine*, Vol. 7, No. 5, 1986, pp. 53-57.

4. A. Newell and H.A. Simon, "Computer Sciences as Empirical Inquiry: Symbols and Search," *Comm. ACM*, Vol. 19, No. 3, 1976, pp. 113-126.

5. E. Feigenbaum, "The Art of Artificial Intelligence," *Proc. IJCAI*, Morgan Kaufmann, Palo Alto, Calif., 1977.

6. E.D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence*, Vol. 5, No. 2, 1974, pp. 115-135.

7. M. Stefik, "Planning with Constraints (MOLGEN: Part 1)," *Artificial Intelligence*, Vol. 16, No. 2, 1981, pp. 111-140.

8. M.S. Fox, *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*, Morgan Kaufmann, Palo Alto, Calif., 1987.

9. L.D. Erman et al., "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty," *ACM Computing Surveys*, Vol. 12, No. 2, 1980, pp. 213-253.

10. S. Smith, M.S. Fox, and P.S. Ow, "Constructing and Maintaining Detailed Production Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems," *AI Magazine*, Vol. 7, No. 4, 1986, pp. 45-61.

11. S.F. Smith and P.S. Ow, "The Use of Multiple Problem Decompositions in Time Constrained Planning Tasks," in *Proc. IJCAI*, Morgan Kaufmann, Palo Alto, Calif., 1985, pp. 1013-1015.

12. J.F. Allen, "Towards a General Theory of Action and Time," *Artificial Intelligence*, Vol. 23, No. 2, 1984, pp. 123-154.

13. R. Michalski, J. Carbonell, and T. Mitchell, *Machine Learning: An Artificial Intelligence Approach*, Tioga, Palo Alto, Calif., 1983.

14. G.S. Kahn, "From Application Shell to Knowledge Acquisition System," in *Proc. IJCAI*, Morgan Kaufmann, Palo Alto, Calif., 1987, pp. 355-359.

15. W.J. Clancey, "Classification Problem Solving," in *Proc. Nat'l AAAI Conf.*, Morgan Kaufmann, Palo Alto, Calif., 1984, pp. 49-55.

16. E.H. Shortliffe, *Computer-Based Medical Consultations: MYCIN*, Elsevier, New York, N.Y., 1976.

17. J. McDermott, "R1: A Rule-Based Configurer of Computer Systems," *Artificial Intelligence*, Vol. 19, No. 1, 1982, pp. 39-88.

18. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988, pp. 61-72.

19. J. Kunz et al., "A Physiological Rule-Based System for Interpreting Pulmonary Function Test Results," Tech. Report HPP-78-19, Computer Science Dept., Stanford University, Stanford, Calif., 1978.

20. H.E. Pople, Jr., "The Formation of Composite Hypotheses in Diagnostic Problem Solving: An Exercise in Synthetic Reasoning," in *Proc. IJCAI*, Morgan Kaufmann, Palo Alto, Calif., 1975.

21. J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, Apr. 1979, pp. 231-272.

22. L. Brownston et al., *Programming Expert Systems in OPS5*, Addison-Wesley, Reading, Mass., 1985.

23. L. Forgy, "OPS5 User's Manual," tech. report, Computer Science Dept., Carnegie Mellon University, Pittsburgh, Pa., 1981.

24. R. Kowalski, "Predicate Logic as a Programming Language," in *Proc. IFIP*, Elsevier North-Holland, New York, N.Y., 1974.

25. W. Clocksin and C. Mellish, *Programming in Prolog*, Springer-Verlag, New York, N.Y., 1981.

26. E.A. Feigenbaum, P. McCorduck, and H.P. Nii, *The Rise of the Expert Company*, Times Books, New York, N.Y., 1988.

27. A. Stolfo and G.T. Vesonder, "Ace: An Expert System Supporting Analysis and Management Decision Making," tech. report, Columbia University, New York, N.Y., 1982.

28. M. Acock and R. Zemel, "Dispatcher: AI Software for Automated Material Handling Systems," in *Proc. SME AI Manufacturing Conf.*, Society of Manufacturing Engineers, Dearborn, Mich., 1986, pp. 116-121.

29. J. Liebowitz, "Common Fallacies about Expert Systems," *Computers and Society*, Vol. 16, No. 4, 1987, pp. 28-33.

Mark S. Fox received his BSc from the University of Toronto in 1975, and his PhD from Carnegie Mellon University in 1983, in computer science. He joined CMU's Robotics Institute as a research scientist in 1979, and was appointed director of CMU's Intelligent Systems Laboratory in 1980. In 1984, he cofounded Carnegie Group Inc., a software company specializing in knowledge-based systems for solving engineering and manufacturing problems. CMU appointed him associate professor of computer science and robotics in 1987 and, in 1988, named him director of the new Center for Integrated Manufacturing Decision Systems — one of the largest US centers for research in intelligent systems to solve engineering and manufacturing problems. He pioneered the application of AI to factory planning and scheduling problems, project management, and material design. He designed PDS/Genaid, a steam turbine generator diagnosis system (which was a recipient of the IR100) and created SRL (from which Knowledge Craft, the commercial knowledge engineering tool, was derived). His research interests include knowledge engineering, constraint-directed reasoning and applications of AI to engineering and manufacturing problems. An active member of *IEEE Expert*'s Editorial Board for four years, he has published over 50 papers and is a member of the IEEE Computer Society, AAAI, ACM, SME, CSCSI, and TIMS.