Introduction to IBM's knowledge-systems products

by A. J. Symonds

The industrialization of artificial intelligence is believed by many to be a technology that will contribute to a new generation of "smart" computer systems. Technical managers who are users or suppliers of computer systems are trying to understand how the technology can help them, and they are finding it an elusive subject to grasp. This introductory paper starts with a technology overview that aims to address this need for understanding and provide a suitable background for the papers that follow.

Artificial Intelligence (AI) is currently receiving a great deal of attention as a source of technology that may prove to be extremely useful to the computer industry. This industrialization of AI conjures up exciting visions of new computer applications in both commercial and government sectors. In the commercial sector, for example, a computer that can run a factory or make better financial decisions than are now possible would be a valuable commercial asset. Opportunities in defense or intelligence applications in government also hold great promise. General managers who are users or suppliers of information processing systems cannot ignore this potential. In fact, they are already asking basic questions such as: "What can AI do to help my business and what is its value?" and "When can I have a solution, and what will it cost?"

As the technical people wrestle with these questions, they find themselves asking equally basic questions. "What is an AI application, and in what ways is it similar to or different from a non-AI application?" As an approach to an answer, this paper contains a brief overview of the technology. Our goal is to focus on the essential differences between an AI application and a non-AI application. However, this is not a comprehensive survey of AI.

Another matter discussed is how to get started on using the possibilities of AI. As a first step, this paper briefly describes software written by IBM that can be used for developing AI applications.

Knowledge-systems overview

The first step is to define some terms. *Artificial Intelligence* (AI) is a broadly defined term that encompasses multidisciplinary research and development activities involved with emulating human intelligence on machines. AI research has commercial potential in a number of fields—robots, artificial limbs, speech recognition, visual recognition, to name but a few. The subdivision of AI that we are concerned with in this paper is known as *knowledge systems* or *knowledge-based systems.* A knowledge system is a form of decision-support system that has some of the features associated with human intelligence. The term *knowledgeable application* can also be used to characterize computer applications that incorporate these features.

Concepts of knowledgeable applications. A knowledgeable application is a computer program that

© **Copyright** 1986 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

performs decision-support functions in a way that exhibits some of the properties associated with human intelligence. Actually, the property that distinguishes a knowledgeable application is its *automatic reasoning* capability. A simple example is the best way to explain what this means. Let us suppose that we are designing a computer program to advise money managers. In this connection, two potential advisory applications are those of collecting the latest economic data, such as interest rates, and analyzing each account to provide recommendations on actions to be taken for each or to respond to an enquiry on what to do with a particular stock in a particular account portfolio.

Before design can begin, the characteristics of the application must first be analyzed. Analysis involves consideration of the data elements relevant to a money-management advisor, together with their relationships. Data elements required may include the following:

- *Company data,* which describe and classify companies from the point of view of industry, size, financial performance, and stock performance
- *Economic data,* which include economic factors to be considered when making investment decisions, e.g., interest rates
- *Account data,* which describe characteristics of each account being managed such as the overall investment strategy, stocks held, cash held, and recommendations made by the money-management advisor

Analysis also involves consideration of the interactions between the computer program and its user, which can be derived from an understanding of the problems the program is to solve.

The results of application analysis are incorporated into the design process. The traditional process is to design data structures based upon the data element definitions and programs to execute the computational logic, i.e., the sequencing of the user interactions and the computations that take place among them. Program design consists of identifying all possible execution paths for each problem to be solved and writing code that defines each path, step by step. Techniques such as the use of decision tables have been devised to ease the coding burden, but the design task of identifying all the paths still exists.

The knowledge-systems approach tries to avoid a rigid separation of the design into program and data objects with their markedly different properties. Instead, application design consists of a *knowledge-base* specification. A knowledge base contains three kinds of objects that use a homogeneous storage representation. Actual data as well as descriptions of properties and relationships are the factual components of a knowledge base and have a superficial resemblance to traditional data structures. The principal difference between a knowledge base and a traditional data structure is that the description and relationship information has much more semantic content. Another component of a knowledge base includes the rules that represent computational relationships among facts. An example of a rule is the following:

RULE100:

IF a specified company's stock fundamentals are sound, and
interest rates are not going up,
THEN the recommended action for the stock is hold.

The IF part of the rule is known as the *premise*, which is a logical combination of functions operating on facts. The THEN part of the rule is known as the *action*, which is a sequence of functions with such side effects as asserting a value for a fact or initiating a user interaction.

Also included among the components of a knowledge base are automatic reasoning strategies for working out an application's execution logic as it goes along.

Facts and rules are designed without including all the details of how they are combined to solve a particular problem. The system uses the information in the knowledge base to figure out which rules should be selected in which order to achieve the desired results. We can illustrate this by showing how a knowledge system reasons to solve two example money-management advisor problems.

The first problem is to collect such economic data as current interest rates, to analyze each account, and to make recommendations on actions to take for each account. The user enters new economic data, and values for the appropriate facts are stored. The program then looks for all the rules whose premises match the facts just entered and adds their actions to an *agenda*. Heuristics are used to select the agenda item to be executed first, which typically results in a change to the stored facts. The cycle is repeated. The program terminates when a problem solution is found. This occurs in the example when

RULE100 fires, i.e., arrives at a recommendation. This reasoning style is known as *forward chaining*, in which the chain of reasoning proceeds forward in an IF-THEN direction until a complete solution is reached. An alternative term for this reasoning style is *data-driven* reasoning, because the process is driven toward its conclusion by the existence of a set of initial data.

The second problem is to respond to an enquiry as to what to do with a particular stock in a particular account. The program starts with an initial *goal*, which is to find a recommendation for a stock, and it then looks for rules whose actions assert a solution to the goal. The program then attempts to evaluate the premises of the rules. RULE100 would therefore be selected and the discovery made that facts about stock fundamentals and interest rates are needed before the premise can be evaluated. Obtaining these facts is established as a subgoal, which the reasoning process solves either by finding rules that assert values or by asking the user for values. The process of successively identifying and solving subgoals continues until all the facts are obtained that allow the initial goal to be solved. This reasoning style is known as *backward chaining*. The chain of reasoning proceeds backward in a THEN-IF direction until the initial goal is solved. Another term for this is *goal-driven* reasoning, because the process is driven toward its conclusion by the existence of an initial goal.

Solving subgoals continues until all the facts are obtained that allow the initial goal to be solved.

A knowledgeable application essentially makes up its execution logic as it goes along, thus relieving its designer of this responsibility. This has some interesting implications. One of these is that the program must remember the execution logic so that it can explain its results to users who want to know why a particular question was asked or how a particular conclusion was reached. Another implication is the method used for development and testing, which must necessarily be very empirical and contain a great deal of end-user involvement. Because the execution paths are not specified in advance, it is impossible to define a rigorous set of test cases that can be run by an objective third party. Development and testing constitute, rather, an iterative process requiring close cooperation between the developer and the end user, where changes are subjected to end-user evaluation in small increments.

The knowledge-systems approach is well suited to problem areas that have the following properties:

- There are many combinations of input data to be evaluated.
- The conclusions that can be reached vary greatly.
- It is difficult or impossible to define precise, exhaustive relationships among the different input data combinations and the corresponding conclusions to be reached. These relationships are more easily defined in terms of individual rules, each of which draws its own conclusion from a small subset of the input data.
- There is continuous modification and enhancement of the application as a result of user feedback or changing requirements.

The knowledge-systems approach is also useful for applications that require some degree of natural-language understanding. A computer program with complete natural-language understanding may be impossible to achieve, because it requires an enormous knowledge of grammar, word meaning, and underlying ideas that are expressible in natural language. However, if the field of expertise is appropriately restricted, it is possible to construct programs that can reason successfully using natural language input concerning specific problems.

Increasingly, there is evidence that serious work on applying knowledge-systems technology is taking place. Some of the more promising application areas are now described.

Examples of knowledgeable applications

Advisory applications. Advisory applications are, as the name suggests, programs that offer advice on some problem domain to a practitioner within that domain. These programs are also known as *expert systems*, but the notion of advice more accurately reflects what they actually do.

Here we use the example of insurance underwriting as an advisory application. Underwriting requires that human beings go through a complex decision-making process; the productivity and quality of underwriting decisions can be improved by a knowledge system that provides suitable advice. A system to be used in production must be accessible from existing terminal networks and must be able to interrogate policyholder data bases.

The computer program encapsulates human expertise, and offers advice to people with less experience.

Another advisory application user is the oil industry, which is investing in knowledge-systems technology. One application is in oil prospecting, in which the computer is programmed to analyze exploration data and advise on the likelihood of finding gas or oil, as well as the desirability of further exploration. The computer program encapsulates human expertise, which is based very much upon long experience, and offers advice to people with less experience.

Two other advisory applications are described in this issue. Hagamen and Gardy[1] describe a medical diagnosis application, and Voelker and Ratica[2] show how the knowledge-systems approach has been used to write agreements between customers and IBM for the movement of IBM equipment.

Problem diagnosis applications. Problem diagnosis is a type of advisory application in which knowledge-systems technology has yielded good results. It is reasonably easy to identify complex systems or pieces of equipment for which downtime is expensive. The process of problem determination is time-consuming, and it requires expertise that might not be readily available. There is often documentation describing heuristics or rules of thumb to guide problem determination that can be encapsulated in a computer program. A simple example of this is the trouble-shooting guide in automobile handbooks.

Several oil companies are using knowledge-systems technology to analyze drilling equipment and well problems. Also, the GTE Compass Program[3] assists in the maintenance of a telephone switching system, and the AT&T ACE program[4] analyzes telephone cable records and is capable of isolating problems much earlier than human experts.

A recurring requirement that must often be met before such applications can be put into full production is the ability to embed the knowledge system into an existing computing environment and allow it to communicate with other programs and data.

Problem diagnosis applications often combine both forward and backward reasoning. Forward reasoning is used to generate possible problem sources from the initial data. Backward reasoning is then used to explore each possibility, to collect supporting data during the process, and to find out which is the most likely source of the problem.

Operational decision-making. The forward-reasoning style has been successfully applied in a number of operational decision-making domains. General properties of this application are the following:

- An existing process that generates large volumes of machine-readable information
- Human intervention that is required to analyze the information and make corresponding operational decisions
- Decisions that must be made rapidly and under conditions in which the quality of the decision is sensitive to such factors of the human condition as degree of experience and fatigue

Knowledge systems that can replace or improve operational decision-making by human beings have proved to be very useful in several applications.

International funds transfer[5] is a banking application that has been successfully implemented. Unformatted messages arrive by Telex and must be read and interpreted, with resultant decisions to move or not move large sums of money. Knowledge-systems technology has provided a computerized solution that had previously eluded more traditional approaches. We term this a very high-leverage application, because one day's delay in handling a large amount of money can translate into a substantial loss of interest income. The application also has a degree of natural-language understanding, in that it analyzes an unformatted message and uses reasoning

techniques to match its contents with the criteria needed to specify an international funds transfer transaction.

YES/MVS[6] is another example of operational decision-making. Current large MVS systems generate operator messages at an enormous rate. As a result, the quality and productivity of system operators' decisions can be improved by means of a knowledge system's automatically proposed actions.

The knowledge-systems approach seems to be tailor-made for the task of simplifying data base access.

As a final example, consider an application implemented at the IBM computer chip manufacturing plant at Burlington, Vermont.[7] A large APL program collects and stores data on batches of computer chips as they pass through the manufacturing process. Until recently, reports were generated from the data and given to manufacturing management, who then made decisions affecting the manufacturing process. As chip manufacturing processes have become more complex, reports are being generated faster than management can act on them. An improved decision-making process is required, and a knowledgeable application has been programmed in APL to solve the problem.

Knowledgeable data bases. Large computerized data bases are information resources whose value depends on ease of access and use of the stored information. In fact, access to data bases is difficult, because it requires a knowledge of special computer languages for data base access as well as knowledge of the contents and structure of the stored information.

The knowledge-systems approach seems to be tailor-made for the task of simplifying data base access. Such systems exhibit an understanding of what the user wants to know (which could be expressed via selection from a list of topics or free-form natural-language input). Then they use reasoning techniques to match the user's input with knowledge about the data base to generate a query in the data base's native language. Intellect®, a trademark of the Artificial Intelligence Corporation, is an example of an application that is knowledgeable about data bases.[8] It accepts free-form natural-language input from users and executes the corresponding data base queries.

Tools for data processing professionals. Many of today's computer applications and tools are designed for data processing professionals, with the objective of making them more effective at meeting their users' needs. Knowledge-systems technology offers the potential for a new generation of tools for data processing professionals.

One example is the area of application analysis, design, and specification.[9] A number of methodologies exist for defining and analyzing application requirements. Examples of what has to be defined are the following:

- Input, output, and storage of data elements
- Relationships among data elements
- User interactions
- Reports required
- Computation required
- Triggering events

Tools exist for editing, analyzing, and displaying a set of application requirements. Translating requirements to an effective design for programs and data bases operating in a specific computing environment requires considerable human expertise, as well as knowledge of existing programs and data bases with which the new application is to communicate. Once a design is complete, a number of specification and prototyping tools are available for converting it to running code. There appears to be considerable potential for an application design advisor that bridges the gap between the analysis and specification phases. Such an advisor could be used in an off-line mode to generate program and data specifications to be executed elsewhere. Alternatively, it could be used in an on-line mode, where it would actually execute the application.

Program conversion is a common yet labor-intensive activity. Every data processing installation needs at times to convert programs either to a different language or to a different operating environment. *Sift tools* are available today that can perform mechanical conversions and identify the parts of the program that require manual conversion. A knowledge sys-

tem, using deeper reasoning, should be able to make program conversion less labor-intensive.

The IBM COBOL Structuring Facility (COBOL/SF)[10] is an example of such a program conversion tool. Data processing users are willing to make a considerable investment in structuring existing COBOL programs so that they can become easier to maintain. COBOL/SF does the structuring automatically. Algorithms exist for converting any program to a structured form. The problem of producing a structured form that is easy for a person to understand does not have an algorithmic solution. Rather, it requires knowledge of what constitutes a good COBOL programming style and how existing code should be changed to make it better. COBOL/SF uses reasoning based on this type of knowledge to arrive at a readable structured form, and it even makes suggestions on how the user can improve the readability.

Application development methodology

Knowledgeable application analysis and design requires a new kind of data processing professional known as a *knowledge engineer*. A knowledge engineer must be able to understand a problem that is a candidate for computer automation and then be able to abstract the problem in knowledge-systems terms, i.e., design the knowledge base. This process involves the following steps:

- Decomposing a problem into discrete subproblems to the extent possible
- Designing a *knowledge-representation schema* for facts
- Characterizing problem solutions in terms of the knowledge-representation schema
- Defining the rules that allow conclusions to be drawn when a particular situation is recognized
- Characterizing the situations and conclusions in terms of the knowledge-representation schema
- Selecting appropriate styles of reasoning to be employed during the search for problem solutions
- Specifying how user interactions should look during the consultation

This methodology is somewhat analogous to more traditional application analysis and design. However, it also requires a good grasp of knowledge-systems concepts and how to apply them in a given situation.

Once abstraction of the application in knowledge-systems terms has taken place, the next step is translation into a program that actually runs on the computer. Application programming is designed to be exploratory in nature. An initial prototype is built for user evaluation and is then extended and modified as a result of discussions between the user and the knowledge engineer. In many cases, this process continues throughout the life of the application as users gain new insights into how the knowledgeable application can help them.

Tools specially designed for knowledge engineers are now commercially available.

Tools and languages for application development

There are essentially two ways of doing the actual application programming: specially designed knowledge-engineering tools and general-purpose programming languages. A number of tools specially designed for knowledge engineers are now commercially available. These tools are also known as *expert-systems shells*, because they provide a hollow shell into which specific expertise is injected to create a finished knowledgeable application. Knowledge-engineering tools provide the following facilities to support the specification, execution, and testing of a knowledgeable application:

- *Languages* for specifying the various elements of the knowledgeable application, i.e., problem decomposition, knowledge-representation schema, rules, reasoning style, and user interactions
- *Editors* for viewing and modifying the application elements that are capable of validating syntax and semantics as the knowledge engineer enters them
- *Run-time interpreter* or inference engine that executes the application and drives the end-user dialog or consultation
- *Services* that can be invoked during the consultation, such as the following: (1) requesting explanations of why a question is being asked or how a particular conclusion has been reached; (2) viewing and modifying the data stored while an appli-

cation is running; and (3) tracing and viewing the application-execution process, setting breakpoints, etc.

Knowledge-engineering tools are analogous to fourth-generation languages and their associated generators that are available for more traditional commercial application development. Fourth-generation languages are designed to support an exploratory programming environment, where changes made to an application can be reviewed very quickly by a user.

General-purpose programming languages can also be used for coding knowledge systems. Comparing a programming language to a knowledge-engineering tool involves the same considerations as those that apply in comparing a programming language to a fourth-generation language. The programming language is more difficult to use than the higher-level tool, but the programming language offers the potential for better function and performance.

Programming a knowledge system using a general-purpose programming language requires specialized AI programming skills, in addition to knowledge-engineering skills. For this reason, most commercial application development is done using knowledge-engineering tools. Programming languages tend to be used for the more basic purposes of developing the knowledge-engineering tools themselves, developing applications custom-tailored for direct delivery to end users, and for AI research.

The complex part of programming a knowledge system lies in the core knowledge representation and reasoning functions. A number of languages have been used to do this. Intellect[8] and COBOL/SF[10] are written in PL/I. Applications have been written in APL.[1,7] Expert System Environment,[11] an IBM knowledge-engineering tool to be discussed later, is written in Pascal. The article by Hodil et al.[12] goes into considerable depth on how PL/I and ISPF (Interactive System Productivity Facility)[13] have been used to build a prototype knowledge-engineering tool that can be embedded in a commercial data processing environment.

This diversity of languages reflects the fact that people like to use languages with which they are familiar. Because most knowledge-systems programming at this level is still done by people with an AI research background, the languages they use deserve special attention.

LISP and PROLOG are two languages that were originally invented by the AI research community. These languages are particularly suitable for the programming of knowledge systems. In their industrialized form, they are actually full-scale general-purpose languages. For that reason, they can be used for writing payroll applications, in addition to their having features that, in the hands of a skilled user, make them the most powerful general-purpose languages available for programming knowledge systems.

Knowledge-systems software uses a symbolic processing subsystem to support its execution environment.

LISP (list processing language) is the most widely used AI programming language, particularly in the U.S.A., and it has been used to write most of the commercial knowledge systems available today. LISP uses the atom and list constructs to represent both functions to be executed and data structures to be interpreted. The invocation of a function is also represented in the same way, thereby yielding two important benefits. For one thing, the essential notion of a knowledge base is the abolition of the distinction between program and data objects. A programming language that has already done this is clearly very suitable as an underlying tool. Also, it is very natural to write LISP programs that manipulate LISP programs. This can be exploited to provide a very powerful and flexible environment for creating, modifying, executing, and debugging programs. LISP contains a vast array of primitives for symbolic and numeric computation, as well as for handling the transfer of control within and between LISP functions. Also, LISP compilers, which, by the way, are written in LISP, can generate efficient machine code from LISP programs. Complete LISP programming environments with highly integrated interpreters, compilers, editors, and debuggers are now available on a variety of computer systems, including specialized workstations. Workstation environments usually exploit bit-mapped graphics, windowing, and pointing devices to make them more friendly to the LISP programmer.

There is evidence that CommonLisp, a standardized version of the LISP language, is being widely adopted. This is still another sign of the movement toward using the language for commercial work as well as research. The first version of YES/MVS[6] was developed using a knowledge-engineering tool written in LISP.

The other AI language that is being increasingly used is PROLOG (programming in logic), which was originally designed as a tool for natural-language research and is now entering the commercial world. Like LISP, PROLOG has a uniform representation for both programs and data, thereby yielding the same benefits for PROLOG as for LISP. On the other hand, PROLOG uses a different set of basic constructs for its knowledge representation. The basic unit of knowledge is a *predicate*, which can be used to represent a fact that is asserted to be true or as a goal to be proved. Knowledge can also be stored as rules. The *head* of a rule is a predicate, and its *body* is a logical combination of predicates that defines the conditions under which the head is true. Because predicate logic is used as a formalism to represent knowledge, these basic constructs provide a higher-level language for programming knowledge systems.

PROLOG solves problems by searching for rules whose heads match a goal. To determine whether one of these rules is true, the predicates in the body are evaluated by defining them as subgoals to be solved. The process is recursive, because solving a subgoal requires the solution to successive levels of consequent subgoals. The process continues until a solution is finally found or until the search fails. The search mechanism automatically backtracks on failure. The technique for matching goals with the heads of rules is called *unification*. This technique can be used for complex pattern matching, as well as simple atomic comparison. PROLOG compilers, written in PROLOG, can generate efficient machine code from PROLOG programs. PROLOG programming environments, complete with highly integrated interpreters, compilers, editors, and debuggers, are now available on a variety of computer systems, including specialized workstations. Workstation environments usually exploit bit-mapped graphics, windowing, and pointing devices to make them more friendly to the PROLOG programmer.

Because many knowledge-systems building blocks (e.g., the use of logic for knowledge representation, pattern matching, and backtracking) are built into the PROLOG language, experienced users of PROLOG feel that it is very suitable for productive development of high-performance knowledge systems and natural-language applications. Wilson[14] discusses in more detail application programming in PROLOG.

Symbolic processing subsystems

We now turn to a discussion of the computer systems that are used for knowledgeable applications and their associated tools and languages. Knowledge-systems software uses what we call a *symbolic processing subsystem* to support its execution environment. Symbolic processing is a term used by the AI community to characterize the kind of computing performed by a program that is doing automatic reasoning. A symbolic processing subsystem provides the underlying support that, in effect, enables a knowledge system to make up and remember its execution logic as it goes along. This implies dynamic creation of data structures and program logic, efficient symbol lookup, flexible management of data and control structures, and large demands on processing power and memory.

AI workstations offer the most productive development environment for knowledge engineers and programmers.

LISP and PROLOG contain their own symbolic processing subsystems, which is one of the reasons why they are very powerful languages for building knowledge systems. Knowledge systems built using traditional programming languages require their own symbolic processing subsystem. Knowledge systems software is available on a range of computing equipment.

AI workstations offer the most productive development environment for knowledge engineers and programmers. Their processing power is supplied by specialized hardware and supports a high-performance symbolic processing subsystem. High-bandwidth communication with a bit-mapped display

and pointing device gives developers the feeling of direct control over their environment. Windows can be opened up at will to watch the internal behavior of programs, edit different programs or run-time data, and continue to monitor and control the external behavior of the application. This feeling of direct control is due very largely to the reactivity provided by immediate system response to user actions.

Super-micro-based engineering workstations, such as the IBM RT PC, are also used for knowledge-systems work. In fact, hardware diagnostics on the RT PC are executed using the Portable Inference Engine (PIE) described by Burns et al.[15] Engineering workstations have characteristics that are similar to those of AI workstations but with less power at less cost.

Significant skills are required to take a generic knowledge-engineering tool and turn it into a production application.

Personal computers do not have the power to support high-performance symbolic processing subsystems, but they can provide a useful environment for learning and prototyping with a highly reactive user interface. They are also capable of running applications that do not need large knowledge bases.

Mainframes with their processing power, large memory, and efficient paging can support a high-performance symbolic processing subsystem. They can also access centralized data base facilities, giving them the capacity to handle large problems. Because mainframes share their resources, they are not as responsive to user actions as are workstations.

Assessment

We have looked at knowledge systems from an application and a system viewpoint. We now assess the impact of knowledge systems on the computer industry. A knowledge-systems industry is gaining momentum. Users are successfully implementing useful applications, and vendors of products and services are actively supplying their needs. Of course, knowledge systems are still a relatively small part of the decision support systems business, because users must make an initial investment in scarce knowledge-engineering and programming talent. For rapid transition to high volumes to take place, more progress needs to be made in several areas.

One area is that of delivering applications to end users without the need for highly skilled knowledge engineers. To be useful, most knowledgeable applications must be customized for individual users or organizations. Significant skills and effort are required to take a generic knowledge-engineering tool or AI programming language and turn it into a production application. Ways must be found to reduce the amount of skill and/or effort required. One approach is to identify application areas for which a starter set of application specifications can be supplied together with the generic tool or language, which is discussed in the paper on YES/MVS.[6] The knowledge engineer thus has a design to start from and becomes productive quickly. However, there is still a need for a knowledge-engineering data processing professional who understands AI concepts, their application, and the use of the tool or language. A longer-term solution is for problem-solvers who are not data processing professionals to develop their own knowledgeable applications. This requires developments in knowledge-acquisition and natural-language technology. Delivering the technology in a user-friendly form requires a high-powered symbolic processing subsystem with fast access to an advanced display. This implies a powerful workstation, operating either in stand-alone configuration or connected to a mainframe.

Knowledge systems must be embedded in existing computing environments, so that they can work with existing terminal networks and data bases to communicate with existing programs. Once users complete a successful knowledgeable-application pilot project, they face the problem of how to deliver a production version to existing operational environments. The pilot application runs in the environment in which it is developed and which is designed to offer a rich, integrated set of facilities for editing and debugging. The application knowledge-base representation is, therefore, designed for ease of editing and debugging, rather than run-time efficiency. For delivery, the trade-off is made in the other direction, because run-time efficiency is the more important aspect.

This suggests that, before knowledgeable applications can be widely used in production, there has to be a translation step that converts application knowledge bases into a form suitable for mass delivery.

Automatic reasoning has a strong dynamic memory-management requirement.

The delivery system must be able to capture trace information and feed it back to the development system. The development system's translator must be capable of sending modifications to the delivery system.

There is general agreement within the industry that performance improvement can be gained by translating an application knowledge base into a more static form for mass delivery. There are differing opinions on how much can be gained, because automatic reasoning has a strong dynamic memory-management requirement. However, the fact remains that users want to minimize the cost of the delivery system, and a breakthrough in this area will be a most welcome development.

Improvement in symbolic processing subsystem performance is another requirement for progress. Knowledge-systems technology offers a very powerful approach to problem-solving but places high demands on the underlying computer system. Extending the technology to new problem areas and making it easier to use creates further demands. Improving symbolic processing subsystem performance thus has a strong effect on the penetration of the technology. A number of research projects are under way to exploit parallel processing as a means to improve performance. The prevailing view today is that inventions in both hardware and software technology must occur before parallelism can be effectively exploited in a symbolic processing subsystem.

Tools and languages for System/370 systems

At present, three software products for users of 43XX and 30XX processors have been developed by IBM. These products run under the control of the VM/SP[16] operating system and are accessed by users from 327X display terminals. There are immediate benefits from providing knowledge systems under VM/SP. VM/SP is a widely used interactive computing environment, the users of which can have access to knowledge-systems technology simply by installing additional products in their existing environment. Also, the processing power of 43XX and 30XX mainframes, together with the large virtual address space capabilities provided by VM/SP, support a high-performance symbolic processing subsystem. By embedding knowledge-systems functions in the VM/SP environment, users can communicate with existing data and programs. Also, systems utilities can be used for saving, restoring, moving, copying, and generally managing knowledge bases stored on disk.

Expert Systems Environment

The Expert Systems Environment/VM product[11] is a knowledge-engineering tool that was announced in the U.S.A. and Canada in August 1985. An MVS[17] version was announced in April 1986. The product originated at the IBM Palo Alto Scientific Center. For example, it was used to build the Contract Support Services consultant[2] mentioned earlier in this paper. Expert Systems Environment/VM and MVS provide a set of functions that one would expect to find in a commercial knowledge-engineering tool.

Consider the language for specifying the elements of a knowledgeable application. The language syntax is designed for users familiar with such programming languages as BASIC or fourth-generation languages. Keywords have been selected so that the meaning of a language statement is intuitively obvious to the reader. Application elements that can be specified are problem decomposition, knowledge-representation schema, rules, reasoning style, and user interactions.

Consider problem decomposition. Application elements can be partitioned into a hierarchical structure. For example, in a systems configuration application, the knowledge engineer might want the end user (or client) to focus initially on the applications, then on the Central Processing Unit, and finally on such peripheral hardware as tape and disk drives.

Partitioning an application into several hierarchically related components has advantages. It encourages top-down design and specification. It also results in more efficient execution, because facts, rules, and reasoning styles can be localized to particular components in the hierarchy.

The knowledge-representation schema defines named parameters, which is how facts are stored. Parameters can have such properties as constraints, prompt messages, explanation messages, and an indication of where the parameter value can be obtained, if it is unknown.

Rules are specified as follows:

IF (condition) THEN (conclusion or action)

For example,

IF INCOME > 50000 THEN TAX_RATE IS 0.5

means "if the value of the parameter INCOME is greater than 50000, then assert that the value of the parameter TAX_RATE is 0.5." Parameter values can be asserted with a certainty factor ranging from +1 (definitely true) to −1 (definitely false). A parameter can have multiple values, each with a different certainty factor.

Regarding reasoning style, both backward and forward reasoning are supported. Control strategies embodying both of the two styles can be individually specified for each component of the application specification hierarchy.

User interactions of prompting and explanation text can be specified for each parameter. Screen designs are automatically created for use during consultations. If desired, the knowledge engineer can interactively design customized screen formats to override the automatically generated screens.

Full-screen editors are provided to assist application specification. The editors are tailored to the application elements being worked on and have built-in automatic checking that can immediately detect many of the errors that might be made. Application elements are compiled automatically into a form that reduces the amount of space required for their storage and improves runtime performance.

There is an Inference Engine to drive the consultations. Among the Consultation Services, both HOW? and WHY? explanations are supported. A client can also ask WHAT? to receive more details on the question being asked. During the testing of an application, the knowledge engineer can request a trace that provides an exact roadmap through the reasoning process. A formatted printout of a complete knowledge base can also be provided for off-line checking.

Extensive on-line help facilities are available in both the consultation and development environments.

An entire consultation can be stored and rerun. During the rerun, answers previously given to any question can be changed to see the effect on the solution. It is also possible during a consultation to undo or change a previously entered response.

Additional important features of the product are on-line help and access to external procedures. Extensive on-line help facilities are available in both the consultation and development environments. As for access to external procedures, user programs written in Pascal or some other language can be invoked during the execution of the condition or action part of a rule. Examples of how this has been exploited within IBM are the following: read/write access to VM/SP data files, writing files that can be processed by DCF (Document Composition Facility)[18] to produce professional-quality printed output, read/write access to relations stored in SQL/DS[19] data bases, and the use of Graphical Device Data Manager (GDDM)[20] to produce presentation graphics output.

LISP/VM

The LISP/VM product[21] is a high-performance LISP programming environment running under VM/SP; it was announced in the U.S.A. and Canada in July 1984. The original development work was done at the IBM Research Center at Yorktown Heights, New York.

LISP/VM includes a full LISP compiler that generates machine code designed to run efficiently in the VM/SP environment. The code can also be shared among virtual machines. A complete development environ-

ment with integrated editing, compilation, and debugging facilities is provided within LISP/VM. Alternatively, users can use standard VM/SP editing and file management tools to prepare LISP/VM programs.

VM/Programming in Logic (VM/PROLOG)

The VM/PROLOG product[22] is a high-performance PROLOG programming environment running under VM/SP that was announced in the U.S.A. in August 1985. The original development work was done at the IBM Paris Scientific Center.

VM/PROLOG is a high-speed interpreter that includes some of the features of a full PROLOG compiler. Rules not modified during the course of an application can be declared as static, thus speeding up their execution. One of the highlights of the product is its ability to communicate with other systems facilities. For example, CMS and CP requests can be submitted. Also, System Product Interpreter or EXEC2 variables can be consulted or updated. Because VM/PROLOG is loaded as a CMS nucleus extension, it can communicate with concurrently loaded modules, including the CMS EXEC processor. When needed, user-defined predicates can be written in assembly language and dynamically loaded. An optional link with LISP/VM is also supported, as well as an optional link with SQL/DS. The latter allows users to explore the potential synergism between knowledge systems and relational data bases.

Distributed development could be architected by combining the symbolic processing power of the mainframe with the reactive workstation capability of the PC.

Future direction

As to the future, a number of other requirements are the basis of continuing investigation. Today's product line represents an important first step in bringing knowledge systems to IBM users in a way that exploits the processing power of large IBM mainframes. Likewise, these knowledge systems introduce the technology as an extension to existing computing facilities, so that knowledgeable applications can be viewed as extensions to existing applications.

A logical next step is to move the technology to MVS,[17] which has already been done for the Expert System Environment product. This provides the opportunity to exploit additional computing power, such as extended addressing, and to communicate with production applications and data bases running under MVS.

Today's products support user access from 327X display terminals, which is a good solution for end users who already have these terminals in their offices. However, the development environment presented to knowledge engineers and AI programmers does not have the reactivity that can be achieved on workstations directly attached to bit-mapped displays and pointing devices. The IBM PC family of workstation products has the potential to improve on this situation. Two ways to exploit the PC potential in a knowledge-systems development environment are being studied. One line of investigation is that of stand-alone PCs, loosely coupled to the mainframe. Knowledge-engineering tools and AI programming languages on the PC could offer a productive environment for application specification, prototyping, and initial testing. Transmitting the results to a mainframe for final testing and production would then be performed via an appropriate file transfer mechanism. Another research approach is toward distributed PCs, tightly coupled to the mainframe. A distributed development environment could be architected by combining the symbolic processing power of the mainframe with the reactive workstation capability of the PC.

As a final comment, we need to address the major obstacles to wide acceptance of knowledge systems mentioned earlier in this paper. One is to find ways of delivering application solutions to end users without the use of scarce knowledge-engineering and/or AI programming skills. Sources of starter sets of application specifications will be investigated to address this need. Because a longer-term solution requires advances in knowledge acquisition and natural-language processing, and because both of these technologies are in the research stage, commercial software products embodying these technologies appear to be far away. Another approach to the wide-

spread use of knowledge systems is to address the requirement for mass delivery of completed applications to existing computing environments, so that they can work with existing terminal networks and data bases.

Concluding remarks

Artificial Intelligence research has spun off the commercially significant knowledge-systems technology. Applications of the technology are being successfully implemented through the use of consulting services, knowledge-engineering tools, AI programming languages, and symbolic processing subsystems. However, progress still needs to be made in several areas before widespread application of the technology is likely to occur. IBM has developed software products that can help users to start on knowledge-systems projects.

Cited references

1. W. D. Hagamen and M. Gardy, "The numeric representation of knowledge and logic—Two artificial intelligence applications in medical education," *IBM Systems Journal* **25,** No. 2, 207–235 (1986, this issue).
2. J. A. Voelker and G. B. Ratica, "The genesis of a knowledge-based expert system," *IBM Systems Journal* **25,** No. 2, 181–189 (1986, this issue).
3. S. K. Goyal, D. S. Prerau, A. V. Lemmon, A. S. Gunderson, and R. E. Reinke, "Compass: An expert system for telephone switch maintenance," *Expert Systems* **2,** No. 3, 112–126 (July 1985).
4. P. Wilkinson, "Humanized system tracks troubles," *Telephone Engineer and Management* **89,** No. 8, 86–91 (1985).
5. T. Johnson, *Natural language computing: The commercial applications*, report published by Ovum Ltd., London (August 1985), pp. 258–262.
6. K. R. Milliken, A. V. Cruise, R. L. Ennis, A. L. Finkel, J. L. Hellerstein, D. J. Loeb, D. A. Klein, M. J. Masullo, H. M. Van Woerkom, and N. B. Waite, "YES/MVS and the automation of operations for large computer complexes," *IBM Systems Journal* **25,** No. 2, 159–180 (1986, this issue).
7. K. J. Fordyce and G. A. Sullivan, report in preparation, IBM Corporation, Neighborhood Road, Kingston, NY 12401.
8. *Intellect General Information Manual*, G320-9199, IBM Corporation; available through IBM branch offices.
9. R. Balzer, T. E. Cheetham, Jr., and C. Green, "Software technology in the 1990's: Using a new paradigm," *IEEE Computer* **16,** No. 11, 39–45 (1983).
10. *COBOL/SF Presentation Guide*, G320-9391, IBM Corporation; available through IBM branch offices.
11. *Expert System Consultation Environment/VM and Expert System Development Environment/VM General Information Manual*, GH20-9597, IBM Corporation; available through IBM branch offices.
12. E. D. Hodil, C. W. Butler, and G. L. Richardson, "Knowledge-based systems in the commercial environment," *IBM Systems Journal* **25,** No. 2, 147–158 (1986, this issue).
13. *What's New in ISPF*, GC34-2172, IBM Corporation; available through IBM branch offices.
14. W. G. Wilson, "Prolog for applications programming," *IBM Systems Journal* **25,** No. 2, 190–206 (1986, this issue).
15. N. A. Burns, T. J. Ashford, C. T. Iwaskiw, R. P. Starbird, and R. L. Flagg, "The Portable Inference Engine: Fitting significant expertise into small systems," *IBM Systems Journal* **25,** No. 2, 236–243 (1986, this issue).
16. *IBM Virtual Machine/System Product: Introduction*, GC19-6200, IBM Corporation; available through IBM branch offices.
17. *MVS General Information Manual*, GC28-1118, IBM Corporation; available through IBM branch offices.
18. *Document Composition Facility and Document Library Facility, General Information Manual*, GH20-9158, IBM Corporation; available through IBM branch offices.
19. *SQL/Data System Concepts and Facilities*, GH24-5013, IBM Corporation; available through IBM branch offices.
20. *GDDM General Information Manual*, GC33-0100, IBM Corporation; available through IBM branch offices.
21. *LISP/VM Program Description/Operations Manual*, SH20-6476, IBM Corporation; available through IBM branch offices.
22. *VM/Programming in Logic Program Description/Operations Manual*, SH20-6541, IBM Corporation; available through IBM branch offices.

Andrew J. Symonds *IBM Data Systems Division, P.O. Box 390, Poughkeepsie, New York 12602.* Dr. Symonds is a senior technical staff member at the Myers Corners Programming Laboratory in Poughkeepsie, New York. He was educated in England, where he graduated from Oxford University with First Class Honours in physics in 1962; he received a D.Phil., also in physics, from the University of Sussex in 1965. Dr. Symonds joined IBM United Kingdom, Ltd., in 1965 as a systems engineer, and transferred to the United States organization in 1970, as data base research manager at the IBM Cambridge Scientific Center. Since then he has performed a variety of professional and managerial assignments related to programming. These have included original work on relational data base implementation techniques and management of the MVS, VSE, and VM IPO (Installation Productivity Option) organizations. Dr. Symonds' most recent assignment was that of manager of special projects in IBM Information Services Software Development, where he was responsible for setting up and directing the plan for IBM's initial set of knowledge-systems offerings.

Reprint Order No. G321-5267.