KNOWLEDGE ENGINEERING

A computer program consists of a sequence of instructions that access and modify a storage space. The instructions are usually directly executed by the hardware. However, an indirect execution is also possible, in which the instructions are executed by another software called an interpreter. The storage space of the interpreter can be divided into two parts, a program part that contains instructions for the interpreter and a data part that contains the data to be manipulated by the interpreted program. The main advantage of this approach is that it provides greater flexibility in designing the instruction set, which is particularly useful for exploratory work. It has also proved to be very desirable in designing artificial intelligence (AI) systems, especially knowledge-based systems, since it provides a richer view of instructions, such as dynamically changing programs and very complex execution semantics. In knowledge-based systems, the interpreter is called the inference engine while the program typically consists of a collection of rules and the storage space consists of a collection of facts.

The term *knowledge engineering* was coined by Feigenbaum in the early 1980s (1) to refer to the systematic steps needed to implement knowledge-based systems. In particular, it refers to systems where the knowledge base is in the form of a single do-loop containing a number of guarded statements called rules. Conceptually, the execution of these rule-based systems consists of a series of cycles. The first step in each cycle is the evaluation of all the guards based on the current content of the storage space. The execution terminates if all the guards are false; otherwise, a true guard is selected and the corresponding actions are performed which results in changes to the storage space. This inference procedure illustrates a simple forward-chaining execution semantic that is similar to the traditional way of executing loops. It can be embellished in several ways, such as constraining the set of true guards and using multithreaded execution, incremental match algorithms that use results of previous iterations, backtracking, and backward chaining. In the last case, the inference engine attempts to find a sequence of rule selections that is guaranteed to result in establishing a given postcondition (or goal).

Knowledge engineering shares many objectives with software engineering, including the development of tools and techniques for making the knowledge base modular and for assessing its performance, reliability, and complexity. However, there are also fundamental differences. Software engineering assumes that programmers can independently design and implement a program once they are given the requirements specification. In knowledge engineering, on the other hand, the programmer (or knowledge engineer) must understand how human experts perform a task and then capture and codify this knowledge in the form of rules that are added to the knowledge base. Thus, in addition to modularity and other software engineering concerns, knowledge engineers must constantly worry whether they have asked the right questions, whether they have asked all the questions, and whether they have correctly encoded the answers in the form of rules.

Knowledge engineering is also tied to data engineering (or data management) which is concerned with methods of storing and accessing large amounts of data. As the size of the data portion of a knowledge base increases, it becomes necessary to use data engineering methods to ensure that the set of rules in the knowledge base is consistent with the facts stored in the system. Thus, when a rule or a fact is modified or when a new rule or fact is added to the knowledge base, it is necessary to ensure that the facts are still consistent with the rules (2). To facilitate this task, several systems have integrated the handling of rules and data into one common framework; examples include Postgres (2) and Starburst (3).

Knowledge-based systems, in the form of rule-based expert systems, have proved to be among the most successful commercial applications of AI research. Most AI research has attempted to solve very general or ill-specified problems, such as understanding natural languages, proving theorems, planning robot motions, and performing inference from first principles. While some of these techniques have shown promise for small (toy) problems, most of these methods have proved to be computationally intractable for realistic problem domains. Expert systems have bypassed this problem by requiring the identification of domain-specific rules to guide the inference process. These systems attempt to emulate the problem solving capabilities of human experts to attain high performance levels in a narrow problem area. Virtually all expert systems rely on a knowledge-based architecture. Also, they must be able to explain and justify their solutions, decisions, and recommendations. This narrow focus has enabled the development of effective expert systems for a variety of practical applications, such as medical diagnosis, system configuration, factory automation, seismic data analysis, etc.

While the narrow focus of rule-based expert systems facilitates the solution of industrial-strength problems, it tends to make these systems very brittle. That is, they can fail miserably for inputs that deviate even in minor respects from the encoded rules. Alternative methods have been proposed to address this problem. Examples include memory-based or case-based reasoning techniques that store a large set of sample inputs and outputs and use statistical techniques to infer appropriate responses to new inputs by matching them with the set of previous inputs. The problem with these methods is the difficulty of ensuring that a reasonably complete sample size has been obtained to reliably bound the behavior of the system.

The rest of this article is organized as follows. The next section gives a precise definition of various components of knowledge engineering. This is followed with discussions of development and assessment procedures. The article concludes with some future perspectives.

KNOWLEDGE-BASED SYSTEMS

The main objective of knowledge engineering is the acquisition and computerization of knowledge. The end product of the knowledge engineering process is a knowledge-based system composed of an inference engine and a knowledge base. In most approaches, the knowledge base consists of facts and rules that use the facts as inputs for making decisions. Rules in this framework consist of an **if**-part and a **then**-part where the **if**-part represents a condition and the **then**-part represents an action that potentially will be taken whenever the condition in the **if**-part is satisfied, that is, whenever the rule is ready to fire. Hence, rules, in contrast to facts, are active objects in the sense that they can perform computations whenever they become eligible to fire.

While rule-based programming is a key paradigm for designing knowledge-based systems, it is not the only approach. In recent years, object-oriented knowledge bases (sometimes also called frame-based systems) have gained some popularity. This method relies on a more passive object-oriented view of knowledge and is not considered further here.

The inference engine is subdivided into an interpreter and a scheduler. The interpreter generates new knowledge by firing rules while the scheduler selects which rules to fire in a particular context. One key idea of knowledge-based systems is the separation of domain specific knowledge (facts and rules in a rule-based knowledge representation framework) from other parts of the system. Due to the difficulty of computerizing domain specific knowledge, knowledge-based systems strongly rely on incremental system design; that is, a knowledge base will be designed and refined several times during the design process. One key claim that advocates of knowledge-based architectures make is that it is much easier, and therefore more cost-effective, to modify and extend a knowledge base compared to modifying a program in conventional programming languages that do not clearly distinguish between domain-specific knowledge and other parts of this system [for more details see (4)].

Another important characteristic of knowledge-based systems is the use of heuristic and approximate methods, and less reliance on traditional algorithmic approaches. Heuristics (derived from the Greek word *heuriskein* which means "to find") are rules of thumb that encode a piece of knowledge on how to solve a particular problem in a particular context. Heuristics are frequently used when it is not feasible to investigate all possible solutions algorithmically due to the complexity of the problems. The role of heuristics is to cut down time and memory requirements of search processes. In general, heuristic methods are not fool-proof and frequently focus on finding a suboptimal, satisfactory solution rather than an optimal solution. Heuristics are usually employed to solve ill-defined problems for which no mathematical technique by itself can yield a solution in a reasonable time. Heuristics are frequently vague and uncertain, and the contexts under which they are applicable are usually difficult to describe and formalize. Heuristic knowledge is frequently derived from experience rather than from scientific analysis.

Heuristics represent special knowledge that is useful in only a small number of application domains. This is in contrast to general knowledge that is useful for solving problems in many application domains. Examples of general knowledge include the rules of logic, probabilistic knowledge, general search techniques, such as backtracking, and so on. In a knowledge-based architecture, domain-specific knowledge is stored in the knowledge base whereas general knowledge is encoded within the inference engine. The history of AI in the 1960s and 1970s demonstrated that systems that rely strongly on general techniques are often not very suitable for solving real-world problems. Reacting to this failure in the 1980s, the focus of AI research shifted to the computerization of specialized knowledge and centered on the creation of systems with very specific problem solving skills. Consequently, the 1980s are frequently considered to be the decade of expert and knowledge-based systems. The belief underlying this period is that the problem solving capabilities of an intelligent computer system are proportional to the amount of problem specific knowledge in its knowledge base. Feigenbaum calls this fact the first principle of knowledge engineering, namely, "that the problem solving exhibited by an intelligent agent's performance is primarily the consequence of its knowledge base, and only secondarily a consequence of the inference method employed . . . . The power resides in the knowledge." There is some evidence that human experts also rely strongly on special knowledge while solving problems; for example, studies with chess and other experts suggest that the knowledge base of human experts in a particular application area can be as large as 70,000 rules [for more details see (5)]. Consequently, a strong belief during this period was that it is best to directly elicit domain-specific knowledge from human experts. Hence, knowledge acquisition, that is, the process of eliciting information from human experts, gained significant attention in the early 1980s.

In conventional procedural languages such as C, C++, or Ada, computations perform data changes or execute other commands, such as "print the value of *x*," "send message *m* to object *o*," or "call procedure *p* with parameter 3." Most importantly, these commands are activated imperatively in procedural languages. Rule-based programming is quite different in the sense that rules are never activated imperatively; that is, a programmer never says "execute rule *r*." Instead, rules are active all the time and can automatically perform computations as soon as their activation condition is satisfied. Two different forms of rule-based programming can be distinguished: data-driven programming and goal-oriented programming. In data-driven rule-based programming, data changes trigger the firing of rules which then perform further data changes that trigger other rules to fire, and so on. Data-driven programming relies on a forward chaining approach in which inference is performed from facts to conclusions. To illustrate the previous discussions, consider the following rule: **if** "the balance of an account becomes negative" **then** "inform the bank manager."

This rule will actively check balances of bank accounts, and perform its action if a data change occurs that makes the balance of an account negative. Typical languages in this group are CLIPS and languages of the OPS-family. Also, research in active databases seeks to integrate data-driven rules with conventional databases, and active database systems such as Postgres (2) and Starburst (3) have emerged from these works.

In goal-oriented rule-based programming, rules are selected and fired with respect to a given goal relying on goal-subgoal mechanisms. In general, goal-oriented approaches rely on a backward chaining approach in which inference is performed from conclusions to antecedents. To illustrate how this approach works, assume that we have a rule for inferring grandchild relationships from child relationships. This rule will be fired in this case if the current goal is to infer all grandchildren of a person named Fred. Languages such as Prolog, EMYCIN, and many diagnostic expert systems and shells rely on this programming style.

In the past decade, rule-based systems have become more object-oriented. For example, CLIPS 6.0 supports the definition of modules and provides encapsulation and several constructs for organizing rule bases more transparently. Also, hybrid shells, such as KEE and NEXPERT, have been developed to support both goal-oriented and data-driven rule-based programming in an object-oriented framework.

Human expertise frequently involves vague and uncertain knowledge, especially in applications that are predictive or diagnostic in nature. In such applications, rules do not lead to decisions directly, but rather provide evidence for or against a particular decision, and the evidence collected from different rules is combined, and the decision with the highest combined evidence is selected. Various models have been proposed to support this form of decision making: Bayesian approaches that rely on probability theory and Bayes's theorem, approaches that rely on Dempster-Shafer's theory of evidence, certainty factors, and other pragmatic approaches. Another problem is that domain experts frequently use terminology whose precise boundaries are very difficult to define. For example, a rule might state **if** "the patient is old" **then** "there is suggestive evidence for not prescribing drug *d*."

However, even two experts will frequently disagree on the precise boundaries of the term old. Is 55 already considered to be old, or should the boundary be 60? Fuzzy sets and their underlying possibility theory have been found to be very useful for approximating the vagueness inherent in terminology and in natural languages in general. Rather than classifying a patient as either old or young, in this approach a number in the interval [0,1] is computed that measures the oldness of a particular patient. The advantages of this approach are smooth decision making (if a patient is only a little older, the negative evidence produced by the rule will increase only slightly) and a very compact and transparent form of representing knowledge. For a more detailed discussion of approaches to cope with possibilistic, probabilistic, and other forms of imperfect knowledge in knowledge bases see (6).

DESIGNING KNOWLEDGE BASES

The following persons are important when designing knowledge bases: The knowledge engineer who usually is an AI expert and is well-versed in knowledge representation, inference techniques, in tools and methodologies that facilitate the design of expert systems, and in hardware and software technologies to be used for implementing expert systems. Knowledge engineers usually have a strong background in computer science but lack expertise in the application domains of knowledge-based systems. Consequently, the participation of a domain expert is essential for the success of developing knowledge-based systems. The knowledge engineer will usually interview the domain expert to become familiar with the application domain and to elicit the domain knowledge. This process of acquiring the domain knowledge of a human expert is called knowledge acquisition. Other persons that participate in the design of a knowledge-based system are the end-users of the system and the clerical staff whose responsibility is to add data to the knowledge base.

It is common practice to subdivide the design of a knowledge-based system into five major stages (7):

- Identification
- Conceptualization
- Formalization
- Implementation
- Testing

The objective of the identification phase is the definition of the scope of the knowledge-based system and identification of the problems that the proposed system must solve. Also, knowledge concerning the characteristics of the application area, the available resources, and the persons who will participate in the design and use of the knowledge-based system has to be acquired.

The main objective of the second phase is the acquisition of the terminology and jargon of the application domain; that is, the key concepts, relations, and control mechanisms that the expert uses in his or her problem solving have to be identified. In addition, subtasks, strategies, and constraints related to the tasks to be automated by the knowledge-based system have to be acquired from the domain expert.

The first two phases are independent of the actual delivery platform of the knowledge-based system. The formalization phase starts with the selection of the language and environment in which the knowledge-based system will be designed and used. (These decisions can also be made earlier in the design cycle.) The key concepts and relations are mapped to a formal representation which is dependent on the languages and tools that are used to design and implement the knowledge-based system.

The objective of the implementation phase is to transform the formalized knowledge into a working prototype system. Representation forms within the framework of the chosen development platform for the knowledge formalized in phase 3 have to be developed. Also, the formalized knowledge has to be made compatible so that it can be integrated into a single system. This step usually involves combination, transformation, and reorganization of various pieces of knowledge to eliminate mismatches between fact representation, rule representation, and control information. Furthermore, the control strategy and control knowledge have to be mapped into code that can be executed by the underlying delivery platform.

Finally, in the testing phase, the prototype system is validated and its problem solving capabilities are evaluated (a more detailed discussion of this phase will be given in the next section).

Knowledge acquisition is currently considered one of the most critical steps for designing knowledge-based systems. Buchanan et al. (7) define knowledge acquisition as "the transfer of problem solving expertise from some knowledge source to a program." In other words, knowledge acquisition centers on the problem of eliciting knowledge from an expert and coverting it into a form so that it can be stored in a knowledge base. The basic model of knowledge acquisition is that the knowledge engineer mediates between the domain expert and the knowledge base, and acquires domain knowledge manually through interviews with the domain expert. Key problems that have to be solved by the knowledge engineer when following this approach [for more detail see (8)] include how to:

- Organize and structure the knowledge acquisition process
- Collaborate efficiently with the domain expert
- Conduct interviews with the domain expert
- Conceptualize the application domain
- Trace the decision making process to acquire knowledge
- Verify and validate the acquired knowledge

However, the approach that considers the knowledge engineer as a mediator between the domain expert and the knowledge base has been recently criticized (9,10), and it has been proposed to develop computerized, interactive tools to assist the domain expert in structuring domain knowledge. Many such tools have been designed in the last decade to directly communicate with the expert with a minimum of intervention from the knowledge engineer (a good survey of these tools can be found in Ref. 11). The main idea of these approaches is to systemize the knowledge-engineering process, thereby increasing the productivity of the involved knowledge engineers and domain experts. However, although these tools facilitate the conceptualization phase, a significant amount of work still has to be done manually by the knowledge engineer in collaboration with the domain expert.

Several more far-reaching approaches to automating knowledge acquisition have been described in the literature. One idea is to develop a meta theory of expertise in a restricted class of application domains (such as equipment malfunctions or for identifying biological organisms) and to provide a knowledge representation and acquisition framework that has been tailored for such applications (12,13). Another very popular approach is to use inductive generalization processes to derive expert-level knowledge from sets of classified examples (10). When using this approach, the expert only provides a set of examples with the class the example belongs to, and an inductive learning algorithm is used to learn the classification algorithm. Approaches that are currently used to learn and represent classification strategies include decision trees, neural networks, naive Bayesian classifiers, and belief networks (for a survey see Ref. 14).

Although there has been significant progress in the development of computerized tools for knowledge acquisition, it faces several challenges for which satisfactory solutions still have to be found (for a more detailed discussion of these and other points see Ref. 15). First, there is the problem of implicit knowledge that refers to the fact that experts are frequently not aware of what they know and, even worse, that often the most relevant knowledge for knowledge bases turns out to be the knowledge that the experts are least able to talk about (16).

The second problem is that knowledge acquisition is a constructive modeling activity (17) in which the expert, jointly with the knowledge engineer, describes and formalizes his knowledge. That is, according to this view, the expert's knowledge is not something that can be directly accessed, but rather needs a creative, cognitive process to be elicited. Current knowledge acquisition tools seem to be too simplistic to support this activity. A third problem is that for a knowledge acquisition tool to be successful, it has to be able to question the expert intelligently. It is unacceptable for the tool to ask the expert redundant or trivial questions that waste the expert's time. However, it turns out that such intelligent questioning strategies are very difficult and expensive to develop, even for application-class specific tools. Finally, the diversity of knowledge poses another challenge for knowledge acquisition tools. For example, a domain expert might use knowledge that consists of simple heuristics, fuzzy sets, Bayesian rules, simple logical rules, frame-based concept hierarchies, hill-climbing, and so on, when solving a particular task. This fact makes it very difficult to develop a comprehensive and complete knowledge acquisition tool.

One critical problem when designing knowledge-based systems is to encode the beliefs and heuristics a domain expert uses in his or her problem solving approach. The following problems complicate the design of knowledge-based systems:

- Heuristics are usually complex, hard to understand, and, therefore, nontrivial to computerize.
- The scope of a heuristic, that is, the context in which a particular heuristics is applicable, is frequently hard to determine.
- Frequently, it is not clear what level of detail is necessary when computerizing heuristics to obtain a satisfactory system performance. In some cases, very simple heuristics are quite suitable to solve the problem at hand.
- Frequently, it is very hard to predict if a particular set of heuristics will solve the problem at hand.

Since knowledge-based systems strongly rely on heuristic information, it is very important in the early design stages to evaluate the problem solving performance of a set of heuristics with respect to a set of example problems. This will validate the acquired heuristics, demonstrate areas in which knowledge is missing or not detailed enough, will reveal discrepancies and inconsistencies between the domain expert's solution and that of the system, and will give a better feeling concerning the complexity of particular tasks to be automated. Consequently, because of the special characteristics of the heuristics outlined in the previous paragraph, rapid prototyping combined with incremental development are the most popular approaches for designing knowledge-based systems. Rapid prototyping is an approach in which first a simplified version, usually a demonstration version, is devised, implemented, tested, and evaluated. This prototype is then extended to obtain a system with complete functionalities. Incremental development refers to an approach in which a system is designed and implemented following multiple iterations. Initially, a version of the system is designed and implemented to provide only basic capabilities and operations. This system is then evolved from solving simple tasks to solving increasingly hard tasks, improving incrementally the organization and representation of knowledge in the knowledge base.

EVALUATION

As knowledge-based systems become larger and larger and as they are used more and more for critical applications, such as medical diagnostic and manufacturing systems, it becomes necessary to develop systematic and rigorous methods for ensuring high quality. Standard software engineering techniques are not directly applicable due to the dynamically evolving nature of knowledge-based systems and the need for close cooperation between knowledge engineers and domain experts. Over the past decade, a variety of approaches have been used to move the development of knowledge-based systems from an ad hoc art form to an engineering discipline with well-defined criteria and methods.

There are two major dimensions to quality assurance for knowledge-based systems. The first one mirrors software engineering and classifies the quality criteria into functional and nonfunctional categories. Some functional criteria that are commonly used are consistency, completeness, correctness, and reliability, while some nonfunctional criteria are modifiability, usability, performance, and cost. Consistency means that the rules in the knowledge base do not contradict other rules or facts, completeness means that the inference engine can find a solution for all possible inputs, correctness means that the output agrees with that of a test oracle (usually a human expert in the application area), reliability is the probability of error-free operation for a specified duration under specified operational conditions, modifiability means that it is easy to make changes to the knowledge base, usability means that it has a user-friendly interface, for example, it can generate easily understandable explanations, performance is a measure of the response time and resource requirements, and cost includes the development time and cost.

The second major dimension, which is not usually considered when assessing conventional software, is the distinction between the quality of the knowledge base and that of the interpreter (inference engine). Functional features, such as correctness and reliability, and nonfunctional features such as usability and performance, can be affected significantly by the quality of the inference engine used in executing the knowledge base. For the same knowledge base, it is possible for a powerful inference engine to yield a better quality response in a shorter time than a naive inference engine.

The above quality criteria can be viewed in a qualitative or a quantitative way. Qualitative criteria include factors such as the thoroughness of independent reviews and checklists, satisfaction of various test coverage criteria, absence of inconsistencies, and so on. Quantitative criteria include reliability, performance, and cost assessment. The following two subsections review methods for ensuring high quality and discuss some quantitative quality measures, respectively.

Assurance Methods

Methods for assuring the quality of knowledge-based systems can be classified into two groups, namely, deterministic methods and probabilistic methods. Deterministic methods, such as consistency checks, ensure that a given quality goal will be definitely achieved while probabilistic methods cannot provide such guarantees.

Deterministic methods consist of a variety of model checking strategies for ensuring the absence of inconsistencies, incompleteness, and livelocks in the knowledge base (18,19). These methods differ depending on the formalism used to represent rules and facts (some formalisms that have been considered are propositional logic, first order predicate calculus, production rules, and frames). A variety of software tools have been developed, mostly for representations in the form of first order predicate calculus, to perform a systematic analysis of the knowledge base (19). Quality objectives that have been targeted include showing the absence of inconsistencies or contradictions in the knowledge base, identifying redundant rules, namely those that are subsumed within other rules or those that can never be fired, checking whether there is a circular dependency between the rules that can result in nonterminating inference procedures, and checking whether all input conditions have been accounted for. Identification of redundant rules and their removal results in a more concise knowledge base which is important for simplifying subsequent maintenance activities (19).

Model checking is computationally expensive and also of limited use. For example, it cannot reveal the presence of missing conditions or incorrect actions. Inspection and review by another expert or systematic testing strategies are more effective at revealing these types of faults. Inspection is usually done on the basis of a checklist containing a list of items that must be verified. This includes checking that all situations have been covered, that the firing conditions are correct, that the actions are correct, that the values of constants are correct, that the explanation text matches the inference chain encountered, that all the rules and facts have been read and found to be correct, and so on. Inspection is a laborious process, and its effort can increase nonlinearly as the size of the knowledge base increases. To be fully effective, it should be ensured that the review is done by an *independent* expert.

Testing is based on the execution of the knowledge-based system in a controlled environment. Three major steps are involved, namely, the selection of test cases, the execution of test cases, and determining the correctness of the result. Test cases can be selected either in a random or a nonrandom way. Random testing according to the operational usage distribution is necessary for reliability assessment (see the next section). Nonrandom testing can be used for ensuring the satisfaction of various test coverage criteria, such as ensuring that every rule is activated at least once or that the conditions in every rule take all possible outcomes at least once. It can also be used to perform stress testing, such as selecting boundary value test cases, selecting extreme and limiting values, ensuring that all critical situations are covered by at least one test case, and so on.

Execution involves running the system in a real or simulated environment. This is easy for applications where each run of the expert system is independent, such as a medical diagnostic program or a system for assisting with decisions, such as a mortgage evaluation system. It is much more difficult for reactive systems, such as process-control systems, patient monitoring systems, and others. In these cases, it is necessary to use a simulator, but this itself can be a source of additional failures.

The final step is checking whether the output of the program is correct. This task is difficult for knowledge-based systems since, unlike in most conventional software testing, there is no formal specification against which the result can be compared. This requires a human expert to give a solution against which the program's output can be compared. The comparison is nontrivial since there may be acceptable variations in the output, so a simple bit-by-bit comparison is not correct. One approach is to use the Turing test, that is, provide the program's answer and the expert's answer to an independent expert and see if the expert can identify which outcome is from the program; if not, then the program is assumed to be correct (19,20).

To facilitate extensive testing, some of the test data generation and output checking effort can be reduced by automatically extending the set of test cases and using interpolation strategies to simulate a test oracle (20). Also, since knowledge-based systems are constantly evolving, regression testing is very effective (20). That is, all the inputs and outputs are retained in a database and automatically re-executed after modifications to the knowledge base. This ensures that new faults will not be introduced as a result of changes to the knowledge base.

Quality Measures

While inspection and testing can result in high quality systems, they do not provide any indication of how good the quality is. Model checking, where applicable, can provide a rudimentary (binary) measure of the quality of the knowledge base. However, in addition to its theoretical and practical difficulties, model checking cannot provide answers to questions such as, "How difficult is it to modify the knowledge base?" or "How much time does a knowledge engineer need to understand the knowledge base?" A number of quantitative measures have been proposed to answer these questions, including complexity, reliability, and performance measures.

Complexity measures can be classified into two categories, namely, bulk measures and rule measures (21). Bulk measures provide some estimate of the size of the knowledge base, such as the number of rules, the number of variables, the number of occurrences of each variable, the depth and breadth of the decision tree, and so on (20,21). Rule measures examine the interaction between the rules and facts in the knowledge base. Some rule measures that have been proposed include the number of variables that occur in a rule, the number of input parameters of a rule, the number of output parameters of a rule, the number of rules that can potentially affect a rule, the number of rules that can potentially be affected by a rule, the length of the longest possible inference chain, and so on (21). While complexity measures provide some guidelines toward the design of more easily understandable and maintainable knowledge bases, there are some problems. For example, it is difficult to relate these measures directly to the parameters of interest, such as the time that is needed to understand the knowledge base. Also, most complexity measures lack adequate scientific foundation and are not very accurate predictors, at least for conventional programs.

In contrast to complexity measures, reliability measures are formally well-defined and have been developed fairly well for hardware and to a lesser extent for software. The first step in estimating statistical software reliability is to determine the operational profile (22) which is defined as the probability that a given input will be selected during operational use. Then, one can use variations of either software reliability growth models or the sampling model (23). These variations must consider the way knowledge-based systems differ from conventional programs, such as the use of heuristics to obtain suboptimal solutions, the improvement in the quality of the output as the depth and breadth of the search space is increased, and the possibility of learning as the system acquires new information.

In software reliability growth models, the software is tested according to the operational profile and, whenever a failure occurs, the fault is removed and the testing is then resumed. The reliability is estimated from the failure history, that is, the time interval between successive failures. A version of the Musa–Okumoto logarithmic model adapted for knowledge-based systems appears in Ref. 24. In the sampling model, the input space of the software is partitioned into a number of equivalence classes. Then, test cases are randomly selected from the partitions according to the operational profile. A model based on this approach appears in (25). These methods work reasonably well for ordinary programs but are not suitable for highly reliable programs (23).

Performance measurement for knowledge-based systems is relatively easy unless there are dependencies between rules. The only problem is to determine the length of the longest inference chain. The system performance also depends on the performance of the inference engine. A Markov process model has been developed for the case where rules are grouped into separate modules (26). The parameters to be estimated in this case are the transition probabilities, that is, the probability of moving from one module to another, and the time that is spent in a module.

For real-time process-control systems, the reliability and performance of the system are both important since the system can fail if either the output is not correct or if it is not produced in a timely way. This is captured in the performability measure developed for real-time knowledge-based systems (24). It uses the distribution of the time to produce an output and the "acceptability" or quality of the output as a function of time.

All the quantitative measures for knowledge-based systems have been developed within the last few years. While these have been applied to pilot projects, more experiments and validation are needed before they can be routinely used.

FUTURE PERSPECTIVES

One major challenge that many companies currently face is how to transform large collections of corporate data into knowledge that can be used to conduct their business more successfully and efficiently. The traditional approach for creating knowledge bases in which a knowledge engineer elicits knowledge from a domain expert who is familiar with a particular data collection seems to be less and less practical in transforming large stores of data into useful knowledge. The recent progress in automated scanners and other automated electronic devices, in database technology, and in the World-Wide Web has resulted in a flood of data which are impossible to analyze manually even by domain experts. For example, satellites in space transmit so many images that it is no longer feasible to manually inspect even a small fraction of the data. Even worse, there may not be any domain expert for some data collections. However, these large data collections frequently contain valuable information. For example, cash register records for supermarkets might provide valuable information regarding customer preferences, which can be very useful at reducing cost and improving customer service.

This trend is characterized by the fact that, on the one hand, knowledge that was not available before is now available in computerized form, whereas, on the other hand, domain experts have less and less knowledge concerning the contents of their data collections. Moreover, the availability of large computerized data collections facilitates the automatic validation of hypothesis and knowledge concerning these data collections.

This new development has created the need for new approaches to designing knowledge bases. Consequently, in recent years, to face this challenge, the new field of knowledge discovery and data mining (KDD) has emerged (for surveys see Refs. 27 and 28). KDD centers on the development of computerized tools that facilitate the domain expert's job of making sense out of large amounts of data. The various tasks addressed by KDD research include finding interesting patterns in databases, creating and testing of hypotheses, dependency analysis, learning class descriptions from examples, cluster analysis, change analysis, detection of instances that significantly deviate from the standard, and creating summaries. Moreover, data warehousing plays an important role in the KDD process. Data warehousing creates an integrated view of a data collection, and cleans and standardizes its content so that data mining algorithms can be applied to it. Technologies that play an important role for KDD include visualization, statistics, machine learning, and databases.

This trend of integrating multiple AI components (pattern recognition, natural language understanding, image processing) to enhance knowledge-based systems is likely to continue in order to cope with unstructured information in various environments. It raises the issue of designing meta-expert systems, that is, systems that have knowledge of and can effectively use multiple expert systems to solve difficult practical problems. For example, an expert system for traffic coordination may use image processing and pattern recognition to classify objects in the environment, a behavior expert to evaluate likely behavior of motorists and pedestrians, a motion analysis expert to determine the likely trajectory of various objects, and a traffic expert to know the best action for minimizing accidents and maximizing traffic flow. All this requires reusable expert systems that can easily interact with other AI components. It also requires rigorous quality assurance since a single poor quality expert system can adversely affect a large number of applications.

BIBLIOGRAPHY

1. E. A. Feigenbaum and P. McCorduck, *The Fifth Generation: AI and Japan's Computer Challenge to the World,* Reading, MA: Addison-Wesley, 1984.
2. M. Stonebraker, The integration of rule systems and database systems, *IEEE Trans. Knowl. Data Eng.,* **4**: 415–423, 1992.
3. J. Widom, The Starburst active database rule system, *IEEE Trans. Knowl. Data Eng.,* **8**: 583–595, 1996.
4. D. A. Waterman, *A Guide to Expert Systems,* Reading, MA: Addison-Wesley, 1985.
5. R. Reddy, The challenge of artificial intelligence, *IEEE Comput.,* **10** (9): 86–98, 1996.
6. S. Parsons, Current approaches to handling imperfect information in data and knowledge bases, *IEEE Trans. Knowl. Data Eng.,* **8**: 353–372, 1996.
7. B. G. Buchanan et al., Constructing expert systems, in F. Hayes-Roth, D. A. Waterman, D. B. Lenat (eds), *Building Expert Systems,* Reading, MA: Addison-Wesley, 1983, pp. 127–167.
8. K. L. McGraw and K. Harbison-Briggs, *Knowledge Acquisition Principles and Guidelines,* Englewood Cliffs, NJ: Prentice-Hall, 1989.
9. B. R. Gaines and M. L. G. Shaw, Eliciting knowledge and transforming it efficiently to a knowledge-based system, *IEEE Trans. Knowl. Data Eng.,* **5**: 4–14, 1993.
10. J. R. Quinlan, Knowledge Acquisition from Structured Data, *IEEE Expert,* **6** (6): 32–37, 1991.
11. *Int. J. of Man-Machine Studies,* **26** (1): 1987; this issue discusses the features of the knowledge acquisition tools MOLE, KNACK, KRITON, OPAL, AQUINAS.
12. S. Marcus, (ed.), *Automating Knowledge Acquisition for Expert Systems,* Norwell, MA: Kluwer Academic, 1989.
13. J. Diederich and J. Milton, Creating domain specific metadata for scientific data and knowledge bases, *IEEE Trans. Knowl. Data Eng.,* **3**: 421–434, 1991.
14. U. M. Fayyad et al., *Advances in Knowledge Discovery and Data Mining,* Cambridge, MA: AAAI/MIT Press, 1996.
15. S. Mussi, Causal knowledge elicitation based on elicitation failures, *IEEE Trans. Knowl. Data Eng.,* **7**: 725–739, 1995.
16. D. C. Berry, The problem of implicit knowledge, *Expert Syst.: The Int. J. Knowl. Eng.,* **4** (3): 144–151, 1987.
17. K. M. Ford et al., Knowledge acquisition as a constructive modeling activity, *Int. J. Intel. Sys.,* **8** (1): 9–32, 1993.
18. C. F. Eick and P. Werstein, Rule-based consistency enforcement for knowledge-based systems, *IEEE Trans. Knowl. Data Eng.,* **5**: 52–64, 1993.
19. G. Guida and G. Mauri, Evaluating performance and quality of knowledge-based systems: foundation and methodology, *IEEE Trans. Knowl. Data Eng.,* **5**: 204–224, 1993.
20. K. Finke et al., Testing expert systems in process control, *IEEE Trans. Knowl. Data Eng.,* **8**: 403–415, 1996.
21. M. B. O'Neal and W. R. Edwards, Jr., Complexity measures for rule-based programs, *IEEE Trans. Knowl. Data Eng.,* **6**: 669–680, 1994.
22. J. D. Musa and K. Okumoto, A logarithmic Poisson execution time model for software reliability measurement, *Proc. 7th Int. Conf. Softw. Eng.,* 230–237, 1984.
23. F. B. Bastani and C. V. Ramamoorthy, Software reliability, in P. R. Krishnaiah and C. R. Rao (eds.), *Handbook of Statistics, vol. 7,* Amsterdam: North-Holland, 1987, pp. 7–25.
24. I.-R. Chen and F. B. Bastani, On the reliability of AI planning software in real-time applications, *IEEE Trans. Knowl. Data Eng.,* **7**: 4–13, 1996.
25. D. E. Brown and J. J. Pomykalsi, Reliability estimation during prototyping of knowledge-based systems, *IEEE Trans. Knowl. Data Eng.,* **7**: 378–390, 1995.
26. I.-R. Chen and B. L. Poole, Performance evaluation of rule grouping on a real-time expert system architecture, *IEEE Trans. Knowl. Data Eng.,* **6**: 883–891, 1994.
27. U. M. Fayyad, Data mining and knowledge discovery: making sense out of data, *IEEE Expert,* **11** (5): 1996.
28. E. Simoudis, J. Han, and U. Fayyad (eds.), *Proc. 2nd Int. Conf. Knowl. Discovery & Data Mining,* AAAI Press, 1996.

FAROKH B. BASTANI
University of Texas at Dallas

CHRISTOPH F. EICK
University of Houston