A Review of Expert Systems Evaluation Techniques*

Peter D. Grogono, Alun D. Preece, Rajjan Shinghal, and **Ching Y. Suen**
Center for Pattern Recognition and Machine Intelligence
Department of Computer Science, Concordia University
1455, de Maisonneuve Blvd West,
Montreal, Canada H3G 1M8

Abstract

Although expert systems have been in use for many years, evaluation techniques are neither frequently applied nor widely known. Our research group has been investigating techniques for evaluating expert systems since 1989, and we have developed a methodology for the systematic design and implementation of expert systems of many different kinds. Our method begins with the preparation of a *specification* that captures the needs of the application. During the implementation of the expert system, we recommend *verification*, to detect internal inconsistencies in the knowledge base, and *validation*, to check that the system is behaving in accordance with the specification. Verification can be partly automated: we review our own tool, COVER, and its application to verification of working expert systems.

Introduction

A large number of expert systems are now being used every day for a great variety of applications. Many of them have been successfully deployed in practice and enormous financial savings have been reported by their users.

As expert systems have become more widely used, their complexity has grown substantially. Many expert systems derive their knowledge base from multiple human experts, while others consist of clusters of different expert systems, each of which contains its own knowledge base performing a specific task. Some expert systems have become so complex that it is not possible to evaluate their performance without reliable and robust techniques. As a result, evaluation has become an active area of research and a number of manual and automatic techniques have been proposed and developed in recent years. Indeed, during the past five years, this subject has attracted the attention of a large number of researchers around the world, leading to the publication of numerous papers ([Grogono *et al.*, 1991] is a recent survey). However, in spite of all these efforts, techniques for evaluating expert systems are still in their infancy when compared with those developed in the field of software engineering.

From the technical point of view, the use of software engineering processes such as modularization, object-oriented design, and formal specification, has lead to products with the desirable properties of maintainability, reusability, verifiability and, hence, correctness and reliability. However, direct application of such techniques to expert systems is not straightforward. We have investigated and developed some new techniques for specifying, verifying, and validating expert systems. In this paper, we summarize our findings and viewpoints.

Specifying an Expert System

The traditional role of the specification in engineering is to act as a contract between suppliers and clients. The assumption that underlies this role is that a specification can state exactly *what* the system is supposed to do without saying too much about *how* the necessary tasks are to be accomplished. Thus the clients are satisfied that they are getting what they need and the suppliers have the greatest possible freedom for implementation.

The role of the specification of an expert system is different because much of the 'how' component of an expert system is typically contained in the inference engine. The 'what' component is contained in the knowledge base which, to a large extent, *is* the expert system. To accomodate the revised role, we believe that an expert system specification should contain both declarative and procedural components: it should serve as both a **contract** between suppliers and clients, and as a **blueprint** for designers and implementors.

There is, however, a tension between these two roles. To the extent that a specification is a contract, it should stabilize early in the development cycle, so that suppliers and clients agree about what is being developed. But the specification as a blueprint should

*The research described in this paper was funded by Bell Canada. We are particularly grateful to F. Coallier and J.-N. Drouin for their support and encouragement.

be allowed to change as implementors gain experience from prototype implementations. We therefore advocate splitting the specification into two parts, called the *problem specification* and the *solution specification*, that provide a contract and a blueprint, respectively. Based on these considerations, Figure 1 shows our proposed structure for an expert system specification [Batarekh *et al.*, 1990]. A small expert system may not require all of the components shown in this diagram.

The Problem Specification

The problem specification must define the problem that is to be solved and the constraints that an acceptable solution must satisfy. It should not state, explicitly or implicitly, how the solution is to be obtained. We divide the problem specification into two components: the *problem description* and the *constraints.*

The **problem description** should usually contain at least the **terms of reference** of the problem; the **objectives** of the project; and the **inputs** and **outputs** of the proposed system.

The *terms of reference* in the specification must provide accurate definitions for the specialized vocabulary, or *jargon*, of the problem domain as well as other terms used in the description of the problem and its solution. The *objectives* of the system include the needs of both the client and the end-users of the expert system. Each *input* of the expert system should be defined according to its source and its content. Similarly, *output* should be defined according to destination and content.

Every project is subject to **constraints**: the budget, the number of people available, the available machine capacity, and so on. In practice, it may be unrealistic to specify an absolute level of performance which the expert system must achieve to be accepted. Instead, performance constraints may be divided into *minimum* performance constraints, which must be met by the system, and *desired* performance constraints, which may be negotiated [Rushby, 1988]. For example, a minimum performance constraint for a medical expert system might stipulate that the system must never prescribe an overdose of any drug; a desired performance constraint for the same system might state that the system should give accurate diagnoses (according to some stated criteria for accuracy) in at least 90% of cases. If the delivered system achieved only 89.5% accuracy, it may still be accepted, but it would not be if it violated any minimum performence constraint.

The Solution Specification

The solution specification is the blueprint for the expert system; it should contain all the information that the design and implementation teams will need to complete the final product. Unlike the problem specification, which treats the expert system as a 'black box', the solution specification treats the expert system as a 'glass box': it makes the interior of the expert system visible. As shown in Figure 1, we divide the solution specification into two parts: the *conceptual model* and the *design model.* The conceptual model explains the proposed solution from the viewpoint of a human expert. The design model shows how the human expertise can be incorporated into a computer program.

The **conceptual model** is obtained by acquiring knowledge of the target domain, usually by discussions with a human expert or by generalizing from solutions obtained by experts in the past. It describes the knowledge that the expert system will contain, not how it will be represented. We have identified four components, or *submodels*, of the conceptual model.

The **problem-solving submodel** contains all of the knowledge needed to solve the problem. The **dialog generator submodel** describes the way in which the expert system interacts with the people who use it. The **cooperation submodel** describes the way in which the expert system interacts with other hardware and software systems. The **upgrading submodel** describes ways in which the system can be enhanced.

The conceptual model incorporates four kinds of knowledge. **Static knowledge** consists of relevant facts, descriptions, entities, relations, and implications drawn from the problem domain. **Primitive inference steps** enable the expert system to infer new knowledge from its static knowledge. Primitive inference steps are combined to form **task procedures**. Finally, **strategic knowledge** selects task procedures that appear to be appropriate for the current problem.

The **design model** is a plan for the implementation of the expert system. In conventional software engineering, design is a separate phase that ideally follows specification. With expert systems, at least at the current state of technology, we believe that specification and design are closely interwoven and that it would be unrealistic and undesirable to separate them.

The **architecture** component of the design model describes the overall structure of the expert system in terms of software components. The **knowledge representation** component determines the ways in which the knowledge can be used. Moreover, the knowledge representation and the inference engine must be compatible with one another. The design model should contain specifications for **utilities** that are needed by the system.

Specification and Evaluation

Every expert system is eventually evaluated. The amount of evaluation that is performed, and the importance attached to it, depend on the size, complexity, criticality, and other aspects of the expert system. Since the purpose of evaluation is to check that the expert system does what it is supposed to do, we can evaluate an expert system only if we already know what to expect. Thus evaluation and specification are closely related: the specification effectively tells us what to look for in evaluation. Evaluation has two aspects,

Specification	
Problem Specification	**Solution Specification**
Description	Conceptual Model
Terms of Reference	Problem Solver
Objectives	Dialog
Inputs and Outputs	Cooperation
	Upgrading
Constraints	Design Model
Performance	Architecture
Interface	Knowledge Representation
Environment	Inference Engine
Safety and Security	Utilities
Maintenance	
Development	

Figure 1: The Structure of an Expert System Specification

verification and validation, described in the next two sections.

Verification of Knowledge Bases

Just as English sentences must respect grammatical principles, the inference engine and the knowledge base of expert systems must respect certain syntactic principles. To verify the inference engine and the knowledge base is to check that they have indeed obeyed these principles. The inference engine is usually a conventional software component, and it can be verified by techniques described in the software engineering literature. Being non-procedural, however, knowledge bases cannot be verified by conventional techniques. In this section, we focus on the verification of knowledge bases, assuming that the inference engine has already been verified.

A knowledge base can be conceptually represented by a set of production rules. Each rule is a logic implication of the form

$$L_1 \wedge L_2 \wedge \cdots \wedge L_n \rightarrow H$$

in which the L's and the H are *literals.* The L's constitute the ***antecedent*** of the rule, and H, the ***hypothesis***, is the ***consequent*** of the rule. A knowledge base implemented as frames or semantic nets, can, for verification purposes, be converted to a set of rules in this form. This gives the knowledge base the clear semantics of logic, without being obscured by the details of implementation.

When formulating the rules of a knowledge base, the system designers must specify the following three sets in addition to the rules. The first set contains ***semantic constraints.*** Each constraint is a set of literals, $\{L_1, L_2, \ldots, L_n\}$. A constraint is satisfied if and only if not all of its literals are simultaneously true. The second set, ***labdata***, is the set of literals from which, in any given situation, a user can select a subset E to be input to the system, provided that E does not contain a semantic constraint. Each such E is called an *environment* for that situation. (We use the term ***labdata*** as an abbreviation for data used during laboratory validation, as described in Section .) The last set contains ***final hypotheses.*** For any given environment, the output of the system is a subset of this set. The set of final hypotheses is a subset of all the hypotheses in the knowledge base. The labdata and the set of final hypotheses will typically be disjoint sets.

When an environment E is input to an expert system, we expect that some or all of the literals in E will unify with the literals in the antecedent of some rule R. The rule R ***fires***, and we ***infer*** the hypothesis in the consequent of R. This inference may cause another rule to fire, which, in turn, can cause more rules to fire. The inference chain (that is, the sequence of rule firings), which is controlled by the inference engine applying logical deduction, terminates with the inferring of one or more final hypotheses.

Knowledge bases reflect the domain expertise of one or more human experts. This expertise may be imperfect, or it may be flawed by the heuristics, often fallible, employed by the experts in their working. Furthermore, the knowledge base designers may unwittingly misrepresent the expertise. As a result, when a knowledge base is built, it may contain ***anomalies.*** An anomaly is not necessarily an error, but it is desirable to check for anomalies, because they may indicate errors. There are four kinds of anomaly: ambivalence, circularity, redundancy and deficiency.

We illustrate the various kinds of anomaly using the following simple knowledge base. We suppose that the labdata is the set $\{A, B, C\}$ and that one of the semantic constraints is $\{D, F\}$.

$$\begin{array}{llll} R_1: & A \rightarrow B & R_4: & A \rightarrow F \\ R_2: & B \rightarrow D & R_5: & D \rightarrow A \\ R_3: & A \rightarrow D & R_6: & G \rightarrow F \end{array}$$

A knowledge base is ***ambivalent*** if, for some environment, the set of final hypotheses inferred contains a semantic constraint. In the example, the environment $\{A\}$ leads to the inferences B, D, and F (by rules R_1, R_3, and R_4). Since $\{D, F\}$ is a semantic constraint, these rules are ambivalent. Ambivalence may arise because the human experts hold incompatible views or because mistakes were made in entering the rules. Ambivalence may be acceptable if the rules contain certainty factors. Algorithms exist to compute the overall certainty even when complementary hypotheses are inferred [Shinghal, 1992].

Circularity exists in a knowledge base if, for some environment, we loop indefinitely in firing the rules. In the example, the rules R_1, R_2, and R_5 produce a loop $A \rightarrow B \rightarrow D \rightarrow A \rightarrow \cdots$ with environment $\{A\}$. Although circularity probably indicates a problem in the knowledge base, it need not be excluded altogether if the inference engine is smart enough to avoid firing rules whose consequents are known to be true.

A literal or rule is *redundant* if its omission from the knowledge base makes no difference to the inferences that can be made. In the example, rule R_3 is redundant because its effect (inferring D from A) can be achieved by applying rule R_1 and then rule R_2. Also, rule R_6 is redundant because its antecedent, G, can never be inferred. Redundancy is often acceptable in a knowledge base. Rule R_3, for instance, improves the efficiency of the knowledge base slightly by inferring D in one rule-firing rather than two. If the rules contain certainty factors, different paths to a conclusion may yield different certainty values.

A knowledge base is *deficient* when, for some environment, we infer no final hypotheses although according to the system specifications we should have inferred some final hypotheses. The sample knowledge base is deficient because it can make no inferences from the environment $\{C\}$.

Although an anomaly does not necessarily indicate an error in the knowledge base, it is nevertheless important to detect anomalies. In many cases, the cause of the anomaly turns out to be a simple clerical error, such as entering the same rule twice or misspelling the name of a literal. For this reason, checking tools do not usually attempt to repair anomalies: their task is simply to detect the anomalies and report them to the system designers.

Manual verification is impractical, for all but the smallest knowledge bases. Consequently, we need automated verification tools. A number of such tools have been proposed recently: we have compared some of them in [Preece *et al.*, 1992a].

COVER is a tool developed within our group. It is written in Prolog and C and runs on SUN workstations. The rules of the knowledge base must be written in, or converted to, a language based on first-order logic. For verifying a given knowledge base, COVER needs to be supplied with the set of semantic constraints, the labdata, and the set of final hypotheses associated with the knowledge base. COVER can be applied flexibly since its user, the system designers, can choose the level at which the knowledge base is to be checked for anomalies.

At the first level, *integrity checking*, COVER checks single rules to detect unfirable rules, dead-end rules, and missing values. The computational complexity is $O(N)$, where N is the number of rules examined. At the second level, COVER needs $O(N^2)$ time to look for pairs of rules that introduce redundancy or ambivalence into the knowledge base. At the third level, COVER traces inference chains to detect more general cases of ambivalence, circularity, redundancy and deficiency. The time required for this check is $O(b^d)$, where b is the average number of literals in a rule (the "breadth" of search) and d is the average length of a chain (the "depth" of search). Third-level checking is therefore slower than first- and second-level checking. Accordingly, whereas we may check single rules and rule pairs frequently during the various stages of the development of a knowledge base, inference chains are usually traced less frequently. The algorithms for the three levels of COVER are given in [Preece *et al.*, 1992b].

Validation of Expert Systems

Conventionally, validation is defined as the process of ensuring that a software system satisfies the requirements of its users. How validation is applied to expert systems depends chiefly upon the nature of the requirements specifications for these systems. In the past, expert system requirements were vague or merely implicit, often stating in general terms only the tasks that the system should perform—for example, that the system should emulate a human expert in a certain domain. In a case such as this, validation would involve comparing the human expert to the expert system, with the system being accepted if it were at least as competent as the human. Setting up such a comparison is not easy, however. One difficulty lies in choosing a set of test problems upon which to measure the competence of humans and machine. Ideally, we would want to select a set of problems which is *representative* of the domain in which we require the expert system to perform, without being too large [O'Keefe *et al.*, 1987]. Such a set is hard to create without a detailed specification of the expert system requirements. Therefore, one of the roles assigned to our expert system problem specification is to provide a test plan for validation.

There are at least two independent aspects to the test plan for an expert system. It is generally agreed in the literature on expert system validation [Gupta, 1990] that expert system must be subjected to two types of validation: **Laboratory validation** measures the performance of the system in an artificial environment, typically with the developers supplying test cases to the system in lieu of actual users. The expert system output is evaluated and, if it is unacceptable, the system is refined until it is satisfactory. **Field**

validation occurs once a system is deemed acceptable in the laboratory. A controlled field study is conducted, with real users and real or synthetic problem cases [Adelman, 1991].

While laboratory validation tends to reveal problem with the knowledge base of the expert system, field validation often reveals problems with the interface aspects of the system [Preece, 1990]. The test plan in the expert system problem specification needs to specify how both aspects of validation will be conducted. For the remainder of this section, we focus on laboratory validation.

Expert System Testing Issues

In testing expert systems, we seek to find the smallest possible *representative* set of test cases. Assuming that it will rarely be possible to test every possible case, we want to test a sufficiently wide range of distinct cases, while minimizing the size of the test set. Several factors make it difficult to cosntruct such a set. First, a representative set of cases may need to be large, due to the diversity of situations in which an expert system must perform. A well-known technique for minimizing this set is to look for *equivalence classes* within the input domain (that is, sets of cases each member of which will be treated similarly by the system, so that only one case from each equivalence class need actually be tested). Second, in many application domains, sufficient documented past cases will not be available for use in testing. In these cases, synthetic test cases must be created. Finally, expert system test cases tend to be complex, because the tasks that expert systems perform are inherently complex. Therefore, it will often be difficult to define each case in terms of an input-output relation: the input may be large and complex, and the 'correct' (or acceptable) output may be hard to determine.

Expert Systems Testing Approach

Some of the above difficulties are common to all software, and there are no 'magic solutions'. The approach we recommend is a combination of *functional testing* and *structural testing*, supported by software tools. An initial test set is developed according to functional testing criteria (described below). This set is run on the expert system, and a *post hoc* analysis is performed to examine the extent to which the testing has exercised all structural components of the system—any deficiencies are pin-pointed, and test cases are generated according to structural criteria (also described below) to 'fill the gaps'. This procedure is described in detail in [Zlatareva and Preece, 1993].

Functional testing bases the generation of test cases upon the requirements stated in the problem specification of the system. Test cases must be created for each task that the system is required to perform, and at each one of several possible levels of difficulty, where appropriate. Particular attention will be paid to testing critical functions of the system, such as those concerned with the protection of users.

Structural testing involves choosing a set of test cases which exercise as many structural components of the system as possible. Methods proposed for structural testing of rule-based systems are based on the notion of an *execution path* through the rule base—this is related to the notion of an inference chain, but more general in that it applies to knowledge bases expressed using procedural representation languages as well as those expressed using declarative ones.

Expert System Test Standards

The above techniques do not address the problem of choosing an appropriate level of performance for the system to achieve in order to be accepted. Another facet of this problem is the difficulty in defining a standard against which to judge the acceptability of the system. One of our objectives in separating the notions of *minimum level of performance* and *desired level of performance* in the problem specification was to recognize this difficulty so that there is some flexibility built into the acceptance testing process.

Two standard approaches may be used for defining the validation standard [O'Keefe *et al.*, 1987]. In some domains, it is possible to define a so-called *gold standard*—a generally-accepted 'correct' response for each test case. If a gold standard is available, then each test produces a boolean result, depending on whether the output of the system matches the gold standard or not. An *agreement method* must be employed when there is no gold standard—the performance of the system is compared with that of other performers (humans or other systems), and the system is deemed to be acceptable if it 'agrees' sufficiently closely with the other performers [O'Keefe *et al.*, 1987; Reggia, 1985].

Conclusion

We have outlined techniques for specifying, verifying, and validating expert systems. The specification of an expert system is important because validation is meaningless without it. Our approach to specification reflects the differences between expert systems and conventional software.

Verification checks the internal consistency of the knowledge base but cannot establish the correctness of its procedural component. The principal difficulty of validating an expert system is the acquisition of a sufficient set of problems and and corresponding solutions. In most cases, validation should include field testing.

In conclusion, our work to date has consisted of laying down a framework to guide practitioners in specifying, verifying, and validating expert systems. So far, certain aspects of the framework (in particular, automated verification and structural validation) have been developed more fully than others. Our future work will aim to develop the other areas, with the ultimate goal of creating an integrated methodology and set of tools for building high-quality expert systems.

References

Adelman, Leonard 1991. Experiments, quasi-experiments, and case studies: A review of empirical methods for evaluating decision support systems. *IEEE Transactions on Systems, Man and Cybernetics* 21(2):293–301.

Batarekh, Aïda; Preece, Alun D.; Bennett, Anne; and Grogono, Peter 1990. Specifying an expert system. Report for Bell Canada, Centre for Pattern Recognition and Machine Intelligence, Concordia University, Montréal, Canada.

Grogono, Peter; Batarekh, Aïda; Preece, Alun; Shinghal, Rajjan; and Suen, Ching 1991. Expert system evaluation techniques: a selected bibliography. *Expert Systems* 8(4):227–239.

Gupta, Uma G. 1990. *Validating and Verifying Knowledge-based Systems.* IEEE Press, Los Alamitos, CA.

O'Keefe, Robert M.; Balci, Osman; and Smith, Eric P. 1987. Validating expert system performance. *IEEE Expert* 2(4):81–90.

Preece, Alun D.; Shinghal, Rajjan; and Batarekh, Aïda 1992a. Principles and practice in verifying rule-based systems. *Knowledge Engineering Review (UK)* 7(2):115–141.

Preece, Alun D.; Shinghal, Rajjan; and Batarekh, Aïda 1992b. Verifying expert systems: a logical framework and a practical tool. *Expert Systems with Applications (US)* 5:421–436. Invited paper.

Preece, Alun D. 1990. DISPLAN: Designing a usable medical expert system. In Berry, D.C. and Hart, A., editors 1990, *Expert Systems: Human Issues.* MIT Press, Boston MA. 25–47.

Reggia, James A. 1985. Evaluation of medical expert systems: A case study in performance assessment. In *Proc. 9th Annual Symposium on Computer Applications in Medical Care (SCAMC 85).* 287–291. Also in Miller, Perry L., ed., *Selected Topics in Medical AI,* New York, Springer, 1988, pp. 222–230.

Rushby, John 1988. Quality measures and assurance for AI software. NASA Contractor Report CR-4187, SRI International, Menlo Park CA. 137 pages.

Shinghal, Rajjan 1992. *Formal Concepts in Artificial Intelligence: Fundamentals.* Van Nostrand, New York.

Zlatareva, Neli and Preece, Alun 1993. State of the art in automated validation of knowledge-based systems. *Expert Systems with Applications (US)* 7. To appear.