Expert System Development and Testing: A Knowledge Engineer's Perspective

R. T. Plant
Department of Computer Information Systems, University of Miami, Coral Gables, Florida

This article discusses the problems found in the validation and verification of a knowledge-based system for equity selection. These problems include the selection of test data, poor methodology, and the difficulties associated with using prototypes. The article then examines the possible techniques available to the knowledge engineer for improving validation and verification. The article discusses exhaustive testing, case-based testing, formal specifications, functional programming, critical testing, mutation testing, and reliability. Finally the article discusses the approach that the knowledge engineer would take in rewriting the equity selection system, one based on a rigorous development methodology that uses as many formal validation techniques as possible to raise the quality of the software produced.

INTRODUCTION

In this article we examine an aspect of expert system development with which we encountered difficulty during the creation of a knowledge-based system for equity selection—validation and verification. In creating our system, we underestimated both the amount of testing and resources required to adequately test the system so that it would satisfy user requirements.

The first section of this article details our original approach to validation and verification of our system, i.e., using random test data, and we discuss the weakness of this approach. A discussion of the alternative strategies to random testing and how each is applicable to testing different aspects of the system follows. We conclude by advocating the use of a rigorous development methodology that incorporates validation techniques and promotes software quality.

INITIAL SYSTEM DEVELOPMENT

The initial development of our equity selection and portfolio advisory system was undertaken using an approach similar to that advocated by Hayes-Roth et al [1] which involved an iterative five-stage model:

1. Identification: characterize the important aspects of the problem, e.g.,
   - Participant identification and roles
   - Problem identification
   - Recourse identification
   - Goal identification
2. Conceptualization: the key concepts and relations identified in the first phase are made explicit, e.g.,
   - What knowledge types to be used?
   - What is the interrelationship of the objects in the domain knowledge?
   - What is the heuristic content of the knowledge?
   - What are the constraints?
   - What is the information flow?
3. Formalization: the third phase of development aims at creating a model of the solution process. This is done through looking at characteristics of the data and the domain, e.g.,
   - Is there a need for certainty factors?
   - Is the data reliable?
   - Is the information a heuristic?
   - Is temporal information important?
   - What are the elicitation considerations?
4. Implementation: the mapping of the formalized conceptual information onto more concrete representations and their associated control structures.
5. Testing: the use of test examples in validating the

Address correspondence to Professor Robert T. Plant, Dept. of Computer Information Science, University of Miami, Coral Gables, FL 33124.

655 Avenue of the Americas, New York, NY 10010
0164-1212/92/$5.00

system with the aim of locating errors in the control structure, knowledge base, and inference rules.

This approach was followed and resulted in the creation of a working system that we considered to be a prototype. However, we found that the size and complexity of our prototype was too large to easily facilitate the recreation and experimentation necessary to achieve satisfactory results at low cost. To a large extent, this was because the system was written in LISP [2] and had extensive I/O operations. Thus, we learned that (1) it is extremely difficult to achieve a balance between scale and complexity such that the results of a prototyping operation are sufficiently significant and not applicable only to a trivial subset of the domain, and (2) prototyping should be approached through the use of a shell or environment as this would facilitate change more easily than a customized LISP system.

Having created a large prototype system, we then began to test it. The initial testing mechanism, advocated in the literature [1, 3] was a case-based approach in which cases solved by the system are compared to the same cases solved by a human expert. This approach is severely limited in its test coverage, as only the most widespread of conditions are considered. In testing our system we applied this method in conjunction with our expert, who checked the system's responses. We found that this approach promoted the testing of obvious situations but did not facilitate the testing of unusual, complex, or boundary conditions. Furthermore, because of the unstructured nature of the testing strategy, the method used a significant amount of the expert's valuable time, mainly because there was no testing plan from which the knowledge engineer and domain expert could test the relevant aspects of the system in a structured manner. We decided that for our final version it was necessary to create such a plan and this required investigation into other approaches to validation and verification.

TEST STRATEGIES

Validation and verification have been defined as follows: "Validation: The process of evaluating software at the end of the software process to ensure compliance with software requirements"; "the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase" [4].

It follows, therefore, that one of the keys to effective evaluation of the software and, consequently, to having valid and verified software, is to use effective techniques. Testing "is the process of executing a program (or part of a program) with the intention of finding errors" [5]. The basic principal on which testing is based is the application of test data (input) to the program in order to examine the correctness of the output with respect to the function of the program over that input.

Output = Program(Input)

One approach to software testing is to test all possible inputs and validate their subsequent outputs. This has the fundamental problem that for a practical system (even if finite in nature), the number of test paths necessary is extremely high; even with the aid of test data generators, the task for nontrivial systems is infeasible.

This problem is magnified for expert systems such as our equity selection system, as they tend to be nondeterministic; they represent partial rather than total functions. In addition, as expert systems often have to reason with incomplete input data, this raises the number of test cases. Therefore, it is our supposition that the use of exhaustive testing is at present implausible.

An alternative testing strategy is the "case" approach, in which the test data are based on some criteria. For example, the following criteria may be used:

- functional
- structural
- data
- random
- extracted
- extreme

In compiling test data for each criterion, the knowledge engineer must consider different aspects of the system, the data types of the system, and its specification, depending on the aspect to be examined.

The data used in "functional" cases is obtained by examining the functionality of the specification; for "structural" cases the logical structure of the code is examined; the test set for "data" cases originates in examining the data elements of the program; while the data used in "extracted" cases are obtained by examining other implementations, for example the prototype. The default strategy that we originally used can be classified as a "random" case strategy that uses random test data.

J. SYSTEMS SOFTWARE 1992; 19:141-146

An especially important test case strategy is testing "extreme" cases, as it is at the boundary conditions that knowledge-based systems are at their most valuable, yet also their most vulnerable, if there is not enough support knowledge to make correct deductions. Testing extreme cases is difficult, for the location of the boundaries is not always known because to a large extent this depends on the interaction of the knowledge in the knowledge base.

Thus, each of these testing strategies examines a different aspect of the system, and collectively they are valuable in raising the level of system correctness. However, note that even when all the strategies are used together, this does not guarantee total correctness.

THE FORMAL APPROACH

An alternative approach to software development is to specify the software system in such a way that it is correct at the design level. This approach, sometimes called "mathematical validation" [6], uses a formal specification style based on a formal language to produce a specification that can be reasoned about. Through a series of refinement steps, this specification can then be transformed into an implementation—fulfilling the program's "correctness argument" [7]. A number of proven methods are currently being applied to real world applications; these include VDM[7] and Z [8].

The language Z has been applied to the problem of specification within artificial intelligence. The language was inherently suitable to game playing, as these domains are finite and have well-defined structures. Teruel [9] has shown how Z could be applied to this problem domain by specifying games such as Ludo, Orthello, and Best of Three.

The use of a formal specification language such as Z can also be demonstrated by specifying the representations and the inference mechanisms that manipulate them. Gold [10] has given formal specification of a production system. The Z language can also be applied to the specification of knowledge bases and, as we experienced difficulties in maintaining the consistency of our prototype knowledge base, it was decided to develop a formal specification for the large-scale implementation of our system. This will allow us to ensure that there will be as little ambiguity, incompleteness, or inconsistency in the knowledge base as possible. This will also allow us to update the knowledge base easily, as the implementation-independent specification will ensure that any knowledge added, deleted, or modified is consistent with the previous knowledge base.

The use of formal specification is therefore limited to the static aspects of the system, i.e., mechanics such as conflict resolution, the structure of the rules, and the rules themselves. However, it is not possible to specify fully the system's dynamic aspects, i.e., the interaction of the rules, the self-modification of the rule base, or the heuristic nature of the rules. This is due to the nature of the systems themselves—the developer cannot know what the system is going to do for all situations and interactions of knowledge. This is analogous to the testing paths problem. Furthermore, even if it were possible to specify a large real world system, there are significant problems in refining this to an implementation.

A second formal approach to specifying a program is functional programming, in which the developer produces an "executable specification" about which mathematical proofs can be performed [11]. Many of the expert systems produced today by hand coding (as opposed to shell-based development) are still produced in LISP [2], which in its pure form, e.g., LISPKIT [12], can be classified as a functional language, yet there has been little or no documented effort in the production of proofs for these systems. This may be in part because most working LISPs, such as the FRANZLISP we used for implementing our equity selection system, are not pure but heavily dependent on side effects. However, it is possible to convert systems into pure LISP and so benefit from the formality imposed on them. This is a direction we are currently investigating; however, note that there is a significant overhead associated with the conversion process.

We feel that the use of formal specifications is currently limited to adapting the formal specification languages to enable partial system specifications to be made, e.g., the knowledge base, while the specification of the whole system is, in all but trivial domains, limited, if not impractical. However, the creation of such specifications considerably raises the level of system correctness. For example, specification of the knowledge base enables any inconsistencies in the domain knowledge or incompleteness in the representation to be identified and corrected.

STATISTICAL APPROACH

In the previous sections we have attempted to indicate why the techniques available to the knowledge engineer—testing formal specification and functional programming—are in reality severely limited in their ability to detect errors in large knowledge-based systems. An alternative approach to testing or specification is to use a statistical approach. This

entails attaching values to each item of knowledge, the values indicating the degree of certainty associated with that knowledge, then as the knowledge is manipulated, a certainty factor algebra can be used to combine certainties and produce a value indicating the degree of certainty associated with the result. This approach is not unique to knowledge engineering is extensively represented in the statistical decision theory literature.

Knowledge engineers initially used Bayes theorem on problems with attached conditional probabilities and the P-Function to manipulate these probabilistic measures. However, as Shortliffe [13] stated with regard to the MYCIN project, these approaches had to be abandoned "because there are large areas of knowledge that, although amenable in theory to the frequency analysis of statistical probability, defy rigorous analysis because of insufficient data and, in a practical sense, because experts resist expressing their reasoning processes in coherent probabilistic terms." Following this, several other approaches to the statistical evaluation of system knowledge have been considered, including the theory of fuzzy sets proposed by Zadah [14]. However, experts also find this an unnatural mechanism in which to relate their knowledge, and thus it has found limited pragmatic use. Artificial intelligence works have also attempted to use confirmation theory [15] and the theory of choices [16], but these have also not met with total success. This led to the adoption of the Dempster-Shafer theory of evidence, a model that has many of the advantages of the certainty factors approach but a stronger mathematical basis [17].

In the creation of our equity selection system, we encountered difficulty in the area certainty factor algebra. At first we decided that a simple certainty factor algebra would be best, as the domain expert indicated that this was the way he worked in making decisions. However, as the system grew in sophistication, it was necessary to adopt more complex algebras for different aspects of the system's reasoning, e.g., the Bonczek-Eagin method was used in one area and the probability sum method in another [18]. The use of these different certainty factor algebras meant that much effort was expended in tuning the system, and even though the expert found it natural to associate certainties with data, he found it difficult to define the certainty factor algebra necessary to reason with combined knowledge items. Consequently, modelling the expert's subjective reasoning became very difficult. The problem was compounded when we used a second expert to correlate some of the findings because the second expert often did not use exactly the same certainties or algebras and thus was not always sure that our system's deductive strategies were correct, even when the results indicated that it was performing accurately.

The use of statistical methods can be seen as positive in that they can assist the domain expert to express subjective or heuristic judgements and allow the user to know the degree of certainty a system has for a result. Alternatively, the use of approaches such as certainties can cause problems because they do not have a complete theoretical foundation and are open to interpretation. In reflection, we feel that we should have spent more resources in consolidating a certainty factor algebra from our experts before system creation and so attempted to minimize the subjective heuristic judgements made by the expert and promote a solution strategy based on a theoretical framework.

ALTERNATIVE STRATEGIES

The conventional approaches to validation and verification discussed above have shown us that although each of the techniques have certain strengths, each is severely limited in its ability to move toward a statement of total system correctness. Two alternative strategies that can be considered in relation to knowledge-based systems and that are currently areas of focused research are critical testing, in which research focuses on adaptive techniques for optimizing the data sets used in the case approach to testing, and reliability theory, in which developers can use and create models that predict the failure of their systems.

The following two sections will outline the theory behind these areas and discuss them in relation to the testing of our equity selection system.

Critical Testing

A program can be defined as correct when the implementation matches the specification:

$$P(D) = f(D)$$

where D = input data (the domain), P = program, and f = formal specification. To do this it is necessary to perform exhaustive testing:

$$P(d_0)..P(d_n)$$

Where n can be very large, if not infinite; therefore, in practice we can only test a limited number of cases. Once tested, however, we can then state that the program will perform correctly with respect to this input set:

$$P^*(D) = f(D)$$

where P^* = the program tested over the test data set.

The selection of the test data is a critical operation and, consequently, the criteria by which the critical test data for an application are selected has been the focus of much research. The area has been influenced by workers such as Gerhart and Goodenough, who considered the theoretical aspects of procedures to select reliable and valid test data for conventional programs [19, 20]. Their method is based in part on the construction of a "condition table," which displays the logically possible combinations of conditions within the program. However, the large number of conditions found in knowledge-based systems can prohibit use of this technique.

An alternative technique is the concept of "adequate" test data, which has been defined as "a test data set T is adequate if P behaves correctly on T but all incorrect programs behave incorrectly" [21]. However, it has been shown that from a theoretical standpoint, it is not possible to construct a general purpose test selection procedure for valid test data, as the function is not computable [22]. Thus, research is now focused on examining test data selection in relation to particular error types, as this would allow the construction of a reliable data set for a certain error type that could then be used on the system. This could be useful in testing aspects such as deductions around thresholds where certainty factors are involved.

Another research direction is that of mutation testing [23–25], and indications are that it could be usefully applied to knowledge-based systems in the identification of both epistemological and structural errors.

We will focus on these research themes when we develop the second version of our system, in that we shall use a focused case-based approach in conjunction with mutation testing. The result will be greater test coverage than before with a low cost/test result ratio. This is important as we will be working from sets of test data that we have already established as valid.

Reliability

A second alternative approach to the measurement of program correctness is to employ a reliability measure. According to Musa et al. [26], "software reliability is defined as the probability of failure-free operation of a computer program in a specified environment for a specified time." The mathematical treatment of software, hardware, and systems reliability has been developing over the last 20 years, and it has been estimated that there are > 40 models for software reliability alone. The problem therefore is in the selection of an appropriate model for evaluating the reliability of a knowledge-based system. Abdel-Ghaly et al. [27] have given an interesting evaluation of competing software reliability models. They state "that no single model can be trusted to perform in all contexts" and they advise software developers "to be eclectic: try many predictions systems and use the reliability metrics which are best for the data under construction."

One comparatively simple model that predicts failures as well as or better than any existing software reliability model is that proposed by Musa and coworkers [26, 28], and we are pursuing research into the applicability of this model for our system. The approach may be useful, for we have compiled data on the failures that have occurred in the system and this can be used to help derive a reliability figure.

CONCLUSION

This article has illustrated the techniques that are available to assist knowledge engineers in the validation and verification of their systems.

In creating our original system several errors occurred: (1) the development methodology we used did not facilitate validation or verification; (2) our prototype was too large and complex to be refined easily; and (3) the testing approach we used with random data was weak. We hope that by following the guidelines given here and using a rigorous development methodology these errors will not recur.

The methodology we encourage includes the use of formal techniques whenever possible, e.g., specify the requirements as far as possible, specify the knowledge base, and produce a denotational semantics and full syntax for the representation. The knowledge engineer should also be able to justify every step in the development and show how each step follows the previous one. It is advantageous to undertake this development through a well-defined prototyping approach for as many cycles as feasibly possible, each cycle being based on the data selected in the critical test data study. Finally, when prototyping becomes impractical, then selective critical testing should be used to limit the amount of testing that has to be performed while maximizing the return on that testing. While this development process is occurring, the knowledge engineer can compile data to produce reliability figures. This approach can be combined with some pragmatic techniques, such as ensuring that in critical situations the system is

fail safe or that multiple systems, developed independently, check each others' results. This approach should enable creation of a higher quality knowledge-based system.

REFERENCES

1. F. Hayes-Roth, D. Waterman, and D. Lenart, *Building Expert Systems*, Addison-Wesley, Reading, Massachusetts, 1983.
2. P. H. Winston and B. K. P. Horn, *LISP*, Addison-Wesley, Reading, Massachusetts, 1989.
3. D. A. Waterman, *A Guide to Expert Systems*, Addison-Wesley, Reading, Massachusetts, 1986.
4. B. W. Boehm, An Experiment in Small Scale Application Software Engineering, *IEEE Trans. Software Eng.* SE-7, 482–493 (1981).
5. G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, 1976.
6. S. L. Pfleeger, *Software Engineering: The Production of Quality Software*, Macmillan, New York, 1987.
7. C. B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.
8. J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
9. A. Teruel, Case Studies in Specification: Four Games, Technical Monograph PRG-30, Oxford University Computing Laboratory Programming Research Group, Oxford University, Oxford, U.K, 1982.
10. D. Gold and R. T. Plant, "Towards the formal specification of an expert system," Working Paper CIS/RTP/90/2 CIS Dept. Univ. of Miami, Coral Gables, Florida.
11. D. A. Turner, Functional programs as executable specifications, in *Mathematical Logic and Programming Languages* (C. A. R. Hoare and J. C. Shepardson, eds.), Prentice-Hall, Englewood Cliffs, New Jersey, 1985, pp. 29–54.
12. P. Henderson, G. A. Jones, and S. B. Jones, The LISPKIT Manual, Programming Research Group Monograph PRG-31, Oxford University, Oxford, U.K., 1983.
13. E. H. Shortliffe, *Computer-Based Medical Consultations: MYCIN*, Elsevier, New York, 1976.
14. L. A. Zadeh, Fuzzy Sets, *Info. Contr.* 8, pp. 338–353, Academic Press, NY (1965).
15. R. G. Swinburne, Choosing Between Confirmation Theories, *Philosophy Sci.* 37, 602–613 (1970).
16. A. Tversky, Elimination of Aspects, *Psychol. Rev.* 79, 281–299 (1972).
17. G. Shafer, *A Mathematical Theory of Evidence*, Princeton University Press, Princeton, New Jersey, 1976.
18. C. W. Holstapple, and A. B. Whinston, *Business Expert Systems*, Irwin Press, Homewood, Illinois, 1987.
19. J. B. Goodenough and S. L. Gerhart, Towards a Theory of Test Data Selection, *IEEE Trans. Software Eng.* SE-1, 156–173 (1975).
20. R. A. DeMillo, R. J. Lipton, and F. G. Sayward, Hints on Test Data Selection: Help For The Practicing Programmer, *Computer* 11, 34–41 (1978).
21. R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume, *Software Testing and Evaluation* Benjamin Cummings, Menlo Park, California, 1987.
22. L. Flon and A. N. Haberman, Towards the Construction of Verifiable Software Systems, *SIGPLAN Not.* 2, 141–148 (1978).
23. M. R. Woodward, M. A. Hennel, and D. Hedley, Experiences with Path Testing and Analysis and Testing of Programs, *IEEE Trans. Software Eng.* SE-6, 278–286 (1980).
24. W. E. Howden, Weak Mutation Testing and Completeness of Test Sets, *IEEE Trans. Software Eng.* SE-8, 371–379 (1982).
25. B. Littlewood, *Software Reliability: Achievement and Assessment*, Blackwell, Oxford, U.K., 1987.
26. J. D. Musa and K. Okumoto, A logarithmic poisson execution time model for software reliability measurement, in *Proceedings of the 7th International Conference on Software Engineering*, IEEE Computer Society Press, Orlando, Florida, 1984, pp. 230–238.
27. A. A. Abdel-Ghaly, P. Y. Chan, and B. Littlewood, Evaluation of Competing Software Reliability Predictions, *IEEE Trans. Software Eng.* SE-12, 950–967 (1986).
28. J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.