DESIGNING EXPERT SYSTEMS FOR EASE OF CHANGE

Judith N. Froscher
Robert J.K. Jacob

Naval Research Laboratory
Washington, D.C. 20375

Abstract. Current expert systems are typically difficult to change once they are built. The objective of this study is to develop a design methodology, which will make a knowledge-based system easier to change, particularly by people other than its original developer. The basic approach for solving this problem is to divide the information in a knowledge base and attempt to reduce the amount of information that each single programmer must understand before he can make a change to the expert system. We thus divide the domain knowledge in an expert system into *groups* and then attempt to limit carefully and specify formally the flow of information between these groups, in order to localize the effects of typical changes within the groups.

Introduction

If expert systems are to come into wide use in practical applications, the problem of continuing maintenance and modification of the knowledge base must be addressed in their development. Most current expert systems began as research tools, often developed in universities and maintained by their originators and their students. Now many observers believe that this technology shows promise of solving practical problems in industry and government, but knowledge-based systems continue to be *ad hoc*, one of a kind, and difficult to maintain. Changing a knowledge base typically requires a knowledge engineer who is well-grounded in the design of the system and the structure of the knowledge base. Most often, it requires the same knowledge engineer who originally built the system.

The objective of the present study is to develop a design methodology similar to those used in software engineering,[6,7] which will make a knowledge-based production system easier to change, particularly by people other than its original developer. We have chosen to concentrate on production systems because they are the most widely-used type of knowledge representation for expert systems, particularly among those existing systems large enough and mature enough to have experienced the types of maintenance problems we hope to alleviate. In the future, we will attempt to extend the approach to suit other, newer knowledge representations.

In this paper, we will describe our approach to developing such a methodology for production systems. The maintenance history of a long-lived, well-documented expert system and the efforts of other researchers to separate the information in a knowledge-based system will be reviewed. Then we will describe the approach we are taking to build maintainability into production systems. Next we will discuss our study to find structure in already existing expert systems. We will then present some algorithms that we have used for separating the information in a knowledge base and our results in using each of them. Finally, we will introduce a programming methodology for production system development and describe support software for it.

Background

While few knowledge-based systems are currently being used in commercial or military environments, one of the most successful exceptions is R1/XCON, developed by McDermott at Carnegie-Mellon University.[5] R1 was designed to configure the many components that make up a DEC VAX 11/780 computer and is implemented as a production system in OPS5. The development history of R1 illustrates the problems encountered when an expert system is used in a practical setting. Since the VAX computer line was constantly being changed and expanded, it was continually necessary to add more knowledge and greater capability to R1. Before long, the system became complex enough that it was necessary to reimplement it entirely. To support R1, DEC established a special software support group, which had to invest considerable effort in understanding R1 before they could make any changes to it.

As commercial promise for expert system technology grows, more researchers have become concerned with the practical problems of building these systems for real users, who have a very limited background in artificial intelligence. They have proposed several methods for partitioning these systems to make the knowledge bases more understandable, easier to maintain, or more efficient. One simple approach is to build a system made up of several knowledge bases; examples of this approach are seen in ACE[12] and PROSPECTOR.[3] The developers of LOOPS[1] use the notion of a rule set, which is called like a subroutine. Each rule set returns a single value, which can then be used elsewhere in the knowledge base. Clancey has abstracted the inference goals in a knowledge base in order to separate the control strategy, encoded as meta-rules, from the specific domain knowledge.[2] Rules that contribute to a particular goal in a knowledge base can then be grouped together; in fact, R1/XCON is partitioned into several subtasks in this fashion.[9] Because expert systems generally use large amounts of computer resources, researchers are studying how both the knowledge base and working memory can be separated so that each independent group can be processed on a parallel processor.[4] Such a separation can also make the system easier to change, although this was not the original purpose of the study.

Approach

The basic approach we have taken for building maintainability into an expert system is to divide the information in a knowledge base and attempt to reduce the amount of information that each single programmer must understand before he can make a change to the knowledge base. We thus divide the domain knowledge in an expert system into *groups* and then attempt to limit carefully and specify formally the flow of information between these groups, in order to localize the effects of typical changes within the groups.

Production systems comprise extensive domain knowledge, expressed as if-then rules, a data base of working memory elements, and a relatively simple inference mechanism or rule interpreter. To determine which rule to activate, the interpreter compares the conditions on the left-hand side of a rule with the current contents of the data base. If the test succeeds, modifications are generally made to specific data base elements according to actions indicated in the right-hand side of the rule; however, the action specified in the consequent part of a rule can also affect the rules themselves. No assumptions have been made about how the condition or action parts of a rule are represented or about how the data itself is represented. In the present approach, we divide the rules in the knowledge base into separate groups. The guiding principle for grouping two rules together is: ***If a change were made to one rule, to what extent would the other rule be affected?*** In this study, a *fact* is that part of the data representation that if changed in one rule would affect another rule in some way; in a simple production system where the condition elements, action elements, and data are represented as sequences of attribute-value pairs, a fact corresponds to an attribute. The knowledge engineer building the system would group together rules that use or produce values for the same sets of facts or data base elements. With this arrangement, a fact in the knowledge base can be characterized either as being generated and used by rules entirely within a single rule group or else as spanning two or more groups.

Whenever rules in a given group use facts generated by rules in other groups, such facts will be specially flagged, so that the knowledge engineer will know that their values may have been set outside this group. More importantly, those facts produced by the given group and used by rules in other groups must be flagged too. For each such fact, the programmer of the group that produces the fact makes an assertion, comprising a brief summary of the information represented by that fact. This assertion is the only information about the fact that should be relied upon by the programmers of other groups that use the fact. It is not a formal specification of the information represented by the fact, but rather an informal summary of what the fact should "mean" to outside users.

Given this structure, a programmer who wants to make changes to the system would assume the responsibility of understanding thoroughly and preserving the correct workings of a single group of rules (but not the entire body of rules, as with conventional systems). He would be free to make changes to the rules in the group provided only that he preserves the validity of the assertions associated with any facts that are produced by his group and used by other groups. Similarly, whenever he used a fact that was produced by another group, he would rely only on the assertion provided for it by the programmer of the other group and not on any specific information about the fact that could be obtained from examining the inner workings of the other group.

An interesting aspect of this approach is that it draws distinctions between the facts, or working memory element names, in a production system. Certain facts are flagged as being important to the overall software structure of the system, while others are "internal" to particular modules and thus less important. Programmers can be advised to pay special attention to rules that involve the "important" facts. This is in contrast to the homogeneous way in which the facts of a rule-based system are usually viewed, where they must all command equal attention or inattention from the programmer.

Connectivity of Production Systems

To decide whether partitioning a knowledge base is a feasible approach, we are analyzing several existing production systems to determine how the rules in the system are related to each other. We have developed a software tool that analyzes the connections between the rules of a production system. The input to the tool is a set of rules expressed in a neutral knowledge representation. Because OPS5 is one of the most widely used production system languages, we have also built software that translates from OPS5 into this representation. For expert systems written in other languages, we have performed the translation manually or semi-automatically with text processing programs.

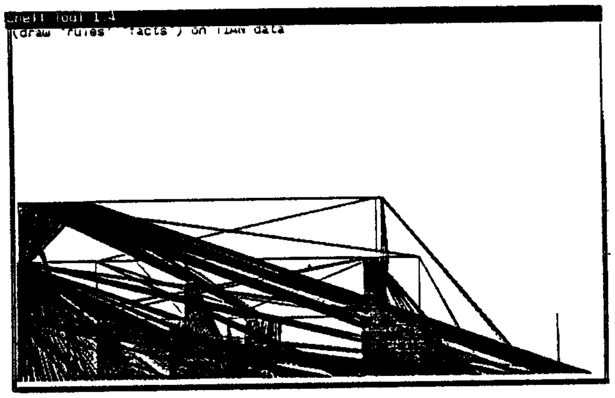
We are using the tool to determine whether the rules are indeed thoroughly intertwined or sufficiently separated that they could be divided into groups. To date, we have analyzed several knowledge bases and found that there is considerable separability and latent structure to the relationships between the rules in these systems, although this structure has not been exploited to improve maintainability.

Next, we are attempting to divide the rules of existing systems into appropriate groups automatically, using several new approaches. By grouping the rules of existing production systems, we hope to determine whether such systems *could* have been cast in the mold required by the new method or whether it would have imposed excessive restrictions and unnatural structure on the developer. Based on the latent structure in rules found thus far, initial results suggest that the present approach can be imposed on many ~~rule-based~~ systems. They also suggest that an ideal, but not always attainable, grouping of rules is one in which each group of rules sets the value of only one fact that is used by rules outside this group.

Algorithms for Partitioning a Knowledge Base

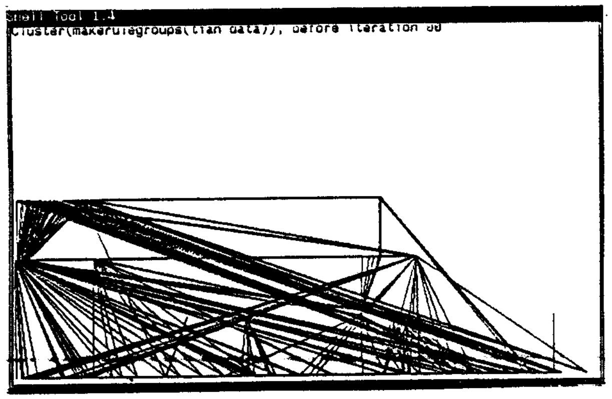
Rules are related to each other through the facts whose values they use or modify. First, we depicted these relationships in a graph, showing the inference hierarchy for the system. Figure 1 shows such a graph for an expert system developed by Reggia at the University of Maryland, using the KES language.[8] It contains 373 rules and 244 facts, and is used to diagnose stroke and related diseases. In Figure 1, each node, or point, represents a rule and each link, or line, between two rules represents a fact whose value is set by one rule and used by the other.

The first algorithm considered attempts to partition this graph into a collection of rooted trees of rules, where the "root" of each such tree is a fact. That is, divide the rules into groups such that each group of rules produces only one fact that is used by other groups. This provides the desideratum mentioned above, since each rule group produces only one external fact. Figure 2 shows the same system as Figure 1, after such an algorithm was applied. Each node, or point, now represents a group of rules, and each link, or line, represents a fact that is produced by rules in one group and used by those in another group. Facts that are produced and used entirely within a single group do not appear in the graph.



```
(draw 'rules' 'facts') on tian data
```

Figure 1. Plot of individual rules of an expert system.



```
Cluster(makerulegroups(tian data)), before iteration 00
```

Figure 2. Rules of expert system, grouped into trees.

These algorithms tend to divide the knowledge base into many small groups. Each such group contains a collection of rules whose effect on other groups is entirely summarized by the fact at the root of the tree. Hence rules in these groups intuitively belong together under any grouping scheme. The problem is that the many small groups now must be combined into larger agglomerations for ease of understanding. One alternative tested was to weaken the criterion for being a "rooted tree." That is, divide the rules into groups such that each group produces no more than **n** external facts, where **n** is now greater than 1. However, as **n** was increased, this approach did not appear to expose any natural structure in the knowledge bases.

Next, an approach based on cluster analysis was developed. Given a collection of objects, a clustering algorithm partitions them into groups of like objects. To use such an algorithm, though, a measure of distance or "relatedness" between rules must be defined. Since our ultimate concern is for a programmer making changes to the knowledge base, the similarity between two rules should measure: *If one rule were changed, how likely is the other rule to have to be changed also.* Rules affect each other through the facts they have in common. Thus a simple measure of the "relatedness" of two rules is simply the number of common facts that are referenced in the left or right hand sides of both rules. Since there are several ways in which two rules could reference the same fact, we decided to weight this count. The two rules **`if A then B`** and **`if B then C`** have **`B`** in common; so do the two rules **`if A then B`** and **`if C then B.`** The rules of the former pair seem to have a greater programming effect on each other than the latter pair, and hence should be more "related." Figure 3 summarizes the three ways in which two rules can reference the same fact, and the weights given to each. The total "relatedness" measure between two rules is, then, a weighted count of all the facts that both rules have in common, where each fact is weighted by the appropriate score, indicating in which of the three possible ways the two rules reference the same fact.

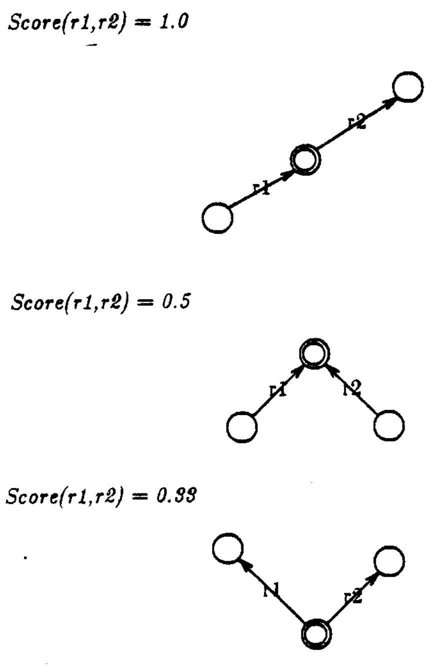


Figure 3. Components of "relatedness" measure between two rules.

Given such a measure of similarity between two rules, we can proceed with a straightforward clustering algorithm. First, measure the similarities between all pairs of rules, select the most related pair, and put those two rules together into one cluster. Then, repeat the procedure, grouping rules with each other or possibly with already-formed clusters. In the latter case, we must measure the "relatedness" between a rule and a cluster of rules. This is simply defined as the mean of the similarities between the individual rule and each of the rules in the cluster, corresponding to an average-linkage clustering procedure. The algorithm proceeds for as many iterations as desired.

The clustering algorithm can also be started with the small groups found by the rooted-tree algorithm above, instead of starting with individual rules. Since the tree groups appeared promising, but just too numerous, this is a reasonable alternative, and it appears to produce slightly better results. Figure 4 shows the rule groups of the system graphed in Figures 1 and 2 after clustering in this fashion. Again, each node, or point, represents a rule group produced by the clustering algorithm, and each link, or line, between two nodes represents a fact produced by one group and used by another group.

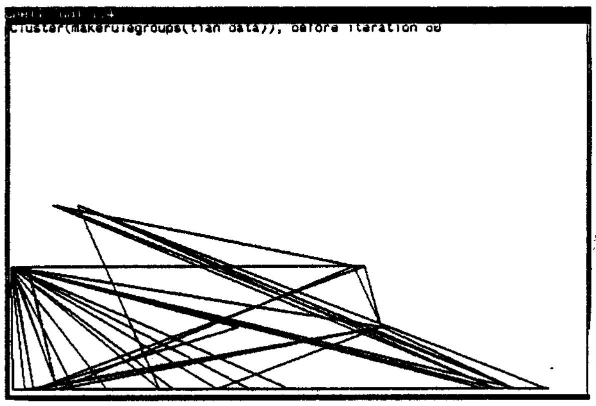


Figure 4. Rules, grouped into trees then clustered.

Thus far, this method appears to do the best job of partitioning a set of rules in an intuitively reasonable way. One drawback to the algorithm is that on each iteration it makes the best possible agglomeration of two groups, but it never backtracks, in case there might be a better grouping for the system considered as a whole. Also, like most clustering algorithms, if it runs for enough iterations it will eventually group all the rules into one large group. Manually examining the sizes and communication paths of the groups that have been produced at each iteration provides a subjective guide for terminating the clustering process.

Measures of Coupling and Cohesion

The division of a set of rules into groups should attempt to minimize the amount of coupling[10] between the groups and maximize the amount of cohesiveness within each group. Defining measures for these notions will provide data to help compare alternative groupings of a given set of rules. Once a set of rules is divided into groups, each fact in the system can be characterized as being used and produced by rules entirely within a single group or else as being used or produced by rules in more than one group. One simple measure of coupling is the proportion of facts in the second category, while cohesion is represented by the proportion of facts in the first.

Another approach to these measures is also being investigated. For coupling, it uses the average "relatedness" between all pairs of rules, where members of the pairs lie in different groups. For overall cohesion, it uses the average "relatedness" of every pair of rules that lie in the same group. For the example shown above, these quantities are 0.0798 average coupling and 0.9238 average cohesion, suggesting a far better than random organization.

Programming Methodology

After prototypes for an expert system have been built and a solid understanding of the domain knowledge and system requirements has been acquired, the knowledge engineer may have to design and develop a version of the expert system that will be used in a production environment. When the system is installed, its maintenance may be undertaken by a software support group that has not been a party to its development. It is thus desirable to provide a method so that the expert system developer can build a system that will be easy to maintain and modify.

Following our methodology, the developer would first divide the rules into groups. This can be done manually or automatically. One approach is to apply an automatic grouping algorithm to the initial prototype expert system and use the resulting grouping to guide the organization and development of the final production version. Then, a software tool will characterize each fact as inter-group or intra-group, and flag the former. They will prove critical to future changes to the knowledge base, since they are the "glue" that holds the groups together. The developer of a rule group that produces inter-group facts then provides an assertion describing each such fact. That description is the only information about the fact that should be used in the development of any other groups containing rules that use the value of the fact.

Thus, the set of rules will be divided into groups, the inter-group facts used and produced by each group will be identified, and descriptions will be entered for those produced by each group. Figure 5 shows the language used to provide this information, using an excerpt from a simple example knowledge base.[11] The figure illustrates the syntax for describing rule groups; a larger system would look exactly the same, except that it would list more rules, facts, and groups.

To modify a group, the programmer must understand ~~the internal operations of that group, but not of the rest of~~ the knowledge base. If he preserves the correct functioning of the rules within the group and does not change the validity of the assertions about its inter-group facts, the programmer can be confident that the change that he has made will not adversely affect the rest of the system. Conversely, if he wants to use additional inter-group facts from other groups, he should rely only on the assertions provided for them, not on the internal workings of the rules in the other group. Of course, changes that pervade several groups would still have to be handled as they always have been, but the grouping is intended to minimize these.

Changes in data representation methods would be confined to the abstract data type or flavors code for the object (which is outside the rules themselves).

```
(GROUP isamammal
     (PRODUCES
          (mammal "is it a mammal,
               by conventional English usage"))
     (RULES
          (r1 (IF hair) (THEN mammal))
          (r2 (IF milk) (THEN mammal))))

(GROUP isabird
     (PRODUCES
          (bird "is it a bird, by English usage"))
     (RULES
          (r3 (IF feather) (THEN bird))
          (r4 (IF flies ovip) (THEN bird))

(GROUP isacarn
     (PRODUCES
          (carn "is it a carnivorous creature"))
     (RULES
          (r5 (IF meat) (THEN carn))
          (r6 (IF pointed claws fwdeyes)
               (THEN carn))))

(GROUP isungulate
     (PRODUCES
          (ungulate "is it an ungulate"))
     (USES

          (mammal))
     (RULES
          (r7 (IF mammal hoofs) (THEN ungulate))))

(GROUP giraffe
     (USES (ungulate))
     (RULES
          (r10 (IF ungulate longn longl darksp)
               (THEN giraffe))))
etc....
```

Figure 5. Example of a grouped set of rules.

Support Software

We have developed software tools to support this programming methodology. The developer can define the grouping of rules and input the knowledge base in the form shown in Figure 5, or he can run one of the grouping algorithms discussed to produce the grouping. These groupings have no impact on system performance since they are invisible to the rule interpreter. Given such a grouping, the software then automatically identifies the intra-group and inter-group facts. It flags all inter-group facts produced by a group, so the programmer can provide assertions for them; it flags all inter-group facts used by a group and retrieves their descriptions, so the programmer can rely on them when using these facts. We have built other software tools that trace all effects of changing a given rule and find any unused rules or groups.

Evaluating the Methodology

To determine how well this programming methodology will work, we are attempting to retro-fit several existing knowledge bases with this approach. This will help determine how well the structure implied by the new programming methodology can fit the structures observed in actual rule bases. We will use the grouping algorithms to divide the rules and then use measures of coupling and cohesion to compare alternative groupings. Next we will attempt to measure the extent to which the new method helps or hinders maintenance of an expert system. We will attempt to make changes both to a conventional expert system and to one divided into groups following the proposed method and compare the results. Based on our preliminary results, the method does not impose unreasonable restrictions on the developer nor does it lead to unnatural structures. We will also investigate alternative partitioning algorithms and measures of coupling and cohesion for knowledge bases.

Conclusions

By studying the connectivity of rules and facts in several typical rule-based expert systems, we found that they seem to have a latent structure, which can be used to support a new programming methodology. Next, we have developed a methodology based on dividing the rules into groups and concentrating on those facts that carry information between rules in different groups. We have also studied several algorithms for grouping the rules automatically and for measuring coupling and cohesion of alternate rule groupings in a knowledge base. Finally, we have developed a simple language and some software tools to support the new method.

The resulting programming methodology requires the knowledge engineer who develops a rule-based system to declare groups of rules, flag all between-group facts, and provide descriptions of those facts to any rule groups that use them. The programmer who wants to modify such a system then gives special attention to the between-group facts and preserves or relies on their descriptions when making changes. In contrast to the homogeneous way in which the facts of a rule-based system are usually viewed, this method distinguishes certain facts as more important than others and directs the programmer's attention to them.

Finally, a future step in this research will be to apply these basic ideas to newer knowledge representations as large systems begin to be written using them. The basic approach will likely remain the same: divide the information in the knowledge base into groups, then specify and limit the flow of information between the groups. The specifics of the programming methodology will, of course, differ..

References

1. D.G. Bobrow and M. Stefik, "The LOOPS Manual," Tech. Rep. KB-VLSI-81-13, Knowledge Systems Area, Xerox Palo Alto Research Center (1981).

2. W.J. Clancey, "The Advantages of Abstract Control Knowledge in Expert System Design," *Proc. National Conference on Artificial Intelligence* pp. 74-78 (1983).

3. R.O. Duda, P.E. Hart, N.J. Nilsson, and G.L. Sutherland, "Semantic Network Representations in Rule-based Inference Systems," pp. 203-221 in *Pattern-directed Inference Systems*, ed. D.A. Waterman and F. Hayes-Roth, Academic Press, New York (1978).

4. A. Gupta and C.L. Forgy, "Measurements on Production Systems," CMU-CS-83-167, Department of Computer Science, Carnegie-Mellon University (1983).

5. J. McDermott, "R1: The Formative Years," *The AI Magazine* pp. 21-29 (1981).

6. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* **15** pp. 1053-1058 (1972).

7. D. L. Parnas, "Software Engineering Principles," *INFOR Canadian Journal of Operations Research and Information Processing* (November, 1984).

8. J. Reggia, "Knowledge-based Decision Support Systems: Development through KMS," *Department of Computer Science, University of Maryland* (1981).

9. M. Stefik, J. Aikins, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth, and E. Sacerdoti, "The Architecture of Expert Systems," pp. 89-126 in *Building Expert Systems*, ed. F. Hayes-Roth, E. A. Waterman, and D. B. Lenat, Addison-Wesley, Reading, Mass. (1983).

10. W.P. Stevens, G.J. Meyers, and L.L. Constantine, "Structured Design," *IBM Systems Journal* **13** pp. 115-139 (1974).

11. P.H. Winston and B.K.P. Horn, *LISP*, Addison-Wesley, Reading, Mass. (1980).

12. J.B. Wright, F.D. Miller, G.V.E. Otto, E.M. Siegfried, G.T. Vesonder, and J.E. Zielinski, "ACE: Going from Prototype to Product with an Expert System," *Proc. 1984 ACM Annual Conference on the 5th Generation Challenge* pp. 24-28 (1984).