1986 Proceedings

FALL JOINT COMPUTER CONFERENCE

November 2-6, 1986—INFOMART®—Dallas, Texas
Sponsored by **ACM and Computer Society of the IEEE**

Harold S. Stone
Stanley Winkler

Computer Society Order Number 743
Library of Congress Number 86-81582
IEEE Catalog Number 86CH2345-7
ACM Order Number 401860
ISBN 0-8186-0743-2

THE COMPUTER SOCIETY OF THE IEEE
Association for Computing Machinery
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE COMPUTER SOCIETY PRESS

Robert Jacob 11/86 NRL

1986 Proceedings
FALL JOINT COMPUTER CONFERENCE

November 2-6, 1986—INFOMART®—Dallas, Texas
Sponsored by **ACM and Computer Society of the IEEE**

Harold S. Stone, Proceedings Editor and Program Chairman
Stanley Winkler, Conference Chairman

Computer Society Order Number 743
Library of Congress Number 86-81582
IEEE Catalog Number 86CH2345-7
ACM Order Number 401860
ISBN 0-8186-0743-2

THE COMPUTER SOCIETY OF THE IEEE
Association for Computing Machinery
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC
IEEE COMPUTER SOCIETY PRESS

The papers appearing in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and are published as presented and without change, in the interests of timely dissemination. Their inclusion in this publication does not necessarily constitute endorsement by the editors, IEEE Computer Society Press, or the Institute of Electrical and Electronics Engineers, Inc.

Published by IEEE Computer Society Press
1730 Massachusetts Avenue, N.W.
Washington, D.C. 20036-1903

COVER DESIGNED BY JACK I. BALLESTERO

IEEE Computer Society Order Number 743
Library of Congress Number 86-81582
IEEE Catalog Number 86CH2345-7
ACM Order Number 401860
ISBN 0-8186-0743-2 (paper)
ISBN 0-8186-4743-4 (microfiche)
ISBN 0-8186-8743-6 (case)

Prices (1986) ACM or IEEE Members: $60.00 prepaid
All others: $120 prepaid

Additional copies of the 1986 Proceedings may be ordered prepaid from:

ACM
Order Department
Post Office Box 64145
Baltimore, MD 21264

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

Computer Society of the IEEE
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

Computer Society of the IEEE
Ave. de la Tanche
1160 Brussels, Belgium

Software Engineering for Rule-based Systems

Robert J.K. Jacob
Judith N. Froscher

Naval Research Laboratory
Washington, D.C. 20375

Abstract. Current expert systems are typically difficult to change once they are built. The objective of the present 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 knowledge engineer 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.

As the commercial promise for expert system technology grows, the problem of ongoing maintenance and modification of knowledge bases is becoming a significant concern. The designs of typical current knowledge-based systems are *ad hoc*, one of a kind, and difficult to maintain. The information in the knowledge base is interconnected in such a way that changing one part of the knowledge base may have unpredictable effects on other parts.

This research attempts to develop a design methodology similar to those used in software engineering,[1,2] 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 in 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, such as frames and semantic nets, as large systems begin to be written using them.

This paper describes the approach we are taking to build maintainability into production systems. It introduces a programming methodology for developing production systems. It discusses our study of structure and connectivity in already existing knowledge bases. It then presents algorithms we have devised for separating the information in a knowledge base and results obtained with them. Finally, it discusses tools for supporting the methodology.

Methodology

The basic approach we have taken for building maintainability into an expert system is to divide the information in the knowledge base and attempt to reduce the amount of information that each single knowledge engineer 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, and a relatively simple inference mechanism or rule interpreter. The interpreter tests the values of the facts on the left-hand side of a rule; if the test succeeds, new values for facts are set according to the right-hand side of the rule. In the present approach, we divide these rules 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* refers to that part of the data representation that, if changed by one rule, would affect another rule in some way; in a simple production system where the 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. 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. The latter will prove critical to future changes to the knowledge base, since they are the "glue" that holds the groups together.

Whenever rules in one 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 one 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 that 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 or she 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 might be obtainable from examining the inner workings of the other group.

Following our methodology, the developer would first divide the rules into groups. This can be done manually or automatically, as described below. One approach is to apply one of the automatic grouping algorithms 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. 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 1 shows the language used to provide this information, using an excerpt from a simple example knowledge base.[5] 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 maintenance 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 maintenance programmer can be confident that the change that has been 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.)

```
(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 1. Example of a grouped set of rules.

Partitioning the Knowledge Base

To decide whether partitioning a knowledge base is a feasible approach, we are analyzing 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 an abstract form.

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, which could be 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.

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 2 shows such a graph for an expert system developed by Reggia at the University of Maryland, using the KES language;[3] it is used to diagnose stroke and related diseases. In Figure 2, 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.

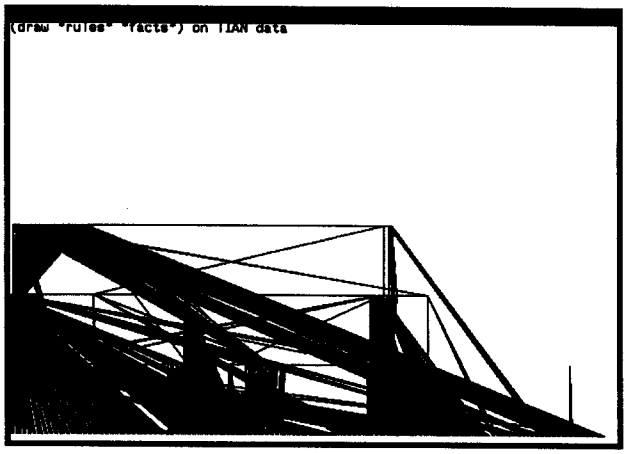


Figure 2. Plot of individual rules of an expert system.

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 sets the value of only one external fact. Figure 3 shows the same system as Figure 2, after such an algorithm was applied. Each node now represents a group of rules, and each link 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.

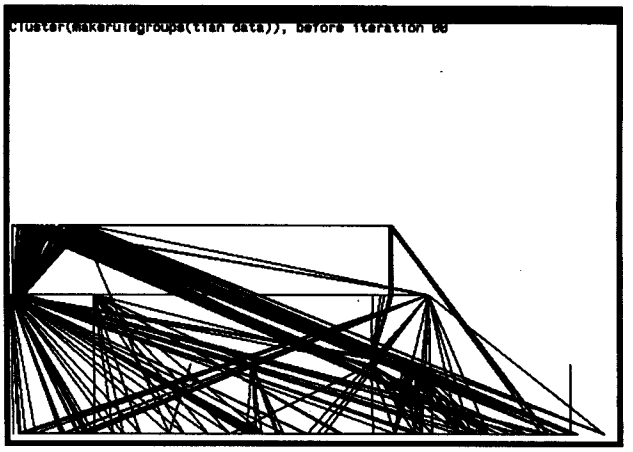


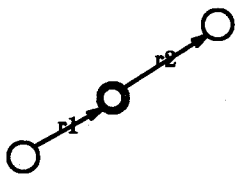
Figure 3. Rules of expert system, grouped into trees.

This algorithm tends 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. 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.

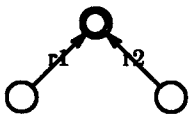
Next, an approach based on cluster analysis was developed. Given a collection of objects, a clustering algorithm partitions them into groups of like objects. To use such an algorithm, though, a measure of distance or "relatedness" between rules must be defined. Since our ultimate concern is for a programmer 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 the number of facts that are mentioned in the left or right hand sides of both rules. Since there are several ways in which two rules could refer to the same fact, we decided to weight this count. The two rules **if A then B** and **if B then C** share fact **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 4 summarizes the three ways in which two rules can

share a fact, and the weights given to each. The total "relatedness" measure between two rules is, then, a weighted count of the facts shared by both rules, where each fact is weighted by the score that indicates in which of the three possible ways the two rules use the fact.

Score(r1,r2) = 1.0



Score(r1,r2) = 0.5



Score(r1,r2) = 0.33

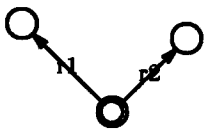


Figure 4. Components of "relatedness" measure between two rules.

Given such a measure, we can proceed with a straightforward clustering algorithm. First, measure the similarities between all pairs of rules, select the closest 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 iteratively.

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 5 shows the rule groups of the system of Figures 2 and 3 after clustering in this fashion.

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

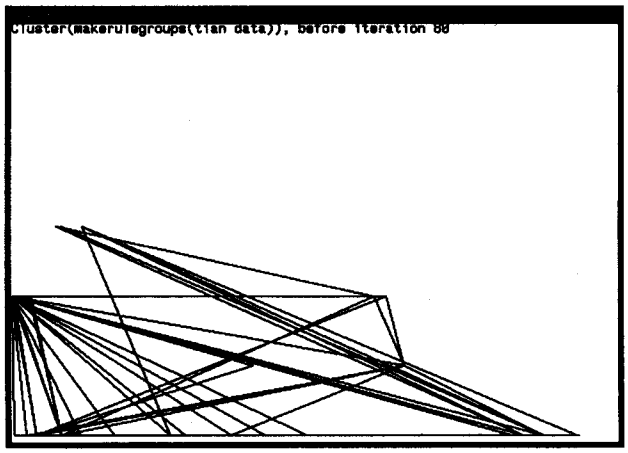


Figure 5. Rules, grouped into trees then clustered.

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; a stopping rule is thus needed.

Software Tools and Measurements

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 1, 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 such facts. Other software tools can trace all effects of changing a given rule and can find any unused rules or groups.

The division of a set of rules into groups should attempt to minimize the amount of coupling between the groups and maximize the amount of cohesiveness within each group.[4] 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.

Conclusions

By studying the connectivity of rules and facts in several typical rule-based expert systems, we found that they indeed have a latent structure, which can be used to support a new programming methodology. 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. Finally, we have developed a simple language and some software tools and measures 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 such facts. The knowledge engineer 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.

An interesting aspect of this approach is that it draws distinctions between the facts contained in working memory of 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.

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.

References

1. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* **15** pp. 1053-1058 (1972).
2. D. L. Parnas, "Software Engineering Principles," *INFOR Canadian Journal of Operations Research and Information Processing* (November, 1984).
3. J. Reggia, "Knowledge-based Decision Support Systems: Development through KMS," *Department of Computer Science, University of Maryland* (1981).
4. W.P. Stevens, G.J. Meyers, and L.L. Constantine, "Structured Design," *IBM Systems Journal* **13** pp. 115-139 (1974).
5. P.H. Winston and B.K.P. Horn, *LISP*, Addison-Wesley, Reading, Mass. (1980).