A Software Engineering Methodology for Rule-based Systems

Robert J.K. Jacob, MEMBER, IEEE
Judith N. Froscher

Abstract– Current expert systems are typically difficult to change once they are built. This paper introduces a method for developing more easily maintainable rule-based expert systems, which is based on dividing the rules into groups and focusing attention on those facts that carry information between rules in different groups. It describes a new algorithm for grouping the rules of a knowledge base automatically and a notation and set of software tools for the proposed method. The approach is supported by a study of the connectivity of rules and facts in rule-based systems, which found that they indeed have the latent structure necessary for the programming methodology, as well as by recent experimental results. In contrast to the homogeneous way in which the facts of a rule-based system are usually viewed, this approach distinguishes certain facts as more important than others with regard to future modifications of the rules.

Index Terms– Knowledge engineering, knowledge maintenance, information hiding, software engineering, knowledge base, rule-based system, production system, cluster analysis

I. INTRODUCTION

Many of the major expert systems in use today began life as research tools, typically developed in universities and maintained by their originators and their students. As long as this was the case, the problems of maintaining and modifying the knowledge base were tractable. However, as expert systems become larger and more complex and come to be used in environments away from their developers, maintenance of the knowledge base becomes an overwhelming problem. Changing a knowledge base of an expert system built with typical current technology requires a knowledge engineer who understands the design of the system and the structure of the knowledge base thoroughly; most often, this means only the original author of the system. If expert systems are to be widely used in practical settings, the problems of continuing maintenance and modification of the knowledge base must be addressed in their development.

The present research develops a design and programming methodology for builders and maintainers of knowledge-based systems. The result is a technique similar to those used in software engineering [16, 17], which can 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. The underlying approach can be extended to suit other, newer knowledge representations.

Other researchers have also begun to be concerned with the problem of designing expert systems for ease of maintenance. Some have proposed methods for partitioning these systems to make the knowledge bases more understandable, maintainable, efficient, or suitable for parallel processing. One simple approach is to build a system made up of several knowledge bases; examples of this approach are seen in ACE [25] and PROSPECTOR [6]. The developers of LOOPS [2] 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 [4]. 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 [20].

Because expert systems generally use large amounts of computer resources, several researchers have investigated how both the knowledge base and working memory can be separated so that each independent group can be processed on a parallel processor [10, 11, 22]. Such a separation can not only make a system easier to change but can also enable parallel firing of the rules.

II. APPROACH

The basic approach we take to make an expert system easier to change is to divide the information in the knowledge base and attempt to reduce both the amount of information that a single knowledge engineer must understand before he or she can make a change to the knowledge base and the effect of such changes. 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.

A basic production system comprises a knowledge base 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 an hierarchy of separate groups. The general principle for deciding whether to group 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 some isolable portion of the data representation that, if changed by one rule, would affect the firing or output of another rule in some way. In a simple production system where the data are attribute-value pairs, a fact corresponds to one attribute (not its value).

This simple view of a production system is not tied to any particular production system language, but represents a general conceptual model of rule-based systems. In it, the actions of rules affect a data base of facts; and the firings of rules are determined by the facts in the data base. The data base is the only medium for interactions or dependencies among the rules. Languages that introduce control constructs and other special features may restrict the data flow given by this view, but they will never expand it. Using this simple view, we trace all potential flows and do not look at restrictions caused by control or other special constructs; the

approach will thus be conservative at times, but never miss a dependency between rules.

The knowledge engineer building a rule-based system now groups together rules that use or produce values for the same sets of facts. Control knowledge is segregated from other rules and handled separately. Each fact in the knowledge base is then 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. Baroff and others [1] propose a complementary approach, in which they divide the facts into groups and then characterize the rules according to which groups of facts they operate on. The method described by Nau [14, 15] also leads to essentially the same result as the present approach.

Whenever rules in one group use facts generated by rules in other groups, such facts are 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 group under study 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 other programmers. The objective is to assemble an overall statement about a group of rules. That is, if the rules in this group are fired (any number of times), the given statement will be true. The collection of the assertions on the facts produced by a group embody such an overall statement about the group.

Specifically, the developer of a rule-based system would perform the following steps, which are discussed in greater detail in the next section:

1. Segregate Control Variables. Flag and then separate all facts or variables that are used for control purposes, that is, those used to enable or disable the firing of rules, as opposed to those that carry information between rules.

2. Divide the Rules into Groups. Partition the rules based on the flow of data (but not control information, which was just removed) between them.

3. Identify Local and Non-local Facts. A software tool characterizes each fact as either being produced and used entirely within one group (a local or *intra-group* fact) or being produced or used by two or more groups (a non-local or *inter-group* fact); the latter are flagged.

4. Write External Descriptions for Non-local Facts. The developer of each rule group that produces inter-group facts then provides an assertion or description of the externally visible properties of each such fact.

5. Separate Control Rules, When Possible. If the production system language permits, the control knowledge should be segregated from the other groups of rules.

Given this structure, programmers who want to change the system would now 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). They could change rules in the group provided that they preserve the validity of the assertions associated with facts produced by this group and used by other groups. Similarly, whenever a programmer used a fact produced by another group, he or she would rely on the assertion provided for it by the programmer of the other group, but not on any specific information about the fact that might be obtainable from examining the inner workings of the other group. Recent experimental results have demonstrated the efficacy of this separation (see Section VII) [5].

III. METHODOLOGY

This section describes the specific steps involved in applying the new method to the development of a rule-based system, gives examples, and discusses some of the details that arise. The description is given at two levels of detail; an overview may be obtained by skipping the parts labeled *Discussion*, which address the finer points of applying the method. They are set off from the main text and could appropriately be skipped on first reading. Also, as noted, recall that a *fact* here refers to a portion of the system's stored data, more nearly like a working memory element or programming language variable than a *fact* in its English usage.

The method may be applied from the start or may be introduced after an early prototype version of a system has been built, but before development of a final production version. Particularly for exploratory work in a new AI domain where requirements are ill-defined, applying the complete method during experimentation with preliminary prototypes could entail more costs than benefits. Since the statements added by this method have no effect on the actual execution of the rules, introducing them later in the development process has no adverse effect on the correctness or runtime behavior of the system.

1. Segregate Control Variables

The first major division of the knowledge base into modules is to separate control knowledge from domain knowledge, as in the approaches of Clancey [4], Fickas [7], and several others. If possible, it would be preferable to segregate control knowledge entirely, perhaps into a separate representation from the rules themselves. Only if the production system language or the design of the system being built does not facilitate such a separation, does this first step then become necessary.

In this step, variables used exclusively to enable or disable the firing of rules are flagged and segregated from those that carry information between rules. "State" or "goal" variables

generally fall into this control category. At this point, control rules are not being identified, only control variables. Segregating the control or state variables ensures that the groupings performed in the next step will be based only on the flow of "real" data (not control) among rules.

Discussion

One could use the opposite approach and concentrate entirely on the state variables as a key to identifying groupings of rules. However, such a method will only divide the rules temporally, that is, according to *when* they operate. That is not always the appropriate grouping from the point of view of modifying the program. For modification, we are concerned with how two rules affect each other over the entire execution of the system, not just whether they affect each other in sequence. While the two characteristics are often the same in well-structured systems, there is no guarantee that they will always coincide.

To aid in this separation of control variables, it is advisable to avoid letting a single variable serve two purposes–one for control and one as other data. For example, where the presence of an item in the data base is used for control, but the value of the item is used as data, it is helpful to use two different variables instead. In general, it is helpful to the subsequent steps of this method to make the system more nearly data-driven, so that firings of rules are determined by ordinary interactions of setting and testing data values, rather than by other programming constructs or shortcuts. The analyses we will want to perform on the system in subsequent steps assume that the system is driven by the data and will attempt to modularize the system based on the interactions of the data. Where the assumption is violated, the analysis will make unnecessarily conservative estimates of the connections between rules.

Example

As an example of Step 1, in one of the rules from the system listed in Section IV

```
(RULE universal
      (phase ^phasename scoring)
      (fmt ^part <y> ^score <z>)
      (entry ^index <y> ^det << all each every >> ^used nil)
      -->
```

```
(modify 2 ^score (compute <z> - 1))
(modify 3 ^used true))
```

`phase` is a control variable whose only purpose is to enable this and other rules. All `phase` variables were flagged, and thus they do not appear in any of the `PRODUCE` or `USE` statements in Section IV.

2. Divide the Rules into Groups

Next, the rules are divided into groups, based on the flow of data (but not control information, which was just removed) between them. In a large system, the groups may form an hierarchy. The division into groups can be done manually or automatically as described in Section V. A hybrid approach may also be used, by applying the automatic grouping algorithm to a prototype of the expert system and then using the resulting grouping to inform the manual organization and subsequent development of the production version.

Discussion

The rules should be divided so that each group contains all the rules relevant to one specific, small area of knowledge. The effect of the operation of the rules in the one group on rules outside that group is via the inter-group facts produced by the group. Ideally, a group should contain all the rules that together produce one or a small number of inter-group facts. The operation of that group is then conveniently summarized by the descriptions of those facts. Any further details about the operation of the group are considered local to the group and may be changed without affecting the rest of the system. The inter-group facts should represent easily-explained concepts around which meaningful modules or work assignments can be structured. Difficult-to-describe intermediate data should be encapsulated within groups as intra-group facts. As shown in the examples below, a group may contain other groups.

Section V defines the notion of relatedness between two rules more precisely and describes the development of algorithms to perform this grouping procedure automatically.

Example

A simplified example of such a group of rules is shown below. Together, the rules in this group determine whether a ship is ready to sail. The group encapsulates the details about what makes a ship ready. (In this simplistic example, its engine and radar must be "ready.") Details about what makes the engine "ready" are further encapsulated into the sub-group `engine-ready`, illustrating the use of hierarchical grouping of rules.

```
(GROUP ship-ready
       (RULE engine-and-radar
             (IF (engine-status = ready)
                 (radar-status = ready))
             (THEN (ship-status = ready)))

       (GROUP engine-ready
              (RULE gas-and-oil
                    (IF (engine-fuel-level >= 100)
                        (engine-oil-level >= 34))
                    (THEN (engine-status = ready)))
              (RULE ... etc.))

       (GROUP radar-ready
              (RULE ... etc.)))
```

3. Identify Local and Non-local Facts

Given a grouping of the rules, the next step is mechanical. A software tool characterizes each fact as being local or inter-group. If all the rules that use or modify the value of a fact are in one group, then that fact is local to the group; all other facts are inter-group facts. Inter-group facts whose values are set or modified by rules in a group are then listed in the `PRODUCE` statements for that group; inter-group facts whose values are examined by rules in a group are listed in the group's `USE` statements.

Discussion

The notion becomes more complicated where there is a hierarchy of groups. A fact produced or used by the rules in more than one top-level group is easily identified as an inter-group

fact. A fact produced and used by several sub-groups, all within one larger group, is considered inter-group *with respect to the sub-groups* but local *with respect to the larger group.* It is thus listed in the `PRODUCE` and `USE` statements of the sub-groups, but not of the larger group.

Example

The example group given above is now shown with the `PRODUCE` and `USE` statements for each of its groups and sub-groups:

```
(GROUP ship-ready
       (PRODUCE ship-status "ready means ship can be sailed on any mission
                             up to 1000 nautical miles")
       (USE engine-lubricated)

       (RULE engine-and-radar
             (IF (engine-status = ready)
                 (radar-status = ready))
             (THEN (ship-status = ready)))

       (GROUP engine-ready
              (PRODUCE engine-status "ready means engine fully operational")
              (USE engine-lubricated)

              (RULE gas-and-oil
                    (IF (engine-fuel-level >= 100)
                        (engine-lubricated = true))
                    (THEN (engine-status = ready)))

              (RULE ... etc.))

       (GROUP radar-ready
              (PRODUCE radar-status "ready means all radars fully
                                     operational")

              (RULE ... etc.)))
```

Now, the effect of group `ship-ready` on rules in other groups is entirely encapsulated in the one fact it `PRODUCE`s, namely `ship-status`; the remaining facts in this group are internal to the group. Facts `engine-status` and `radar-status` are part of the internal workings by which this group determines its conclusion about the ship status. They are examples of facts produced and used by sub-groups within a larger group, and are thus considered inter-group

with respect to the sub-groups `engine-ready` and `radar-ready` but local with respect to the larger group `ship-ready`. They are listed in the `PRODUCE` and `USE` statements of the sub-groups, but not of the larger group. In this example, fact `engine-lubricated` is assumed to be produced by some other group that analyzes the engine; while `engine-fuel-level` is an example of a primitive input to this rule-based system (i.e., not generated by any other rules) and hence does not appear in any of the `PRODUCE` or `USE` statements.

Several investigators have prescribed or demonstrated the division of rules into smaller groups to improve modularity [3]. Such divisions are often made around control or state variables, as described above, with the hope that the resulting groups will be sufficiently independent of one another to aid in maintenance. The present method takes this approach further in precision and explicitness. First, it verifies the actual independence of the groups from one another. Second, where two groups are connected, it identifies the specific data elements through which they interact. The result is not just a set of modules, but a set of modules with a known and enforceable degree of independence and an explicit declaration of all the data paths between them.

4. Write External Descriptions for Non-local Facts

The developer of each rule group that produces inter-group facts then provides an assertion or description of the externally-visible properties of each such fact. The assertion describes what the programmer of the fact-producing group asserts will remain true of that fact in the future– and hence upon which the programmers of the groups using that fact can rely. It summarizes the workings of the group that produces it. It is generally inappropriate for this assertion to provide a formal specification of the conditions for producing the fact, because that would essentially repeat the entire group of rules as they are presently. Rather, what is desired is a higher-level informal statement of the aspects of the output that will not change and may be considered externally visible. For example: *Fact X gives the system's best estimate of*

whether the patient has heart disease; rather than: *X will be true iff input A < 0.6 and B ≥ 2.1*. This assertion is the only information about the fact that should be used in the development of other groups containing rules that use the value of the fact; information about the internal workings of the rules in the group should not be used, because such details are not guaranteed to remain unchanged.

Discussion

This step is the crucial one and is the only part of the method that is inherently resistant to automation. It is discussed in some detail here, particularly as it applies in situations that are not completely straightforward, such as hierarchical rule groups.

Basically, this step should be viewed as making an overall assertion about each group or sub-group of rules that will be true regardless of how often the rules in that group fire. Specifically, that assertion would state that:

- the rules in this group can fire any number of times and they will have no effect on any inter-group variables not in the `PRODUCE` statements of this group; and
- if the variables in the `USE` statements of this group meet some stated conditions *c*, then the rules in this group will maintain some stated properties *p* of the variables in the `PRODUCE` statements.

The developer of a group fills in the conditions *c* and properties *p* for each of the facts produced by this group. For each such fact, he or she provides a statement or assertion indicating *p*, the effect of the rules in this group on the fact, qualified by a condition *c* if necessary. Such an assertion resembles a program slice [23] in that it summarizes the effect of several different rules on a single fact, even though the same rules might also affect other facts.

In the example above, simple assertions (without preconditions) were given for the effect of the rules on facts `ship-status`, `engine-status`, and `radar-status`. Rules in other groups may use the fact `ship-status` in accordance with its assertion. That is, they may use `ship-status` only to determine whether the ship "can be sailed on any mission up to 1000 nautical miles." They are not based on the knowledge of what actually constitutes being able to sail 1000 nm. The developer of this group is thereby free by change the

rules that define "can be sailed up to 1000 nm." (for example, to check the electrical generators before sailing). Such a change would be confined entirely to group `ship-ready`; it does not change the validity of the assertion given for fact `ship-ready`; and rules in other groups that relied on this assertion will not be changed.

In languages that permit rules to create and delete data base elements, the assertions may also involve the configuration of the data base. For example, an assertion might say that rules in a group expect exactly one element of some class to be in the data base and always leave exactly one. This enables one to follow an approach in which "legal" and "illegal" configurations of the data base elements are specified by the programmer [3]. Then, every group is simply permitted to assume a "legal" configuration of the inter-group facts it uses and required to preserve the configuration on the inter-group facts it produces. With the present method, however, such restrictions need only be placed on the configurations of the inter-group facts.

Another complication arises when facts are produced by more than one group. A fact could be produced by several top-level groups or by several sub-groups that all fall within one larger group. In either case, different assertions for such facts might be attached to the individual producer groups or a single global assertion could be made at a higher level. The distinction between these two alternatives is useful.

For the first case, attaching separate assertions to the individual groups or sub-groups that produce the same fact allows the designer to specify the separate effects each of the groups has upon that fact. The overall statement one could make about the fact is then the logical "or" of the assertions given by the individual groups. Attaching a single higher-level assertion allows the designer to specify an overall interpretation or mode of usage for that fact.

To accommodate this second case, the assertion is given in a `GLOBAL` statement either at the top level or else within the innermost group that contains all rules that produce or use the fact. Facts produced by a large number of groups are generally handled in this way, since they constitute the "global" data for the system. Their assertions are part of the overall system design, rather than the design of the individual groups. The system designer will identify these facts in the early design and provide more informative statements, which indicate how he or she

expects the global data to be used by the groups together.

Specifically, the assertions for the different kinds of inter-group facts that can arise from hierarchical groupings are handled as follows. (Examples of these cases taken from the larger set of rules shown in Section IV are referenced here.)

- The innermost group that is the sole producer of a fact gives its assertion in a `PRODUCE` statement in this group (e.g., `conn-score` in group `compute-conn-scores` shown in Section IV). Groups that enclose this group and are thus by extension also sole producers of the fact give the symbol `DOWN` for the assertion to point to the assertion made at the lower level of the hierarchy (e.g., `conn-score` in `compute-scores`).

- When a fact is produced by several groups and a single high-level assertion can be made to describe it, that assertion is attached to the innermost group which contains all groups that produce this fact. If the fact is used outside that innermost containing group, the assertion is given in a `PRODUCE` statement in that group (e.g., `fmt-score` in group `scoring`). Inner groups that produce this fact give the symbol `UP` as their assertion, indicating that the correct assertion for this fact is found at a higher level of the hierarchy (e.g., `fmt-score` in groups `compute-scores` and `compute-conn-scores`). If the fact is produced by several groups within a larger group, but not used outside the larger group, there would be no `PRODUCE` statement for the larger group. The assertion then is given in a `GLOBAL` statement in the scope of the larger group (e.g., `entry-headcat` in `compute-conn-scores`). Finally, if the fact is produced by several outermost level groups, the overall assertion is given in a `GLOBAL` statement at the outermost level (e.g., `entry-index`).

- When a fact is produced by several groups and separate, specific assertions are made by each group, then each innermost group that produces the fact and for which a unique assertion about the fact can be made gives the assertion in a `PRODUCE` statement. That assertion is preceded by the symbol `UP`, indicating that the other portions of the full assertion for this fact are found at a higher level. A

larger, enclosing group that is still not the sole producer of the fact also gives UP as its assertion. A still larger enclosing group that is the sole producer gives DOWN for its assertion, indicating that the correct assertion is the logical "or" of the assertions of all the lower-level component groups (does not occur in the system shown in Section IV).

To summarize, the innermost group that is sole producer of a fact gives an assertion for it. Enclosing groups that are therefore also sole producers of the fact give DOWN for the assertion. A group that is one of several producers of a fact may give (1) UP as the assertion or else (2) UP plus a more specific assertion. The innermost enclosing group that is the sole producer for the fact (i.e., group that contains all sub-groups that produce this fact) then either gives (for case 1) the general assertion or else (for case 2) DOWN, meaning the "or" of all the more specific UP assertions made within the group.

5. Separate Control Rules, When Possible

At this point, control knowledge (rules or statements that enable or disable the firing of other rules) should be separated from the rest of the knowledge base, if the production system language permits. As noted in step 1, in some languages, control knowledge is expressed in a notation other than rules, so that this separation is already made (e.g., KES, ORBS [7]). In an object-oriented language, the control knowledge about changes in state or goals might also be encapsulated in an object, and the left-hand sides of other rules could send messages to the control object to determine whether to fire.

Discussion

One approach in a simple production system language where state or goal variables have been used is to make a separate top-level group (perhaps containing hierarchical sub-groups) for control. All rules that set the values of control variables are then collected into one group, which describes all the transitions between states or goals. The rules in the control group are the only ones that can change the values of the control variables; antecedents of rules in other groups can query those variables.

Example

For example, in the system shown in Section IV, there is a single control variable `phase`. Most of the rules test the value of `phase` to determine whether they can fire, as in this example:

```
(RULE universal
      (phase ^phasename scoring)
      (fmt ^part <y> ^score <z>)
      (entry ^index <y> ^det << all each every >> ^used nil)
      -->
      (modify 2 ^score (compute <z> - 1))
      (modify 3 ^used true)).
```

Thus, all rules associated with `scoring` can be enabled or disabled together by setting `phase-phasename` to `scoring`. Rules in the group named `control` are the only ones that set the value of `phase` and thereby determine the transitions between the phases of the message analysis process.

6. Other Issues

Some other issues arise in trying to map various production system languages into the present method. The method is based on tracking the flow of data between rules and groups by means of the names of variables set or used by rules. To alleviate such problems, it is desirable to make the actual flow of data correspond as closely as possible to the use of the variable names. Techniques involving more sophisticated uses of variable names (such as serial re-use of temporary variables) can therefore "fool" this strategy (into being more conservative) and should be eschewed or handled specially.

Discussion

For example, using the same variable for two different purposes (but possibly at different times) will cause such confusion. Temporary variables used at different times should be given distinct names. Variables used in different states and which thus cannot exist at the same time

should have different names. In general, wherever the programmer can distinguish two unrelated uses of a variable or two uses that can never occur simultaneously, the two should be given different names.

A further wrinkle is added by production system languages that permit creation of new data base elements or facts during execution (such as OPS5 or ART), since they modify the straightforward conception of static facts examined and changed by rules. Such languages typically permit the left-hand side of a rule to ask whether there exists any fact with certain attributes, rather than simply testing the values of pre-declared facts or variables. A rule in OPS5, for example, can test and set values of one or more attributes of one or more "working memory elements" and can create or remove such elements. The "facts" of interest are thus dynamic– they cannot be statically enumerated in advance. Each working memory element is identified as belonging to some *class*, and rules generally identify the class of each element they examine or change.

For example, the following OPS5 rule

```
(p example
   (fmt ^part <x>)
   (entry ^index <x> ^headcat assembly)
   -->
   (modify 1 ^score 5)
   (remove 2)
   (make ship ^status ready))
```

examines the `part` attribute of a working memory element of class `fmt` and the `index` and `headcat` attributes of an element of class `entry`. It changes the `score` attribute of the `fmt` element, removes the `entry` element, and creates a new element of class `ship` and sets its `status` attribute. For the purposes of the software engineering method, each of the possible classes of elements in a system could be treated as a separate "fact," with a multi-faceted value consisting of all the attribute values of elements of that class. The above rule would then be considered to use facts `fmt` and `entry` and produce facts `fmt`, `entry`, and `ship`. In the notation we use for abstract description of rules, it would be written:

```
(RULE example
      (IF fmt entry)
      (THEN fmt entry ship))
```

The problem with this interpretation is that it is a rather coarse-grained abstraction of the working of the rules. For example, nearly every rule in the system shown in Section IV tests some attribute of `fmt`. A finer-grained abstraction of the information used and produced by each rule is needed. The basic goal of this abstraction is to identify which items in the right-hand sides of rules could affect the firing or not firing of which items in the left-hand sides. Observe that the right-hand side item

```
(modify 1 ^score 5)
```

in the above, which changes the `score` attribute of the `fmt` element matched by the left-hand side, would not affect any other rule with a left-hand side containing

```
(fmt ^part foo).
```

It would affect only those rules that test the `score` attribute of working memory elements of class `fmt`. Thus each attribute of each class is treated as a separate "fact," e.g., `fmt-score` and `fmt-part` are separate facts.

Further, since working memory elements are created and destroyed dynamically, there is not really a single static variable `fmt-score` in the system; it is considered to occur whenever any working memory element of class `fmt` uses an attribute named `score`. The fact `fmt-score` is an abstraction of all possible uses of the `score` attribute of *all* working memory elements of class `fmt`.

We observe, similarly, that the right-hand side

```
(make ship ^status ready)
```

can only affect rules whose left-hand sides test the `status` attribute of class `ship`, since no other attributes of `ship` were set. Hence it is considered to set "fact" `ship-status` only. However, the right-hand side

```
(remove 2)
```

in the example above, which deletes an entire working memory element of class `entry`, potentially affects any rule that examines any portion of an `entry` element. It is considered to set a new "fact" `entry`, which may be thought of as an abbreviation for *the effect of removing an element of class* `entry`. All left-hand sides that examine any part of an

`entry` working memory element are then considered to use the fact `entry` in addition to the facts for any individual attributes of `entry` they use.

The OPS5 rule above thus finally becomes, in our representation:

```
(RULE example
      (IF fmt fmt-part entry entry-index entry-headcat)
      (THEN fmt-score entry ship-status))
```

As noted, these "facts" are not actual memory elements but abstractions of the manipulations of working memory data by which two rules can affect each other. During execution of the system, various working memory elements may be created and removed dynamically. All possible effects of arbitrary sequences of such operations are summarized in the connections between the pseudo-facts `fmt`, `fmt-part`, `entry`, etc. that we use.

However, undisciplined use of "there-exists" predicates can make the rules unnecessarily difficult to analyze and group. It is helpful to restrict it further. Wherever the full power of such a predicate is not strictly necessary, it should be avoided. For example, if a system is designed so that there could never be more than one element of some class *c*, then the predicate *Does there exist any element of class* c can be treated simply as *Has the variable* c *been set*[3] . Class *c* becomes a single static variable *c*. Similarly, if it were known that there will be precisely three instances of class *c*, then class *c* could become three separate static variables. In general, then, whenever the creation of instances of a class can be predicted statically in advance, it is helpful to give a separate class name to each known instance to permit a more precise, finer-grained analysis of the interaction between rules.

Syntax

Following the steps listed, a set of rules is thus divided into groups, the inter-group facts used and produced by each group are identified, and descriptions are entered for those facts produced by each group. The example below illustrates the language used to represent this information, using a simple example knowledge base [24]. This is a forward chaining system with no control rules; steps 1, 5, and 6 above were thus inapplicable.

```
(GROUP isamammal
```

```
        (PRODUCE mammal "is it a mammal, by conventional English usage")

        (RULE r1 (IF hair) (THEN mammal))
        (RULE r2 (IF milk) (THEN mammal)))

    (GROUP isabird
        (PRODUCE bird "is it a bird, by English usage")

        (RULE r3 (IF feather) (THEN bird))
        (RULE r4 (IF flies ovip) (THEN bird))

        (RULE rx (IF a) (THEN c))
        (RULE ry (IF b) (THEN c))
        (RULE rz (IF c) (THEN bird)))

    (GROUP isacarn
        (PRODUCE carn "is it a carnivorous creature")

        (RULE r5 (IF meat) (THEN carn))
        (RULE r6 (IF pointed claws fwdeyes) (THEN carn)))

    (GROUP isungulate
        (PRODUCE ungulate "is it an ungulate")
        (USE mammal)

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

    (GROUP kind-of-carn
        (USE mammal)
        (USE carn)

        (RULE r8 (IF mammal carn tawny darksp) (THEN cheetah))
        (RULE r9 (IF mammal carn tawny blackst) (THEN tiger)))

    (GROUP kind-of-ungulate
        (USE ungulate)

        (RULE r10 (IF ungulate longn longl darksp) (THEN giraffe))
        (RULE r11 (IF ungulate blackst) (THEN zebra)))

    (GROUP kind-of-bird
        (USE bird)

        (RULE r12 (IF bird notfly longn longl blackwh) (THEN ostrich))
        (RULE r13 (IF bird notfly swims blackwh) (THEN penguin))
        (RULE r14 (IF bird flyswell) (THEN albatross)))
```

The rule bodies above are given in an abstract notation that lists their input and output facts, but no further details. This is the same representation used as input for the analyses described

in Section VII.

Note that three extra rules have been added for illustration to group `isabird` above. They indicate that the animal is considered a bird if `c` is true, which in turn depends on `a` and `b`. These rules appear to produce fact `c`, but `c` is not shown as being `PRODUCE`d by group `isabird` and does not have an assertion. This is because `c` is not used by rules in any other group– it is thus an intra-group fact, analogous to a local variable with no bearing on the connectivity of the modules of the system. Also, group `kind-of-carn` appears to produce facts `cheetah` and `tiger`, but they, too, are not listed. The reason is that those facts are not used by rules in any other groups and, again, have no effect on the connections between groups. In fact, they are top-level outputs of this expert system.

Also, observe, in the above example, that all information except for the rule bodies themselves– i.e., the groupings, `PRODUCE` and `USE` statements, and assertions– are essentially comments. While it might be possible to exploit them to improve efficiency or enable parallel firing of rules, they need have no effect on the actual execution (or correctness) of the expert system. Further, since the groupings have no effect on the execution of the rules, the method applies equally to any expert system shell or language, provided only that its concept of rules and facts can be mapped onto our rather abstract view. The method would be particularly convenient in a system where rules, facts, relationships, and assertions were all kept in a single knowledge or data base, but it does not require such support.

After a knowledge base is developed in this fashion, the knowledge engineer who wants to modify a group must understand the internal operations of that group, but not of the rest of the knowledge base. If he or she preserves the correct functioning of the rules within the group and does not change the validity of the assertions about its inter-group facts, he can be confident that the change that has been made will not adversely affect the rest of the system. Conversely, if he or she 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.)

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.

IV. AN EXAMPLE

A larger set of rules is shown in the example in this section. It is an excerpt from a rule-based system developed at NRL for summarizing natural language messages about failures of electronic equipment on ships [9]. Navy personnel currently perform this task manually, collecting the summary data for use in selecting reliable equipment to outfit new ships. To use this system, the narrative section of each incoming message is first parsed using a broad coverage string parser [19], and the data are stored in an information format table. The rules of the system shown here then examine and modify the entries in this table. The rows of the information format table represent clauses of the text, and the columns represent semantic categories specific to the messages, such as function, organization, part, process, repair, signal, and status. Each clause is also syntactically regularized so that it is in the active voice and in a standard order.

Each line of the information format table is converted into one OPS5 working memory element, where it will be accessible to the rules. Each line of the format table and each semantic category has an index, which is used to point to the actual entries in the table. Each entry

in the information format table thus becomes an OPS5 element of class `entry` with attributes containing its `index` and its actual contents, such as its text, its syntactic and semantic properties, and whether it has been scored yet. Each format line similarly becomes an OPS5 element of class `fmt`, whose attribute values contain the `index`es of the corresponding `entry`s.

The rule-based system then selects one clause from the information format which best describes the equipment failure and its cause. The rules score each line of the table based on characteristics such as how specific the description is, whether the information shows causality, and whether the line describes a component or a whole system. The format line with the highest score is chosen for the summary.

The system is written in the OPS5 language [3, 8]. Using the new method, this set of rules would appear as shown below, with the rules divided into groups, the inter-group facts declared, and the control knowledge segregated. To save space, most of the actual rules are omitted from the listing, but the `GROUP` declarations and `PRODUCE` and `USE` statements are shown for all groups. Within the group `compute-conn-scores`, the system is shown in full, including all rules, just as it would be entered in practice. The overall hierarchy of the groups and sub-groups is also depicted in Figure 1.

The control variable throughout this system is `phase-phasename`; all rules can test its value, while rules in the group `control` change its value. "Facts" in OPS5 are not always simple static variables, and they have been treated as discussed in the previous section. However, since there are no `remove` statements in the right-hand sides of these rules, the additional facts such as `fmt` and `entry` discussed previously are not required. The example illustrates the programming method, but it should be remembered that it is used for exposition; the proposed method may seem excessive for a system of this size and complexity.

```
(GLOBAL entry-index
    "Contains symbol used by attributes of fmt to point to this entry")
(GLOBAL entry-used
```

```
   "Contains nil if this text entry has not been scored by a rule, else true")
(GLOBAL fmt-status
   "Gives entry-index of text describing equipment status")

(GROUP scoring
   (PRODUCE entry-index UP)
   (PRODUCE entry-used UP)
   (PRODUCE fmt-score
      "Set to numerical weight assigned to this line of information format")
   (PRODUCE fmt-status UP)
   (PRODUCE vars-bigscore "Set to largest value of fmt-score in the table")

   (USE entry-index)
   (USE entry-used)
   (USE fmt-func)
   (USE fmt-part)
   (USE fmt-process)
   (USE fmt-score)
   (USE fmt-signal)
   (USE fmt-status)
   (USE vars-bigscore)
   ...rules omitted here...

   (GROUP compute-scores
      (PRODUCE conn-score DOWN)
      (PRODUCE entry-index UP)
      (PRODUCE entry-used UP)
      (PRODUCE fmt-score UP)
      (PRODUCE fmt-status UP)

      (USE conn-score)
      (USE entry-index)
      (USE entry-used)
      (USE fmt-func)
      (USE fmt-part)
      (USE fmt-process)
      (USE fmt-score)
      (USE fmt-signal)
      (USE fmt-status)
      ...rules omitted here...

      (GROUP compute-conn-scores
         (PRODUCE conn-score
            "Set to numerical weight of format line containing a connective word")
         (PRODUCE entry-index UP)
         (PRODUCE entry-used UP)
         (PRODUCE fmt-score UP)
         (PRODUCE fmt-status UP)

         (USE conn-score)
```

```
(USE entry-index)
(USE entry-used)
(USE fmt-score)
(USE fmt-status)

(GLOBAL entry-headcat
   "Word describing semantic category of head of phrase of this entry")

(RULE problem-is
   (phase ^phasename scoring)
   (conn ^op <x> ^arg1 <y> ^arg2 <z>)
   (entry ^index <x> ^head be)
   (fmt ^index <y> ^status <u>)
   (entry ^index <u> ^headcat problem ^used nil)
   (<< conn fmt >> ^index <z> ^score <v>)
   -->
   (modify 6 ^score (compute <v> + 1))
   (modify 5 ^used true))

(RULE find
   (phase ^phasename scoring)
   (conn ^op <x> ^arg2 <y>)
   (entry ^index <x> ^head << find determine >> ^used nil)
   (<< conn fmt >> ^index <y> ^score <z>)
   -->
   (modify 4 ^score (compute <z> + 1))
   (modify 3 ^used true))

(RULE invest-conn
   (phase ^phasename scoring)
   (conn ^op <x> ^arg1 <y> ^arg2 <z>)
   (entry ^index <x> ^headcat show ^used nil)
   (fmt ^index <y> ^invest <> nil)
   (<< conn fmt >> ^index <z> ^score <w>)
   -->
   (modify 5 ^score (compute <w> + 1))
   (modify 3 ^used true))

(RULE bequeath
   (phase ^phasename scoring)
   (conn ^score { <> nil <> 0 <x> } ^arg1 <y> ^arg2 <z>)
   (<< conn fmt >> ^index <y> ^score <u>)
   (<< conn fmt >> ^index <z> ^score <v>)
   -->
   (modify 3 ^score (compute <u> + <x>))
   (modify 4 ^score (compute <v> + <x>))
   (modify 2 ^score 0))

(RULE modal-obj
   (phase ^phasename scoring)
```

```
    (conn ^op <x> ^arg2 <y>)
    (entry ^index <x> ^head << suspect appear believe >> ^used nil)
    (fmt ^index <y> ^score <z>)
    -->
    (modify 4 ^score (compute <z> + 1))
    (modify 3 ^used true))

(RULE modal-move
    (phase ^phasename scoring)
    (conn ^op <x> ^arg2 <y>)
    (entry ^index <x> ^head { << suspect appear believe >> <z> }
          ^used nil)
    (conn ^index <y> ^op <u>)
    (entry ^index <u>)
    -->
    (modify 5 ^modal <z>)
    (modify 3 ^used true))

(RULE cause
    (phase ^phasename scoring)
    (conn ^op <x> ^arg1 <y>)
    (entry ^index <x> ^headcat cause ^modal nil ^used nil)
    (fmt ^index <y> ^score <z>)
    -->
    (modify 4 ^score (compute <z> + 2))
    (modify 3 ^used true))

(RULE cause-may
    (phase ^phasename scoring)
    (conn ^op <x> ^arg1 <y>)
    (entry ^index <x> ^headcat cause ^modal <> nil ^used nil)
    (fmt ^index <y> ^score <z>)
    -->
    (modify 4 ^score (compute <z> + 1))
    (modify 3 ^used true))

(GROUP determine-headcat
    (PRODUCE entry-headcat UP)

    (USE entry-head)
    (USE entry-headcat)

    (RULE catgz-bad
        (phase ^phasename initial)
        (entry ^headcat nil ^head << bad break broken burn crack
              damage error fail failure fault inop inoperative lack
              lose loss malfunction poor spot wear wrong >>)
        -->
        (modify 2 ^headcat bad))
```

```
    (RULE catgz-cause
       (phase ^phasename initial)
       (entry ^headcat nil ^head << cause make produce render result >>)
       -->
       (modify 2 ^headcat cause))

    (RULE catgz-code
       (phase ^phasename initial)
       (entry ^headcat nil ^code << KWR-37 KY-8 >>)
       -->
       (modify 2 ^headcat assembly))

    (RULE catgz-component
       (phase ^phasename initial)
       (entry ^headcat nil ^head << amplifier antenna apc assembly
             circuit driver exciter pa ppc rf >>)
       -->
       (modify 2 ^headcat assembly))

    (RULE catgz-impair
       (phase ^phasename initial)
       (entry ^headcat nil ^head << impair inhibit prevent stop >>)
       -->
       (modify 2 ^headcat impair))

    (RULE catgz-problem
       (phase ^phasename initial)
       (entry ^headcat nil ^head << problem damage failure fault
             malfunction difficulty defect >>)
       -->
       (modify 2 ^headcat problem))

    (RULE catgz-show
       (phase ^phasename initial)
       (entry ^headcat nil ^head << show indicate reveal >>)
       -->
       (modify 2 ^headcat show)))

(GROUP score-preparation
    (PRODUCE conn-arg1
       "Gives fmt-index or conn-index of fmt line preceding connective")
    (PRODUCE conn-arg2
       "Gives fmt-index or conn-index of fmt line following connective")
    (PRODUCE entry-head "Contains first word of phrase of this entry")
    (PRODUCE entry-headcat UP)
    (PRODUCE entry-index UP)
    (PRODUCE entry-neg
       "Value is non-nil if head of phrase for this entry is negated")
    (PRODUCE fmt-status UP)
```

```
(USE conn-arg1)
(USE conn-arg2)
(USE entry-head)
(USE entry-headcat)
(USE entry-index)
(USE entry-neg)
(USE entry-used)
(USE fmt-status)

(RULE cause-bad
    (phase ^phasename initial)
    (conn ^op <x> ^arg2 <y>)
    (entry ^index <x> ^headcat impair)
    (fmt ^index <y> ^status nil)
    -->
    (bind <z>)
    (modify 4 ^status <z>)
    (make entry <z> ^head bad ^headcat bad ^text bad)
    (modify 3 ^headcat cause)
    (call new-entry <y> status bad))

(RULE smoke-fire
    (phase ^phasename initial)
    (conn ^op <x> ^arg1 <y> ^arg2 <z>)
    (entry ^index <x> ^headcat cause)
    (fmt ^index <y> ^status nil)
    (fmt ^index <z> ^status <w>)
    (entry ^index <w> ^headcat bad)
    -->
    (bind <u>)
    (modify 4 ^status <u>)
    (make entry <u> ^head bad ^headcat bad ^text bad)
    (call new-entry <y> status bad))

(RULE due-to
    (phase ^phasename scoring)
    (conn ^op <x> ^arg1 <y> ^arg2 <z>)
    (entry ^index <x> ^head << due_to result >> ^used nil)
    -->
    (modify 3 ^headcat cause)
    (modify 2 ^arg1 <z> arg2 <y>))

(RULE neg-to-bad-status
    (phase ^phasename scoring)
    (fmt ^index <y> ^status <x>)
    (entry ^index <x> ^neg <> nil ^headcat <> bad)
    -->
    (modify 3 ^neg nil ^head bad ^headcat bad ^text bad)
    (call new-entry <y> status bad))
```

```
        (RULE zero-to-bad
            (phase ^phasename scoring)
            (fmt ^index <y> ^status <x>)
            (entry ^index <x> ^quant << zero 0 >> ^headcat <> bad)
            -->
            (modify 3 ^quant nil ^head bad ^headcat bad ^text bad)
            (call new-entry <y> status bad))

        (RULE quant-to-bad
            (phase ^phasename scoring)
            (fmt ^index <y> ^status <x>)
            (entry ^index <x> ^quant-mod in_excess_of)
            -->
            (modify 3 ^quant-mod nil ^head bad ^headcat bad ^text bad)
            (call new-entry <y> status bad))))))

(GROUP output-preparation
    (PRODUCE entry-index UP)
    (PRODUCE entry-used UP)
    (PRODUCE fmt-func "Gives entry-index of text describing function of equipment")
    (PRODUCE fmt-part
        "entry-index of text describing component level of equipment in system")
    (PRODUCE fmt-process
        "entry-index of text describing how equipment processes electrical signal")
    (PRODUCE fmt-signal
        "entry-index of text describing electrical signal from equipment")
    (PRODUCE fmt-status UP)

    (USE fmt-func)
    (USE fmt-part)
    (USE fmt-process)
    (USE fmt-score)
    (USE fmt-signal)
    (USE fmt-status)
    (USE vars-bigscore)
    ...rules omitted here...)

(GROUP final-output
    ;; N.B. this group displays final output of system to user,
    ;; hence does not PRODUCE any data USED by other groups

    (USE entry-index)
    (USE fmt-func)
    (USE fmt-part)
    (USE fmt-process)
    (USE fmt-score)
    (USE fmt-signal)
    (USE fmt-status)
    (USE vars-bigscore)
    ...rules omitted here...)
```

```
(GROUP control
   ;; Produces only control variable phase-phasename, which is never listed
   ...remainder of group omitted...)
```

V. AN AUTOMATED PROCEDURE FOR PARTITIONING A KNOWLEDGE BASE

It is possible to develop an algorithm that will take a set of rules and divide them into groups suitable for use with this method. Such an algorithm can help in the development of a rule-based system (in step 2 above), and it will also be useful in evaluating the wider applicability of the proposed method by attempting to group the rules of existing systems, as described in Section VII. It is somewhat easier to develop a grouping algorithm for rule-based systems than for code written in conventional programming languages because both the syntax and semantics of production languages are simpler and more regular.

The basic problem is to partition a set of rules into groups that will aid in their maintenance. Rules that affect each other and are likely to be changed at the same time should be grouped together. Several approaches to the grouping problem have been explored [12, 13], and the best results were obtained from an algorithm based on cluster analysis. A clustering algorithm takes a collection of objects and partitions them into groups of like objects.

In order to use a clustering algorithm, a measure of distance or "relatedness" between the objects to be clustered (the rules) must be defined. Since our ultimate concern is for a programmer making changes to the knowledge base, this "relatedness" between two rules should approximate the likelihood that a change made to one rule would require a change in the other rule. The rules in a production system are related through the facts whose values they use or modify. The strict definition of *rule dependency* is thus that two rules are related if there is any non-control fact mentioned by both rules (on their left- or right-hand sides); they are independent if they have no non-control facts in common. For the clustering algorithm we need a measure of the *extent* of the relationship between two rules, rather than a binary-valued

definition. The *relatedness* between two rules is thus measured by the number of non-control facts that are mentioned in both rules. Since there are several ways in which two rules could refer to the same fact, a weighting factor is applied. The two rules

```
if A then B
if B then C
```

share fact `B` in common; so do the two rules

```
if A then B
if C then B .
```

The rules of the former pair have a greater programming dependency on each other than the latter pair, and hence should be more "related." The top three illustrations in Figure 2 summarize the three ways in which two rules can share a fact and the weight given to each. The values of the weights shown in Figure 2 were observed to yield the best groupings, but experiments with other values suggested that the clusterings are relatively insensitive to any plausible choice of weights. So the precise values of these weights are somewhat arbitrary and non-critical. The total "relatedness" measure between two rules is a weighted count of the facts shared by both rules. Each fact is weighted by the score that indicates in which of the three possible ways the two rules use the fact.

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 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. As currently implemented, it does not generate hierarchical clusters, although this is envisioned. The view of rule dependency incorporated

into this "relatedness" measure is quite simple and is essentially limited to a single cycle of production system firing. Nevertheless, as seen in Section VII, it can produce a useful partitioning of a rule base, specifically a partitioning that satisfies our initial goal that those rules likely to be modified at the same time should be grouped together.

The algorithm presently used contains some further refinements. While they have a relatively small effect on its overall operation, they prove significant at iterations where the best and next-best possible pairings are close in average similarity but differ substantially on other criteria. First, an additional case was added to cover facts *not* shared by two rules. Without it, the pair of rules

```
if A then B
if B then C
```

and the pair

```
if A then B
if B then C and D and E and F
```

would have the same similarity; but, in determining a grouping, the former pairing is preferable to the latter. Hence a negative weight, as shown in the last illustration of Figure 2, is added to the similarity between two rules for facts that appear in one of the rules but not the other. In addition, the square root of each between-rule similarity is taken before combining them into the between-group similarity measure. These two changes reduce the tendency of the algorithm to use a few pivotal rules to form a long, "spindly" group that gradually annexes its neighbors in all directions.

The other refinement is an additional term added to the similarity between two groups, which slightly favors combinations of smaller groups over larger ones. This takes into account the practical consideration that the groupings are ultimately going to be used as the basis for assignment of programming responsibilities to individuals, hence very small or very large

groups would likely be manually overridden in the end. The quantity *2/(size of the combined group)* is added to the similarity between two groups for this purpose.

A drawback to algorithms of this general type is that on each iteration the algorithm 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 that has been used successfully is to stop when the similarity between the next two groups to be combined is no longer positive (since there is a term with a negative weight in the similarity between rules). Another approach is to use the distribution of the sizes of the groups as a guide.

This clustering algorithm implies a bottom-up approach, which requires no high-level information from the knowledge engineer. It derives a high-level structure for the rule system based only on the information contained in the rules themselves. Finally, recall that the use of this or any algorithm to divide the rules into groups is an optional aspect of the new method. The partitioning may equally well be done manually or by some other algorithm.

The algorithm and relatedness measure are presented more formally below:

Algorithm

1. Put each rule into a separate group.
2. Measure relatedness (defined below) between each pair of groups.
3. Choose the pair of groups, g1 and g2, with the largest relatedness, R(g1,g2).
4. If $R(g1,g2) \leq 0$, stop (or other stopping rule, see text).
5. Combine g1 and g2 into a single group.
6. Go to step 2.

Relatedness between two groups, g1 and g2

R(g1,g2) ≡

mean over all pairs of rules r1 and r2, where r1∈g1 and r2∈g2 (

sqrt (

no. of "in-out" shared facts in r1 and r2 * W(in-out) [see Fig. 2]

+ no. of share-outs facts in r1 and r2 * W(share-outs)

+ no. of share-ins facts in r1 and r2 * W(share-ins)

+ no. of not-shared facts in r1 and r2 * W(not-shared)

))^2

+ 2 / (no. of rules in g1 + no. of rules in g2)

VI. SUPPORT SOFTWARE

Some of the steps required by the new method are amenable to automation, and software tools have been developed for them. Step 2, the division of rules into groups can be performed by the clustering algorithm described in Section V. Given such a grouping, whether generated manually or automatically, a software tool then performs Step 3, identifying 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; and it flags all inter-group facts used by a group and retrieves their assertions, so the programmer can rely on them when using such facts. Step 4, writing the external descriptions of the inter-group facts produced by each group, represents the capture and documentation of additional high-level "plan" or design information in the system designer’s head, describing his expectations about what aspects of a fact will not change in the future. This information cannot be deduced from the rules themselves. As noted, these assertions are most useful when, like high-level comments, they describe the designer’s intentions rather contain a formal specification of the conditions for producing a fact. This step is therefore inherently manual and is critical to the success of the method. The tools find and

highlight the inter-group facts but then require the programmer to give them special attention and provide special declarations for them.

Additional software tools have been developed to support the method and to analyze the connections between the rules of a production system. The input to all the software tools is a set of rules expressed in the abstract form shown in Section III. The developer of a rule-based system can define the grouping of rules and input the knowledge base in the form shown in Section III or he or she can enter the un-grouped rules in the same form and use the clustering algorithm to produce the grouping. We have built software that translates rules from OPS5 into the abstract representation, including the separation of OPS5 working memory elements into their component "facts" (as discussed in Step 6). For expert systems written in other languages, we have performed the translation manually or semi-automatically with text processing programs.

One of the software tools used for analyzing rule connectivity traces all effects of changing a given rule and can find any unused rules or groups. Statistics about the distribution of inter- and intra-group facts in the system are also produced; an example is shown in Section VII. A graphical plot of the relationships between rules or groups may be obtained. Figure 3 shows such a plot for the animal system given in Section III. In it, each node, or circle, 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. Plots of a larger system are described in Section VII.

Finally, we have developed tools to compute measures of coupling and cohesion for a set of rules. As with other software engineering methods, the division of the rules into groups should attempt to minimize the amount of coupling between the groups and maximize the amount of cohesiveness within each group [21]. By defining numerical measures for the informal notions of coupling and cohesion in this domain, alternative groupings of a set of rules may be compared. One simple measure of overall coupling is the proportion of inter-group

facts, while cohesion is represented by the proportion of intra-group facts. (top- and bottom-level facts are excluded from these counts). Another approach uses, for coupling, 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.

VII. INVESTIGATING THE FEASIBILITY OF THE NEW METHOD

To decide whether partitioning a knowledge base is a feasible approach, we analyze existing expert systems to determine how the rules in the system are related to each other. We use the software tools described in Section VI to determine whether the rules are indeed thoroughly intertwined or sufficiently separated that they could be divided cleanly into the groups required by the method. The approach is to use the clustering algorithm to divide the rules of an existing system into groups. By examining the resulting groupings, we hope to determine how well the structure implied by the new programming method can fit the structures observed in actual rule bases, that is, whether the existing systems *could* have been cast in the mold required by the method or whether it would have imposed excessive restrictions and unnatural structure on the developers. 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 permits the present approach to be imposed.

For example, Figure 4 shows the graphical plot for a larger expert system. It was developed by James Reggia of the University of Maryland, using the KES language and is used to diagnose stroke and related diseases [18]. It contains 373 rules and 116 distinct variables. Unlike OPS5, the KES language uses static variables, in which all instantiations are declared in advance. However, 21 of the variables in this system can be assigned more than one value simultaneously. The presence of each possible value for each multi-valued variable becomes a separate fact in our framework, so that the system has 244 facts. In Figure 4, like

Figure 3, each node represents a rule and each link between two rules represents a fact whose value is set by one rule and used by the other. To reduce clutter, labels like those in Figure 3 have been suppressed, and rules, which were represented by circles in Figure 3, are now shown as points.

Figure 5 shows the same system as Figure 4, after clustering into 30 groups. 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, and the resulting structure is visibly simpler. (This graph provides an indication of how well the *method* is working; it is not intended to serve as documentation for the rule base maintainer.) Statistics about the facts for this grouping are as follows:

```
373      Rules
 30      Groups

244      Facts

 94      bottom
 30      top
 66      x-x
 12      x-y
  6      x-xy
  7      xy-x
  0      xy-xy
 21      x-any
  5      any-x
  3      any-any
```

This means that of the 244 facts found, 94 of them were characterized as *bottom*, meaning input data, not produced by any rules, and 30 *top*, meaning outputs, not used by any rules. The *x-x* category denotes intra-group facts. The remaining categories describe subspecies of inter-group facts: *x-y* denotes facts produced by just one group and used by just one other group; *x-xy* are those produced by one group and used by that group and one other group; *xy-x* are produced by two groups and used only by one of those two; *xy-xy* are produced by two

groups and used by the same two; *x-any* are produced by one group and used by two or more groups; *any-x* are produced by two or more groups and used by one group; and *any-any* covers the remaining more complex cases. Observe that 66 of the 120 non-top- or bottom-level facts in this system have become intra-group facts or local variables, while 54 are now inter-group. The groupings were also found to be substantially similar to the arrangement suggested by the rule author's naming of the original rules and his comments in the code; approximately 90 per cent of all pairs of rules were handled in the same way (either both separated or both joined) by the two arrangements. This provides an empirical estimate of the extent to which the structure of this knowledge base was successfully represented by the new method.

Finally, some support for the efficacy of this approach may be gained from a series of experiments conducted by J. Steve Davis [5]. Experimental subjects were given a simple rule-based system, and they were asked to modify it to incorporate an additional piece of knowledge. Some of the subjects received a rule-based system built according to the present method, while the others received essentially the same system constructed in the traditional way. Those in the first group made the required modifications significantly faster than those in the second, and their changes were judged to be of higher quality (i.e., they fit the structure of the original rule system better, in contrast to an ad hoc "patch" to handle a new case). This suggests that, at least in a limited experimental setting, the present method is indeed effective in its goal of reducing the effort required to maintain a knowledge base.

VIII. CONCLUSIONS

This paper has described a new method for developing more easily maintainable rule-based expert systems, which is based on dividing the rules into groups and concentrating on those facts that carry information between rules in different groups. It described a new algorithm for grouping the rules of a knowledge base automatically and a simple notation and set of software tools for the proposed method. The method was supported by a study of the

connectivity of rules and facts in rule-based systems, which found that they indeed have a latent structure, which can be used to support the programming methodology.

The resulting programming method requires the programmer 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 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.

ACKNOWLEDGMENTS

We are grateful to several AI researchers who have made knowledge bases they developed available to us: Bonnie Holte Bennett of Honeywell Research Center, Ralph Fink of the Naval Air Development Center, Mark Lerner of Columbia University, James Reggia of the University of Maryland, Alan Schultz of the Naval Research Laboratory, and Anne Werkheiser of the Army Engineer Topographic Laboratory. We also thank James Slagle for his helpful suggestions at the outset of this work and David Parnas for his comments on an earlier version of this paper.

REFERENCES

1. J. Baroff, R. Simon, F. Gilman, and B. Shneiderman, ‘‘Direct Manipulation User Interfaces for Expert Systems,’’ pp. 99-125 in *Expert Systems: The User Interface*, ed. J.A. Hendler, Ablex Publishing Co., Norwood, N.J. (1988).

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

3. L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming*, Addison-Wesley, Reading, Mass. (1985).

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

5. J.S. Davis, ‘‘Effect of Modularity on Maintainability of Rule-based Systems,’’ *International Journal of Man-Machine Studies* **32** pp. 439-447 (1990).

6. 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).

7. S. Fickas, ‘‘Development Tools for Rule-based Systems,’’ pp. 127-151 in *Expert Systems: The User Interface*, ed. J.A. Hendler, Ablex Publishing Co., Norwood, N.J. (1988).

8. C.L. Forgy, ‘‘OPS5 User’s Manual,’’ Technical Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University (1981).

9. J. Froscher, R. Grishman, J. Bachenko, and E. Marsh, ‘‘A Linguistically Motivated Approach to Automated Analysis of Military Messages,’’ *Proc. 1983 Conference on Intelligent Systems and Machines*, Oakland University (1983).

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

11. A. Gupta, C.L. Forgy, A. Newell, and R. Wedig, ‘‘Parallel Algorithms for Rule-Based Systems,’’ *Proc. 13th Annual International Symposium on Computer Architecture* pp. 116-120 (1986).

12. R.J.K. Jacob and J.N. Froscher, ‘‘Developing a Software Engineering Methodology for Rule-based Systems,’’ *Proc. 1985 Conference on Intelligent Systems and Machines* pp. 179-183, Oakland University (1985).

13. R.J.K. Jacob and J.N. Froscher, ‘‘Software Engineering for Rule-based Systems,’’ *Proc. Fall Joint Computer Conference* pp. 185-189, Dallas, Tex. (1986).

14. D.S. Nau and M. Gray, ‘‘Hierarchical Knowledge Clustering: A Way to Represent and Use Problem-solving Knowledge,’’ pp. 81-98 in *Expert Systems: The User Interface*, ed. J.A. Hendler, Ablex Publishing Co., Norwood, N.J. (1988).

15. D.S. Nau, ‘‘Automated Process Planning Using Hierarchical Abstraction,’’ Award Winner, Texas Instruments Call for Papers on Applications of AI in Industry, To appear in Texas Instruments Technical Journal (1988).

16. D.L. Parnas, ‘‘On the Criteria to be Used in Decomposing Systems into Modules,’’ *Comm. ACM* **15** pp. 1053-1058 (1972).

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

18. J.A. Reggia, R. Tabb, T.R. Price, M. Banko, and R. Hebel, ‘‘Computer-aided Assessment of Transient Ischemic Attacks: A Clinical Evaluation,’’ *Archives of Neurology* **41** pp. 1248-1254 (1984).

19. N. Sager, *Natural Language Information Processing*, Addison-Wesley, Reading, Mass. (1981).

20. 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).

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

22. M.F.M. Tenorio and D.I. Moldovan, ‘‘Mapping Production Systems into Multiprocessors,’’ *Proc. IEEE International Symposium on Parallel Processing* pp. 116-120 (1985).

23. M. Weiser, ‘‘Programmers Use Slices When Debugging,’’ *Comm. ACM* **25** pp. 446-452 (1982).

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

25. 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).

Figure Captions

Fig. 1. An illustration of the overall hierarchy of rule groups and sub-groups for the example system shown in Section IV. Each group of rules is shown as a box; each sub-group is shown as a box within a larger box.

Fig. 2. Components of ‘‘relatedness’’ measure between two rules. Rules are shown as lines, facts are shown as circles, and the fact whose score is given is shown as a double circle.

Fig. 3. Plot of individual rules of the simple expert system shown in Section III. Rules are shown as circles and facts, as lines connecting them.

Fig. 4. Plot of rules of a more complex expert system, described in Section VII. Rules are shown as dots and facts, as lines.

Fig. 5. Rules of Figure 4, clustered. Rule groups are shown as dots and inter-group facts, as lines.

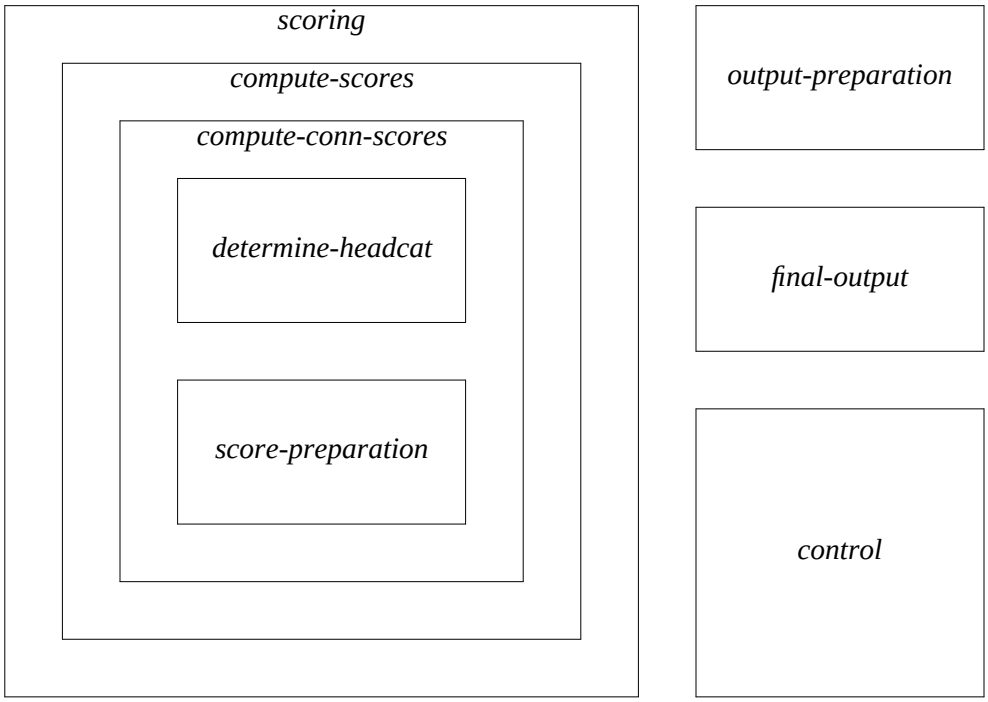
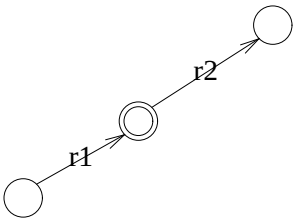
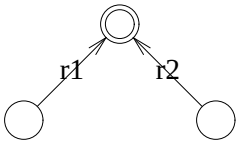

scoring
compute-scores
compute-conn-scores
determine-headcat
score-preparation
output-preparation
final-output
control

FIGURE 1.

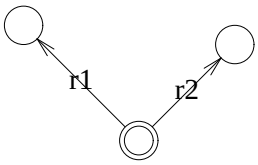
W(in-out) = 1.0



W(share-outs) = 0.75



W(share-ins) = 0.5



W(not-shared) = -0.25

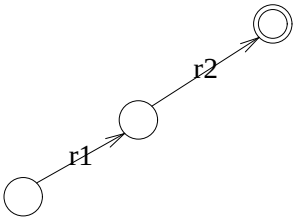


FIGURE 2.

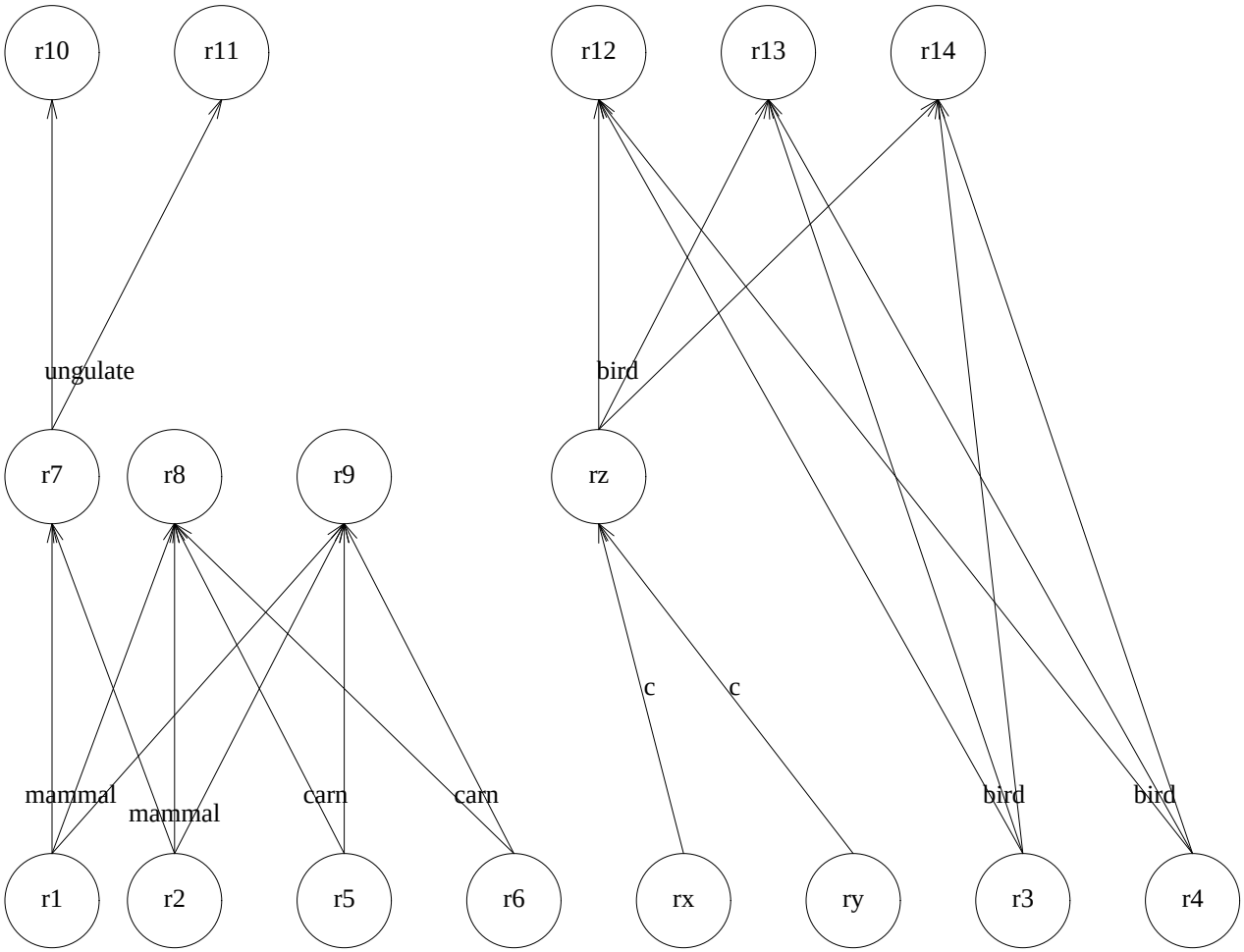

r10
r11
r12
r13
r14
ungulate
bird
r7
r8
r9
rz
c
c
mammal
mammal
carn
carn
bird
bird
r1
r2
r5
r6
rx
ry
r3
r4

FIGURE 3.

(draw *rules* *facts*) on TIAN data

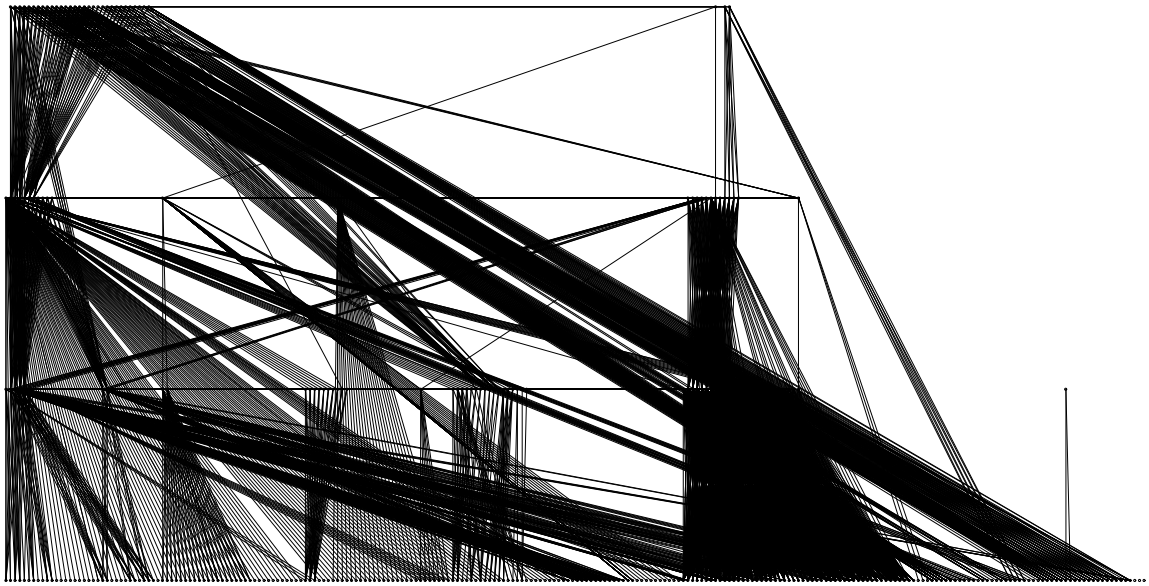


FIGURE 4.

Cluster(makerulegroups(tian data)), before iteration 80

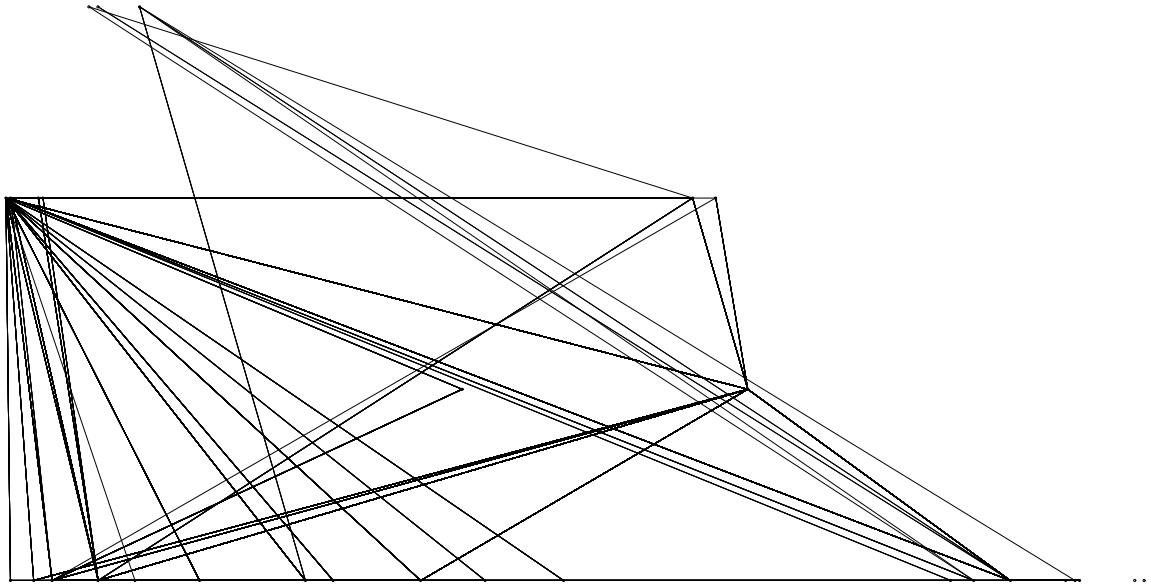


FIGURE 5.

Footnotes

The authors are with the Naval Research Laboratory, Washington, D.C. 20375.

Index terms

cluster analysis

information hiding

knowledge base

knowledge engineering

knowledge maintenance

production system

rule-based system

software engineering

Biography – Jacob

Robert J.K. Jacob (M'79) is a Computer Scientist in the Human-computer Interaction Lab at the Naval Research Laboratory in Washington, D.C. His current research interests are in the use of eye movements in human-computer interaction techniques, formal specification methods for describing user-computer dialogues, and software engineering methods to make expert systems easier for people to comprehend and modify.

Jacob received his B.A., M.S.E., and Ph.D. (EECS) degrees from The Johns Hopkins University, Baltimore, Md. in 1972, 1974, and 1976, respectively. He is also on the faculty of George Washington University, where he teaches courses in Computer Science. He is a member of IEEE, ACM, AAAI, and Sigma Xi and serves on the ACM SIGCHI Conference Planning Committee.

Biography – Froscher

Judith N. Froscher is....

Address for correspondence

Robert J. K. Jacob

Code 5530

Naval Research Laboratory

Washington, D.C. 20375