

ADAPTING SOFTWARE ENGINEERING PRINCIPLES
TO DIAGRAM, MODULARIZE, AND ANALYZE
RULE-BASED EXPERT SYSTEMS

By

STEVEN BRUCE CUDD

Bachelor of Science in Arts and Sciences
Oklahoma State University

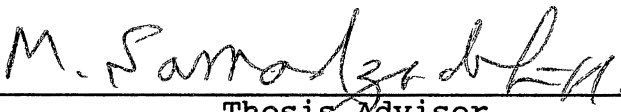
1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1990

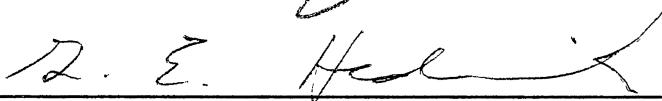
Thesis
1990
C967a
cop. 3

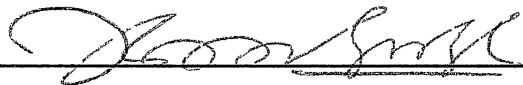
ADAPTING SOFTWARE ENGINEERING PRINCIPLES
TO DIAGRAM, MODULARIZE, AND ANALYZE
RULE-BASED EXPERT SYSTEMS

Approved :



Thesis Adviser







Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to Dr. Mansur Samadzadeh for his guidance and direction throughout my graduate program. Many thanks also go to Drs. George and Hedrick for serving on my graduate committee.

I also wish to thank my colleagues at MPSI for their ongoing support throughout my graduate studies. To Dr. Kyle Glover for his ideas and discussions that made this thesis subject possible. To Dr. Jacques LaFrance for his support and understanding of the work involved in receiving a Master's degree. To Vernon Sharp who made the long hours of work and study easier. To Bill and Judi Brody who set the stage for the final product.

Special thanks are due to my wife, Lisa, for all of her patience, understanding, and support for my graduate studies. Her love and caring have made the long hours, long nights, and the long drives all worthwhile.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION AND MOTIVATION.....	1
Background.....	1
Focus of Thesis.....	3
II. CHAINING FLOW DIAGRAMS	5
Chaining Types.....	5
Chaining Flow Diagram.....	6
Rule Infrastructure.....	7
Goal Symbol.....	8
Flow Connections.....	8
Generating a Chaining Flow Diagram.....	10
Algorithm 1.....	12
III. RULE GROUPS.....	15
Motivation.....	15
Dependent Relation.....	16
Rule Groups.....	17
IV. THE CHAINING FLUX.....	19
Background.....	19
Chaining Flux.....	20
Terminating and Initiating Rules.....	23
Control Rules.....	24
V. KNOWLEDGE MODULES.....	26
Background.....	26
Knowledge Modules.....	27
Algorithm 2.....	30
Intermediate Representation.....	32
VI. COMPLEXITY.....	34
Background.....	34
Spanning Trees.....	34
Knowledge Module Complexity.....	36

Chapter	Page
VII. CONCLUSION.....	39
Benefits.....	39
Future Work.....	40
REFERENCES..	42
APPENDIXES..	44
APPENDIX A - COMPUTER PROGRAM TO PARTITION A RULE-BASED KNOWLEDGE BASE.....	45
APPENDIX B - SAMPLE INPUT FOR THE COMPUTER PROGRAM.....	53
APPENDIX C - SAMPLE OUTPUT FROM THE COMPUTER PROGRAM.....	57
APPENDIX D - USERS' GUIDE FOR THE COMPUTER PROGRAM.....	76

LIST OF FIGURES

Figure		Page
1.	Rule Infrastructure Symbols.....	9
2.	Example of a Rule Infrastructure.....	9
3.	The Goal Symbol.....	9
4.	A Partial Chaining Flow Diagram.....	11
5.	Example of a Dependent Relation.....	16
6.	Examples of Chaining Flux.....	23
7.	A Knowledge Base Partitioned into Knowledge Modules.....	29
8.	Intermediate Knowledge Representation Tuples.	33
9.	Tree Generating Matrix.....	36
10.	Example of Knowledge Module Complexity.....	38

CHAPTER I

INTRODUCTION AND MOTIVATION

Background

One definition of artificial intelligence states that it is a subfield of computer science concerned with the possibility that a computer can be made to behave in ways that humans recognize as intelligent behavior in each other [4]. Thus, when presented with knowledge and data about a subject, the computer program could present an answer in much the same way a human would. To do this, the computer needs a way to represent the requisite knowledge and data.

One such computer program is an expert system. An expert system is a computer program that contains knowledge about objects, events, situations, and courses of actions which emulate the reasoning process of human experts in a particular domain [24]. An expert system consists of three parts: an inference engine, a knowledge base, and a user interface. The inference engine infers a conclusion from the knowledge and data provided. The knowledge base stores the knowledge and references the data in a format which the inference engine can manipulate. The user interface communicates the state of the inference process to the user.

Although all expert systems attempt to emulate the human reasoning process, their representations of knowledge and data vary. The two most popular forms of knowledge representation are production (if-then) rules and frames. Production systems are more widely used and more highly developed [15]. Therefore, this paper focuses on production rules as the form of knowledge representation scrutinized. However, the concepts discussed in this work are not limited to production systems and can be extended to frames and other similar knowledge representation forms.

As the use of expert systems becomes more widespread, several issues develop. Some of the issues involved are the creation, maintenance, and validation of knowledge bases. Since expert systems do not depend upon sequential processing as conventional programming does, these issues cannot be solved through conventional software development methodologies. Instead, some principles from software engineering can be utilized to devise a fresh approach to the aforementioned issues.

Development and maintenance of a large knowledge base is a non-trivial task. When a developer adds a new rule or edits an existing rule, the interaction between rules may change. The results may not be completely predictable. When using software engineering techniques for conventional program development, a developer can reduce the cost of future maintenance by increasing the accuracy of the program (i.e., confidence in the correctness of the program) and the

efficiency of the program development process. By adapting some of the software engineering techniques, the knowledge base developer may reduce the knowledge base complexity and still bring a level of predictability to the expert system development process.

The IEEE standard glossary on software engineering terminology defines software engineering as "the systematic approach to the development, operation, maintenance, and retirement of software" [9]. Previous attempts to link software engineering and expert system development include the partitioning of rules [10,11], measuring the complexity control in PROLOG and rule-based systems [14,16], data modeling of expert systems [7,13], and constructing well-structured knowledge bases [19,20,21].

Focus of Thesis

This work focuses on the adaptations of the software engineering principle of flow diagrams (data and control flow diagrams) to create a chaining flow diagram - a tool for conceptualization, communication, and abstraction. At the outset, the idea was to diagram a knowledge base to show the interactions of the rules to enhance the understandability of an expert system. Subsequently, the main objective of using flow diagrams, which subsumes the initial notion, became facilitating and expediting expert system development

and validation with the subsidiary goal of decreasing maintenance costs.

Included in this work is also the definition and proposal of the chaining flux as a software metric for rule-based knowledge systems. This metric assesses the manner in which a rule interacts with the other rules of a knowledge base. This metric is used as a basis for knowledge base partitioning and refinement.

CHAPTER II

CHAINING FLOW DIAGRAMS

Chaining Types

Two concepts govern rule-based knowledge systems: one is the exploration of the consequences of an action; the other is the investigation of ancillary goals. The forward and backward chains of a rule-based system accomplish these objectives.

The forward chain is data driven or data directed [6,15,25]. The inference engine explores the consequence of assigning a value to a variable. Thus, the data determines when rules are executed. An analogy can be made to a program data flow diagram. Whereas, the data flow diagram indicates the flow of data between procedures of a program, the forward chains in a knowledge base reveal the flow of data through the knowledge base. Analogously to a data flow diagram, this chart initially presents the overall information flow through the knowledge base.

The backward chain is goal driven or goal directed [6,15,25]. The inference engine seeks a value for a needed variable. The variable is usually associated with a subgoal - a task to be completed before further advancement of the

current objective. In their search for a variable's value, subgoals dictate which rules are fired. This search utilizes a program control flow diagram from software engineering. Similar to a procedural flowchart, the backward chains in a knowledge base outlines the flow of control through the knowledge base rules. This chart depicts the control flow of the knowledge base.

Chaining Flow Diagram

By combining forward and backward chaining into one diagram, the chaining flow diagram is developed. The chaining flow diagram shows the interaction of the rules through the backward and forward chains and unifies both the control and data flow concepts from software engineering.

Three basic components compose a chaining flow diagram. The first is the rule infrastructure. The rule infrastructure consists of a rule's name and all variables referenced in that rule which either initiate or complete a forward or backward chain. The second is a line, a flow connection, with a single arrowhead that connects different rules with common forward and backward chains. The flow connections show the interaction and information (data/control) flow through the rules due to the forward and backward chains. The third component is the overall goal of the knowledge system. The chaining flow diagram shows the flow through the knowledge base when ascertaining a value for the goal.

Rule Infrastructure

The rule infrastructure represents the rule's name, forward chaining variables, and backward chaining variables. The chaining flow diagram employs three different symbols to distinguish the components of the rule. The rule name is indicated by a rectangle with the rule's name within the rectangle. The name of a variable within an elongated hexagon symbolizes a forward chaining variable. An elongated hourglass containing the name of a variable denotes a backward chaining variable. The rule infrastructure displays the rule name first, followed by the names of the chaining variables in the same order that they appear in the rule. Figure 1 contains the rule infrastructure symbols.

One problem that immediately arises is whether the variable appearing in the rule infrastructure is referenced in a hypothesis or conclusion of a rule. To resolve this confusion, a bold line is drawn between the symbols holding the last chaining variable appearing in a hypothesis and the first chaining variable appearing in a conclusion. The variables appearing above the bold line are in the rule's hypotheses and the variables below the bold line are in the rule's conclusions. A color system also could solve the problem by providing different colors for the variable names in the hypotheses and conclusions. For example, the rule name could be in black, the chaining variables in the hy-

potheses could be in red, and the chaining variables in the conclusions could be in blue. Both of these solutions can be extended to support the case of a counter-conclusion if the expert system supports the IF-THEN-ELSE construct (another line is drawn between the variables in the conclusions and those in the counter-conclusions; or, in a color system, the chaining variables in a counter-conclusion could be in a different color). Figure 2 contains an example of a rule infrastructure.

Goal Symbol

A special symbol designates the overall goal of the knowledge base. An elongated hexagon with one end pointing inward surrounding the goal variable's name designates the goal. It resembles an arrow since it is always the goal that triggers the start of the inference process. See Figure 3 for a depiction of the goal symbol.

Flow Connections

If one rule modifies a variable's value and another rule references the same variable, then a directed line, a flow connection, is drawn to connect the two rules. The flow connection indicates the source rule and the target rule. The source rule is the rule at which the chaining originates. The target rule is the rule at which the chaining

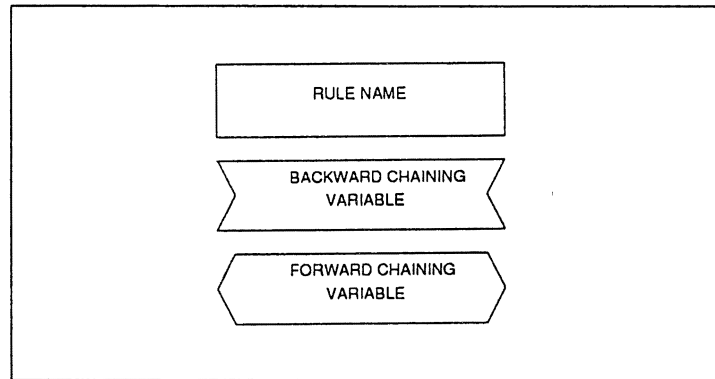


Figure 1. Rule Infrastructure Symbols

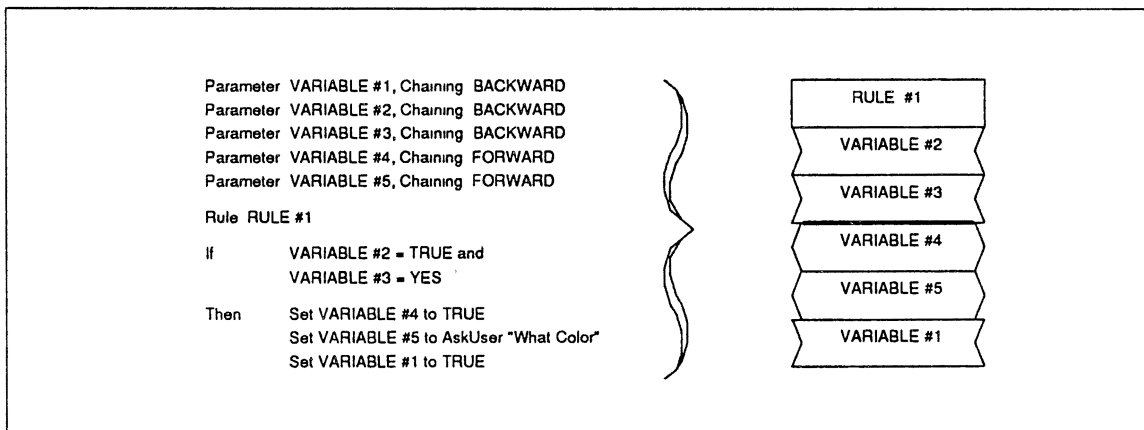


Figure 2. Example of a Rule Infrastructure

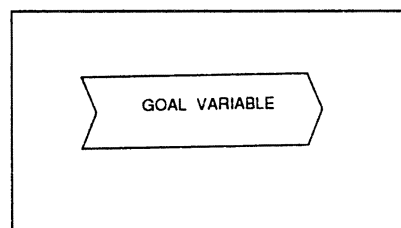


Figure 3. The Goal Symbol

terminates. The line's arrowhead points from the source rule to the target rule.

Note that in the case of a forward chain, the source rule is the rule that modifies the variable's value and the target rule is the rule that references the variable's value. For a backward chain, the opposite is true. This is because a backward chain is initiated when a rule references a variable without a value. The source rule is the rule that references the variable's value and the target rule is the rule that modifies the variable's value.

The relationship between source rules and target rules is not necessarily one-to-one. A source rule may have several target rules, and a target rule may have several source rules. Furthermore, a variable chain may have multiple source and target rules. Figure 4 is an example of a partial chaining flow diagram.

Generating a Chaining Flow Diagram

The creation of a chaining flow diagram is a recursive procedure. It is built in a left-to-right, top-down fashion. Before generating the part of the chaining flow diagram originating at a given rule, all sections originating at the given rule's target rule(s) are generated first.

The algorithm that generates the chaining flow diagram appears after this paragraph. The algorithm assumes, without loss of generality, that the knowledge base has only one

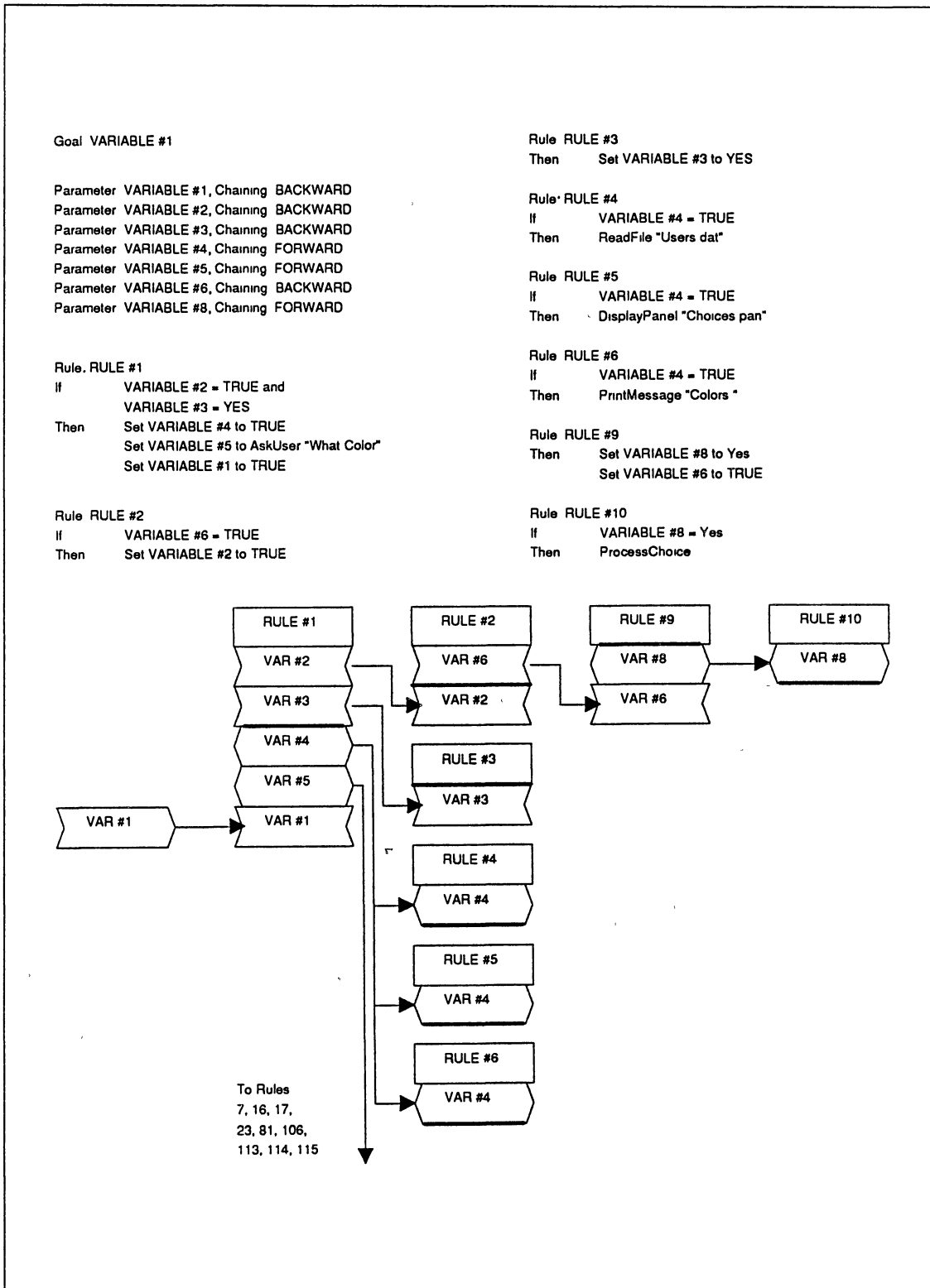


Figure 4. A Partial Chaining Flow Diagram

goal variable. Two other assumptions for the algorithm are as follows: no rule can reference itself and a variable cannot be both forward and backward chaining. In the algorithm, the set V_s is the current set of chaining variables within a given source rules (in the initialization, the goal variable is chosen), V_o is the initial chaining variable set (the goal variable) which, when matched by V_s , signals the end of the algorithm, the set B is the set of all backward chaining variables in the knowledge base, the set F is the set of all forward chaining variables in the knowledge base, and the set I is the set of all rules whose infrastructures have been drawn. As a post-processing step, at least one pass may be necessary to make the chaining flow diagram aesthetically pleasing (e.g., to reduce the number of intercepts and to distribute the nodes and edges evenly).

Algorithm 1

Generate a Chaining Flow Diagram.

Input. A knowledge base and its goal variable.

Output. A Chaining Flow Diagram.

Method.

Initialization:

- A. Place the name of the goal variable within the goal symbol.
- B. Mark the goal symbol as the current source rule, s.

- C. Construct $V_O = V_S = \{ \text{goal variable} \}$.
- D. Construct $B = \{ v \mid v \text{ is a backward chaining variable} \}$.
- E. Construct $F = \{ v \mid v \text{ is a forward chaining variable} \}$.
- F. Set $I = \{ \}$.

Body:

For every v in V_S , perform the following:

- A. Mark v as the current active variable.
- B. If v is in B , construct a set of rules $R_v = \{ r \mid r \text{ is a rule that concludes a value for } v \}$.
- C. If v is in F , construct a set of rules $R_v = \{ r \mid r \text{ is a rule that references } v\text{'s value in its hypothesis} \}$.
- D. For every rule r in R_v , perform the following:
 1. Mark r as the current target rule t .
 2. If t is not in I , then draw t 's infrastructure to the right of the current source rule s and add t to I .
 3. Draw a connecting arrow originating at the current active variable v in the current source rule s and terminating at the current active variable v in the current target rule t .

- E. For every rule r in R_V , perform the following:
1. Mark r as the current source rule s .
 2. Construct $V_s = \{ v \mid v \text{ is a chaining variable of the current source rule } s \text{ that originates at } s \}$.
 3. If V_s is non-null, recursively repeat the Body.
- F. If $V_s = V_O$, then halt.

The algorithm terminates when the chaining variable set V_s returns to its initial state V_O , the goal variable. This signals that all the chaining flow diagram sections originating at rules which conclude a value for the goal variable have been generated. Since the number of rules, the number of variables, and the number of variables referenced in a rule are all finite, the algorithm always terminates. This algorithm focuses on just one goal variable. However, it can be easily extended to include multiple goal variables by including all of the goal variables in the sets V_s and V_O in the initialization step.

CHAPTER III

RULE GROUPS

Motivation

To improve the comprehensibility, maintainability, and general structure of a knowledge system, one can divide the information in the knowledge base into modules with each module containing a group of rules that interact closely among one another. An individual module should have a reasonably distinct purpose and essentially should be isolated from other modules. Ideally, a modification in one module should have no effect on the execution of other modules.

When modularization is achieved, it delivers clear benefits for the knowledge engineer and maintainer. The prime benefit is expected to be localizing the effects of a rule modification in well-behaved systems. As stated previously, the results of adding or modifying a rule can affect other rules. If a rule is within an isolated module, the effects of any rule change can be contained within its rule group. As shown in previous studies with conventional programs, modularization leads to increased program understanding and lower maintenance costs [17,18].

Dependent Relation

A dependent rule is a rule that relies upon the execution of another rule before it can evaluate all of its hypotheses and execute its conclusions. In terms of a chaining flow diagram, a dependency between two rules exists if a chaining variable, either backward or forward, connects two rules. The arrowhead on the line indicates which rule is dependent upon the other one. The target rule is dependent upon the source rule. We define this relationship between two rules as a dependent relation. A dependent relation is critical since it governs the firing of a rule. See Figure 5 for an example of a dependent relation.

The dependent relation is transitive. If rule R3 is dependent upon rule R2 and rule R2 is dependent upon rule R1, then R3 is also dependent upon R1 by transitivity. To distinguish between the two dependencies, R3 is said to be directly dependent upon R2 and level-1 dependent upon R1. Generalizing, if a rule is shown to be dependent upon

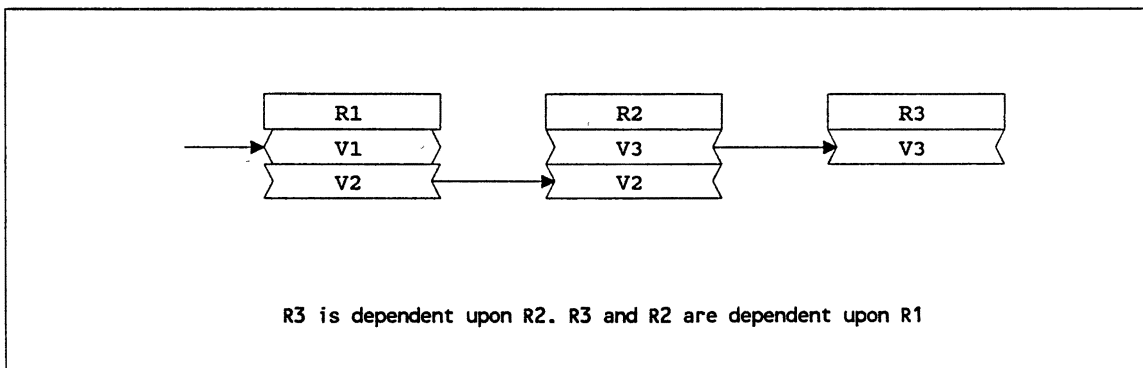


Figure 5. Example of a Dependent Relation

another rule either directly or transitively, it is level-X dependent upon that rule with X being the number of rules in the connection. Thus, a direct dependency is a level-0 dependency.

The dependent relation is neither reflexive nor symmetric, which is to be expected. If a rule were allowed to be dependent solely upon itself either directly or transitively, then an infinite loop would result. A nonresolvable paradox develops: the rule cannot execute its conclusions because it needs the variable's value to evaluate a hypothesis; however, the only rule to conclude a value for the variable is that same rule whose hypotheses can never be satisfied.

Symbolically, we can denote the dependency, either directly or transitively, of rule x upon rule y by " $x \leftarrow y$ " or " $y \rightarrow x$ ". To show the level of dependency between two rules, we define a dependency function $D(x,y)$. For any two rules x and y, $D(x,y)$ is the level of dependency of rule x upon rule y. For instance in the above example, $R3 \leftarrow R2$, $R3 \leftarrow R1$, $R2 \leftarrow R1$, $D(R3,R2)=0$, $D(R3,R1)=1$, and $D(R2,R1)=0$.

Rule Groups

Having defined the dependent relation, we can characterize a rule group. Intuitively speaking, a rule group is a collection of rules that are related. Formally, a rule

group is a collection of rules that are level-N, level-(N-1), ..., level-0, dependent upon a given rule. Thus, each rule group has two distinguishing features: the rule it is based upon and the level of dependency. Adding one to the highest level of dependency in the rule group will be called the depth of the rule group. The depth of a rule group is defined to be one plus the maximum of the set $\{n_1, n_2, \dots, n_k\}$ where n_i is the dependency level of rule r_i . One is added to prevent a depth of zero when a rule group consists of one source rule with one or more target rules.

CHAPTER IV

THE CHAINING FLUX

Background

In this section, a metric that measures a rule's influence in the knowledge base is introduced. This measure, the chaining flux, is based solely upon a rule's chaining variables. The foundation of this measure is derived from the work done by Henry and Kafura in the area of information flow metrics [8] and extended to rule-based systems by O'Neal and Edwards [16].

Henry and Kafura define two measures, fan-in and fan-out, that are used in the construction of a metric to measure the complexity of a procedure's connection to its environment. The fan-in of a procedure is the number of local flows into the procedure plus the number of data structures from which the procedure retrieves information. The fan-out of a procedure is the number of local flows from the procedure plus the number of data structures which the procedure updates. They proceed to demonstrate that the metric fan-in multiplied by fan-out is a good indicator of a procedure's complexity as correlated with the number of reported faults.

O'Neal and Edwards extended Henry and Kafura's concept of fan-in and fan-out to rule-based systems. They state that "the complexity of a rule is based upon the number of interactions between the rule and the rest of the program and, to a lesser degree, the internal complexity of that rule" [16]. So, to measure the complexity of a rule, O'Neal and Edwards describe five complexity measures: the data fan-in, the data fan-out, the object transfer, the rule fan-in, and the rule fan-out. Of these measures, the only two of concern here are the rule fan-in and the rule fan-out. These two measures quantify the way rules interact with one another; the other measures are concerned with data manipulation, which does not pertain to this study. They define the rule fan-in as the number of rules which directly could have caused the rule in question to fire. Thus, the rule fan-in is the number of possible "predecessors" to the rule. They define the rule fan-out as the number of rules which could become eligible to fire as a direct result of the rule in question firing. Thus, the rule fan-out is the number of possible "successors" to the rule.

Chaining Flux

Analogously to O'Neal and Edwards, this work adapts the fan-in and fan-out metrics to a rule-based system. In this regard, a rule in a knowledge system is likened to a proce-

ture in a conventional program. Whereas, a procedure is connected to its environment by procedure calls and argument transmissions, a rule is connected to its environment by chaining variables and their values. A rule's connections to its environment are measured as a function of that rule's fan-in and fan-out.

We define the fan-in of a rule as the number of source rules with variable chains to that rule. This expresses the number of rules which possibly can transfer control to the rule (which is consistent with O'Neal and Edward's definition of rule fan-in). The number of source rules with variable chains to a rule may be greater than the number of incoming variable chains to that rule. This is due to the fact that an incoming variable chain can originate from more than one source rule. The fan-in of a rule measures the potential flow of control into a rule.

We define the fan-out of a rule as the number of target rules that receive chains from that rule. This indicates the number of rules that can possibly inherit control from the rule (again this is consistent with O'Neal and Edward's definition of rule fan-out). Since a source rule may have multiple target rules, the fan-out of a rule may be greater than the number of variable chains that initiate outgoing chains. The fan-out measures the potential flow of control out of a rule.

Based on the fan-in and fan-out measures for a rule, we define the chaining flux metric for a rule. The chaining

flux, $F(X)$, of a rule is the number of chaining variables within the rule multiplied by the square of product of the rule's fan-in and fan-out.

$$F(X) = n * (\text{fan-in} * \text{fan-out})^2 . \quad (1)$$

The chaining flux measures the complexity of a rule's connection to its environment; i.e., all the other relevant rules. The factor n denotes the number of chaining variables in rule X . This factor represents a bulk component within the rule since bulk metrics like program statements correlate well with program complexity [3]. The product of fan-in and fan-out represents the total possible number of combinations for the transfer of control from a source rule of rule X to a target rule of rule X . The raising of the fan-in times fan-out factor to the power of two is consistent with Henry and Kafura's paper [8]. One justification that can be offered for the squaring of the second factor is that the number of rules which can potentially transfer control to rule X and the number of rules to which rule X potentially can transfer control is more critical in measuring rule X 's connection to the rest of the rules than just the number of chaining variables within rule X . Further refinement of the formula for chaining flux through empirical analysis is part of the suggested future work. Figure 6 has an example of the chaining flux.

Terminating and Initiating Rules

We define a terminating rule as a rule that accepts variable chains from source rules but does not initiate variable chains to any target rules. Thus, a terminating rule does not transfer control to any other rule; i.e., the flow of control is ended with a terminating rule. Since a terminating rule does not transfer control to another rule,

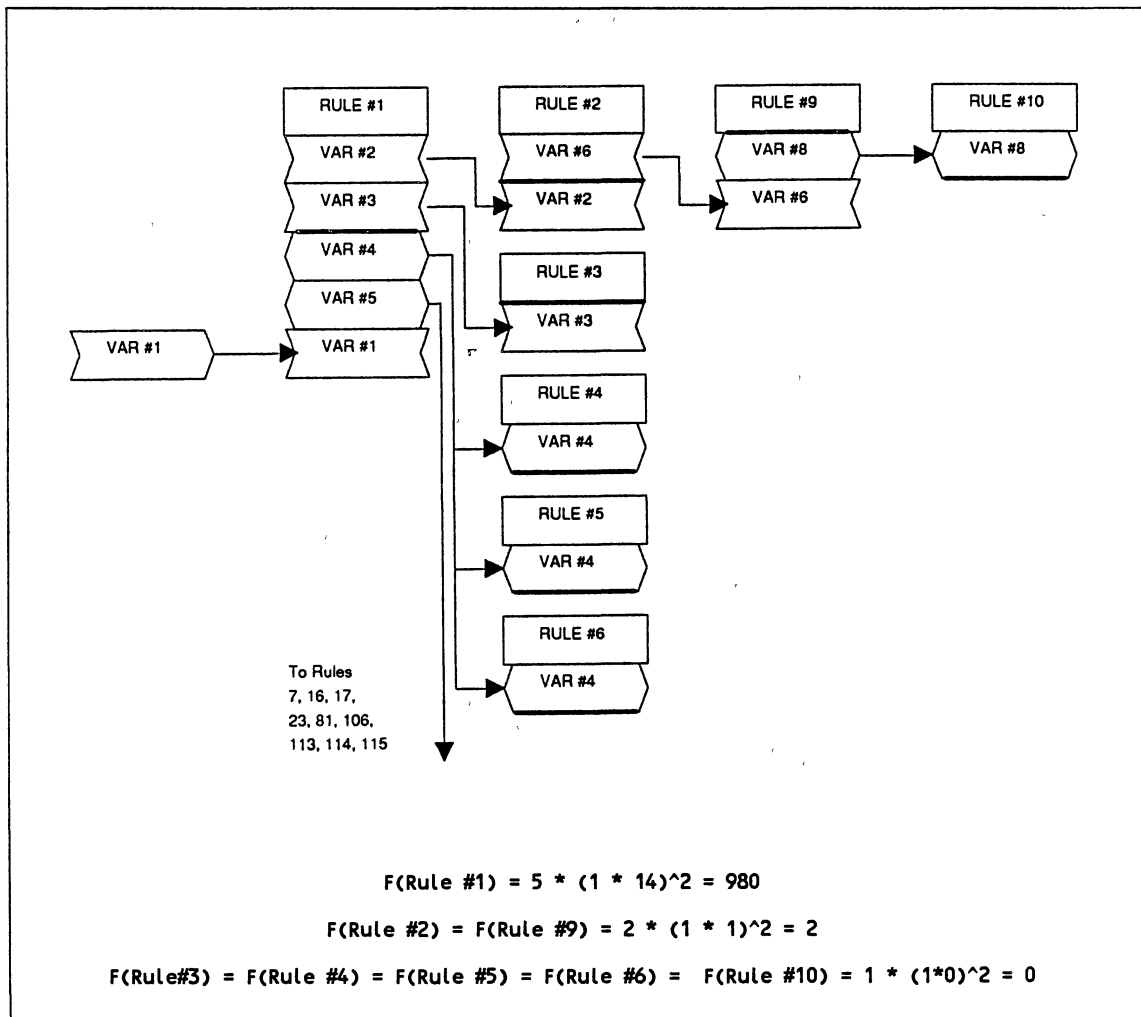


Figure 6. Examples of Chaining Flux

the fan-out of a terminating rule is zero. With the fan-out being zero, the chaining flux of a terminating rule is zero.

An initiating rule is a rule that only transfers control to target rules, but the rule itself is not a target rule for any other rule. This results in a fan-in of zero and leads to a chaining flux of zero. But what does an initiating rule mean? Since no other rules can transfer control to an initiating rule, the initiating rule can never execute. Also any rules dependent upon an initiating rule can never execute. These rules are isolated and cannot be considered part of the knowledge system since they can never be executed. Thus, in a knowledge system, the only useful rules with a chaining flux of zero are the terminating rules.

Control Rules

One significant consequence of the chaining flux is that it permits us to single out particular rules that dominate the flow of control through the knowledge base. These dominating rules have relatively higher chaining flux scores than the rest of the rules which reflects their greater influence on the flow of control in the knowledge base. A rule with a high chaining flux (to be defined below) is referred to as a control rule.

A high chaining flux is knowledge base dependent. The greater the number of rules and chaining variables in a

knowledge base, the higher the possible chaining flux scores for individual rules and hence the higher the threshold for the control rules. Also, the chaining flux depends on how the knowledge system is written. If all of the rules are strung along in a chain with few terminating rules, then the chaining flux for each rule may not vary. If the rules are grouped together with a number of terminating rules, then there exists a high possibility that the chaining flux for each rule will vary widely.

A control rule is distinguished from other rules by calculating the average chaining flux of the knowledge system. Thus, a control rule can be defined as a rule with a chaining flux greater than the average chaining flux of the knowledge system. This provides a good mark since these rules are more complex than the average rule and they can transfer control from more source rules to more target rules than the average rule. The concept of the control rule is used in the next section to partition the rules in a knowledge base.

CHAPTER V

KNOWLEDGE MODULES

Background

In this section, an algorithm to partition the rules in a knowledge system into different modules is presented. Partitioning the rules is similar to breaking a conventional program into smaller procedures for conceptual manageability. A software developer can employ several software engineering techniques to aid in the design and implementation of conventional programs. However, there are very few software engineering techniques to help a knowledge engineer in designing an expert system.

One such technique is outlined by Jacob and Froscher [10,11,12]. They define a rule relatedness measure as the basis for a rule partitioning algorithm. They state that two rules are related if they share a common non-chaining variable. Depending on the type of relation, they empirically score the relatedness between the rules, then use that score to partition the rules into distinct groups.

A similar strategy for partitioning rules now follows; however, there is one distinct difference in the philosophy of rule partitioning. Jacob and Froscher base their rule

"relatedness" measure solely on the strength of non-chaining variables. They ignore chaining variables entirely and advocate that one should segregate the control knowledge from the domain knowledge. Since this is not always possible or desired, we take the opposite approach. The control knowledge emulates the problem solving techniques of the expert. Thus, by focusing our attention on the control knowledge, the partitioning algorithm can create modules that reflect the sequence of rule execution. This can help a knowledge engineer to organize rules anticipating inference paths.

The chaining flux score from the previous section is used as the basis for rule partitioning. The chaining flux is derived solely from the control (chaining) variables. This gives us the advantage of partitioning the system in accordance to when the rules execute and which rules execute together.

Knowledge Modules

We define a knowledge module as a level-N rule group based upon a base rule which is a target rule that has one and only one chaining variable leading to it. We refer to this chaining variable as an interface chain. The interface chain connects the knowledge module with the remainder of the knowledge system. It is the interface chain that dictates when the rules within the knowledge module are considered for execution. Note that the interface chain may

be derived from several different source rules. Thus, the interface chain may be one-to-one or many-to-one but never one-to-many or many-to-many when describing the relationship of the source rules to the base rule. The interface chain does not rely on a specific chaining strategy; it may be either a forward or a backward chaining variable.

The notion of a knowledge module is critical since it defines a specialized set of related rules that can be accessed only through the interface chain to the knowledge module. If one disconnects the interface chain, an entire knowledge module and any dependent knowledge modules will be isolated. Thus, an expert system can lose integral knowledge if an interface chain is severed. We can now visualize a knowledge system as a collection of specialized knowledge modules connected by interface chains. Figure 7 contains a relatively small knowledge base that has been partitioned into knowledge modules.

A knowledge module is defined as being based upon a single rule - a base rule. At the outset, a control rule is selected as the base rule for each knowledge module. Thus, each knowledge module is based upon a rule that dominates the flow of control of a part of the knowledge system. Each knowledge module consists of a control rule and rules dependent upon that control rule.

Partitioning a knowledge base into separate knowledge modules is a recursive procedure. After a rule is assigned to a knowledge module, all of its dependent rules up to a

base rule are assigned to the same knowledge module before proceeding to a different rule at the same level. The following algorithm partitions a knowledge base into knowledge

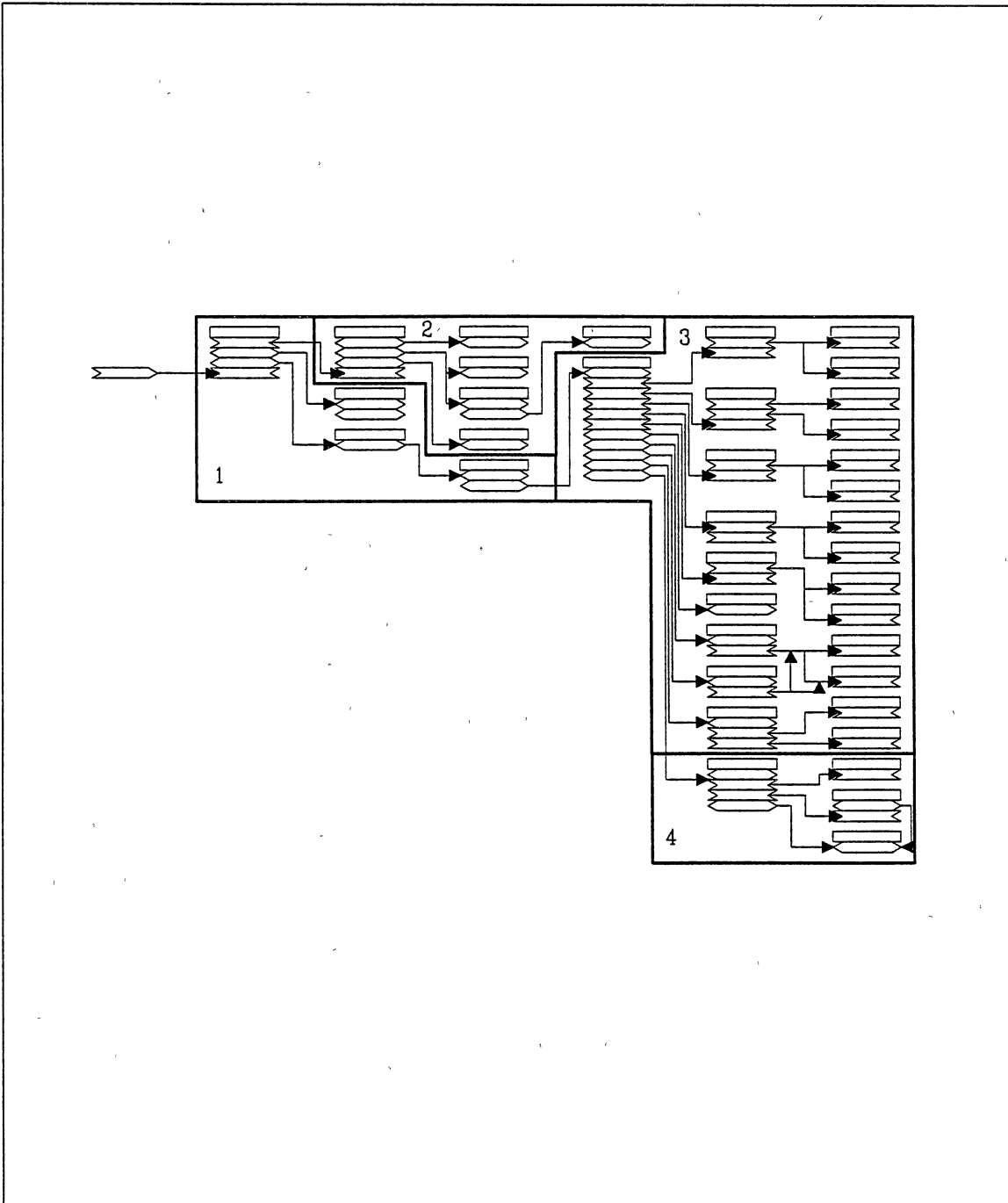


Figure 7. A Knowledge Base Partitioned into Knowledge Modules.

modules according to control rules. In Algorithm 2 below, the set R is the set of rules within the knowledge base, the set B is the set of base rules, the set D is a set of dependent rules, the set M is the set of rules that have been marked as already having been included in a knowledge module, and the set K_b is the set of rules corresponding to a knowledge module based upon rule b .

Algorithm 2

Partition a Knowledge Base.

Input. A knowledge base and its goal variable.

Output. Knowledge base modules.

Method.

Initialization:

- A. Construct $R = \{ r \mid r \text{ is a rule in the knowledge base } \}$.
- B. Set $M = \{ \}$.
- C. For every rule r in R , calculate r 's chaining flux.
- D. Construct $B = \{ r \mid F(r) > \text{"threshold value" for the chaining flux } \}$.
- E. For every rule r that concludes a value for the goal variable, if r is not in B , add r to B .

Body:

- A. Set B_0 to B .

- B. For every b in B , perform the following:
1. Construct $K_b = \{b\}$.
 2. Construct $D = \{ r \mid r \text{ is in } R, r \text{ is not in } B, r \leftarrow b \text{ (} r \text{ is dependent upon } b \text{), and } r \leftarrow n \text{ (} r \text{ is not dependent upon } n \text{) for any } n \text{ in } B \text{ such that } b \leftarrow n \}$.
 3. For every r in D , perform the following:
 - a. If r is not in M , add r to M and add r to K_b .
 - b. If r is in M and r is not in K_b , then
 1. $K_i = K_i - \{ r \}$.
 2. Construct $S = \{ s \mid s \text{ is in } R, s \text{ is not in } B, s \leftarrow r, \text{ and } s \leftarrow n \text{ for any } n \text{ in } B \text{ such that } b \leftarrow n \}$.
 3. For every s in S , remove s from M .
 4. Add r to B .
- C. If $B_0 = B$ and $M = R$, then halt, else repeat the body.

The algorithm terminates when all rules have been assigned to a knowledge module and no new base rules have been added. Because the number of rules, the number of variables, and the number of variables referenced in a rule are all finite, this algorithm always terminates. Even in the worst possible case in which every rule references every other rule, the algorithm terminates because it creates one knowledge module for each rule.

This algorithm creates new base rules in step 3b of the body. A base rule is created when a rule is assigned to more than one knowledge module. This is done to avoid overlapping knowledge modules. Also, it insures us that each knowledge module has one and only one interface chain. This results in having more knowledge modules than control rules; and in the worst possible case, a knowledge module for each rule. If a knowledge module is created for each rule, then the knowledge system either is very simplistic (i.e., all rules execute in a single chain) or very complex (i.e., all rules are connected to one another).

The algorithm partitions a knowledge system. This algorithm can be extended to partition a knowledge system on several levels. Once the top level is partitioned, another application of this algorithm to each separate knowledge module leads to an inner-partitioning of each knowledge module. With a recursive invocation of the algorithm, not only can a knowledge base be partitioned, but all of its knowledge modules can be partitioned into smaller, finer modules also.

Intermediate Representation

To insure implementation and language independence of the algorithm, an intermediate form of representation for knowledge systems is developed. This representation does not show the rule's intent; rather, it captures the rule

chaining aspects needed for the algorithm. This transformation is different from the representation presented by O'Neal and Edwards in [16]. Their representation does not emphasize the rule's chaining aspects; rather, it emphasizes how the rule accesses, creates, modifies, and deletes data.

Each rule can be represented by a set of tuples. Each tuple contains the source rule's name, the target rule's name, and the name of the chaining variable which connects the two rules. Since a tuple contains both a source and target rule, it identifies two rules. Once a system is built to partition a knowledge system based upon this representation, we have to build the transformation from the original implementation to the tuple representation. Figure 8 contains an example of this representation. An implementation of this algorithm using this representation is given in Appendix A. An example set of tuples is listed in Appendix B. The output of the example set of tuples from Appendix B using the program from Appendix A is given in Appendix C.

Tuple Representation: < Source Rule, Target Rule, Variable Chain >	
Tuples for the Partial Knowledge Base from Figure 4:	
< Rule #1, Rule #2, Variable #2 >	< Rule #1, Rule #3, Variable #3 >
< Rule #1, Rule #4, Variable #4 >	< Rule #1, Rule #5, Variable #5 >
< Rule #1, Rule #6, Variable #4 >	< Rule #2, Rule #9, Variable #6 >
< Rule #9, Rule #10, Variable #8 >	

Figure 8. Intermediate Knowledge Representation Tuples

CHAPTER VI

COMPLEXITY

Background

Once a knowledge system is partitioned, it is possible to adapt some of the existing software metrics to measure the complexity of the newly created knowledge modules and their interdependencies. The complexity measure uses the graphical representation of the knowledge system provided by the chaining flow diagram. The approach taken here is similar to that of Bieman and Edwards to measure the complexity of data dependency diagrams [2].

Spanning Trees

By basing our complexity measure on the graphical representation of the chaining flow diagram, we can draw on results from graph theory for the metric. First, we present some preliminary definitions from graph theory. A graph $G=(N,E)$ consists of a finite set of nodes N and a finite set of edges E [5]. To each edge there corresponds a pair of nodes; if the pair is ordered, then the graph is said to be directed [5]. A cycle in a graph is a path from some node

back to itself where no edge appears more than once and the initial node is the only node appearing more than once [5]. A graph is connected if there exists a path from any node to any other node [5]. A tree is a connected graph with no cycles [5]. The root node of the tree is a node that is the predecessor of all other nodes in the tree [5]. A spanning tree of a directed graph G is a graph $ST(G) = (N, E')$, where E' is a subset of E and $ST(G)$ is a tree that includes every node in G [5]. The rooted spanning tree complexity with root node n ($RSTC(n)$) of a graph G is the number of distinct spanning trees with root n that can be constructed from the graph consisting of the nodes and edges of G that are successors of n [2]. The number of spanning trees within a graph can be used as a complexity measure. The idea of using the number of spanning trees as a measure of complexity is not new; it has been described before in graph theory literature [1,23, as cited in 2].

The rooted spanning tree complexity is calculated as the determinant of a tree-generating matrix [2,23]. A tree-generating matrix is defined as follows: Let $G(N, E)$ be a directed graph, let n_1 be a member of N , and associate a variable n_{ij} with the number of directed edges from node n_i to node n_j . The matrix is defined in Figure 9.

Knowledge Module Complexity

Since the chaining flow diagram is a tree, by definition, it contains no cycles. Thus, the tree-generating matrix of a chaining flow diagram becomes upper triangular. The determinant of an upper triangular matrix is the product of the terms on the main diagonal. In this matrix, each diagonal term represents the number of incoming edges to one node. For a chaining flow diagram, the number of incoming edges to a node is the fan-in of the rule represented by the node. Thus, the RSTC for a chaining flow diagram rooted at node n is the product of the fan-in of all the rules dependent upon n . The rooted spanning tree complexity of a chaining flow diagram rooted at node n is a simple measure of the complexity of a knowledge module or module hierarchy.

Other factors in the complexity of a knowledge module include the number of rules in the knowledge module and the depth of the knowledge module. Weighting the rooted spanning tree complexity by the number of rules and the depth of

$$\begin{bmatrix} \sum_{i \neq 2} a_{i,2} & -a_{23} & \dots & -a_{2n} \\ -a_{32} & \sum_{i \neq 3} a_{i,3} & \dots & -a_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{n2} & -a_{n3} & \dots & \sum_{i \neq n} a_{i,n} \end{bmatrix}$$

Figure 9. Tree Generating Matrix

the knowledge module yields the knowledge module complexity, K , for a module X as follows.

$$K(X) = n * d * RSTC(X) \quad (2)$$

The number of rules, n , represents a bulk complexity component and the depth, d , of the knowledge module is an indicator of the nesting level complexity component. Weighting the rooted spanning tree complexity with these two factors creates a metric that is sensitive to three important components. Figure 10 contains some examples of knowledge module complexity.

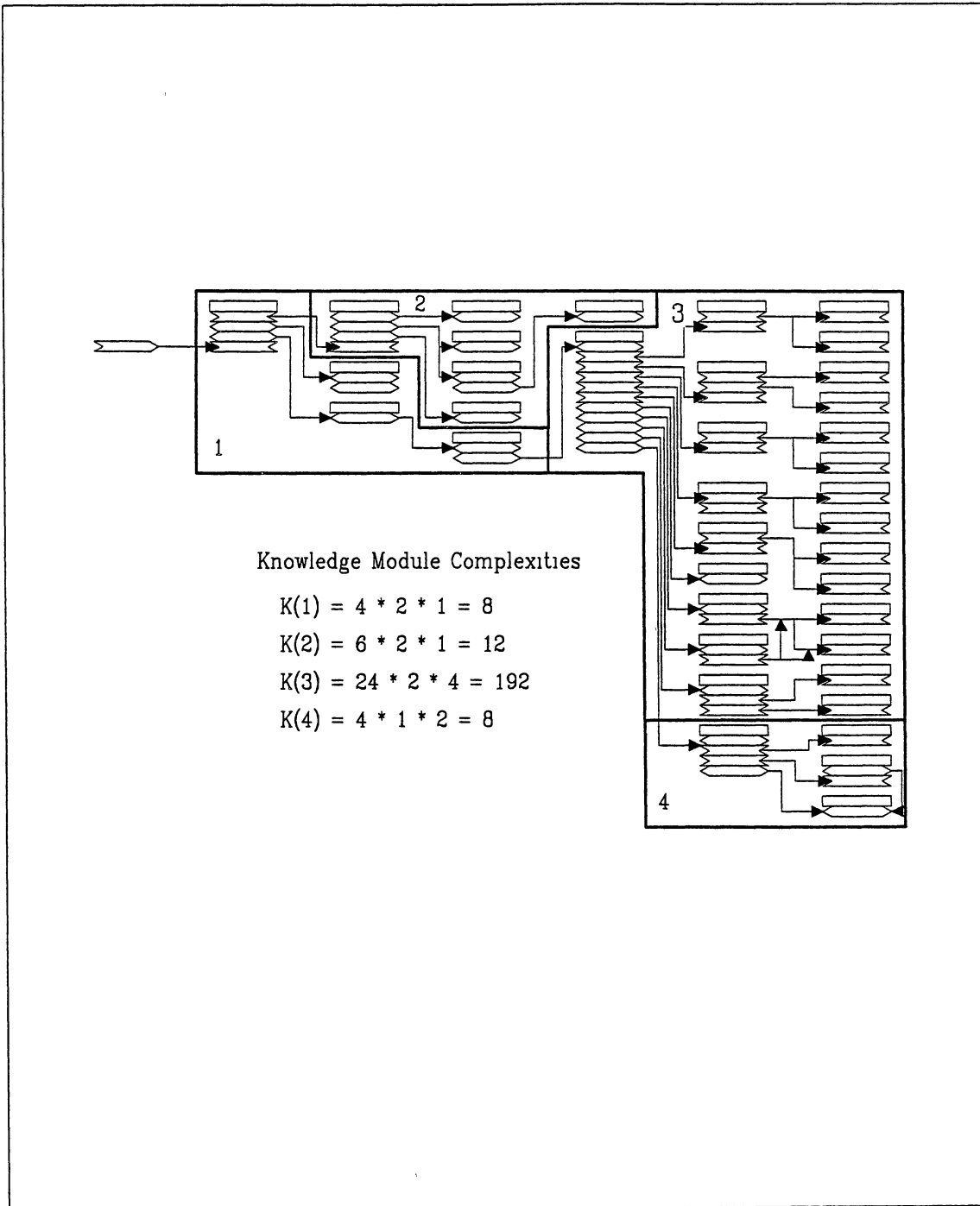


Figure 10. Example of Knowledge Module Complexity

CHAPTER VII

CONCLUSION

Benefits

The chaining flow diagram delivers some clear benefits to the developer, maintainer, and end-user. It gives the developer a clear representation of the system. It shows the maintainer which rules are dependent upon which other rules as he or she updates the knowledge base for information for better and more cost-effective maintenance. It demonstrates the flow of information through the system for the end-user. Overall, it helps an individual to gain a better understanding of the knowledge system as a whole.

The chaining flux metric presents the knowledge developer and maintainer with a measure of the flow of control into and out of a given rule. Used in conjunction with the other rules of the knowledge system, it shows which rules dominate the flow of control. It helps the developer and maintainer quickly grasp which rules control the inference process.

With the chaining flux for each rule calculated, the knowledge base can then be partitioned according to when rules execute and which rules operate together. This makes the knowledge system easier to maintain and understand. It

helps the developer and maintainer see how rules interact with one another and which rules handle certain processes.

Future Work

Several areas of this work can be expanded upon or clarified with additional time and effort. With respect to the chaining flow diagram, a program should be written to take a knowledge system as input and create the corresponding chaining flow diagram as its output. Also, a study should compare the chaining flow diagram as a representation of a knowledge system for a developer, maintainer, and end-user to other forms of representation.

For the chaining flux metric, knowledge base modularization, and knowledge module complexity, several knowledge systems written in different environments need to be compared and contrasted. This will allow for further refining of the factors in the chaining flux definition and it is conjectured that it also will demonstrate that the idea of the chaining flux can be extended to several different expert system representations.

By studying and comparing different knowledge systems, the knowledge modularization techniques will be refined along with the knowledge base complexity measures. Overall, a methodology for building knowledge systems can be created using the ideas presented in this paper to deliver expert

system applications that will be easier to understand, to develop, and to maintain.

REFERENCES

- [1] Berge, C. 1973. Graphs and Hypergraphs. Amsterdam, The Netherlands: North-Holland.
- [2] Bieman, J., and W. R. Edwards, Jr. June 1985. Modeling and Measuring Software Data Dependency Complexity. Ames: ISU Department of Computer Science TR #85-14.
- [3] Evangelist, W. M. 1983. Software Complexity Metric Sensitivity to Program Structuring Rules. The Journal of Systems and Software 3. 231-243.
- [4] Feigenbaum, E., and P. McCorduck. 1983. The Fifth Generation. Reading, MA: Addison-Wesley Publishing Co.
- [5] Gondran, M., and M. Minoux. 1984. Graphs and Algorithms. New York: John Wiley & Sons.
- [6] Harmon, P., and D. King. 1985. Expert Systems: Artificial Intelligence in Business. New York: John Wiley & Sons, Inc.
- [7] Held, J., and J. Carlis. 1989. Conceptual Data Modeling of Expert Systems. IEEE Expert. 4,1: 50-61.
- [8] Henry, S., and D. Kafura. 1981. Software Structure Metrics Based on Information Flow. IEEE Transactions on Software Engineering. 7,5: 510-518.
- [9] IEEE. 1983. IEEE Glossary of Software Engineering Terminology, IEEE Std. 729-1983. New York: IEEE.
- [10] Jacob, R., and J. Froscher. 1986. Software Engineering for Rule-Based Systems. Proceedings of the 1986 Fall Joint Computer Conference. 1986: 185-189.
- [11] Jacob, R., and J. Froscher. Naval Research Laboratory. December 17, 1986. Developing a Software Engineering Methodology for Knowledge-Based Systems. Washington D.C.: Naval Research Laboratory Report 9109.
- [12] Jacob, R. 1989. Private correspondence.

- [13] Kiernan G., Kolton, A., and E. Schwartz. 1988. Constructing an Expert System - Software Engineering of a Different Kind. Proceedings of the 1988 ACM Sixteenth Annual Computer Science Conference. 1988: 223-231.
- [14] Markusz, Z., and A. Kaposi. 1985. Complexity Control in Logic-Based Programming. The Computer Journal. 28,5: 487-495
- [15] Martin, J., and S. Oxman. 1988. Building Expert Systems: A Tutorial. Englewood Cliffs, NJ: Prentice-Hall.
- [16] O'Neal, M., and W. R. Edwards, Jr. 1988. Measuring and Controlling Complexity in Rule-Based Programs. Proceedings of the IASTED International Symposium EXPERT SYSTEMS. 1988: 6-9.
- [17] Parnas, D. L. 1972. A Technique for Software Module Specification with Examples. Communications of the ACM. 15,5: 330-336.
- [18] Parnas, D. L. 1972. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM. 15,12: 1053-1058.
- [19] Pedersen, K. 1989. Well-Structured Knowledge Bases. AI Expert. 4,4: 44-55.
- [20] Pedersen, K. 1989. Well-Structured Knowledge Bases, Part II. AI Expert. 4,7: 45-48.
- [21] Pedersen, K. 1989. Well-Structured Knowledge Bases, Part III. AI Expert. 4,11: 36-41.
- [22] Swigger, K., and R. Brazile. 1989. Experimental Comparison of Design/Documentation Formats for Expert Systems. International Journal of Man-Machine Studies. 31: 47-60.
- [23] Temperley, H.N.V. 1981. Graph Theory and Applications. Chichester, England: Ellis Horwood Limited.
- [24] Texas Instruments. 1988. AI Glossary. Notes of the Fourth Artificial Intelligence Satellite Symposium. 18-19.
- [25] Wiess, S., and C. Kulikowski. 1984. A Practical Guide to Designing Expert Systems. Tetowa, NJ: Rowan & Allanheld.

APPENDIXES

APPENDIX A

COMPUTER PROGRAM TO PARTITION
A RULE-BASED KNOWLEDGE BASE

```

/*****
/* Knowledge Partitioning Program */
/*****
/* Based upon the knowledge partitioning algorithm using the */
/* chaining flux metric. Also uses the intermediate knowledge */
/* representation of rule and variable tuples to make the program */
/* knowledge base independent. */
/*****
/* Input */
/* A rule/variable tuple of the form */
/* <source rule number> <target rule number> <variable number> */
/*****
/* Output */
/* The tuples are placed into a rule representation that is then */
/* partitioned according to the chaining flux metric. */
/*****

#include <stdio.h>

#define true 1
#define false 0

#define is ==
#define isnt !=
#define and &&
#define or ||
#define not !
#define begin {
#define end }

#define MAXVARS 500
#define MAXRULES 500
#define MAXGROUPS 500
#define MAXTUPLES 500

#define SOURCE_FOR 0
#define TARGET_FOR 1

struct infrastructure
begin
    int var[MAXVARS];
    int control;
    int chains_in;
    int chains_out;
    int no_vars;
    int flux;
    int group;
    int dependent[MAXRULES];
end;

struct rule_group
begin
    int control_rule;
    int rule[MAXRULES];
    int no_rules;
end;

struct tuple
begin
    int source_rule;
    int target_rule;
    int var_no;
end;

struct infrastructure Rule[MAXRULES];
struct rule_group Group[MAXGROUPS];
struct tuple Tuple[MAXTUPLES];

main ()
begin
    int numtuples, numrules, numgroups;

```

```

int avg_flux;

/* initialize the program */
initialize( &numtuples, &numrules, &numgroups );

/* load the tuple file into memory */
if ( ( numtuples = load_tuples() ) isnt 0 )
begin
/* put the tuple representation into a rule representation */
numrules = load_rules ( numtuples );
/* find the chaining flux for each rule */
find_flux ( numrules );
/* calculate the average chaining flux of the system */
avg_flux = average_flux ( numrules );
/* designate rules > average chaining flux as control rules */
find_control_rules ( numrules, &numgroups, avg_flux );
/* group the rules according to the control rules */
if ( numgroups > 0 )
group_rules ( &numgroups );
/* print the results according to the users choice */
menu ( numtuples, numrules, numgroups, avg_flux );
end
end

menu ( numtuples, numrules, numgroups, avg_flux )
int numtuples, numrules, numgroups, avg_flux;
begin
int i, done, choice, max;

done = false;
while ( not done )
begin
/* print the menu of choices for the user */
printf ( "\n" );
printf ( "***** MAIN MENU *****\n" );
printf ( " 1. Statistics\n" );
printf ( " 2. Show Rule\n" );
printf ( " 3. Show All Rules\n" );
printf ( " 4. Show Knowledge Module\n" );
printf ( " 5. Show All Knowledge Modules\n" );
printf ( " 6. Exit\n" );
printf ( "\n" );
printf ( " Choice ==> " );
scanf ( "%d", &choice );
switch ( choice )
begin
/* print the statistics of the entire system */
case 1: printf ( "\n" );
printf ( " Number of Tuples : %d\n", numtuples );
printf ( " Number of Rules : %d\n", numrules );
printf ( " Number of Partitions : %d\n", numgroups );
printf ( "Average Chaining Flux : %d\n", avg_flux );
for ( i = 1, max = 0 ; i <= numrules ; i++ )
if ( Rule[i].flux > max )
max = Rule[i].flux;
printf ( "Maximum Chaining Flux : %d\n", max );
break;

/* print the statistics of a given rule */
case 2: printf ( "\n" );
printf ( " Enter the Rule Number ==> " );
scanf ( "%d", &choice );
if ( choice >= 1 and choice <= numrules )
print_rule ( choice );
else
printf ( "*** Error...Invalid Rule Number ***\n" );
break;

/* print the statistics of all rules */
case 3: for ( i = 1 ; i <= numrules ; i++ )
print_rule ( i );

```

```

        break;

/* print the statistics of a given group */
case 4: printf ( "\n" );
        printf ( " Enter the Partition Number ==> " );
        scanf ( "%d", &choice );
        if ( choice >= 1 and choice <= numgroups )
            print_group ( choice );
        else
            printf ( "**** Error...Invalid Partition Number ****\n" );
        break;

/* print the statistics of all groups */
case 5: for ( i = 1 ; i <= numgroups ; i++ )
        print_group ( i );
        break;

case 6: done = true;
        break;

default: printf ( "**** Error...Invalid Menu Selection ****\n" );
        break;
    end
end
end

initialize ( numtuples, numrules, numgroups )
int *numtuples, *numrules, *numgroups;
begin
    int i;

    *numtuples = 0;
    *numrules = 0;
    *numgroups = 0;
    /* initialize the rule structures */
    for ( i = 0 ; i < MAXRULES ; i++ )
        begin
            Rule[i].chains_in = 0;
            Rule[i].chains_out = 0;
            Rule[i].flux = 0;
            Rule[i].no_vars = 0;
            Rule[i].group = 0;
            Rule[i].control = 0;
        end
    /* initialize the group structures */
    for ( i = 0 ; i < MAXGROUPS ; i++ )
        begin
            Group[i].control_rule = 0;
            Group[i].no_rules = 0;
        end
    /* initialize the tuple structures */
    for ( i = 0 ; i < MAXTUPLES ; i++ )
        begin
            Tuple[i].source_rule = 0;
            Tuple[i].target_rule = 0;
            Tuple[i].var_no = 0;
        end
    end

load_tuples ( )
begin
    int count, source, target, var;
    char filename[32];
    FILE *fopen(), *fp;

    /* get the file name that contains the tuples */
    printf ( "Enter the tuple file name ==> " );
    scanf ( "%s", filename );

    if ( (fp = fopen ( filename, "r" )) is NULL )
        begin

```

```

        printf ( "**** Error...Cannot open %s ****\n", filename );
        return ( 0 );
    end

    /* cycle thru the file reading in the tuples */
    count = 0;
    while ( fscanf ( fp,"%d %d %d", &source, &target, &var ) isnt EOF )
    begin
        if ( count < MAXTUPLES )
            begin
                Tuple[count].source_rule = source;
                Tuple[count].target_rule = target;
                Tuple[count].var_no = var;
                count++;
            end
        else
            printf ( "ERROR...Too many tuples\n" );
        end

        fclose ( fp );
        return ( count );
    end

load_rules ( numtuples )
int numtuples;
begin
    int i, j, k, n, source, target, found, count;

    /* loop through the tuples and put information into the rule structures */
    for ( i = 0 ; i < numtuples ; i++ )
        begin
            source = Tuple[i].source_rule;
            target = Tuple[i].target_rule;
            Rule[source].chains_out++;
            Rule[target].chains_in++;
            Rule[source].dependent[Rule[source].chains_out] = target;
            /* first is for source rule, second is for target rule */
            for ( k = 1 ; k <= 2 ; k++ )
                begin
                    if ( k is 1 )
                        n = source;
                    else
                        n = target;
                    /* see if variable is already on the rule's list */
                    j = 0;
                    found = false;
                    while ( not found and j < Rule[n].no_vars )
                        begin
                            if ( Rule[n].var[j] is Tuple[i].var_no )
                                found = true;
                            else
                                j++;
                        end
                    /* if not on rule's list, put the variable on the list */
                    if ( not found )
                        begin
                            Rule[n].var[j] = Tuple[i].var_no;
                            Rule[n].no_vars++;
                        end
                end
            end

            count = 1;
            while ( Rule[count].no_vars isnt 0 )
                count++;

            return ( count - 1 );
        end

end

flux ( n, in, out )
int n, in, out;

```



```

begin
  return ( n * in * in * out * out );
end

find_flux ( numrules )
int numrules;
begin
  int i;

  for ( i = 1 ; i <= numrules ; i++ )
    Rule[i].flux = flux ( Rule[i].no_vars, Rule[i].chains_in,
                        Rule[i].chains_out );
end

average_flux ( numrules )
int numrules;
begin
  int i, total;

  total = 0;
  for ( i = 1 ; i <= numrules ; i++ )
    total = total + Rule[i].flux;

  return ( total / numrules );
end

find_control_rules ( numrules, numgroups, avg )
int numrules, *numgroups, avg;
begin
  int i;

  for ( i = 1 ; i <= numrules ; i++ )
    if ( Rule[i].flux > avg )
      create_group ( i, numgroups );
end

group_rules ( numgroups )
int *numgroups;
begin
  int i, j, done;

  done = false;
  while ( not done )
    begin
      done = true;
      j = *numgroups;
      for ( i = 1 ; i <= j ; i++ )
        if ( get_dependents ( Group[i].control_rule, numgroups, true ) )
          done = false;
    end
end

get_dependents ( rulenum, numgroups, newgroup )
int rulenum, *numgroups, newgroup;
begin
  int i, j, n, assigned;

  /* loop through the target rules trying to assign them */
  /* to the knowledge module of the current marked rule */
  assigned = false;
  for ( i = 1 ; i <= Rule[rulenum].chains_out ; i++ )
    begin
      j = Rule[rulenum].dependent[i];
      n = Rule[rulenum].group;
      /* make sure the target rule is not a control rule */
      if ( not Rule[j].control )
        begin
          /* assign the target rule to the group if unaffiliated */
          if ( Rule[j].group is 0 )
            begin
              /* recursively try to assign all its */

```

```

        /* target rule's to its knowledge module */
        group_rule ( j, n, false );
        get_dependents ( j, numgroups, newgroup );
        assigned = true;
    end
    /* if the target rule belongs to another group, */
    /* remove it from that group and create another */
    /* group around the target rule */
    else if ( Rule[j].group isnt n )
    begin
        delete_rule ( j );
        if ( newgroup )
            create_group ( j, numgroups );
        else
            group_rule ( j, n, false );
            get_dependents ( j, numgroups, false );
            assigned = true;
        end
    end
    end
    return ( assigned );
end

create_group ( control_rule, numgroups )
int control_rule, *numgroups;
begin
    (*numgroups)++;
    group_rule ( control_rule, *numgroups, true );
end

group_rule ( rulenum, groupnum, control )
int rulenum, groupnum, control;
begin
    if ( control )
        Group[groupnum].control_rule = rulenum;
        Rule[rulenum].control = control;
        Rule[rulenum].group = groupnum;
        Group[groupnum].no_rules++;
        Group[groupnum].rule[Group[groupnum].no_rules] = rulenum;
    end

delete_rule ( rulenum )
int rulenum;
begin
    int i, j, oldgroup;

    /* take rule out of old group */
    oldgroup = Rule[rulenum].group;
    for ( i = 1 ; i <= Group[oldgroup].no_rules ; i++ )
        begin
            /* shift down by one */
            if ( Group[oldgroup].rule[i] is rulenum )
                begin
                    for ( j = i ; j < Group[oldgroup].no_rules ; j++ )
                        Group[oldgroup].rule[j] = Group[oldgroup].rule[j+1];
                    end
                end
            end
        Group[oldgroup].no_rules--;
    end

print_rule ( i )
int i;
begin
    int j;

    printf ( "\n" );
    printf ( "***** RULE %d *****\n", i );
    printf ( "    Variables :" );
    for ( j = 0 ; j < Rule[i].no_vars ; j++ )
        printf ( " %d ", Rule[i].var[j] );
    printf ( "\n" );

```

```

printf ( "   Number of Vars : %d \n", Rule[i].no_vars );
printf ( "           Fan In : %d \n", Rule[i].chains_in );
printf ( "           Fan Out : %d \n", Rule[i].chains_out );
printf ( "   Chaining Flux : %d \n", Rule[i].flux );
printf ( " Knowledge Module : %d \n", Rule[i].group );
printf ( " Dependent Rules :" );
for ( j = 1 ; j <= Rule[i].chains_out ; j++ )
    printf ( " %d ", Rule[i].dependent[j] );
printf ( "\n" );
end

print_group ( i )
int i;
begin
    int j;

    printf ( "\n" );
    printf ( "***** GROUP %d *****\n", i );
    printf ( "   Base Rule : %d \n", Group[i].control_rule );
    printf ( " Number of Rules : %d \n", Group[i].no_rules );
    printf ( "   Rules :" );
    for ( j = 1 ; j <= Group[i].no_rules ; j++ )
        printf ( " %d ", Group[i].rule[j] );
    printf ( "\n" );
end

```

APPENDIX B

SAMPLE INPUT FOR THE COMPUTER PROGRAM

0	1	1
1	2	2
1	3	3
1	4	4
1	5	4
1	6	4
1	7	5
1	16	5
1	17	5
1	81	5
1	33	5
1	106	5
1	113	5
1	114	5
1	115	5
2	9	6
7	8	80
8	96	7
8	10	8
8	11	9
8	12	10
8	13	11
8	14	11
8	15	11
9	10	8
16	18	6
16	19	7
16	20	8
16	37	9
17	93	10
17	19	7
18	21	11
19	22	12
19	23	13
19	24	14
19	25	15
19	26	16
19	27	17
19	28	18
19	29	19
19	30	20
21	31	21
25	47	22
25	121	23
27	34	24
27	36	25
30	32	26
31	92	27
33	34	24
33	28	18
33	29	19
33	30	20
33	35	28
33	25	15
33	36	25
33	37	9
33	20	8
34	38	29
34	39	30
34	40	31
34	41	32
34	37	9
39	42	33
39	43	34
39	44	34
40	42	33
40	43	34
40	44	34
41	42	33
41	43	34
41	44	34

43	45	35
45	46	36
47	48	37
47	49	38
47	50	39
47	51	40
47	52	41
47	53	41
47	54	41
47	55	41
47	56	41
47	57	41
47	58	42
47	59	42
47	60	42
47	61	42
47	62	42
47	63	42
47	64	42
47	65	42
47	66	42
47	67	42
47	68	42
47	69	42
47	70	42
47	71	42
47	72	42
47	73	43
47	74	44
47	75	45
49	76	46
51	77	47
51	78	48
72	79	49
73	80	50
81	82	51
81	83	52
81	92	27
81	93	10
81	84	53
83	85	57
84	86	54
84	87	55
84	88	56
86	89	58
86	90	59
89	91	60
92	94	61
92	95	62
93	94	61
93	95	62
94	96	7
94	97	63
94	98	64
95	96	7
95	97	63
95	100	65
95	101	66
95	102	67
95	103	68
95	104	68
95	105	68
97	99	69
106	107	70
107	108	71
107	109	72
108	110	73
109	111	74
110	111	74
111	112	75
113	117	76

113	118	76
113	116	77
117	119	78
117	120	79
118	119	78
118	120	79

APPENDIX C

SAMPLE OUTPUT FROM THE COMPUTER PROGRAM

Enter the tuple file name ==> tuples.dat

***** MAIN MENU *****

1. Statistics
2. See Rule
3. See All Rules
4. See Group
5. See All Groups
6. Exit

Choice ==> 1

Number of Tuples : 148
 Number of Rules : 121
 Number of Groups : 21
 Average Chaining Flux : 135
 Maximum Chaining Flux : 7840

***** MAIN MENU *****

1. Statistics
2. See Rule
3. See All Rules
4. See Group
5. See All Groups
6. Exit

Choice ==> 3

***** RULE 1 *****

Variables : 1 2 3 4 5
 Number of Vars : 5
 Chains In : 1
 Chains Out : 14
 Chaining Flux : 980
 Rule Group : 1
 Dependent Rules : 2 3 4 5 6 7 16 17 81 33 106 113 114 115

***** RULE 2 *****

Variables : 2 6
 Number of Vars : 2
 Chains In : 1
 Chains Out : 1
 Chaining Flux : 2
 Rule Group : 1
 Dependent Rules : 9

***** RULE 3 *****

Variables : 3
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 1
 Dependent Rules :

***** RULE 4 *****

Variables : 4
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 1
 Dependent Rules :

***** RULE 5 *****

Variables : 4
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 1

Dependent Rules :

***** RULE 6 *****

Variables : 4
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 1
Dependent Rules :

***** RULE 7 *****

Variables : 5 80
Number of Vars : 2
Chains In : 1
Chains Out : 1
Chaining Flux : 2
Rule Group : 1
Dependent Rules : 8

***** RULE 8 *****

Variables : 80 7 8 9 10 11
Number of Vars : 6
Chains In : 1
Chains Out : 7
Chaining Flux : 294
Rule Group : 2
Dependent Rules : 96 10 11 12 13 14 15

***** RULE 9 *****

Variables : 6 8
Number of Vars : 2
Chains In : 1
Chains Out : 1
Chaining Flux : 2
Rule Group : 1
Dependent Rules : 10

***** RULE 10 *****

Variables : 8
Number of Vars : 1
Chains In : 2
Chains Out : 0
Chaining Flux : 0
Rule Group : 10
Dependent Rules :

***** RULE 11 *****

Variables : 9
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 2
Dependent Rules :

***** RULE 12 *****

Variables : 10
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 2
Dependent Rules :

***** RULE 13 *****

Variables : 11
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0

Rule Group : 2
Dependent Rules :

***** RULE 14 *****

Variables : 11
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 2
Dependent Rules :

***** RULE 15 *****

Variables : 11
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 2
Dependent Rules :

***** RULE 16 *****

Variables : 5 6 7 8 9
Number of Vars : 5
Chains In : 1
Chains Out : 4
Chaining Flux : 80
Rule Group : 1
Dependent Rules : 18 19 20 37

***** RULE 17 *****

Variables : 5 10 7
Number of Vars : 3
Chains In : 1
Chains Out : 2
Chaining Flux : 12
Rule Group : 1
Dependent Rules : 93 19

***** RULE 18 *****

Variables : 6 11
Number of Vars : 2
Chains In : 1
Chains Out : 1
Chaining Flux : 2
Rule Group : 1
Dependent Rules : 21

***** RULE 19 *****

Variables : 7 12 13 14 15 16 17 18 19 20
Number of Vars : 10
Chains In : 2
Chains Out : 9
Chaining Flux : 3240
Rule Group : 3
Dependent Rules : 22 23 24 25 26 27 28 29 30

***** RULE 20 *****

Variables : 8
Number of Vars : 1
Chains In : 2
Chains Out : 0
Chaining Flux : 0
Rule Group : 17
Dependent Rules :

***** RULE 21 *****

Variables : 11 21
Number of Vars : 2
Chains In : 1
Chains Out : 1

Chaining Flux : 2
Rule Group : 1
Dependent Rules : 31

***** RULE 22 *****

Variables : 12
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 3
Dependent Rules :

***** RULE 23 *****

Variables : 13
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 3
Dependent Rules :

***** RULE 24 *****

Variables : 14
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 3
Dependent Rules :

***** RULE 25 *****

Variables : 15 22 23
Number of Vars : 3
Chains In : 2
Chains Out : 2
Chaining Flux : 48
Rule Group : 14
Dependent Rules : 47 121

***** RULE 26 *****

Variables : 16
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 3
Dependent Rules :

***** RULE 27 *****

Variables : 17 24 25
Number of Vars : 3
Chains In : 1
Chains Out : 2
Chaining Flux : 12
Rule Group : 3
Dependent Rules : 34 36

***** RULE 28 *****

Variables : 18
Number of Vars : 1
Chains In : 2
Chains Out : 0
Chaining Flux : 0
Rule Group : 11
Dependent Rules :

***** RULE 29 *****

Variables : 19
Number of Vars : 1
Chains In : 2

Chains Out : 0
 Chaining Flux : 0
 Rule Group : 12
 Dependent Rules :

***** RULE 30 *****

Variables : 20 26
 Number of Vars : 2
 Chains In : 2
 Chains Out : 1
 Chaining Flux : 8
 Rule Group : 13
 Dependent Rules : 32

***** RULE 31 *****

Variables : 21 27
 Number of Vars : 2
 Chains In : 1
 Chains Out : 1
 Chaining Flux : 2
 Rule Group : 1
 Dependent Rules : 92

***** RULE 32 *****

Variables : 26
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 13
 Dependent Rules :

***** RULE 33 *****

Variables : 5 24 18 19 20 28 15 25 9 8
 Number of Vars : 10
 Chains In : 1
 Chains Out : 9
 Chaining Flux : 810
 Rule Group : 4
 Dependent Rules : 34 28 29 30 35 25 36 37 20

***** RULE 34 *****

Variables : 24 29 30 31 32 9
 Number of Vars : 6
 Chains In : 2
 Chains Out : 5
 Chaining Flux : 600
 Rule Group : 5
 Dependent Rules : 38 39 40 41 37

***** RULE 35 *****

Variables : 28
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 4
 Dependent Rules :

***** RULE 36 *****

Variables : 25
 Number of Vars : 1
 Chains In : 2
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 15
 Dependent Rules :

***** RULE 37 *****

Variables : 9
 Number of Vars : 1

Chains In : 3
 Chains Out : 0
 Chaining Flux : 0'
 Rule Group : 16
 Dependent Rules :

***** RULE 38 *****

Variables : 29
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 5
 Dependent Rules :

***** RULE 39 *****

Variables : 30 33 34
 Number of Vars : 3
 Chains In : 1
 Chains Out : 3
 Chaining Flux : 27
 Rule Group : 5
 Dependent Rules : 42 43 44

***** RULE 40 *****

Variables : 31 33 34
 Number of Vars : 3
 Chains In : 1
 Chains Out : 3
 Chaining Flux : 27
 Rule Group : 5
 Dependent Rules : 42 43 44

***** RULE 41 *****

Variables : 32 33 34
 Number of Vars : 3
 Chains In : 1
 Chains Out : 3
 Chaining Flux : 27
 Rule Group : 5
 Dependent Rules : 42 43 44

***** RULE 42 *****

Variables : 33
 Number of Vars : 1
 Chains In : 3
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 5
 Dependent Rules :

***** RULE 43 *****

Variables : 34 35
 Number of Vars : 2
 Chains In : 3
 Chains Out : 1
 Chaining Flux : 18
 Rule Group : 5
 Dependent Rules : 45

***** RULE 44 *****

Variables : 34
 Number of Vars : 1
 Chains In : 3
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 5
 Dependent Rules :

***** RULE 45 *****

Variables : 35 36

Number of Vars : 2
 Chains In : 1
 Chains Out : 1
 Chaining Flux : 2
 Rule Group : 5
 Dependent Rules : 46

***** RULE 46 *****

Variables : 36
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 5
 Dependent Rules :

***** RULE 47 *****

Variables : 22 37 38 39 40 41 42 43 44 45
 Number of Vars : 10
 Chains In : 1
 Chains Out : 28
 Chaining Flux : 7840
 Rule Group : 6
 Dependent Rules : 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
 68 69 70 71 72 73 74 75

***** RULE 48 *****

Variables : 37
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 6
 Dependent Rules :

***** RULE 49 *****

Variables : 38 46
 Number of Vars : 2
 Chains In : 1
 Chains Out : 1
 Chaining Flux : 2
 Rule Group : 6
 Dependent Rules : 76

***** RULE 50 *****

Variables : 39
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 6
 Dependent Rules :

***** RULE 51 *****

Variables : 40 47 48
 Number of Vars : 3
 Chains In : 1
 Chains Out : 2
 Chaining Flux : 12
 Rule Group : 6
 Dependent Rules : 77 78

***** RULE 52 *****

Variables : 41
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 6
 Dependent Rules :

***** RULE 53 *****

Variables : 41
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 54 *****

Variables : 41
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 55 *****

Variables : 41
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 56 *****

Variables : 41
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 57 *****

Variables : 41
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 58 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 59 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 60 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 61 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 62 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 63 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 64 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 65 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 66 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 67 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 68 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6

Dependent Rules :

***** RULE 69 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 70 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 71 *****

Variables : 42
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 72 *****

Variables : 42 49
Number of Vars : 2
Chains In : 1
Chains Out : 1
Chaining Flux : 2
Rule Group : 6
Dependent Rules : 79

***** RULE 73 *****

Variables : 43 50
Number of Vars : 2
Chains In : 1
Chains Out : 1
Chaining Flux : 2
Rule Group : 6
Dependent Rules : 80

***** RULE 74 *****

Variables : 44
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 75 *****

Variables : 45
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 76 *****

Variables : 46
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0

Rule Group : 6
Dependent Rules :

***** RULE 77 *****

Variables : 47
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 78 *****

Variables : 48
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 79 *****

Variables : 49
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 80 *****

Variables : 50
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 6
Dependent Rules :

***** RULE 81 *****

Variables : 5 51 52 27 10 53
Number of Vars : 6
Chains In : 1
Chains Out : 5
Chaining Flux : 150
Rule Group : 7
Dependent Rules : 82 83 92 93 84

***** RULE 82 *****

Variables : 51
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 7
Dependent Rules :

***** RULE 83 *****

Variables : 52 57
Number of Vars : 2
Chains In : 1
Chains Out : 1
Chaining Flux : 2
Rule Group : 7
Dependent Rules : 85

***** RULE 84 *****

Variables : 53 54 55 56
Number of Vars : 4
Chains In : 1
Chains Out : 3

Chaining Flux : 36
Rule Group : 7
Dependent Rules : 86 87 88

***** RULE 85 *****

Variables : 57
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 7
Dependent Rules :

***** RULE 86 *****

Variables : 54 58 59
Number of Vars : 3
Chains In : 1
Chains Out : 2
Chaining Flux : 12
Rule Group : 7
Dependent Rules : 89 90

***** RULE 87 *****

Variables : 55
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 7
Dependent Rules :

***** RULE 88 *****

Variables : 56
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 7
Dependent Rules :

***** RULE 89 *****

Variables : 58 60
Number of Vars : 2
Chains In : 1
Chains Out : 1
Chaining Flux : 2
Rule Group : 7
Dependent Rules : 91

***** RULE 90 *****

Variables : 59
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 7
Dependent Rules :

***** RULE 91 *****

Variables : 60
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 7
Dependent Rules :

***** RULE 92 *****

Variables : 27 61 62
Number of Vars : 3
Chains In : 2

Chains Out : 2
 Chaining Flux : 48
 Rule Group : 18
 Dependent Rules : 94 95

***** RULE 93 *****
 Variables : 10 61 62
 Number of Vars : 3
 Chains In : 2
 Chains Out : 2
 Chaining Flux : 48
 Rule Group : 19
 Dependent Rules : 94 95

***** RULE 94 *****
 Variables : 61 7 63 64
 Number of Vars : 4
 Chains In : 2
 Chains Out : 3
 Chaining Flux : 144
 Rule Group : 8
 Dependent Rules : 96 97 98

***** RULE 95 *****
 Variables : 62 7 63 65 66 67 68
 Number of Vars : 7
 Chains In : 2
 Chains Out : 8
 Chaining Flux : 1792
 Rule Group : 9
 Dependent Rules : 96 97 100 101 102 103 104 105

***** RULE 96 *****
 Variables : 7
 Number of Vars : 1
 Chains In : 3
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 20
 Dependent Rules :

***** RULE 97 *****
 Variables : 63 69
 Number of Vars : 2
 Chains In : 2
 Chains Out : 1
 Chaining Flux : 8
 Rule Group : 21
 Dependent Rules : 99

***** RULE 98 *****
 Variables : 64
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 8
 Dependent Rules :

***** RULE 99 *****
 Variables : 69
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 21
 Dependent Rules :

***** RULE 100 *****
 Variables : 65
 Number of Vars : 1

```
    Chains In : 1
    Chains Out : 0
    Chaining Flux : 0
    Rule Group : 9
    Dependent Rules :

***** RULE 101 *****
    Variables : 66
    Number of Vars : 1
    Chains In : 1
    Chains Out : 0
    Chaining Flux : 0
    Rule Group : 9
    Dependent Rules :

***** RULE 102 *****
    Variables : 67
    Number of Vars : 1
    Chains In : 1
    Chains Out : 0
    Chaining Flux : 0
    Rule Group : 9
    Dependent Rules :

***** RULE 103 *****
    Variables : 68
    Number of Vars : 1
    Chains In : 1
    Chains Out : 0
    Chaining Flux : 0
    Rule Group : 9
    Dependent Rules :

***** RULE 104 *****
    Variables : 68
    Number of Vars : 1
    Chains In : 1
    Chains Out : 0
    Chaining Flux : 0
    Rule Group : 9
    Dependent Rules :

***** RULE 105 *****
    Variables : 68
    Number of Vars : 1
    Chains In : 1
    Chains Out : 0
    Chaining Flux : 0
    Rule Group : 9
    Dependent Rules :

***** RULE 106 *****
    Variables : 5 70
    Number of Vars : 2
    Chains In : 1
    Chains Out : 1
    Chaining Flux : 2
    Rule Group : 1
    Dependent Rules : 107

***** RULE 107 *****
    Variables : 70 71 72
    Number of Vars : 3
    Chains In : 1
    Chains Out : 2
    Chaining Flux : 12
    Rule Group : 1
    Dependent Rules : 108 109

***** RULE 108 *****
    Variables : 71 73
```

Number of Vars : 2
Chains In : 1
Chains Out : 1
Chaining Flux : 2
Rule Group : 1
Dependent Rules : 110

***** RULE 109 *****
Variables : 72 74
Number of Vars : 2
Chains In : 1
Chains Out : 1
Chaining Flux : 2
Rule Group : 1
Dependent Rules : 111

***** RULE 110 *****
Variables : 73 74
Number of Vars : 2
Chains In : 1
Chains Out : 1
Chaining Flux : 2
Rule Group : 1
Dependent Rules : 111

***** RULE 111 *****
Variables : 74 75
Number of Vars : 2
Chains In : 2
Chains Out : 1
Chaining Flux : 8
Rule Group : 1
Dependent Rules : 112

***** RULE 112 *****
Variables : 75
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 1
Dependent Rules :

***** RULE 113 *****
Variables : 5 76 77
Number of Vars : 3
Chains In : 1
Chains Out : 3
Chaining Flux : 27
Rule Group : 1
Dependent Rules : 117 118 116

***** RULE 114 *****
Variables : 5
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 1
Dependent Rules :

***** RULE 115 *****
Variables : 5
Number of Vars : 1
Chains In : 1
Chains Out : 0
Chaining Flux : 0
Rule Group : 1
Dependent Rules :

***** RULE 116 *****

Variables : 77
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 1
 Dependent Rules :

***** RULE 117 *****

Variables : 76 78 79
 Number of Vars : 3
 Chains In : 1
 Chains Out : 2
 Chaining Flux : 12
 Rule Group : 1
 Dependent Rules : 119 120

***** RULE 118 *****

Variables : 76 78 79
 Number of Vars : 3
 Chains In : 1
 Chains Out : 2
 Chaining Flux : 12
 Rule Group : 1
 Dependent Rules : 119 120

***** RULE 119 *****

Variables : 78
 Number of Vars : 1
 Chains In : 2
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 1
 Dependent Rules :

***** RULE 120 *****

Variables : 79
 Number of Vars : 1
 Chains In : 2
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 1
 Dependent Rules :

***** RULE 121 *****

Variables : 23
 Number of Vars : 1
 Chains In : 1
 Chains Out : 0
 Chaining Flux : 0
 Rule Group : 14
 Dependent Rules :

***** MAIN MENU *****

1. Statistics
2. See Rule
3. See All Rules
4. See Group
5. See All Groups
6. Exit

Choice ==> 5

***** GROUP 1 *****

Control Rule : 1
 Number of Rules : 28
 Rules : 1 2 9 3 4 5 6 7 16 18 21 31 17 106 107 108 110 111 112 109
 113 117 119 120 118 116 114 115

***** GROUP 2 *****

Control Rule : 8


```

Number of Rules : 6
Rules : 8 11 12 13 14 15

***** GROUP 3 *****
Control Rule : 19
Number of Rules : 6
Rules : 19 22 23 24 26 27

***** GROUP 4 *****
Control Rule : 33
Number of Rules : 2
Rules : 33 35

***** GROUP 5 *****
Control Rule : 34
Number of Rules : 10
Rules : 34 38 39 42 43 45 46 44 40 41

***** GROUP 6 *****
Control Rule : 47
Number of Rules : 34
Rules : 47 48 49 76 50 51 77 78 52 53 54 55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72 79 73 80 74 75

***** GROUP 7 *****
Control Rule : 81
Number of Rules : 11
Rules : 81 82 83 85 84 86 89 91 90 87 88

***** GROUP 8 *****
Control Rule : 94
Number of Rules : 2
Rules : 94 98

***** GROUP 9 *****
Control Rule : 95
Number of Rules : 7
Rules : 95 100 101 102 103 104 105

***** GROUP 10 *****
Control Rule : 10
Number of Rules : 1
Rules : 10

***** GROUP 11 *****
Control Rule : 28
Number of Rules : 1
Rules : 28

***** GROUP 12 *****
Control Rule : 29
Number of Rules : 1
Rules : 29

***** GROUP 13 *****
Control Rule : 30
Number of Rules : 2
Rules : 30 32

***** GROUP 14 *****
Control Rule : 25
Number of Rules : 2
Rules : 25 121

***** GROUP 15 *****
Control Rule : 36
Number of Rules : 1
Rules : 36

***** GROUP 16 *****
Control Rule : 37

```

Number of Rules : 1
Rules : 37

***** GROUP 17 *****

Control Rule : 20
Number of Rules : 1
Rules : 20

***** GROUP 18 *****

Control Rule : 92
Number of Rules : 1
Rules : 92

***** GROUP 19 *****

Control Rule : 93
Number of Rules : 1
Rules : 93

***** GROUP 20 *****

Control Rule : 96
Number of Rules : 1
Rules : 96

***** GROUP 21 *****

Control Rule : 97
Number of Rules : 2
Rules : 97 99

***** MAIN MENU *****

1. Statistics
2. See Rule
3. See All Rules
4. See Group
5. See All Groups
6. Exit

Choice ==> 6

APPENDIX D

USERS' GUIDE FOR THE COMPUTER PROGRAM

Users' Guide for the Program

To start the program, enter the command 'flow' at the unix prompt. The program will first ask you name of the input file. The input file contains the intermediate knowledge representation tuples describing the knowledge base. An intermediate knowledge representation tuple for this program is the form

< Source Rule Number, Target Rule Number, Variable Number>.

Once the correct file name is entered, the program will calculate the chaining flux of each rule, determine the control rules of the knowledge base, and then partition the knowledge base into knowledge modules based upon the control rules.

After the knowledge base is partitioned, the program will then display the main menu and a prompt for your selection.

```
***** MAIN MENU *****  
1. Statistics  
2. Show Rule  
3. Show All Rules  
4. Show Knowledge Module  
5. Show All Knowledge Modules  
6. Exit  
Choice ==> _
```

The first choice, statistics, shows the statistics of the entire knowledge system. These include the number of tuples read from the input file, the number of rules in the knowledge base, the number of partitions, the average chaining flux of the knowledge base, and the maximum chaining flux of the knowledge base.

The second choice, show a rule, displays information about a selected rule. The information printed includes the variables referenced in the rule, the fan-in of the rule, the fan-out of the rule, the chaining flux of the rule, the knowledge module to which the rule belongs, and the rules which are directly dependent upon the selected rule.

The third option, show all rules, displays the above information for each rule in the knowledge base.

The fourth choice, show a knowledge module, displays information about a partition of the knowledge base. The information displayed includes the base rule of the knowledge modules, the number of rules in the knowledge modules, and the rules assigned to the knowledge module.

The fifth choice, show all knowledge modules, displays the above information for each partition of the knowledge system.

The sixth and final choice, exit, halts execution of the program and returns you back to the unix prompt.

VITA

Steven Bruce Cudd

Candidate for the Degree of
Master of Science

Thesis: ADAPTING SOFTWARE ENGINEERING PRINCIPLES TO
DIAGRAM, MODULARIZE, AND ANALYZE RULE-BASED
EXPERT SYSTEMS

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Warrensburg, Missouri, August
5, 1965, son of George S. and Barabara J. Cudd.

Education: Graduated from Minco High School, Minco,
Oklahoma, in May 1983; received Bachelor of Science
Degree with a double major in Mathematics and Com-
puter Science from Oklahoma State University in May
1987; completed requirements for the Master of Sci-
ence degree at Oklahoma State University in May
1990.

Professional Experience: Senior Programmer Analyst,
MPSI Systems Inc., June 1987 to July 1989; Senior
Systems Analyst, MPSI Software, July 1989 to pres-
ent.