1994

A Directed Hypergraph Approach for the Verification of Rule-Based Expert Systems.

Mysore Ramaswamy
Louisiana State University and Agricultural & Mechanical College

A DIRECTED HYPERGRAPH APPROACH FOR THE VERIFICATION OF RULE-BASED EXPERT SYSTEMS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Interdepartmental Program in Business Administration

by
Mysore Ramaswamy
B.E., Bangalore University, 1970
M.S., University of Southwestern Louisiana, 1988
December 1994

ACKNOWLEDGEMENTS

I would like to express my appreciation and gratitude to my advisors Dr. Sumit Sarkar and Dr. Ye-Sho Chen for their guidance and support throughout this research which made all this possible. I wish to express my sincere thanks to Dr. Deb Ghosh, Dr. Kwei Tang, Dr. Jianhua Chen, and Dr. E. I. Meletis, members of my dissertation committee, for their helpful suggestions.

I would also like to thank my wife Vatsala for her support over the years and assistance during this period. Thanks are also due to little Omar for putting up with me during this long process.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Rule-based representation techniques have become popular for storing and manipulation of domain knowledge in expert systems. It is important that systems using such a representation are verified for accuracy before implementation. In recent years, graphical techniques have been found to provide a good framework for the detection of errors that may appear in a rule base. In this dissertation, we develop a technique that uses a *directed hypergraph* to *accurately* detect all the different types of errors that appear in a rule base. This technique overcomes limitations of existing graphical techniques that are unable to accurately detect all the errors that appear in a rule base, without misdiagnosing error-free instances. The directed hypergraph technique allows rules to be represented in a manner that clearly identifies complex dependencies across compound clauses in the rule base. Since connectivity across compound clauses are accurately represented, the verification procedure can detect errors in an accurate fashion. We have developed a verification procedure that uses the adjacency matrix of the directed hypergraph. The procedure detects different types of errors by using simple operations on the adjacency matrix.

In practice, expert systems are often used to make inferences based on multiple observed facts. Most existing techniques have ignored this aspect, since the selection of valid combinations of rule antecedents from a large number of rule antecedents to be considered is difficult. To address this issue, the directed hypergraph technique has been extended to perform verification checks when sets of feasible multiple assertions are made available to the system. As the size of the rule base increases, execution of the algorithm can be hard due to storage and computational considerations. It has been empirically found that sets of rules in large rule bases are sufficiently separated to allow decomposition into

smaller sets. The directed hypergraph technique has been enhanced to accurately detect all errors in large rule bases by performing verification checks over the smaller groups of rules separately, and propagating the results from one group to other linked groups.

CHAPTER 1
INTRODUCTION

Knowledge acquisition is one of the most important, yet least formalized steps in expert system development. It has been widely acknowledged as a major bottleneck in the development of expert system (Agarwal and Tanniru, 1992a; Bundy, 1988; Geissman and Schultz, 1988; Hayes-Roth, 1985; Nazareth, 1989; Zahedi, 1993). This is because obtaining a complete and consistent specification of relevant domain knowledge for an application is a hard problem. Knowledge verification procedures can significantly alleviate the above problem by detecting the errors that may exist in knowledge that is obtained from experts.

Rule-based representation techniques have become one of the most popular techniques for storing and manipulation of domain knowledge in expert systems. Two important reasons for this popularity of rule-based expert systems are (1) the modularity of the rule-based framework, and (2) the ability to use knowledge stored as rules in a non-procedural manner (Davis, 1987; Hayes-Roth, 1985). Unfortunately, these features of rule-based knowledge representation also render the reasoning process to be largely invisible to developers and users (Agarwal and Tanniru, 1992). There is a strong potential for errors during the elicitation of the rules from domain experts. Experts often think intuitively and consequently omit many steps in the reasoning process. In addition, expert systems are usually developed over a period of time in an incremental manner which often leads to inconsistent specifications. This can be further magnified when multiple experts provide input to a system. As a result, it is very important that such a system is verified for correctness before it is used in a commercial environment. The verification task requires

checking that the rules in the system are consistent and complete. These checks detect superfluous, incorrect or missing rules which affect the performance of the expert system and have been termed as *structural errors* (Nazareth, 1989).

1.1 Common Errors in Rule Bases

There are many types of structural errors that may exist in rules obtained from experts. One common error is when a collection of rules leads to contradictory conclusions. For example, consider the following three rules from an R&D Project Evaluation rule base:

Rule 1: If likelihood of commercial success is low
Then financial viability is poor.

Rule 2: If financial viability is poor
Then likelihood of raising a loan is low.

Rule 3: If likelihood of commercial success is low
Then likelihood of raising a loan is high.

The first two rules taken together indicate that the likelihood of raising a loan is low when the likelihood of commercial success of the project is low. This contradicts the third rule which states that he likelihood of raising a loan is high when the likelihood of commercial success of the project is low. In this example, it is easy to see the contradiction across these rules. However, in real applications consisting of hundreds of rules, detection of contradiction among rules is difficult.

Another common problem is when a rule, while not contradicting other rules, does not provide any additional knowledge about the application. Such rules are redundant and reduce the efficiency of the system, since more rules than required must be processed to

answer queries. For instance, in the example considered earlier, suppose the first two rules are retained and the third rule is modified as follows:

Rule 3. If likelihood of commercial success is low

Then likelihood of raising a loan is low.

In this rule base, Rule 3 conveys the same information which is conveyed by the first two rules, and thus renders the first two rules redundant.

More importantly, redundant rules cause problems when rules are revised. Suppose Rule 3 is revised to "If likelihood of commercial success is low, Then

likelihood of raising a loan is high." After the revision, if the first two rules are still in the rule base, it leads to contradiction. Thus, even when reduced efficiency is not a major consideration, a rule base should be free of redundant rules.

In complex applications with large number of attributes, it is hard at times to determine the causal dependencies among them. Different sets of rules may indicate different causal dependencies. These sets of rules taken together, may lead to cyclic dependencies. Such cyclic dependencies must be avoided since these can cause a system to go into an infinite loop when being used for consultation.

Specification of decision rules elicited from experts can at times be incomplete, especially in large rule bases. This is due to some rules not being spelt out as they are considered "obvious" by the expert. Also, when multiple experts contribute to the rule base, rules that link different portions of the rule base can be missed. If the rule base is not complete, some queries will not get answered, since many intermediate rules may need to be processed to reach a final conclusion. Therefore it is necessary to ensure that the rule base is complete.

1.2 Limitations of Existing Approaches

A number of different techniques have been developed to perform verification checks for rule bases (Agarwal and Tanniru, 1992; Cragun and Steudel, 1987; Ginsberg, 1987; Marek, 1987; Nazareeth, 1993; Nazareth and Kennedy, 1991; Nguyen et al, 1987; Suwa et al, 1982). Unfortunately, as noted by Nazareth (1989), this work has been fragmentary in nature, with the scope varying considerably across different techniques. Earlier work in this area focussed on those errors that could be detected by the pairwise checking of rules (Cragun and Steudel, 1987; Nguyen et al, 1987; Suwa et al, 1982). Recent techniques have enhanced the scope to include errors that occur over longer chains of rules (Agarwal and Tanniru, 1992; Ginsberg, 1988; Nazareth, 1993; Nazareth and Kennedy, 1991). Many of the newer techniques use graphical representations of the rule base in order to detect the different errors that may appear in the system. Topological properties of graphical structures are used to deduce dependencies across propositions. These graphical representations allow the verification problem to be reformulated as one of reachability of specific states in the graph.

Graphical techniques are attractive since graphs provide an easy-to-use framework to represent conceptual dependencies. Analytical techniques to detect connectivity in graphs allow the verification of rule-bases to be performed in a rigorous manner. A directed graph approach to represent a rule base for the verification problem was presented in Nazareth and Kennedy (1991). Petri nets have also been used to model this problem (Agarwal and Tanniru, 1992; Nazareth, 1993). However, there are some important drawbacks with these graphical techniques. These drawbacks are related to the fact that a rule often indicates a dependency from a compound clause to a simple clause. The existing techniques use nodes that correspond only to simple clauses. As a result, dependencies involving compound

clauses are artificially incorporated into the graphical representation. This is accomplished by incorporating additional nodes that correspond to rules in the directed graph approach, and by the use of transitions in the Petri net based approaches. The suggested procedures to detect errors are not accurate. They either fail to detect some of the common errors, or indicate errors in instances which are actually error-free. For instance, the directed graph approach indicates spurious errors whenever rules have compound antecedent clauses that overlap (Nazareth and Kennedy, 1991). As noted by the author, the Petri net approach in Nazareth (1993) has the same limitation. The Petri net approach presented in Agarwal and Tanniru (1992) is unable to detect errors which are caused by multiple inference paths of different lengths.

1.3 Contribution of This Research

In this dissertation, we present a new graphical technique to verify rule bases that overcomes the limitations of existing techniques. We use *directed hypergraphs* to model the dependency across attribute-values as indicated by the rules. The directed hypergraph allows compound antecedents of rules to be modeled as compound nodes. This allows rules to be graphically represented in a manner that clearly identifies complex dependencies across compound clauses in the rule base. Since connectivity across compound clauses are *precisely represented*, the verification procedure can detect errors in an accurate fashion. Analogous to the other graphical techniques, we have developed a verification procedure that uses the adjacency matrix of the directed hypergraph. The procedure detects different types of errors by using simple operations on the adjacency matrix. The technique has been implemented, and shown to have a computational complexity that is comparable to that of the other graphical techniques.

In practice, expert systems are often used to make inferences based on multiple observed facts (which are termed as assertions). This renders verification of rule bases hard for the following reasons. First, the number of different combinations of rule antecedents that have to be considered can grow exponentially in large rule bases. In addition, among the numerous combinations, selection of valid combinations is difficult. For these reasons, most existing techniques have ignored this aspect of the verification problem. In order to address this issue, the directed hypergraph technique has been extended to perform verification checks when sets of feasible multiple assertions are made available to the system. It is shown that if the feasible sets of assertions are indicated beforehand, the verification process can identify errors with very little increase in computational effort.

Although the verification algorithm is of polynomial complexity, as the size of the rule base increases, the matrix manipulation operations can be hard to perform (primarily due to storage considerations). However, it has been empirically noted that sets of rules in large rule bases are usually sufficiently separated that the rule base can be decomposed into smaller sets (Jacob and Froscher, 1986). Rules in these smaller sets are usually intertwined in complex ways. We have enhanced the hypergraph based verification procedure to detect errors in large rule bases which can be decomposed into smaller groups. The technique is shown to accurately detect all errors in the large rule base by performing verification checks over the smaller groups of rules separately, and propagating the results from one group to other linked groups. The procedure significantly reduces the storage and computation requirements associated with verifying the large set of rules simultaneously.

The remainder of this dissertation is organized as follows. In Chapter 2, we discuss the different kinds of structural errors that may occur in a rule base, followed by a brief

literature review. In Chapter 3, we present the directed hypergraph representation for a rule base, and define the various notations that are used with this representation. We then define the adjacency matrix of a hypergraph, and discuss some special matrix operations that are used to detect errors. The verification algorithm, its proof, and computational complexity are also presented in this chapter. The verification procedure is demonstrated with an example in Chapter 4. In Chapter 5, we present an extension of the verification algorithm to cover cases where antecedents of several rules are instantiated simultaneously. Decomposition issues are discussed in Chapter 6. Concluding remarks are presented in Chapter 7. The code of the Program Hypergraph based on the verification algorithm appears in the Appendix.

CHAPTER 2
CLASSIFICATION OF ERRORS

In this chapter, we provide an overview of the type of errors that may exist in large rule bases. Next, we review current literature that addresses this problem. We assume that all rules are in *unitized form* before they are used to detect errors (Pederson, 1989). This implies that compound antecedent clauses in rules only allow conjunctions, and only simple clauses are allowed as conclusions. Each simple clause corresponds to an attribute-value pair. The above requirements ensure a structured rule base, with no loss of generality.

Our verification algorithm focuses on the structural (or logical) errors that arise from the interaction of the rules with each other. Nazareth (1989) has provided a taxonomy of these structural errors, which consist of *redundancy*, *conflict*, *circularity* and *incompleteness* (*deadend premises* and *unreachable goals*). In order to check for incompleteness, we assume that each attribute-value pair belongs to one of three categories. An attribute-value pair is an *input variable* if it is provided by the user during runtime. *Goal variables* are those that refer to the final desired recommendations of the expert system. All other variables are used to provide inference paths from the input variables to the goal variables, and are called *intermediate variables*.

2.1 Errors Affecting Consistency

Among researchers, there is no unanimity regarding errors that cause inconsistency. Redundancy, contradiction, and circularity are identified as the common errors that causes inconsistency in rule bases [Nazareth (1989); Marek (1987); and Preece (1993)]. Ignizio (1991) considers subsumption and unnecessary premises as additional errors that cause

inconsistency. Synonyms are to be considered as a cause of inconsistency, especially when knowledge is elicited from multiple experts (Trice, 1993). Illegal attribute values are considered as an error which causes incompleteness [Ignizio (1991) and Nguyen et al (1987)]. But it is more appropriate to treat illegal attribute values as an error that causes inconsistency.

We classify errors that cause inconsistency as follows:

(a) Synonyms,

(b) Redundancy,

(c) Contradiction,

(d) Circularity, and

(e) Illegal Attribute Values.

2.1.1 Synonyms

In a rule base, different variable names might actually refer to the same attribute. These are referred to as synonyms. In particular, when knowledge bases are developed by gathering expert information from different sources, synonyms are likely to be present. Also, when a rule base is incrementally developed, an expert may refer to the same attribute by different names at different times. It is necessary to eliminate synonyms because a rule base with synonyms may not lead to desired inferences.

Synonyms can also hinder detection of other errors that cause inconsistency and incompleteness. For example consider the following rules from an R&D Project Evaluation rule base:

Rule 1:	IF	Consumer Preference is high
	THEN	Probability of Commercial Success is high.

Rule 2: IF Market Acceptability is high

THEN Probability of Commercial Success is low.

In the above case, if the attributes "Consumer Preference" and "Market Accepability" refer to the same factor, then these two rules contradict each other. But if the synonyms are not identified, the two rules seem to be consistent. Therefore, it is necessary to remove all synonyms before other checks for verification are performed.

2.1.2 Redundancy

There are two types of redundancies that can occur in rule bases, viz., (a) Redundancy due to combination of rules; and (b) Redundancy due to subsumption. The first type of redundancy occurs when a single rule can lead to an inference which is conveyed by another set of rules. For example, consider the following three rules:

Rule 1: If probability of commercial success is low

Then financial viability is poor.

Rule 2: If financial viability is poor

Then probability of raising a loan is low.

Rule 3: If probability of commercial success is low

Then probability of raising a loan is low.

In the above case, Rule 3 leads to the same conclusion as that derived from the first two rules. Thus, if the purpose of including Rules 1 and 2 is to deduce whether the likelihood of raising a loan is low or not when the likelihood of commercial success is low, then the first two rules are redundant. Alternately, if Rules 1 and 2 are required for other inference paths, then Rule 3 is redundant. We note that while it is possible to identify multiple paths from one clause to another, such a condition does not necessarily indicate

redundant rules. As pointed out in (Agarwal and Tanniru, 1992), if the intermediate clauses in an inference path are required to establish paths across some other clauses, then there may not exist any redundant rules even when there are multiple paths between two sets of clauses. Thus, we identify multiple paths as *potentially* implying redundancy.

The second type of redundancy is caused when a more general rule makes a restrictive rule unnecessary. This is termed as subsumption. Rules which subsume other rules get included in the rule base either due to a more general rule being included during the knowledge elicitation stage or due to changed problem environment.

For example, consider the following rule:

Rule 1: If probability of commercial success is low, and
anticipated growth rate is low, and
capital requirement is high,
Then financial viability is poor.

Suppose the expert comes up with following rule later on:

Rule 2: If probability of commercial success is low, and
capital requirement is high,
Then financial viability is poor.

Now, the more general latter rule subsumes the former rule.

2.1.3 Contradiction or Conflict

Contradiction among rules can appear either due to (1) the mutually exclusive conclusions being ignored during the knowledge elicitation stage or (2) a rule being mis-stated. Consider the following three rules:

Rule 1: If the probability of commercial success is "medium",
Then project can be funded for two years.

Rule 2: If project can be funded for two years,
Then continue the project.

Rule 3: If the probability of commercial success is "medium",
Then discontinue the project.

The first two rules taken together indicate that the project is to be continued if the probability of commercial success is "medium". This contradicts the third rule which states that the project has to be discontinued if the probability of commercial success is "medium". The contradiction existing across the above three rules is easy to detect. However, in large rule bases, inferences are often made after processing several rules in sequence and it is difficult to detect the inconsistency.

2.1.4 Circularity

Circularity among rules can occur due to difficulty in determining the causal dependencies among variables. It is often difficult to determine the dependencies between two variables when both of them have been caused by a third variable. For example consider the following rules:

Rule 1: If disposition of major consumer groups is encouraging
Then commercial viability is optimistic.

Rule 2: If commercial viability is optimistic
Then disposition of major consumer groups is encouraging.

An extraneous factor such as interest rate can affect both "disposition of major consumer groups" and "commercial viability." Therefore both the above rules may be included in the

rule base due to lack of clear information regarding causal dependencies between the two variables. Such circular rules must be removed from the rule base as they can cause the system to go into an infinite loop.

2.1.5 Illegal Attribute Values

This is an instance of a rule containing an attribute-value pair which is not valid or legal. For example, consider the following rule:

Rule1: If Anticipated Return on Investment is 12%
Then Abandon the Project.

If the attribute "Anticipated Return on Investment" can take on values "High", "Moderate", or "Low", then the above is an instance of an illegal attribute value.

In the existing literature - for example, Ignizio (1991), Chung (1989), Nguyen et al (1987) - the above discrepancy has been considered as a feature of incompleteness. However, it is more appropriate to consider this as a consistency problem since the error stems from the fact that the attribute-value pair(s) used in the rule are not consistent with the valid list of attribute-value pairs.

2.2 Errors Affecting Completeness

Rule bases are rendered incomplete due to missing rules. This can manifest in one of the following ways:

(a) Deadend Premises

(b) Unreachable Goals

(c) Unreferenced Attribute Values

2.2.1 Deadend Premises

When the conclusion of a rule is neither a final conclusion, nor is part of the condition of some other rule, then that rule constitutes a deadend premise. A likely reason for this is a rule which has been missed by the expert during the knowledge elicitation stage. For example, consider the following rule:

Rule 1: If probability of commercial success is low,
Then financial viability is pessimistic.

If there are no rules which have "financial viability is pessimistic" as part of its condition and if "financial viability is pessimistic" is not a final conclusion, this is a case of a deadend premise.

2.2.2 Unreachable Goals

When the condition of a rule is neither an input variable, nor a part of conclusion of some other rule, then that rule constitutes a unreachable goal. A missing rule at the knowledge elicitation stage can cause such a discrepancy. Consider the following rules.

Rule 1: IF Probability of Commercial Success is low
THEN Profitability is low.
Rule 2: IF Anticipated Return on Investment is low
THEN Abandon the project.

The premise of Rule 2 is unachievable without a rule that links the conclusion of the first rule to the premise of the second rule such as the following rule.

Rule 3: IF Profitability is low
THEN Anticipated Return on Investment is low.

2.2.3 *Unreferenced Attribute Values*

This refers to the absence of any rules containing some valid attribute-value pairs. For example, consider an R&D Project Evaluation rule base where the attribute "Anticipated Growth Rate" has the three values High, Moderate, and Low. If the rule base does not have any rule pertaining to Anticipated Growth Rate being Moderate, it is an instance of unreferenced attribute value and is indicative of a missing rule containing that attribute-value pair.

In addition to the errors discussed, previous studies have identified other errors that can appear in a rule base (Ignizio, 1991; Kang and Bahill, 1990; Nazareth, 1989; Suwa et al, 1982). Some of these errors are syntactic in nature, and can usually be easily detected by the development shell. We assume that such errors have been eliminated from the rule base before detection of structural errors. Other errors have to do with incorrect representation of the domain knowledge, and cannot be detected without extensive interaction with domain experts - these errors are called *semantic errors* (Nazareth, 1989). Such errors are usually identified during the validation stage of the development process, and are not addressed in this research.

2.3 *Type 'A' and Type 'B' Errors*

All of the consistency errors illustrated in Section 2.1 are those that result from inferences based on assertions which form the antecedent of one rule in the rule base. These assertions could be either simple clauses or compound clauses. For instance, in the example considered in Section 2.1.3, the assertion of the antecedent "the likelihood of commercial success is 'medium'" leads to a contradiction, since both "continue project" and

"discontinue project" are concluded. We classify such errors that are caused when antecedents of single rules are instantiated to start the inference process as Type 'A' errors. Most existing verification techniques have focussed on the detection of errors that result from such assertions. In practice, assertions used to trigger the inference mechanism often form antecedents for multiple rules. For example, consider a rule base with the following rules:

Rule 1:	IF	the likelihood of technical success is high
	THEN	the R&D project can be completed within three years.
Rule 2:	IF	the likelihood of commercial success is high
	THEN	Federal grants are not available.
Rule 3:	IF	the estimated R.O.I is low
	THEN	Federal grants are not available.
Rule 4:	IF	the estimated R.O.I is low
	THEN	do not allocate more personnel.
Rule 5:	IF	the R&D project can be completed within three years
	AND	Federal grants are not available
	THEN	allocate more personnel.

Let both the factors 'the likelihood of high technical success' and 'the likelihood of high commercial success' be true for a particular project. If 'the estimated R.O.I. is low' is not true for the same project, then the conclusion 'Allocate more personnel' is reached. Suppose it is known that both the clauses 'the likelihood of high technical success' and 'the estimated R.O.I. is low' are true for an R&D project. This results in a contradiction since both 'Allocate more personnel' and 'Do not allocate more personnel' are concluded by the rule base. It is important to note that this kind of inconsistency is not detected if

assertions used to trigger the inference mechanism correspond to antecedents of single rules. Errors that are caused when antecedents of several rules are instantiated simultaneously are classified as Type 'B' errors.

Detection of Type 'B' errors is a hard problem because of the following reasons. First, the number of different combinations of variables to be considered can grow exponentially in large rule bases. In addition, among the numerous combinations of input variables, selection of valid combinations for testing is difficult. If input variables are considered that cannot occur simultaneously, then spurious errors are likely to be detected. For this reason, it is not possible to completely automate the detection of such errors in a rule base.

2.4 Existing Verification Techniques

Existing techniques to verify rule bases can be broadly categorized as follows depending on the underlying principle:

(a) Pairwise Check Techniques,

(b) Rule Label Technique, and

(c) Graph-Theoretic Techniques.

2.4.1 Pairwise Check Techniques

Suwa, Scott, and Shortliffe (1982) were the first to propose a domain independent methodology to verify rule bases. In this technique, the checking of rules is done in the following way:

(a) The program finds all the attributes used as conditions of the rules,

(b) Then the program makes a table, which displays all possible combinations of attributes used as conditions and the corresponding conclusions.

(c) The program checks the table for conflict, redundancy, subsumption and missing rules by performing pairwise checks.

This verification procedure detects contradiction, redundancy and subsumption between pairs of rules. It also detects missing rules. The above technique does not detect conflicts in chains of rules. Circular rules also go undetected.

Nguyen, Perkins, Laffey, and Pecora (1987) describe an automated rule verifier called CHECK. In this technique, each IF and THEN clause of every rule in the set is compared against the IF and THEN clauses of every other rule in the rule base. The comparison of one clause against another results in a label of SAME, DIFFERENT, CONFLICT, SUBSET, or SUPERSET being stored in a two-dimensional table maintaining the interclause relationships. Unreachable goals are identifiedby finding those then clauses which have the different relationshipfor all if clauses and goals. Dead-end premises are identified when the different relationship exists for all conclusions, and the attribute these goals and conditions refer to is not askable.

In addition to pairwise checks for redundancy, subsumption and contradiction, this technique also detects circular chains made up of pairs of rules. This verification procedure does not detect conflict or redundancies that occur across chains of rules.

Cragun and Steudel (1987) propose a decision-table based method for rule base verification. In this technique, the rule base is represented as a decision table with each rule requiring one column and each different condition and action requiring one row. Decision tables facilitate the testing of a set of rules for contradiction, redundancy, and

completeness. Contradiction occurs when the same set of logical conditions satisfy two or more different rules that have different actions. Redundancy occurs when the same logical conditions satisfy more than one rule, but the actions are the same. Completeness is present when all possible combinations of logic are addressed by by rules in the table. But this technique also fails to detect conflicts across chains of rules and circular rule chains.

2.4.2 Rule Label Technique

Ginsberg(1987) has proposed a verification scheme based on the principle of truth maintenance systems. In this scheme, errors are detected in the following way. To begin with, all the rule elements are categorized into three mutually exclusive categories viz. "findings," "hypotheses," and "default-hypotheses." A finding is any literal that appears only on the left hand side of rules and is not the logical negation of a literal on the right hand side of any rule. A hypothesis is any literal that either occurs solely on the right hand side of rules or on the left hand side of some rules and the right hand side of others. A default-hypothesis is any literal that occurs only on the left hand sides of rules and is the logical negation of some hypothesis.

After categorizing the rule elements as noted above, all the rules in the knowledge base are ordered into levels based on a "depends-on" relationship. An "environment" for a hypothesis is defined as the set of findings which conclude that hypothesis. The set of minimal logically consistent environments for a hypothesis is called its "label". The verification procedure consists of determining labels for all the hypotheses (conclusion portions of rules) and detecting inconsistencies by comparing the rule labels. Using this procedure, Ginsberg shows how different types of errors may be detected using this

procedure. His procedure illustrates how the detection of errors due to redundancy and contradiction which appear in chains of rules is a computationally hard problem.

2.4.3 Graph-Theoretic Techniques

Nazareth and Kennedy (1991) have proposed a verification procedure using directed graphs. In this representation, each attribute-value pair in the rule base appears as a node in the graph. In addition, there is a node associated with every rule in the rule base. By using nodes for each rule, the graph displays inference paths in the rule base. For example, consider the following rules:

Rule 1. IF (A = a1) & (B = b1) THEN (C = c1) and

Rule 2. IF (C = c1) & (D = d1) THEN (E = e1).

The digraph representation of the above rules is as shown in Figure 2.1.

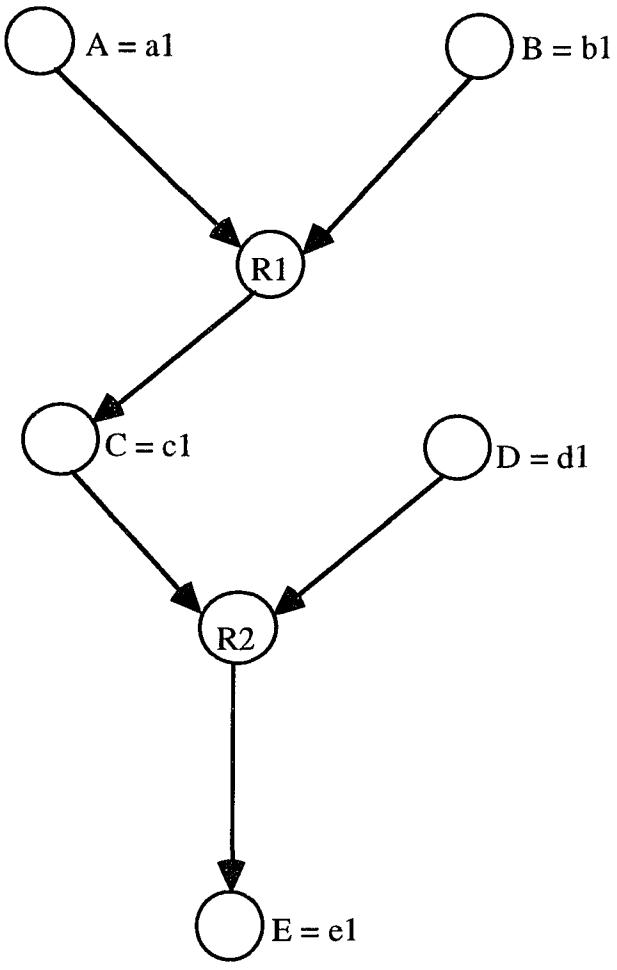

Figure 2.1 Rule Representation Using Directed Graph

In this procedure, an initial adjacency matrix is first constructed for the directed graph representation. This matrix, called the **A** matrix shows the dependencies across all the attribute-value pairs and the rules in which they appear. When this **A** matrix is multiplied by itself it yields a new matrix ($\mathbf{A}^2$) which denotes inference paths indicated by each rule. By repeating this multiplication process, the inference paths across chains of rules are identified. For instance, the matrix $\mathbf{A}^4$ displays inference paths over all sets of

two rules. This process is continued until more multiplications do not lead to additional paths being displayed. The final reachability matrix, **T** is obtained by summing all matrices $(\mathbf{A} + \mathbf{A}^2 + \mathbf{A}^3 + ...)$.

In this technique, error detection is done in the following way. By inspecting the **T** matrix, we can find out all the conclusions that can be reached starting from an attribute-value pair as well as the number of ways it can be reached. Redundancy is indicated when the same conclusion can be reached in more than one way. Contradiction is flagged when the same attribute-value pair leads to mutually exclusive conclusions. However, when simple nodes are part of a compound antecedent, the inference paths indicated by this scheme are not accurate. This is illustrated below:

Example 1. Rule 1: a1 —> b1

Rule 2: b1 + c1 —> e1

Rule 3: b1 + d1 —> e2 and e1, e2 mutually exclusive.

The directed graph for the above rules is shown in Figure 2.2. While there is no contradiction inherent in these rules, the matrix operations indicate that both (e1) and (e2) are reached from (a1), thereby implying inconsistency. Similar spurious errors of redundancy are also detected.

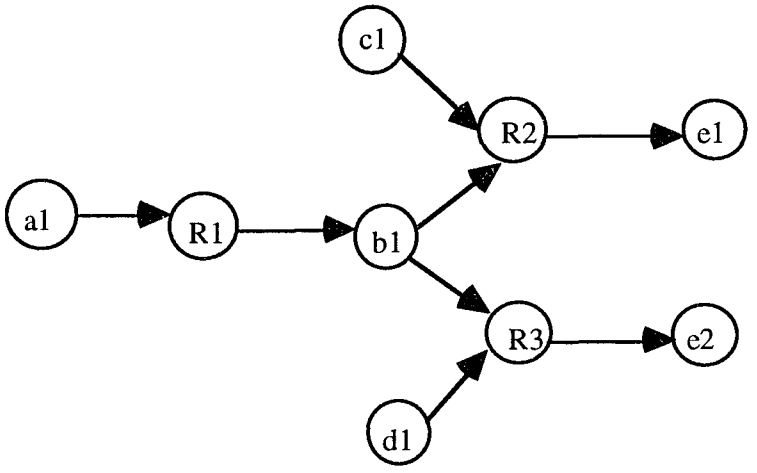

Figure 2.2 Directed Graph for Example 1

Two different verification procedures that model the rule base as a Petri net have been proposed. Agarwal and Tanniru (1992) present a verification procedure in which they model a Petri net by an incidence matrix. In an incidence matrix, each attribute-pair value corresponds to a column and each rule corresponds to a row. If an attribute-value pair is present in the condition part of a rule, then a '-1' is placed in the corresponding column and if it is a conclusion, a '+1' is placed. In order to test the consistency of a new rule being added to the rule base, a row vector is constructed for the new rule. An output vector is determined from the product of incidence matrix and the transpose of the row vector for the new rule. Verification involves the use of the following parameters:

S(i): the total number of attribute-value pairs in rule i,

O(i): the value of row i in the Output Vector.

Tests for redundancy are performed as follows:

If S(i) > O(i) and O(i) = S(k), then the new rule k subsumes rule i,

If S(i) = O(i) and O(i) < S(k), then rule i subsumes the new rule k,

If S(i) = O(i) = S(k), then rule i redundant to the new rule k.

We note that in (Agarwal and Tanniru; 1992), the third condition is given as S(i) = O(i) = O(k). This appears to be a misprint since O(k) is not defined when k is the new rule.

The above procedure fails to detect errors that are caused by inference paths of different lengths. For example, consider a rule base with the following three rules:

Example 2: Rule 1. a1 + b1 —> c1

Rule 2. a1 —> c1

Rule 3. b1 —> d1

Let the new rule to be added, Rule 4, be: c1 + d1 —> e1. The addition of this rule results in two different paths from (a1, b1) to (e1). The Incidence Matrix for the first three rules, Row Vector for the fourth rule, and the resulting Output vector are shown in Figures 2.3, 2.4, and 2.5 respectively.

	a1	b1	c1	d1	e1
Rule 1	-1	-1	0	0	1
Rule 2	-1	0	1	0	0
Rule 3	0	-1	0	1	0

Figure 2.3 Incidence Matrix for Example 1

	a1	b1	c1	d1	e1
Rule 4	0	0	-1	-1	1

Figure 2.4 Row Vector for Rule 4

$$\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$$

Figure 2.5 Output Vector

In this example, we have:

S(1) = 3; S(2) = 2; S(3) = 2; S(4) = 3

O(1) = 1; O(2) = -1; O(3) = -1

Examining the above values, we find that none of the conditions for redundancy are met in this example, although there are two different paths from (a1,b1) to (e1). In this Petri Net-based scheme, conflicts are detected by replacing the conclusion of a rule by one of its mutually exclusive outcomes and then performing tests for redundancy. Consequently, the algorithm is also unable to detect conflicts that occur due to paths of different lengths.

Nazareth (1993) has proposed a different Petri Net approach. A special type of representation is used which accurately represents the conclusion of a rule after its firing.

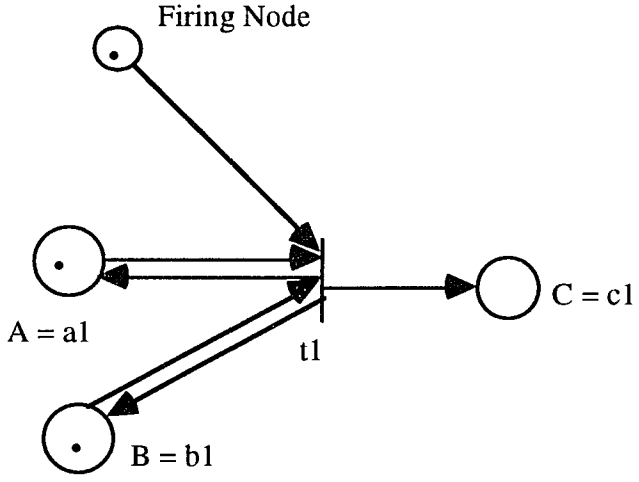

Figure 2.6 Rule Representation Before Firing of Rule

For example, consider the rule IF (A = a1) and (B = b1) THEN (C = c1). It is represented as shown in Figure 2.6.

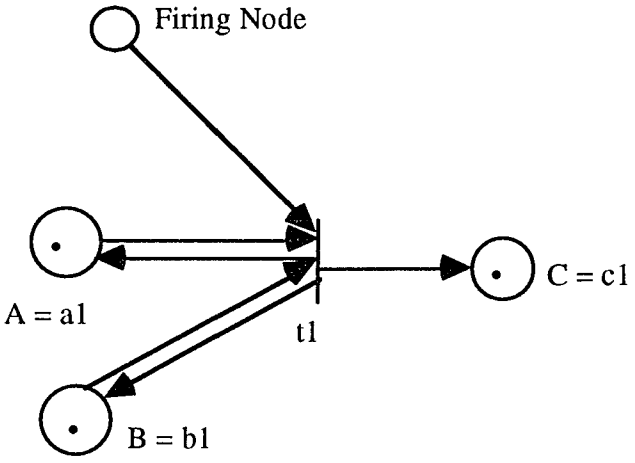

Figure 2.7. Resultant State After Firing of Rule.

The nodes corresponding to the conclusions that are mutually exclusive are connected to 'integrity constraint' nodes.

The error detection procedure in the above scheme works in the following way. The verification procedure starts with a given set of 'markings' which corresponds to the attribute-value pairs of the rules that are being fired simultaneously. Based on these initial markings, the conclusions which can be reached by each attribute-value pair as well as the transitions needed are determined. If the same conclusion is reached via two different sets of transitions, it indicates redundancy. The procedure incorporates a dummy "integrity constraint" node which is asserted when two or more mutually exclusive attribute-value pairs are concluded. These integrity constraint nodes are utilized to detect contradiction. The Petri Net representation for the rules in Example 1 are shown in Figure 2.8.

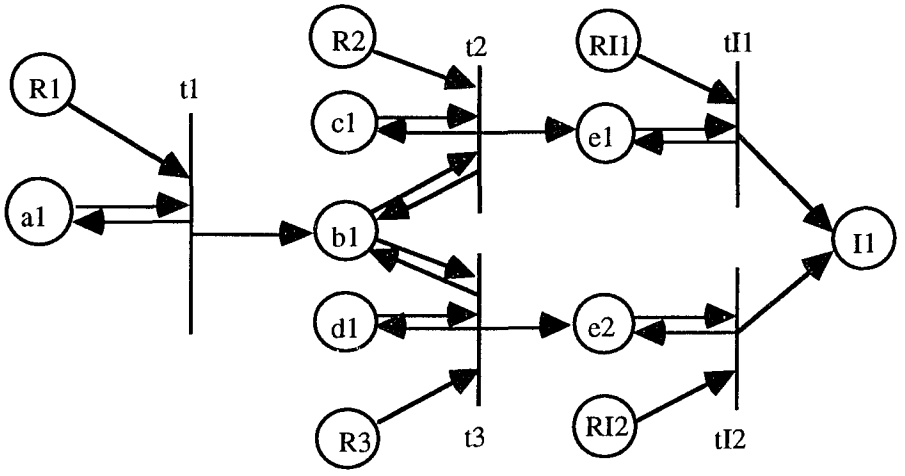

Figure 2.8 Petri Net Representation for Example 1

While the the Petri net representation itself is accurate, the associated matrix operations indicate spurious paths much like the directed graph approach. As noted by Nazareth, these spurious paths are indicated since the matrix operations do not check if a transition is enabled prior to being fired [Nazareth (1993), pp. 406]. In Example 1, if a1 is the only observed variable and we wish to identify reachabilities using all possible rules, the procedure would indicate two paths to I1, thereby indicating contradiction.

CHAPTER 3
DIRECTED HYPERGRAPHS

The flexibility and compactness of representation, coupled with the ability to analyze connectivity, makes graphical representation of rules well suited for verification of rule-based systems. In the existing verification techniques based on graphical representation (Nazareth and Kennedy, 1991; Agarwal and Tanniru, 1992; Nazareth, 1993) each attribute-value pair is considered as a separate node. But there is no special representation to capture a group of attribute-value pairs which form an antecedent of a rule. When different rules share common attribute-value pairs, the single node representation cannot capture the dependencies accurately.

We have developed a graphical representation scheme based on hypergraphs (Berge, 1989) which accurately represent all the dependencies across simple and compound clauses in a rule base. In our directed hypergraph representation of rule bases, we use simple and compound nodes to represent simple and compound clauses respectively. A compound node is merely a collection of simple nodes. However, by including such compound nodes, we can clearly represent the dependencies across a compound antecedent clause and a conclusion clause.

In this chapter, we introduce directed hypergraphs and provide the important notation and definitions that we use. An adjacency matrix representation for the directed hypergraph is discussed, and some special matrix operations used in the verification procedure are defined. Finally, the verification algorithm is presented.

3.1 Directed Hypergraph Representation

The main reason for using hypergraphs is that compound clauses in rules can be explicitly represented by such graphs without recourse to contrived representations (as is the case in simple graphical structures). In our directed hypergraph representation, simple and compound clauses in a rule base are represented as simple and compound nodes, respectively. Thus, a node is compound if it consists of a conjunction of two or more attribute-value pairs. A rule is represented as a directed arc across nodes (simple or compound as the case may be) that correspond to its antecedent and conclusion clauses. The direction of the arc is determined by the causality implied by the rule. We illustrate the directed hypergraph representation using the following rules:

Rule 1:	IF	the likelihood of technical success is high
	AND	the likelihood of commercial success is high
	THEN	the R&D project can be completed within three years .
Rule 2:	IF	the likelihood of technical success is high
	AND	the likelihood of commercial success is high
	THEN	Federal grants are available .
Rule 3:	IF	the R&D project can be completed within three years
	AND	Federal grants are available
	THEN	continue the project .
Rule 4:	IF	Federal grants are available
	THEN	allocate more personnel .

The directed hypergraph representation for the above rules is shown in Figure 3.1.

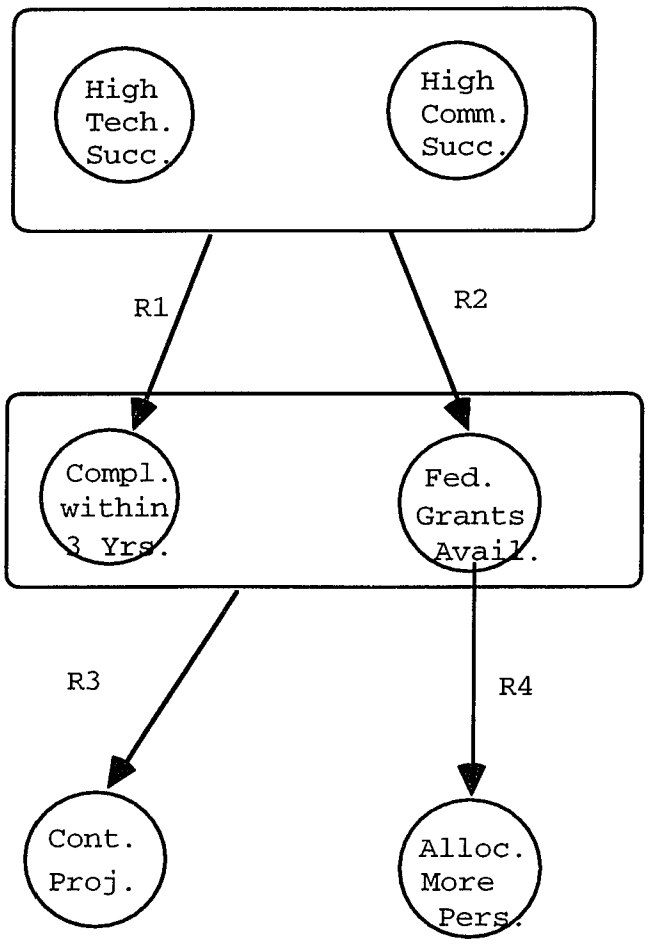

Fig. 3.1 Directed Hypergraph for Rules 1 through 4

Since the condition clause of Rule 1 consists of two attribute-value pairs, the rule is represented by a directed arc from a compound node (corresponding to the compound condition clause *High Technical Success* AND *High Commercial Success*) to a simple node (corresponding to the rule's conclusion *Completed Within 3 Years*). Such a straightforward representation is not possible when simple graphs are used.

The attribute-values that are inferred when the premise of a rule is known to be true can be determined from the hypergraph by identifying those nodes that can be reached by following the directed arcs. Thus, feasible inference paths from one set of clauses to another are indicated by the reachability across the corresponding nodes in the hypergraph.

3.2 Notation and Definitions

The directed hypergraph representation that we use is a variation of hypergraphs as discussed by Berge (1989). We provide a formal definition of the different terms and notations that are used in the verification procedure. The notation allows us to provide a concise and accurate representation of the dependencies in the rule base.

Rule: Rules are represented symbolically as follows. Each attribute-value pair is represented by a lower case alphabet indicating an attribute and a numeric digit indicating the value for that attribute. Thus, an attribute P with three legal values would be represented as p1, p2 and p3. The rule: *IF (P = p1) & (Q = q2) THEN (R = r3)* is denoted as: p1 + q2 —> r3. Using this notation, the four rules in Section 3.1 can be represented as follows:

Rule 1:	a1 + b1	—> c1
Rule 2:	a1 + b1	—> d1
Rule 3:	c1 + d1	—> e1
Rule 4:	d1	—> f1

The symbols a1, b1, c1, d1, e1 and f1 refer to the attribute-value pairs in the corresponding rules.

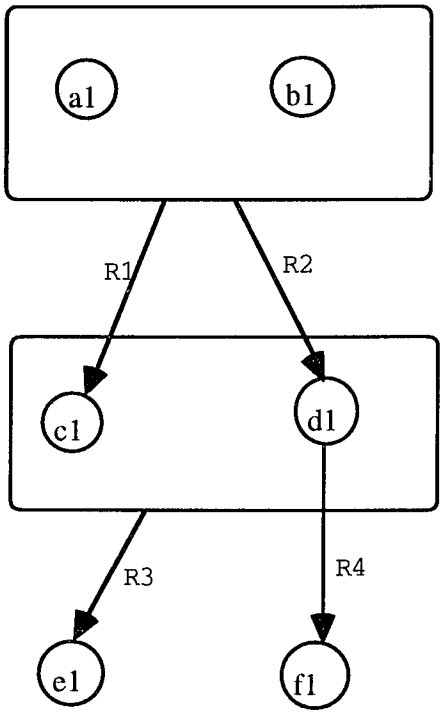

Figure 3.2 Directed Hypergraph For Rules in Symbolic Form

Figure 3.2 shows the directed hypergraph for the rules in their symbolic form. At times, for the sake of compactness, we also use the form <a1,b1; c1> to represent a rule. This is particularly useful when an entire inference path is to be represented.

Hypernode: Each node corresponding to either the condition clause or the conclusion clause of a rule is called a hypernode. A hypernode is a *simple node* if it comprises a single attribute-value pair. It is a *compound node* if it consists of two or more attribute-value pairs. We refer to a hypernode by either using the attribute-value(s) of the corresponding clause, or an uppercase character with an underscore. For example, Rule 1 above can be written as <U;V>, where U = (a1,b1) and V = (c1). In order to differentiate between a simple and a compound node, we sometimes use lowercase and uppercase characters to

denote each, respectively. In that case, Rule 1 is written as <U;v> where U = (a1,b1) and v = (c1).

Hyperpath: A hyperpath is a collection of rules that comprises the inference path from one hypernode to some other hypernode (we use the term *path* interchangeably with *hyperpath*). A hyperpath is uniquely identified by specifying the originating and terminal hypernodes, and the collection of rules that comprises the hyperpath. For example, in Figure 3.2, the inference path between (a1,b1) and (f1) can be written as P{a1,b1;f1 \ <a1,b1; d1>, <d1;f1>}. The notation specifies that the hyperpath with the originating and terminal hypernodes as indicated consists of the arcs that correspond to the rules <a1,b1; d1> and <d1;f1>. We wish to point out that it is necessary to provide all of the above information to identify a hyperpath uniquely (which we call a *distinct hyperpath*). Providing only the originating and terminal hypernodes is not enough because there may be multiple inference paths (through different rules) between them. Providing only the rules is also not adequate, because the same set of rules can be used to represent hyperpaths with different originating hypernodes. Thus, in Figure 2, the two hyperpaths P{c1,d1;f1 \ <d1;f1>} and P{d1;f1 \ <d1;f1>} have the same rule set. We consider these two paths as distinct because the originating hypernodes are different. This allows us to easily detect errors that result from different paths, when the originating hypernode of one path subsumes the originating hypernode of another path. Since the representation for a hyperpath can consist of a large number of rules, we use a shorthand notation consisting of only the originating and terminal hypernodes *in those cases where there is no ambiguity associated with that representation.* In such cases, the hyperpath from (a1,b1) to (f1) would be represented as P{a1,b1;f1}. In general, the hyperpath between a hypernode $\underline{U}$ to a hypernode $\underline{V}$ is denoted as P{$\underline{U}$;$\underline{V}$} when rules are not being specified.

An important implication of having compound nodes in a hyperpath is that there can be more than one sequence in which the rules may be fired to reach the terminal hypernode. Thus, there may be multiple sequences in which the rules may appear in the hyperpath. In Figure 2, the hyperpath between the two compound nodes (a1,b1) and (c1,d1) can be written as either P{a1,b1;c1,d1 \ <a1,b1;c1>,<a1,b1;d1>} or as P{a1,b1;c1,d1 \ <a1,b1;d1>,<a1,b1;c1>}. This can be confusing when the inference chain consists of a large number of rules that are interdependent. The actual sequence in which rules are fired will depend on the control strategy of the inference engine that is used. In order to make our notation implementation independent, we use the notion of a *segment* to identify the logical sequence of rules that constitutes a hyperpath. For the sake of illustration, we assume the following: (1) rules are fired using forward chaining, and (2) when one or more attribute-values are known, then the inference mechanism examines every rule in each pass of the rule base (thus, multiple rules may be fired in each pass of the rule base). With this scenario in mind, we define the first segment of a hyperpath as those rules in the inference path that are fired in the first pass, when the attribute-value(s) of the originating hypernode is (are) known to be true. The next segment corresponds to those rules in the inference path that are fired when the conclusions of the first segment are added to the set of known assertions, and so on. Thus, each hyperpath can be viewed as an ordered sequence of segments, with rules in a segment being fired in the same pass of the rule base. If the rules <a1,b1;c1> and <a1,b1;d1> are part of the same segment, then we denote them as: «<a1,b1;c1>, <a1,b1;d1>». Thus, if we wish to explicitly identify each segment when representing the hyperpath P{a1,b1;e1}, we denote it as: P{a1,b1;e1 \ «<a1,b1;c1>, <a1,b1;d1>», «<c1,d1; e1>»}. The order of rules within each segment is not important.

However, the order of each segment is important and helps identify the flow of reasoning in an inference path. We should note that the assumption of forward chaining is used merely to help define a segment, and does not in any way restrict the actual inference engine that is used in practice.

The verification problem for rule bases is complicated by the presence of compound clauses in rules, and hence compound nodes that appear in an inference path. In order to identify the impact of such nodes in a clear fashion, we classify hyperpaths into two categories, *simple paths* and *strict hyperpaths.* A path from U to V is *simple* if every segment of the path has exactly one rule, and the first rule has U as the condition clause (i.e., the condition part of the rule in the first segment includes the entire set U and not a subset of U). Since we have assumed that all the rules are in unitized form, all intermediate nodes in a simple path are necessarily simple nodes. The path P{a1,b1;f1} in Figure 3.2 is a simple path, and is written as SP {a1,b1;f1 \ «<a1,b1;d1>», «<d1; f1>»}. A path from U to V is a *strict hyperpath* if one of the following two conditions hold: (1) at least one segment of the path has two or more rules; or (2) the rule in the first segment has U' Ã U as the condition clause (i.e. the condition part of the rule in the first segment is a proper subset of U). For example, the hyperpath P{a1,b1;e1} is a strict hyperpath due to condition (1), and is written as HP{a1,b1;e1\«<a1,b1;c1>,<a1,b1;d1>», «<c1,d1;e1>»}. The path P{c1,d1; f1} is a strict hyperpath due to condition (2) and is written as HP{c1,d1; f1\ «<d1;f1>»}.

The notion of a segment also allows us to identify the subpaths that make up a complete hyperpath. For example, the hyperpath P{a1,b1;e1\ «<a1,b1;c1>,<a1,b1;d1>», «<c1,d1;e1>»} has a subpath from (a1,b1) to (c1,d1) which is the strict hyperpath HP{a1,b1;c1,d1\ «<a1,b1;c1>,<a1,b1;d1>»}, and a subpath from (c1,d1) to (e1) which is

the simple path SP{c1,d1;e1\ «<c1,d1;e1>»}. When checking for errors that show up over a chain of rules, we essentially need to consider those paths that end in a simple node (since the rule base is assumed to be unitized, existence of a path to a compound node necessarily implies the existence of paths to each component of the compound node). For this reason, when looking for errors, we examine strict hyperpaths that end in a single node. The last segment of such strict hyperpaths consist of a single rule. This implies that the sub path of a strict hyperpath, that is obtained by dropping the last segment of the hyperpath, must be a strict hyperpath by itself (the subpath could end in either a simple or a compound node). Subsequently, we use this property to identify the strict hyperpaths that exist in the rule base.

Length of a path: We define the length of an inference path as the number of segments that constitutes the path. Thus, in Figure 3.2, P{a1,b1;e1} is of length 2, while P{a1,b1; c1,d1} is of length 1. The length of the path indicates the maximum number of times the inference mechanism may need to examine the rule base in order to establish all the attribute-value pairs in the terminal hypernode when the attribute-value pairs in the originating hypernode are asserted to be true.

3.3 *The Adjacency Matrix and Matrix Operations*

We use an adjacency matrix representation of the directed hypergraph to perform the verification checks for a rule base. In addition to standard matrix operations such as matrix multiplication, matrix addition, and matrix subtraction, we also use two special operations that we call Column Revision and Row Revision. Our definitions for the adjacency matrix and the special operations are provided below:

Adjacency Matrix: The adjacency matrix of a directed hypergraph is a square matrix, **A**, which has a row and a column for each hypernode in the hypergraph. If there is an arc from a hypernode U to a hypernode V, then the corresponding entry in the matrix, **A**[U,V], takes on the value 1. If there is no such arc then **A**[U,V] takes on the value 0. Thus, the adjacency matrix displays all simple paths of length 1 in the rule base. The directed hypergraph in Figure 3.2 has an adjacency matrix **A** as shown in Figure 3.3.

	a1,b1	c1,d1	c1	d1	e1	f1
a1,b1			1	1		
c1,d1					1	
c1						
d1						1
e1						
f1						

Figure 3.3 Adjacency Matrix for the Example Hypergraph

Since each hypernode corresponds to either an antecedent or a conclusion of a rule, the maximum size the adjacency matrix can have is $2n \times 2n$, where n is the number of rules in the rule base. However, hypernodes common to several rules typically reduce the number of rows and columns that are required. For example, there are six rows and columns in the adjacency matrix in Figure 3.3, even though there are four rules. This is because the hypernode (a1,b1) is a common antecedent for two rules and (d1) is a conclusion of one rule and an antecedent of another rule. Since rules are in their unitized form, there must be a unique row and column for every attribute-value pair corresponding to intermediate and output variables. We also note that rows/columns corresponding to compound nodes are included in the matrix only if the compound node corresponds to a

compound antecedent for a rule. Subsequently, rows corresponding to compound nodes must have at least one non-zero entry.

Column Revision (CR) Operation: The column revision (CR) operation establishes the reachability from a compound node to a simple node when any subset of the compound node has a path to the simple node. Consider the adjacency matrix shown in Figure 3.3. While the matrix shows that node (f1) can be reached from node (d1), it *does not* show that (f1) can be reached from the hypernode (c1,d1). This reachability is important for detecting errors in the rule base, and is captured in the adjacency matrix by the Column Revision operation. Formally, the CR operation for a matrix **M** is defined as follows:

$$CR\ (\mathbf{M}[\underline{U}, \underline{V}]) = \mathbf{M}[\underline{U}, \underline{V}] + \Sigma\, \mathbf{M}[\underline{U}', \underline{V}] \quad \forall\, \underline{U}' \subset \underline{U}.$$

Thus, if $\mathbf{M}[\underline{U}, \underline{V}] = 0$ and $\mathbf{M}[\underline{U}', \underline{V}] = 1$, where $\underline{U}' \subset \underline{U}$, then after column revision, we have $\mathbf{M}[\underline{U}, \underline{V}] = 1$, indicating that $\underline{V}$ can be reached from $\underline{U}$. If there is a path from $\underline{U}$ to $\underline{V}$ as well as a path from a subset of $\underline{U}$ to $\underline{V}$ then it results in the entry $\mathbf{M}[\underline{U}, \underline{V}] = 2$ thereby indicating multiple paths from $\underline{U}$ to $\underline{V}$. The matrix CR(**A**), obtained after performing the Column Revision operation on matrix **A** (of Figure 3.3) is as shown in Figure 3.4. The reachability established by column revision is indicated by a highlighted entry.

	a1,b1	c1,d1	c1	d1	e1	f1
a1,b1			1	1		
c1,d1					1	1
c1						
d1						1
e1						
f1						

Figure 3.4 Adjacency Matrix for the Directed Hypergraph after Column Revision

Row Revision (RR) Operation: The Row Revision operation establishes the reachability from an originating hypernode to a terminal hypernode that is a compound node, when the originating hypernode has a path to each component (attribute-value pair) of the terminal hypernode. The Row Revision operation for a matrix **M** is defined as follows:

$$RR\ (\mathbf{M}[\underline{U}, \underline{V}]) = \mathbf{M}[\underline{U}, \underline{V}] + \prod_i \mathbf{M}[\underline{U}, v_i] \quad \text{s.t.} \cup_i v_i = \underline{V}.$$

The adjacency matrix in Figure 3 fails to show that the compound node (c1,d1) can be reached from (a1,b1), although it shows that the two simple nodes (c1) and (d1) can be reached from (a1,b1). After performing Row Revision on matrix **A**, we get RR(**A**) as shown in Figure 3.5. New reachabilities established by Row Revision are highlighted in the matrix. The product operator helps ensure that if there are multiple paths from the originating hypernode U to one or more components of the terminal hypernode V, then all the distinct paths from U to V are reflected in the revised matrix.

	a1,b1	c1,d1	c1	d1	e1	f1
a1,b1		1	1	1		
c1,d1					1	
c1						
d1						1
e1						
f1						

Figure 3.5 Adjacency Matrix for the Directed Hypergraph after Row Revision

3.4 Procedure to Perform Checks

The basic idea behind our procedure is to accurately identify all simple paths and strict hyperpaths, and then examine them for omissions and errors. Paths consisting of more than one segment are detected using matrix multiplication. All feasible reachabilities are established using the defined or standard matrix operations. To illustrate how the algorithm works, we provide the following notation for the different matrices that are used in the verification process.

A: The adjacency matrix for the hypergraph.

$\mathbf{A}^i$: Displays simple paths of length i. For example, $\mathbf{A}^3$ displays all *simple paths* of length exactly three. Note that $\mathbf{A}^1 = \mathbf{A}$.

$\mathbf{A}_i$: Displays strict hyperpaths of length i that end in simple nodes. Therefore, $\mathbf{A}_3$ displays all *strict hyperpaths* of length exactly three.

$\mathbf{B}_i$: Displays all simple paths and strict hyperpaths of length $\leq$ i ending in simple nodes.

$\mathbf{C}_i$: Displays all simple paths and strict hyperpaths of length $\leq$ i ending in simple as well as compound nodes.

$\mathbf{D}_i$: Displays only strict hyperpaths of length i ending in either simple or compound nodes.

All of these matrices are of the same size as the adjacency matrix for the hypergraph. The notation we have used makes it easy to detect errors that exist in a rule base. For example, if an entry in matrix $\mathbf{B}_i$ has a value greater than 1, then it indicates multiple paths across the hypernodes associated with the column and row of the matrix. Since all hyperpaths (simple paths and strict hyperpaths) of length greater than one comprise hyperpaths of smaller lengths, the algorithm we use is iterative in nature, i.e. we

first determine all hyperpaths of length one, and then use them to determine hyperpaths of length two, etc. The algorithm halts when no additional paths can be detected. In addition to the above matrices, we use a list, called the Circular Path List, where we store the hypernodes that form both the originating and terminal nodes for a circular hyperpath. When circular paths show up in the above matrices, they result in infinite path lengths, and the halting condition in the algorithm is never satisfied. In order to avoid such situations, occurrences of circular paths are eliminated from the matrices themselves and stored in the Circular Path List. For expositional purposes, we nevertheless indicate them in the matrices using an asterisk (*). Thus, an asterisk in a matrix indicates the presence of a circular path; however, it is treated as a zero for matrix computations such as multiplication, addition, etc.

The hyperpaths that end in simple nodes are adequate for detecting all errors of interest. Thus, we can detect all the anomalies from the $\mathbf{B}_i$ matrix that is obtained after the longest inference path has been identified. However, to establish the existence of strict hyperpaths of length i, we need to establish all hyperpaths of length i-1 that end in either simple *or compound nodes*. Thus, in order to obtain the final $\mathbf{B}_i$ that is required to check for errors, we need to evaluate $\mathbf{C}_j$ and $\mathbf{D}_j$ for all $j < i$. We first present the algorithm that is used to generate the different matrices, and then discuss the tests that are used to detect each type of error.

Verification Algorithm

Step 1: Initialize i = 1

$\mathbf{B}_1 = CR(\mathbf{A}^1)$

$\mathbf{C}_1 = RR(\mathbf{B}_1)$

$\mathbf{D}_1 = \mathbf{C}_1 - \mathbf{A}^1$

Step 2: While $\mathbf{A}^i + \mathbf{A}_i \neq 0$ do steps 3 - 9

Step 3: Set i = i + 1

Step 4: $\mathbf{A}^i = \mathbf{A}^{i-1} ¥ \mathbf{A}$

If an element $\mathbf{A}^i$ [U,V] ≥ 1, and U V then include P{U;V} in the Circular Path List and reset $\mathbf{A}^i$ [U,V] = 0

Step 5: $\mathbf{A}_i = \mathbf{D}_{i-1} ¥ \mathbf{A}$

If an element $\mathbf{A}_i$ [U,V] ≥ 1, and U V then include P{U;V} in the Circular Path List and reset $\mathbf{A}_i$ [U,V] = 0

Step 6: If $\mathbf{A}^i + \mathbf{A}_i = 0$, go to Step 10

Step 7: $\mathbf{B}_i = \mathbf{A}^i + \mathbf{A}_i + \mathbf{B}_{i-1}$

Step 8: $\mathbf{C}_i = RR(\mathbf{B}_i)$

Step 9: $\mathbf{D}_i = \mathbf{C}_i - \mathbf{C}_{i-1} - \mathbf{A}^i$

Step 10: Examine **A** and $\mathbf{B}_i$ for errors

The incompleteness errors, *deadend premises* and *unreachable goals*, can be detected from the adjacency matrix **A** itself, and could be performed before the algorithm is executed. All of the other errors are detected by inspecting the final $\mathbf{B}_i$ matrix. We present the important results in the form of the following propositions and corollaries, proofs of which are provided in the next section.

Proposition 1: If there are k distinct paths of length ≤ i from a hypernode (simple or compound) U to a simple node *v*, then $\mathbf{B}_i[U,v] = k$.

Corollary 1: If $\mathbf{B}_i[U, v] > 1$, then there potentially exists a redundant rule, or, a chain of redundant rules, from U to *v* in the rule base.

Corollary 2: If $\mathbf{B}_i[U, v] \geq 1$, $\mathbf{B}_i[U, w] \geq 1$, and *v* and *w* are mutually exclusive, then the rule sets that comprise P{U; v} and P{U; w} are conflicting.

Corollary 3a: If an element $\mathbf{A}^i[U,V] \geq 1$, and $U \supseteq V$, then the rules that comprise the path(s) P{U;V} are circular.

Corollary 3b: If an element $\mathbf{A}_i[U,V] \geq 1$, and $U \supseteq V$, then the rules that comprise the path(s) P{U;V} are circular.

Proposition 2: If every element of a row in matrix **A** is zero, the row does not correspond to a goal variable, and the corresponding node is not a subset of a compound node, then it indicates a deadend premise.

Proposition 3: If every element of a column in matrix **A** is zero, the column corresponds to a simple node, and the column does not correspond to an input variable, then it indicates an unreachable goal.

3.5 Proof of Correctness

The correctness of the verification algorithm based on the directed hypergraph is shown by proving the following propositions:

Proposition 1: If there are k distinct paths of length ≤ i from a hypernode (simple or compound) U to a simple node v, then $\mathbf{B}_i[U, v] = k$.

Proof: In order to prove the above proposition, we need to show that the matrices $\mathbf{A}^i$, $\mathbf{A}_i$, $\mathbf{B}_i$, $\mathbf{C}_i$, and $\mathbf{D}_i$ will display the simple and strict hyperpaths as defined in Section 3.4. The proof is by induction. First, we prove the proposition for i = 1 and i = 2. Then we show that if the proposition is true for any arbitrary *i-1*, then it is also true for *i*.

Let $k_i(\underline{X}, \underline{Y})$ be the number of distinct paths of length ≤ i from a hypernode $\underline{X}$ to a hypernode $\underline{Y}$;

$k_{s_i}(\underline{X}, \underline{Y})$ be the number of distinct simple paths of length = i from $\underline{X}$ to $\underline{Y}$; and

$k_{h_i}(\underline{X}, \underline{Y})$ be the number of distinct strict hyperpaths of length = i from $\underline{X}$ to $\underline{Y}$.

a. Paths of length 1

Paths of length 1 can be categorized as (i) simple paths of length 1, and (ii) strict hyperpaths of length 1. We show that the matrices $\mathbf{A}$, $\mathbf{B}_1$, $\mathbf{C}_1$, and $\mathbf{D}_1$ are as defined.

A: *Simple paths of length 1*

We assume that duplicate rules are eliminated before any additional checks are performed. A path of length 1 is simple if and only if it corresponds to a single rule in the rule base. By definition, all such rules are displayed as entries in the adjacency matrix **A**.

$\mathbf{B}_1$: *Simple and strict hyperpaths of length 1 ending in simple nodes*

A strict hyperpath of length 1 can exist from a compound node $\underline{U}$ to a simple node v if and only if there exists a rule $\underline{U}' \rightarrow$ v where $\underline{U}' \tilde{A} \underline{U}$. The column revision operation establishes the reachability from a compound node to a simple node when any subset of the compound node has a path to the simple node. Since $\mathbf{B}_1$ is obtained after performing a column revision on $\mathbf{A}_1$, $\mathbf{B}_1$ contains all strict hyperpaths of length 1 from compound nodes to simple nodes. All simple paths in A are also preserved in $\mathbf{B}_1$. Therefore, $\mathbf{B}_1[\underline{U}, v]$ =

$k_{s_1}(\underline{U},v) + k_{h_1}(\underline{U},v) = k_1(\underline{U},v)$. Since $\mathbf{A}[\underline{U}, W] = 0$ where W is a compound node, and column revision does not affect such entries, we have $\mathbf{B}_1[\underline{U}, W] = 0$ for all such W.

$\mathbf{C}_1$: *Simple and strict hyperpaths of length 1 ending in simple or compound nodes*

$\mathbf{C}_1$ is obtained by performing a row revision on $\mathbf{B}_1$. Row revision establishes the reachability to a compound node when each component simple node is reached from a hypernode. The number of paths established is the product of the number of paths to the component simple nodes. For example, this number is zero if at least one constituent simple node is not reached by the originating hypernode, and is one if all the constituent simple nodes are reached uniquely. Since all the paths in $\mathbf{B}_1$ are also retained in $\mathbf{C}_1$, it follows that $\mathbf{C}_1[\underline{U}, \underline{V}] = k_1(\underline{U}, \underline{V})$, where $\underline{V}$ could be either a simple node or a compound node.

$\mathbf{D}_1$: *Strict hyperpaths of length 1 ending in simple or compound nodes*

$\mathbf{D}_1$ is obtained by subtracting **A** from $\mathbf{C}_1$. This ensures that only strict hyperpaths of length 1 (ending in simple or compound nodes) are displayed in $\mathbf{D}_1$, i.e. $\mathbf{D}_1[\underline{U}, \underline{V}] = k_{h1}(\underline{U}, \underline{V})$.

b. Paths of length ≤ 2

Paths of length ≤ 2 can be categorized as (i) Paths of length 1, (ii) Simple paths of length 2, and (iii) Strict hyperpaths of length 2.

$\mathbf{A}^2$: *Simple paths of length 2*

A simple path of length 2 from U to v is formed whenever a simple path of length 1 that ends in v has its antecedent reached from U through a simple path of length 1. Since rules are unitized, the general form of such a path is {U, v \ «<U, x>», «<x, v>»}. Thus, if x is an intermediate node in a path from U to v, it implies that $\mathbf{A}$[U, x] =1 and $\mathbf{A}$[x, v] =1. These terms will contribute 1 to the value of $\mathbf{A}^2$[U, v]. If there are k_{s_2}(U, v) simple paths of length 2 from U to v then each such path must traverse through a distinct simple node (since duplicate rules are not allowed). Therefore, after multiplying $\mathbf{A}$ with itself, we have $\mathbf{A}^2$[U, v] = k_{s_2}(U, v). We also have $\mathbf{A}^2$[U, W] = 0 where W is a compound node, since $\mathbf{A}$ does not include paths ending in compound nodes.

$\mathbf{A}_2$: *Strict hyperpaths of length 2 ending in simple nodes*

The last segment of a strict hyperpath of length 2 that ends in a simple node must be a simple path of length 1. From this it follows that the first segment must be a strict hyperpath of length 1. The antecedent of the last segment can be either a simple node or a compound node.

Case 1: Simple Node Antecedent

In this case, the general form of the path is {U, v \ «<U', x>», «<x, v>»}, where U' Ã U and x is a simple node. If x is an intermediate node in one or more strict hyperpaths from U to v then $\mathbf{D}_1$[U, x] = k_{h_1}(U, x) ≥ 1 and $\mathbf{A}$[x, v] = 1. These entries will contribute k_{h_1}(U, x) strict hyperpaths to $\mathbf{A}_2$[U, v] (since $\mathbf{A}_2$ is obtained by multiplying $\mathbf{D}_1$ by $\mathbf{A}$). Let there be *r* strict hyperpaths from U to v where the intermediate node is a simple node. Further, let these hyperpaths pass through *q* distinct intermediate nodes x_m, m=1,...,q.

Then $\mathbf{A}[x_m,v]$ is equal to 1 for all such x_m, and $r = \sum_{m=1}^{q} k_{h_1}(\underline{U}, x_m)$. As a result of multiplying $\mathbf{D}_1$ by $\mathbf{A}$, r will be the contribution to $\mathbf{A}_2[\underline{U},v]$ of strict hyperpaths with simple intermediate nodes.

Case 2: Compound Node Antecedent

In this case, the first segment of such a hyperpath must be a strict hyperpath from $\underline{U}$ to some compound node X, and the last segment is a simple path from X to the simple node v. Let there be *t* strict hyperpaths from $\underline{U}$ to v where the intermediate node is a compound node, which pass through *s* distinct intermediate nodes X_m, m=1,...,s. Then $\mathbf{A}[X_m,v]$ is equal to 1 for all such X_m, and $t = \sum_{m=1}^{s} k_{h_1}(\underline{U}, X_m)$. Therefore, after multiplying $\mathbf{D}_1$ by $\mathbf{A}$, *t* will be the contribution to $\mathbf{A}_2[\underline{U},v]$ of strict hyperpaths with compound intermediate nodes.

As a result, $\mathbf{A}_2[\underline{U}, v] = r+t$, which is the total number of strict hyperpaths of length 2 from $\underline{U}$ to v. Once again, $\mathbf{A}_2[\underline{U}, W] = 0$ where W is a compound node, since $\mathbf{A}[\cdot, W] = 0$ for all such W.

$\mathbf{B}_2$: *Simple and strict hyperpaths of length ≤ 2 ending in simple nodes*

We have $\mathbf{B}_2[\underline{U}, v] = \mathbf{A}^2[\underline{U}, v] + \mathbf{A}_2[\underline{U}, v] + \mathbf{B}_1[\underline{U}, v]$

$= k_{s_2}(\underline{U}, v) + k_{h_2}(\underline{U}, v) + k_1(\underline{U}, v)$

$= k_2(\underline{U}, v)$ where v is a simple node;

and, $\mathbf{B}_2[\underline{U}, W] = \mathbf{A}^2[\underline{U}, W] + \mathbf{A}_2[\underline{U}, W] + \mathbf{B}_1[\underline{U}, W] = 0$ where W is a compound node.

Therefore, the matrix $\mathbf{B}_2$ is as defined.

$\mathbf{C}_2$: *Simple and strict hyperpaths of length ≤ 2 ending in simple or compound nodes*

$\mathbf{C}_2$ is obtained by performing a row revision on $\mathbf{B}_2$. Thus, reachabilities to compound nodes are also captured in $\mathbf{C}_2$. We note that since all reachabilities in $\mathbf{B}_1$ are included in $\mathbf{B}_2$, the hyperpaths of length 1 that end in compound nodes are also retained.

$\mathbf{D}_2$: *Strict hyperpaths of length 2 ending in simple or compound nodes*

$\mathbf{D}_2$ is obtained by subtracting $\mathbf{C}_1$ (simple and strict hyperpaths of length 1 ending in simple and compound nodes) and $\mathbf{A}^2$ (simple paths of length 2) from $\mathbf{C}_2$ (simple and strict hyperpaths of length ≤ 2 ending in simple or compound nodes). This ensures that only strict hyperpaths of length 2 ending in simple or compound nodes are displayed in $\mathbf{D}_2$.

c. Paths of Length ≤ i

We show that if $\mathbf{A}^{i-1}$, $\mathbf{A}_{i-1}$, $\mathbf{B}_{i-1}$, $\mathbf{C}_{i-1}$, and $\mathbf{D}_{i-1}$ are as defined, then our matrix operations ensure that the matrices $\mathbf{A}^i$, $\mathbf{A}_i$, $\mathbf{B}_i$, $\mathbf{C}_i$, and $\mathbf{D}_i$ are also as defined. Paths of length ≤ i can be categorized as: (i) Paths of length ≤ (i-1); (ii) Simple paths of length i; and (iii) Strict hyperpaths of length i.

$\mathbf{A}^i$: *Simple paths of length i*

A simple path of length i from $\underline{U}$ to v is formed whenever a simple path of length 1 that ends in v has its antecedent reached from $\underline{U}$ through a simple path of length (i-1). If x is a penultimate intermediate node of one or more such paths, then $\mathbf{A}^{i-1}[\underline{U}, x] = k_{S_{i-1}}(\underline{U}, x) \geq 1$ and $\mathbf{A}[x, v] = 1$. Let the $k_{S_i}(\underline{U},v)$ simple paths of length i from $\underline{U}$ to v pass through

s distinct penultimate intermediate nodes x_m, m=1,...,s. Then, $\mathbf{A}[x_m,v]$ is equal to 1 for all such x_m, and $k_{S_i}(\underline{U}, v) = \sum_{m=1}^{s} k_{S_{i-1}}(\underline{U}, x_m)$. It follows that $\mathbf{A}^i[\underline{U}, v] = k_{S_i}(\underline{U}, v)$. Note that $\mathbf{A}^i[\underline{U}, W] = 0$ where W is a compound node since $\mathbf{A}[\cdot,W] = 0$ for all such W.

$\mathbf{A}_i$: *Strict hyperpaths of length i ending in simple nodes*

The last segment of a strict hyperpath of length i that ends in a simple node must be a simple path of length 1. The subpath leading to the last segment of such a strict hyperpath must be a strict hyperpath of length i-1. The antecedent of the last segment is either a simple node or a compound node.

Case 1: Simple Node Antecedent

If x is a simple penultimate node in one or more strict hyperpaths from $\underline{U}$ to v, then we have $\mathbf{D}_{i-1}[\underline{U}, x] = k_{h_{i-1}}(\underline{U}, x) \geq 1$ and $\mathbf{A}[x, v] = 1$. These entries will contribute $k_{h_{i-1}}(\underline{U}, x)$ strict hyperpaths to $\mathbf{A}_i[\underline{U}, v]$ (since $\mathbf{A}_i$ is obtained by multiplying $\mathbf{D}_{i-1}$ by $\mathbf{A}$). Let there be *r* strict hyperpaths from $\underline{U}$ to v where the penultimate node is a simple node. Further, let these hyperpaths pass through *q* distinct intermediate nodes x_m, m=1,...,q. Then $\mathbf{A}[x_m,v]$ is equal to 1 for all such x_m, and $r = \sum_{m=1}^{q} k_{h_{i-1}}(\underline{U}, x_m)$. As a result of multiplying $\mathbf{D}_{i-1}$ by $\mathbf{A}$, r will be the contribution to $\mathbf{A}_i[\underline{U},v]$ of strict hyperpaths with simple intermediate nodes.

Case 2: Compound Node Antecedent

For a strict hyperpath from $\underline{U}$ to v that falls in this category, the last segment is a simple path from a compound node X to the simple node v. The subpath leading to the last

segment is a strict hyperpath of length i-1 from $\underline{U}$ to the penultimate compound node X. Let there be *t* strict hyperpaths from $\underline{U}$ to v of length i where the penultimate intermediate node is a compound node. Let these hyperpaths pass through *s* distinct penultimate nodes X_m, m=1,...,s. Then, $\mathbf{A}[X_m,v]$ is equal to 1 for all such X_m, and $t = \sum_{m=1}^{s} k_{h_{i-1}}(\underline{U}, X_m)$. Therefore, after multiplying $\mathbf{D}_{i-1}$ by $\mathbf{A}$, *t* will be the contribution to $\mathbf{A}_i[\underline{U},v]$ of strict hyperpaths with compound intermediate nodes.

As a result, $\mathbf{A}_i[\underline{U}, v] = r+t$, which is the total number of strict hyperpaths of length i from $\underline{U}$ to v. Once again, $\mathbf{A}_i[\underline{U}, W] = 0$ where W is a compound node, since $\mathbf{A}[\cdot,W] = 0$ for all such W.

$\mathbf{B}_i$: *Simple and strict hyperpaths of length ≤2 ending in simple nodes*

We have $\mathbf{B}_i[\underline{U}, v] = \mathbf{A}^i[\underline{U}, v] + \mathbf{A}_i[\underline{U}, v] + \mathbf{B}_{i-1}[\underline{U}, v]$

$= k_{s_i}(\underline{U}, v) + k_{h_i}(\underline{U}, v) + k_{i-1}(\underline{U}, v)$

$= k_i(\underline{U}, v)$ where v is a simple node;

and $\mathbf{B}_i[\underline{U}, W] = \mathbf{A}^i[\underline{U}, W] + \mathbf{A}_i[\underline{U}, W] + \mathbf{B}_{i-1}[\underline{U}, W] = 0$ where W is a compound node.

Therefore, the matrix $\mathbf{B}_i$ is as defined.

$\mathbf{C}_i$: *Simple and strict hyperpaths of length ≤ i ending in simple or compound nodes*

$\mathbf{C}_i$ is obtained by performing a row revision on $\mathbf{B}_i$, and therefore includes all reachabilities to compound nodes as well. Since $\mathbf{B}_i$ includes all reachabilities indicated in $\mathbf{B}_1, ..., \mathbf{B}_{i-1}$, all hyperpaths of length less than i are also included.

$\mathbf{D_i}$: *Strict hyperpaths of length i ending in simple or compound nodes*

$\mathbf{D_i}$ is obtained by subtracting $\mathbf{C_{i-1}}$ and $\mathbf{A^i}$ from $\mathbf{C_i}$, so it follows that only strict hyperpaths of length i ending in simple or compound nodes are displayed in $\mathbf{D_i}$.

Therefore, by induction, the proposition is true for all i. •

Corollary 1: If $\mathbf{B_i}[\underline{U}, v] > 1$, then there potentially exists a redundant rule, or a chain of redundant rules, from $\underline{U}$ to v in the rule base.

Proof: $\mathbf{B_i}[\underline{U}, v] > 1$ indicates that there are multiple paths from $\underline{U}$ to v, which indicates the potential existence of a redundant rule or a chain of redundant rules. •

Corollary 2: If $\mathbf{B_i}[\underline{U}, v] \geq 1$, $\mathbf{B_i}[\underline{U}, w] \geq 1$, and v and w are mutually exclusive, then the rule sets that comprise $P\{\underline{U}; v\}$ and $P\{\underline{U}; w\}$ are conflicting.

Proof: From Proposition 3 it follows that both v and w are reached from $\underline{U}$, indicating contradiction. •

Corollary 3a: If $\mathbf{A^i}[\underline{U}, \underline{V}] \geq 1$ and $\underline{U} \supseteq \underline{V}$, then the rule sets that comprise the path(s) $P\{\underline{U}; \underline{V}\}$ are circular.

Proof: From Proposition 3, it follows that $\mathbf{A^i}[\underline{U}, \underline{V}] \geq 1$ implies the existence of one or more simple paths of length i from $\underline{U}$ to $\underline{V}$. Hence, if $\underline{U} \supseteq \underline{V}$, it indicates that a path exists from a hypernode to some subset of itself, implying circularity. •

Corollary 3b: If $\mathbf{A_i}[\underline{U}, \underline{V}] \geq 1$ and $U \supseteq V$, then the rules that comprise the path(s) $P\{\underline{U}; \underline{V}\}$ are circular.

Proof: $\mathbf{A_i}[\underline{U}, \underline{V}] \geq 1$ implies the existence of a strict hyperpath of length i from $\underline{U}$ to $\underline{V}$, indicating circularity if $\underline{U} \supseteq \underline{V}$. •

Proposition 2: If every element in a row in matrix **A** is zero, the row does not correspond to a goal variable, and the corresponding node is not a subset of a compound node, then it indicates a deadend premise.

Proof: An empty row in matrix **A** indicates that there are no paths or subpaths originating from the corresponding hypernode. If the hypernode is not a subset of a compound node, then it can only be the terminal node of some inference path. In that case, if it is not a goal variable, then it indicates a deadend premise. •

Proposition 3: If every element of a column in matrix **A** is zero, the column corresponds to a simple node, and the column does not correspond to an input variable, then it indicates an unreachable goal.

Proof: Since the rule base is unitized, every simple node that corresponds to an intermediate or an output variable must be the conclusion of some rule. In that case the column in **A** corresponding to such a node must have a nonzero entry. The absence of such an entry will indicate that the corresponding node is unreachable. •

3.6 Computational Complexity

We analyze the computational complexity of our technique in this section. While our technique can identify existence of certain types of errors that are not feasible with other graph based techniques [Agarwal and Tanniru, 1992; Nazareth, 1991; Nazareth, 1993], we show that the complexity of our technique is nevertheless of the same order as the other techniques.

In our algorithm, all of the computationally intense tasks involve matrix manipulations. In particular, we perform matrix additions and subtractions, matrix multiplications, and the Column Revision and Row Revision operations. The

computational complexity associated with each such operation, and the performance of the overall algorithm, are presented. The parameters that affect the computational complexity are:

n: the number of rules in the rule base;

r: the maximum number of attribute-value pairs in any hypernode; and

m: the length of the longest chain of rules (i.e. length of the longest inference path).

Matrix Addition and Subtraction

Since there are *n* rules, the size of all the matrices are no greater than $2n \times 2n$. Therefore, each matrix addition and subtraction involves at most $O(4n^2)$ operations.

Matrix Multiplication

Each matrix multiplication involves at most $O(8n^3)$ operations.

Column Revision

Performing Column Revision on a matrix requires the following steps. Each pair of hypernodes are examined to see if one is a superset of the other or not. If one hypernode is a superset of the other, then the entries of the corresponding row in the matrix are examined, and used to update the entries for the row corresponding to the subset. Since a hypernode can have upto *r* simple attribute-value pairs, comparing two hypernodes for subsumption requires $O(r^2)$ operations. Performing the actual updates for a row requires O(2n) operations. As each matrix can have at most 2n rows, the total number of pairwise comparisons required is $O(2n^2)$. Therefore, the worst case complexity associated with the Column Revision operation is $O(2n^2(r^2+2n)) = O(4n^3+ 2n^2r^2)$.

Row Revision

This operation is performed as follows. For each compound hypernode, we first identify the rows in a matrix that correspond to the simple nodes that constitute the

compound hypernode. Since this can require comparing the elements of every hypernode with the compound hypernode of interest, the number of operations is of the order $O(2nr^2)$. Then, the entries in rows that correspond to the simple nodes are examined, and used to update the row corresponding to the compound hypernode. Since *r* different rows may have to be examined, O(2nr) operations are required. The above operations have to be performed for each compound hypernode, of which there can be at most 2n instances (of course, in practice it will usually be substantially lesser than 2n). Therefore, the worst case performance is bounded by $O(2n(2nr^2 + 2nr)) = O(4n^2r^2 + 2n^2r)$.

Complete Algorithm

The computational complexity of matrix additions and subtractions are clearly dominated by the other operations and are subsequently ignored. The algorithm requires performing the Column Revision operation once, and the matrix multiplication and Row Revision operations during each iteration. The number of iterations required is equal to *m*. Therefore, the overall complexity is bounded by $O(m(8n^3 + 4n^2r^2 + 2n^2r) + 4n^3 + 2n^2r^2) \approx O(8mn^3 + 4mn^2r^2)$.

The above expression indicates the worst case complexity for the algorithm. In practice, the size of the matrix will usually be substantially smaller than $2n \times 2n$. Typically, the size of the adjacency matrix is expected to be a little more than $n \times n$. If a is the size of the adjacency matrix, then the complexity is bounded by $O(ma^3 + ma^2r^2)$. As noted in (Nazareth and Kennedy, 1991), the adjacency matrix for a directed graph representation is usually sparse for most applications. This is true for the directed hypergraph representation as well, and therefore more efficient matrix multiplication algorithms can be used. Since the row revision operation is repeated in every iteration, it can be made more efficient by performing the checks to identify the components of

compound nodes once, and then storing this information in a list for future reference. Therefore, the actual computational complexity will be considerably less than the worst case expression that we have presented.

CHAPTER 4
APPLICATION OF VERIFICATION ALGORITHM

In the preceding chapters, it was implicitly assumed that domain knowledge not explicitly specified in rules, is available when performing verification checks. In practice, the availability of such knowledge is essential for any automatic verification procedure. In this chapter, first, the different types of metaknowledge and their usefulness in the verification of rule-based expert systems are discussed. Next, an example is provided which demonstrates the verification procedure using the directed hypergraph technique. This technique is shown to overcome limitations of existing graphical techniques. Details about implementation of the verification algorithm conclude this chapter.

4.1 Metaknowledge Essential for Verification

The following types of metaknowledge are essential to detect Type 'A' errors (as defined in Section 2.3) in rule bases: synonyms, mutually exclusive attributes, and input (or observable) & goal attribute values.

4.1.1 Synonyms

A common phenomenon that influences all checks for consistency and completeness is the presence of synonyms as described in Section 2.1.1. To some extent, information about potential synonyms can be obtained by automated procedures (Trice and Davis, 1993). The procedure to detect potential synonyms is as follows. Rules are compared in a pairwise fashion and whenever two rules differ by only one variable, it is indicative of a potential synonym. After identifying potential synonyms, the expert has to

be consulted to finalize a list of synonyms. The problems posed by synonyms can be handled in the following way. Only one accepted form of the many synonyms of a variable is allowed to be used in the rule base. In other words, checking for synonyms constitutes a pre-processing stage which yields a synonym free rule base. On such a rule base various checks can be performed to ensure consistency and completeness. The information to be stored consists of a master list of synonyms. If a new rule is to be added, first the variables in the rules are checked with the list of synonyms, and should a synonym be found, its accepted form is incorporated in the additional rule.

4.1.2 Mutually Exclusive Attribute Values

In order to detect contradicting conclusions that result from a set of rules an automatic verification system must incorporate information regarding which attribute values are mutually exclusive. For example, consider the following two rules from an R&D Project Evaluation rule base:

Rule 1:	IF	Number of Alternative Uses is high
	THEN	Likelihood of Commercial Success is high.
Rule 2:	IF	Number of Alternative Uses is high
	THEN	Likelihood of Project Approval is low.

Suppose it is known that "Likelihood of Commercial Success being high" and "Likelihood of Project Approval being low" are mutually exclusive, then it can be inferred that the above two rules are contradictory since the same condition gives rise to mutually exclusive conclusions. In the absence of this kind of information, contradiction cannot be detected in rule bases.

The information on mutually exclusive attribute values can be categorized into two groups. The first group consists of mutually exclusive values of the same attribute. An example of this group is 'Likelihood of Commercial Success is High' and 'Likelihood of Commercial Success is Low.' The second group consists of mutually exclusive values of different attributes. An example of this group would be 'Likelihood of Commercial Success is High' and 'Likelihood of Project Approval is Low.' Both types of mutually exclusive attribute-value pairs must be stored in order for the verification procedure to detect contradiction.

4.1.3 Input & Goal Attribute Values

To detect missing rules either in the form of deadend premises or unreachable goals, information about input and goal attribute values is required. When an antecedent of a rule is not a part of conclusion of any other rule, it has to be checked whether it is an input variable. If it is not an input variable, then it is a case of deadend premise. Similarly, when a conclusion of a goal is not a part of condition of any other rule, it has to be checked with the list of goal attribute values before concluding that it is an unreachable goal.

4.2 An Illustrative Example

We demonstrate the verification procedure with the help of an example rule base. The rule base is adapted from a *project termination model* for R&D projects (Balachandra and Raelin, 1980). We have modified the existing knowledge base to highlight how different types of errors can occur in such a rule base.

Rule 1: IF the likelihood of technical success is high (a1)

THEN the degree of commitment of project leader is high(c1).

Rule 2: IF the likelihood of technical success is high (a1)

THEN continue funding (j1).

Rule 3: IF the likelihood of cost over-runs is high (b2)

THEN funding from industry is not available (f1).

Rule 4: IF the likelihood of cost over-runs is high (b2)

THEN priority classification is C (g1).

Rule 5: IF the likelihood of technical success is high (a1)

AND the likelihood of cost over-runs is high (b2)

THEN anticipated return on investment is low (e1).

Rule 6: IF anticipated return on investment is low (e1)

THEN priority classification is C (g1).

Rule 7: IF funding from industry is not available (f1)

THEN the project cannot be completed within three years (d1).

Rule 8: IF funding from industry is not available (f1)

AND priority classification is C (g1)

THEN discontinue funding (j2).

Rule 9: IF the project cannot be completed within three years (d1)

THEN the likelihood of cost over-runs is high (b2).

Rule 10: IF profitability is low (i2)

THEN discontinue funding (j2).

It is assumed that the observable facts (input variables) are "likelihood of technical success being high" and "likelihood of cost over-runs becoming high" (i.e. a1 and b2). The goal is to decide whether to continue funding the R&D Project or not (i.e. j1 or j2). These assumptions are necessary to detect deadend premises and unreachable goals in the rule base. The directed hypergraph representation of the above rule base is given in Figure 4.1.

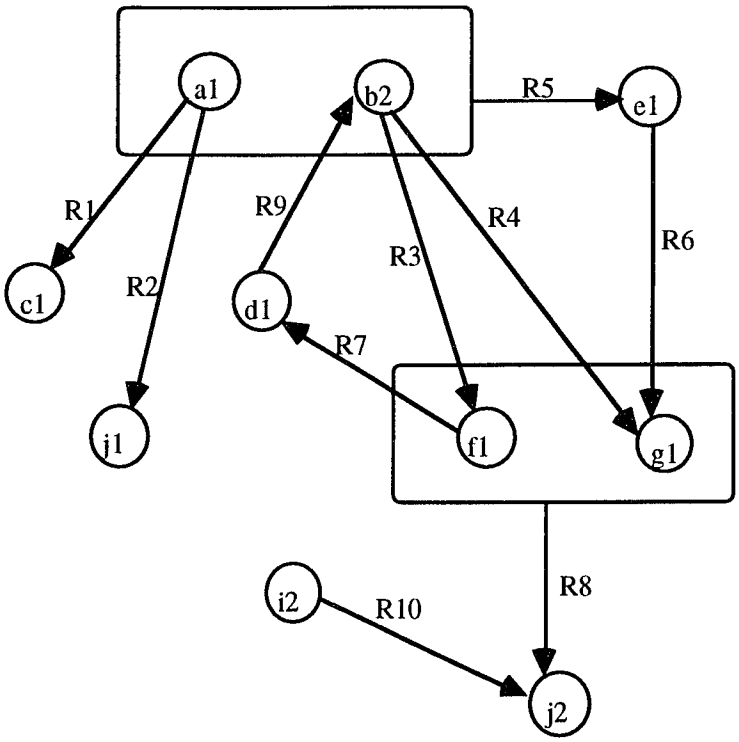

Figure 4.1 Directed Hypergraph for R & D Example

The errors that exist in the example rule base are shown below:

i. There are two distinct paths from (a1,b2) to (g1), SP{a1,b2; g1 \ «<a1,b2; e1>», «<e1; g1>»}, and HP{a1, b; g1 \ «<b2; g1>»}. Since paths from (a1,b2) to (j2) traverse through (g1), there are two distinct paths from (a1,b2) to (j2) as well.

ii. The path from (a1) to (j1) contradicts the paths from (a1,b2) to (j2).

iii. There is a circular path involving nodes (b2), (f1) and (d1).

iv. The node (c1) is a deadend premise.

v. The node (i2) is an unreachable goal.

We show how our verification procedure detects the above errors accurately. The adjacency matrix, **A**, is shown in Figure 4.2. This matrix displays all of the reachabilities directly implied by the rules, which are simple paths of length 1.

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1									1		1	
b2					1					1		
a1,b2				1								
e1										1		
f1						1						
d1		1										
f1,g1												1
i2												1
c1												
g1												
j1												
j2												

Figure 4.2 The Adjacency Matrix **A** for the R & D Example

Performing the column revision operation on **A** yields $\mathbf{B}_1$ as shown in Figure 4.3. $\mathbf{B}_1$ indicates all hyperpaths (simple paths as well as strict hyperpaths) of length 1 which end in simple nodes. The following strict hyperpaths of length 1 are established: HP{a1,b2 ; c1 \ «<a1; c1>»}, HP{a1,b2 ; f1 \ «<b2; f1>»}, HP{a1,b2 ; g1 \ «<b2; g1>»} and HP{a1,b2 ; j1 \ «<a1; j1>»}. The corresponding entries are highlighted in Figure 4.3.

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1									1		1	
b2					1					1		
a1,b2				1	1				1	1	1	
e1										1		
f1						1						
d1		1										
f1,g1						1						1
i2												1
c1												
g1												
j1												
j2												

Figure 4.3 The Matrix $\mathbf{B}_1$ for the R & D Example

The matrix $\mathbf{C}_1$, which is obtained by performing a row revision on $\mathbf{B}_1$, is shown in Figure 4.4. $\mathbf{C}_1$ establishes additional paths of length 1 that *end in compound nodes*. In this example, the new paths established are HP{b2; f1, g1 \ «<b2; f1>,<b2; g1>»} and HP{a1,b2 ; f1,g1 \ «<b2; f1>,<b2; g1>»} which are highlighted in Figure 4.4.

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1									1		1	
b2					1		1			1		
a1,b2				1	1		1		1	1	1	
e1										1		
f1						1						
d1		1										
f1,g1						1						1
i2												1
c1												
g1												
j1												
j2												

Figure 4.4 The Matrix $\mathbf{C}_1$ for the R & D Example

The matrix $\mathbf{D}_1$ displays all strict hyperpaths of length 1 that end in simple or compound nodes. This matrix is important for identifying *strict hyperpaths* of length 2 that end in simple nodes. Since such strict hyperpaths necessarily have a single rule in the last segment, the subpath corresponding to the first segment must be a strict hyperpath. Subsequently, when $\mathbf{D}_1$ is obtained by subtracting **A** from $\mathbf{C}_1$, it displays all the potential first segments of strict hyperpaths of length 2 that end in simple nodes. When the matrix $\mathbf{D}_1$ is multiplied with the matrix **A**, we obtain $\mathbf{A}_2$ which displays all strict hyperpaths of length 2 that end in simple nodes. The matrices $\mathbf{D}_1$ and $\mathbf{A}_2$ for this example are shown in Figure 4.5 and Figure 4.6, respectively. The matrix $\mathbf{A}_2$ displays the following four strict hyperpaths of length 2: HP{b2; j2 \ «<b2;f1>,<b2;g1>», «<f1,g1; j2>»}, HP{a1,b2; d1 \ «<b2;f1>», «<f1;d1>»}, HP{a1,b2; j2 \ «<b2;f1>,<b2;g1>», «<f1,g1; j2>»}, and HP{f1,g1; b2 \ «<f1; d1>», «<d1; b2>»}.

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1												
b2							1					
a1,b2					1		1		1	1	1	
e1												
f1												
d1												
f1,g1						1						
i2												
c1												
g1												
j1												
j2												

Figure 4.5 The Matrix $\mathbf{D}_1$ for the R & D Example

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1												
b2												1
a1,b2						1						1
e1												
f1												
d1												
f1,g1		1										
i2												
c1												
g1												
j1												
j2												

Figure 4.6 The Matrix $\mathbf{A}_2$ for the R & D Example

The matrix $\mathbf{A}^2$, obtained by multiplying **A** by itself, displays all the simple paths of length 2. There are five such simple paths in this example: SP{b2; d1}; SP{a1,b2; g1}; SP{f1; b2}; SP{d1; f1}; and SP{d1; g1}. The matrix displaying these paths is shown in Figure 4.7.

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1												
b2						1						
a1,b2										1		
e1												
f1		1										
d1					1					1		
f1,g1												
i2												
c1												
g1												
j1												
j2												

Figure 4.7 The Matrix $\mathbf{A}^2$ for the R & D Example

We have defined the matrix $\mathbf{B}_2$ to contain all hyperpaths of length less than or equal to 2 that end in simple nodes. From our definitions of the matrices $\mathbf{B}_1$, $\mathbf{A}^2$ and $\mathbf{A}_2$, it is

clear that adding all of them results in the matrix $\mathbf{B}_2$. In this example, there is a strict hyperpath of length 1 from (a1,b2) to (g1), as well as a simple path of length 2 from (a1,b2) to (g1). Matrix $\mathbf{B}_2$ identifies these multiple paths from (a1,b2) to (g1) by the entry 2 in the cell corresponding to row (a1,b2) and column (g1) in Figure 4.8.

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1									1		1	
b2					1	1				1		1
a1,b2				1	1	1			1	2	1	1
e1										1		
f1		1				1						
d1		1			1					1		
f1,g1		1				1						1
i2												1
c1												
g1												
j1												
j2												

Figure 4.8 The Matrix $\mathbf{B}_2$ for the R & D Example

The matrices $\mathbf{C}_2$, $\mathbf{D}_2$, $\mathbf{A}_3$, $\mathbf{A}^3$ and $\mathbf{B}_3$ are obtained as discussed in the algorithm. They are shown in Figures 4.9 through 4.13. Strict hyperpaths of length 2 that are identified as a result of Row Revision have been highlighted in matrix $\mathbf{C}_2$.

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1									1		1	
b2					1	1	1			1		1
a1,b2				1	1	1	2		1	2	1	1
e1										1		
f1		1				1						
d1		1			1		1			1		
f1,g1		1				1						1
i2												1
c1												
g1												
j1												
j2												

Figure 4.9 The Matrix $\mathbf{C}_2$ for the R & D Example

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1												
b2												1
a1,b2						1	1					1
e1												
f1												
d1							1					
f1,g1		1										
i2												
c1												
g1												
j1												
j2												

Figure 4.10 The Matrix $\mathbf{D}_2$ for the R & D Example

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1												
b2												
a1,b2		*										1
e1												
f1												
d1												1
f1,g1					*					*		
i2												
c1												
g1												
j1												
j2												

Figure 4.11 The Matrix $\mathbf{A}_3$ for the R & D Example

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1												
b2		*										
a1,b2												
e1												
f1					*					1		
d1						*						
f1,g1												
i2												
c1												
g1												
j1												
j2												

Figure 4.12 The Matrix $\mathbf{A}^3$ for the R & D Example

	a1	b2	a1,b2	e1	f1	d1	f1,g1	i2	c1	g1	j1	j2
a1									1		1	
b2		*			1	1				1		1
a1,b2		*		1	1	1			1	2	1	2
e1										1		
f1		1			*	1				1		
d1		1			1	*				1		1
f1,g1					*					*		1
i2												1
c1												
g1												
j1												
j2												

Figure 4.13 The Matrix $\mathbf{B}_3$ for the R & D Example

The iterative procedure stops when both $\mathbf{A}^i$ and $\mathbf{A}_i$ are null matrices. In this example, both $\mathbf{A}^4$ and $\mathbf{A}_4$ are null matrices and therefore $\mathbf{C}_3$ and $\mathbf{D}_3$ are the final matrices constructed. We have not shown $\mathbf{C}_3$ and $\mathbf{D}_3$ explicitly; however, they are evaluated to obtain $\mathbf{A}_4$. The different errors are detected by inspecting matrices $\mathbf{A}$ and $\mathbf{B}_3$ as follows:

i. Redundancy: $\mathbf{B}_3$ [a1,b2; g1] = 2 indicates the presence of two distinct paths from (a1,b2) to (g1). Similarly, $\mathbf{B}_3$ [a1,b2; j2] = 2 indicates the presence of two distinct paths from (a1,b2) to (j2).

ii. Contradiction: $\mathbf{B}_3$ [a1,b2; j1] = 1 indicates the path from (a1,b2) to (j1), while $\mathbf{B}_3$ [a1,b2; j2] = 2 indicates the two paths from (a1,b2) to (j2), implying contradiction.

iii. Circularity: Circular paths, identified by an asterisk, appear in $\mathbf{B}_3$[b2;b2], $\mathbf{B}_3$[f1;f1], $\mathbf{B}_3$[d1;d1], $\mathbf{B}_3$[a1,b2;b2], $\mathbf{B}_3$[f1,g1;f1] and $\mathbf{B}_3$[f1,g1;g1]. All of these occurences are due to the circular path involving nodes b2, f1 and d1. However, by the very nature of such paths, there may not be a unique originating or terminal node. Therefore, circular paths are separately flagged for each node that appears in such a path. In addition, since b2 and f1 are part of compound nodes, the corresponding compound nodes also display circularity.

iv. Deadend Premises: The row corresponding to the node (c1) has no entries in matrix **A**, which implies the presence of a deadend. This indicates the possibility that a rule including c1 in its antecedent is missing from the rule base.

v. Unreachable Goals: The column corresponding to i2 has no entries in matrix **A**. Since i2 is not an input variable, it implies an unreachable goal, i.e., node i2 cannot be reached in the rule base.

4.3 Comparison with Other Approaches

In Chapter 2, it was shown that in some instances, current graphical techniques are not capable of detecting errors accurately. In this section, we use the same examples used in Chapter 2 to show how the directed hypergraph algorithm can detect errors more accurately when compared with other approaches.

Example 1.

The directed hypergraph for the example in Chapter 2 is shown in Figure 4.14.

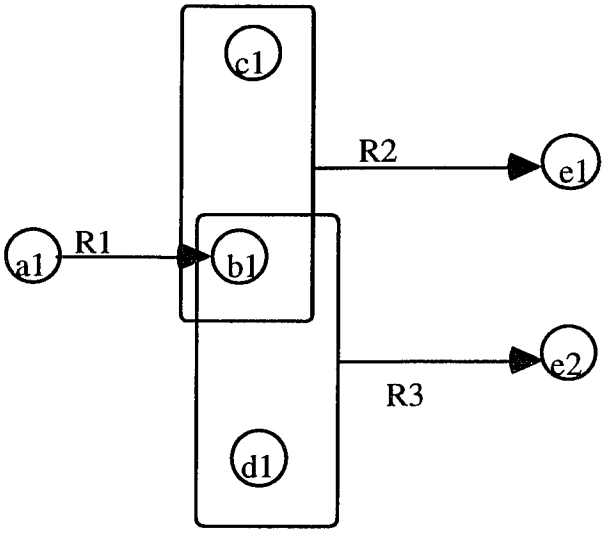

Figure 4.14 Directed Hypergraph for Example 1

In the directed hypergraph, no spurious reachabilities from a1 to e1 and e2 are indicated.

Example 2.

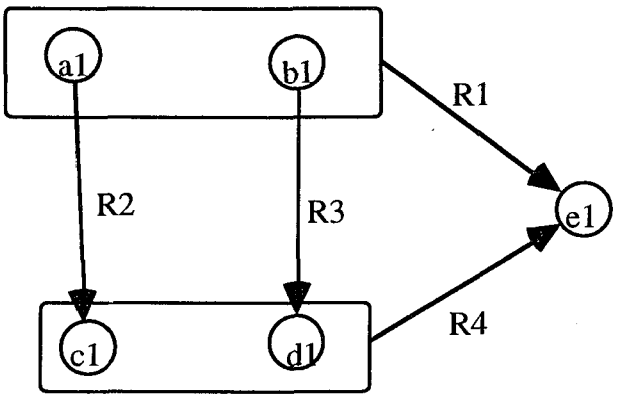

Figure 4.15 Directed Hypergraph for Example 2.

In the directed hypergraph, multiple paths from (a1, b1) to e1 are indicated.

4.4 Implementation of Algorithm

A computer program based on the directed hypergraph algorithm, PROGRAM HYPERGRAPH, is included as an Appendix. Information about the attribute-value pairs used in the rules are stored as integers. Hypernodes are stored as sets of integers in four different arrays. Array NODELABL contains all the hypernodes in the hypergraph. Input variables and goal variables are contained in arrays INSET and GOALSET respectively. Array DOMLABL stores information about mutually exclusive attribute-value pairs. Information regarding the adjacency matrix **A** is stored in a two dimensional array. Deadend premises are detected by checking the empty columns and then comparing the set of node labels of those columns with GOALSET. Unreachable goals are detected by checking the empty rows and then comparing the set of node labels of those rows with INSET.

Separate procedures have been written for all the matrix operations required to obtain the intermediate matrices and the final matrix $\mathbf{B}_i$. While the intermediate matrices $\mathbf{A}^i$ and $\mathbf{A}_i$ are being computed, circularity is detected by checking whether the node reached is a subset of the node from which the path has originated. Redundancy is detected by checking those entries with value ≥ 2 in the final matrix $\mathbf{B}_i$. Contradiction is detected by creating a set of those node labels that have a value ≥ 1 and comparing that set with the entries in array DOMLABL.

CHAPTER 5
DETECTION OF TYPE 'B' ERRORS

Detection of errors considered so far assumed that antecedents of rules are asserted one at a time. As stated earlier in Chapter 2, such errors are classified as Type 'A' errors. Type 'B' errors are caused when antecedents of several rules are instantiated simultaneously. Detection of Type 'B' errors is hard due to (i) difficulty in extending the techniques suitable for detection of Type 'A' errors to detect Type 'B' errors, and (ii) difficulty in selecting the appropriate combinations of input variables to be considered for simultaneous instantiation. In this chapter, an extension of the directed hypergraph algorithm that considers cases where several input variables occur simultaneously is discussed.

5.1 Simultaneous Instantiation

We have shown in Section 2.3 how a contradiction that shows up when multiple input variables are asserted might not show up when input variables that correspond to the antecedent of a single rule is true. The directed hypergraph for the example in Section 2.3 is shown in Figure 5.1. If the antecedents for rules R1 and R4 can be simultaneously true, then the rule base leads to contradictory conclusions.

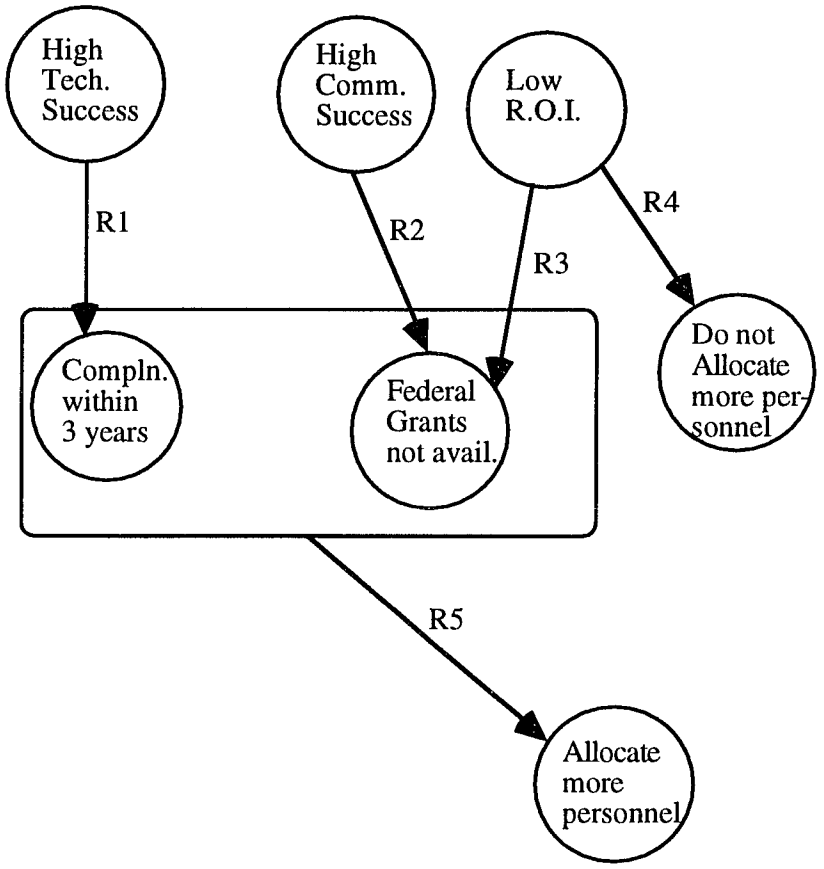

Figure 5.1 Directed Hypergraph for Rules 1 - 5

5.2 *Extension of the Hypergraph Algorithm*

The verification algorithm as discussed in Chapter 3 indicates all the Type 'A' errors in the final $\mathbf{B}_i$ matrix. We now discuss how the algorithm can be extended to detect Type 'B' errors as well. As discussed in Section 2, it is not possible to completely automate the detection of Type 'B' errors. Any verification procedure must have knowledge of valid sets of input variables that may be simultaneously asserted. We show

that with this knowledge, the directed hypergraph technique can be easily applied to detect such errors.

In the revised procedure, the directed hypergraph is modified such that new compound nodes corresponding to each valid combination of input attribute-values are added to the existing hypergraph. These additional compound nodes are different from the existing compound nodes in that they do not have any outgoing arcs connecting them to other nodes.

The adjacency matrix **A** corresponding to the directed hypergraph is modified such that there are additional rows and columns for each combination of input attribute-values that can occur simultaneously. The entries in rows corresponding to such combinations of input variables are zero since there are no rules with such antecedents. However, when $\mathbf{B}_1$ is computed, the column revision operation ensures that the new compound nodes reach all the simple nodes that are reached by their constituent hypernodes. $\mathbf{B}_i$ is obtained using the verification algorithm as defined earlier. **A** and $\mathbf{B}_i$ are then examined for errors in a similar manner.

Consider a rule base with the following rules:

Example 1:

Rule 1. a1 ——————> c1

Rule 2. b1 ——————> f1

Rule 3. b1 ——————> g2

Rule 4. c1 ——————> d1

Rule 5. c1 ——————> e1

Rule 6. d1 + e1 ——————> g1

Rule 7. d1 + e1 + f1 ——————> h1.

The attribute values g1 & g2 are assumed to be mutually exclusive. The directed hypergraph of this rule base is shown in Figure 5.2.

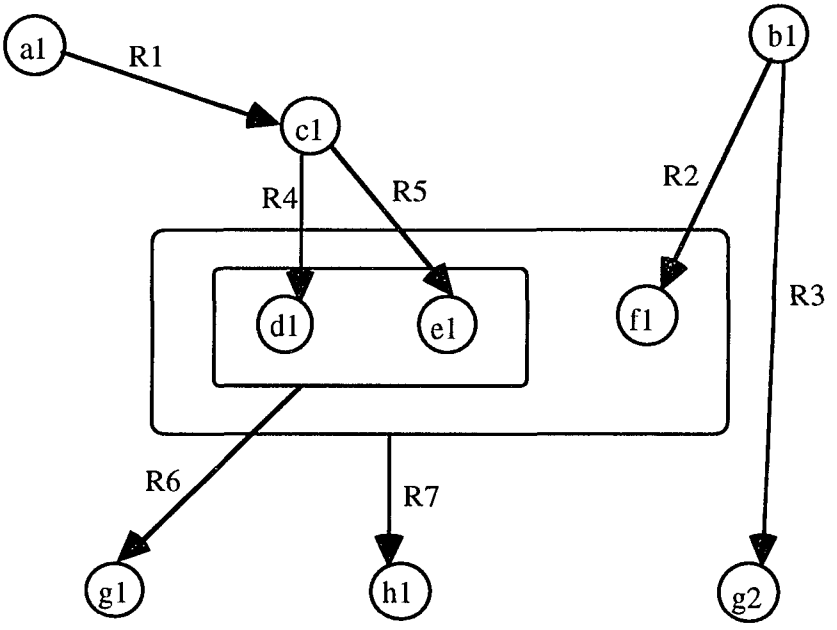

Fig. 5.2 Directed Hypergraph of Rules 1 - 8

In the above rules, there is no contradiction when either variables a1 or b1 occur individually. But when a1 and b1 occur simultaneously, it leads to a contradiction since both g1 and g2 are inferred from (a1, b1).

The revised hypergraph that takes the simultaneous instantiation of a1 and b1 into consideration is shown in Figure 5.3. Let **A**, shown in Figure 5.4, be the revised adjacency matrix for this case. The matrices $\mathbf{B}_1$, $\mathbf{B}_2$, and $\mathbf{B}_3$ obtained from applying the directed hypergraph algorithm are shown in Figures 5.5, 5.6, and 5.7. In matrix $\mathbf{B}_1$, the following paths from (a1, b1) are established: HP{a1,b1;c1}, HP{a1,b1;f1}, and HP{a1,b1;g2}. The new paths established from (a1, b1) that are indicated in matrix $\mathbf{B}_2$ are: HP{a1,b1;d1}, and HP{a1,b1;e1}. In matrix $\mathbf{B}_3$, paths HP{a1,b1;g1} and

HP{a1,b1;g2} are displayed indicating contradiction, since g1 and g2 are mutually exclusive.

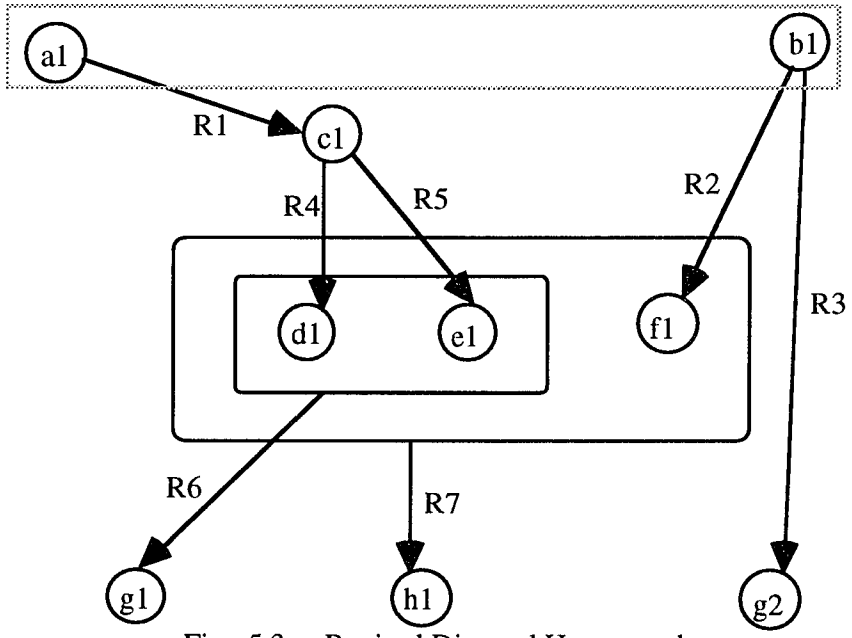

Fig. 5.3 Revised Directed Hypergraph

	a1	b1	a1, b1	c1	d1	e1	d1, e1	f1	d1, e1,f1	g1	h1	g2
a1				1								
b1								1				1
a1, b1												
c1					1	1						
d1												
e1												
d1, e1										1		
f1												
d1, e1,f1											1	
g1												
h1												
g2												

Fig. 5.4 Matrix A Modified to Include (a1, b1)

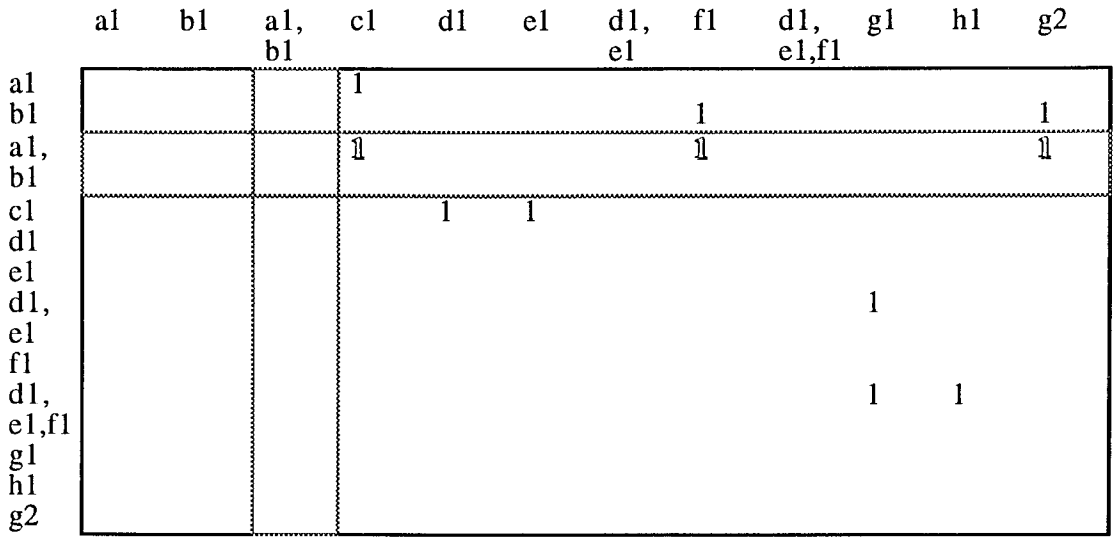

Fig. 5.5 Matrix $\mathbf{B}_1$ Derived from **A** Shown in Fig. 5.4

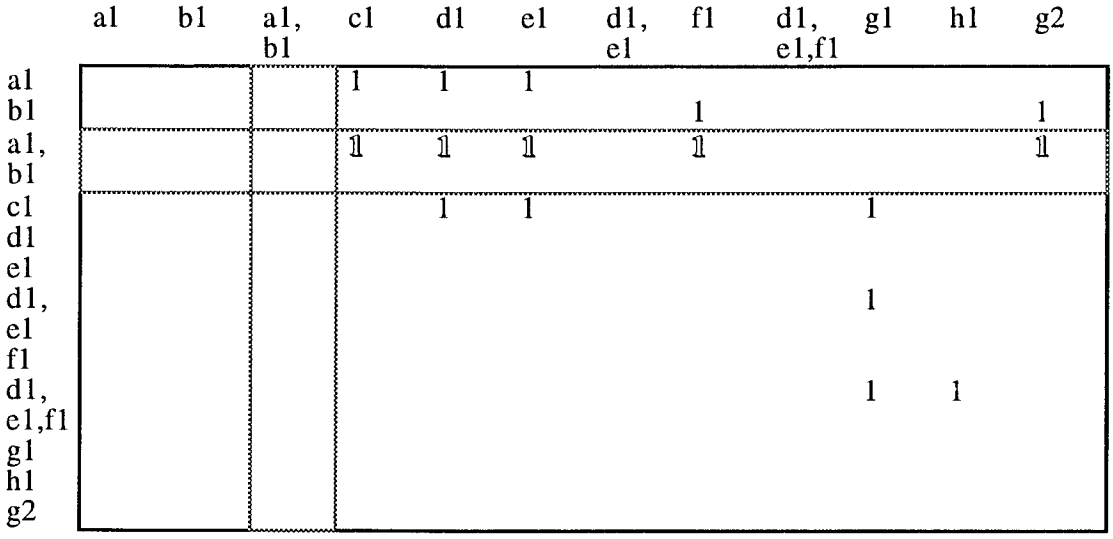

Fig. 5.6 Matrix $\mathbf{B}_2$ Derived from **A** Shown in Fig. 5.4

	a1	b1	a1, b1	c1	d1	e1	d1, e1	f1	d1, e1,f1	g1	h1	g2
a1				1	1	1				1		
b1								1				1
a1, b1				1	1	1		1		1	1	1
c1					1	1				1		
d1												
e1												
d1, e1										1		
f1												
d1, e1,f1										1	1	
g1												
h1												
g2												

Fig. 5.7 Matrix $\mathbf{B}_3$ Derived from **A** Shown in Fig. 5.4

The procedure to detect Type 'B' errors is shown to identify all the possible errors in an accurate fashion, as established by the following proposition.

Lemma 5.1: The addition of a row and a column to the adjacency matrix **A** for each valid combination of feasible input attribute values enables the accurate display of all paths originating from such combinations in matrix $\mathbf{B}_1$ and all subsequent matrices.

Proof: Each valid combination identified by the expert results in an additional row and column in the adjacency matrix **A**. All entries in these new rows are zeroes since there are no rules which have these particular combinations of input variables as antecedents. In order to obtain B1, the operation column-revision is performed on **A**. As a result, if any subset of a new compound node is the antecedent of a rule, then the column-revision operation ensures that there is a path from the compound node to the conclusion of that rule.

Thus, although there is no rule that has the new compound node as as its antecedent, all reachabilities from that compound node are captured by the revised procedure. Once all such reachabilities of length 1 have been captured, all reachabilities of longer lengths are identified by the verification algorithm.

5.3 Computational Complexity

As shown in Chapter 3, the expression for the computational complexity is $O(8ma^3 + 4ma^2r^2)$. The revised procedure is similar to the earlier one. In the revised procedure, *a* represents the number of rows in the revised adjacency matrix, and *r* represents the maximum number of attribute-value pairs in any hypernode (including the compound nodes of feasible combinations of input variables). Thus the increase in complexity depends on the number of valid combinations that are provided by the expert. When the number of such combinations is small compared to the size of the original matrix, the increased computational requirements is minimal. When the number of such combinations is relatively large, they can be processed in batches where in each batch, a part of the valid combinations is examined separately.

CHAPTER 6
DECOMPOSITION OF RULE BASES

Expert systems that are designed to solve complex problems are large. Some large expert systems have been reported to contain about 10,000 rules (Segev and Zhao, 1994). When rule bases contain thousands of rules, it is difficult to use verification techniques that need to consider all the rules simultaneously. For instance, the adjacency matrix of a rule base with 1000 rules would consist of upwards of 10^6 elements that would be difficult to store and manipulate for verification purposes.

A common way to solve complex problems is by decomposing such problems into subproblems that are smaller (Simon, 1981). This phenomenon is observed in rule base development and has been documented by many researchers (Jacob and Froscher, 1986; Agarwal and Tanniru, 1992a). It would be pragmatic to decompose large rule bases into smaller sets of rules that can be independently verified. There is empirical evidence to indicate that sets of rules in large rule bases are sufficiently separated to allow the rule base to be decomposed into smaller sets (Jacob and Froscher, 1986). Most existing verification techniques explicitly or implicitly assume that such decomposition is feasible (Cragun and Steudel, 1987). In this chapter, we show that the directed hypergraph representation allows us to characterize when a set of partitions for a composite rule base can be separately verified, without misdiagnosing errors that may appear in the rule base. For the sake of illustration, we first present a simple partitioning scheme, where adjacent partitions share nodes that correspond to different values of the same attribute. We discuss how a rule base consisting of such partitions can be verified by examining the different partitions sequentially. Next, we provide the general characterization of partitions that share nodes

corresponding to different values of several attributes. In this dissertation, it is assumed that the expert is able to decompose the rule base into smaller partitions based on the context of rules.

6.1 Partitions with Single Shared Variables

This decomposition refers to instances where adjacent partitions of rules share nodes that correspond to different values of exactly one attribute. Consider two adjacent partitions P1 and P2 having a common attribute A. Such partitions can be verified separately if the following conditions hold:

(i) If a value of a variable is in some partition, then all other values of that attribute must also be in that partition. However, values of the shared attribute can be present in both the adjacent partitions.

(ii) Any path from a hypernode U in P1 to a simple node v in P2 must go through some value of A (say a1);

(iii) P{U; a1} must lie entirely in P1; and P{a1; v} must lie entirely in P2.

When a path consists of rules in three or more partitions, then the above conditions must be satisfied for each pair of adjacent partitions. In addition, if there is a path from a hypernode U in partition P_1 to a simple node y in partition P_r, that traverses through partitions P_2,..., P_r, then all paths from U to y must also traverse through partitions P_2,..., P_r.

For example, consider the following rules adapted from an R&D Project Evaluation Model (Balachandra and Raelin, 1980):

Rule 1:	IF	Need for Innovation is High (a1)
	AND	Number of Technologies Required is Low (b2)
	THEN	Likelihood of Technical Success is High (c1).

Rule 2:	IF	Need for Innovation is High (a1)
	AND	Number of Technologies Required is High (b1)
	THEN	Likelihood of Technical Succ. is Medium (c2).
Rule 3:	IF	Anticipated R.O.I is High (d1)
	AND	Market Competition is Medium (e2)
	THEN	Likelihood of Commercial Succ. is High (f1).
Rule 4:	IF	Anticipated R.O.I is Low (d2)
	AND	Market Competition is Medium (e2)
	THEN	Likelihood of Commercial Succ. is Medium (f2).
Rule 5:	IF	Likelihood of Technical Success is High (c1)
	AND	Likelihood of Commercial Succ. is High (f1)
	THEN	Project can be completed within 3 years (g1).
Rule 6:	IF	Likelihood of Technical Success is Low (c2)
	AND	Likelihood of Commercial Succ. is Medium (f2)
	THEN	Project cannot be compltd. within 3 yrs (g2).
Rule 7:	IF	Project can be completed within 3 years (g1)
	THEN	Fund the Project (h1).
Rule 8:	IF	Project cannot be compltd within 3 yrs (g2)
	THEN	Do Not Fund the Project (h2).

The directed hypergraph of the above rule base is shown in Figure 6.1. The above rule base can be decomposed into three partitions as shown in Figures 6.2 through 6.4.

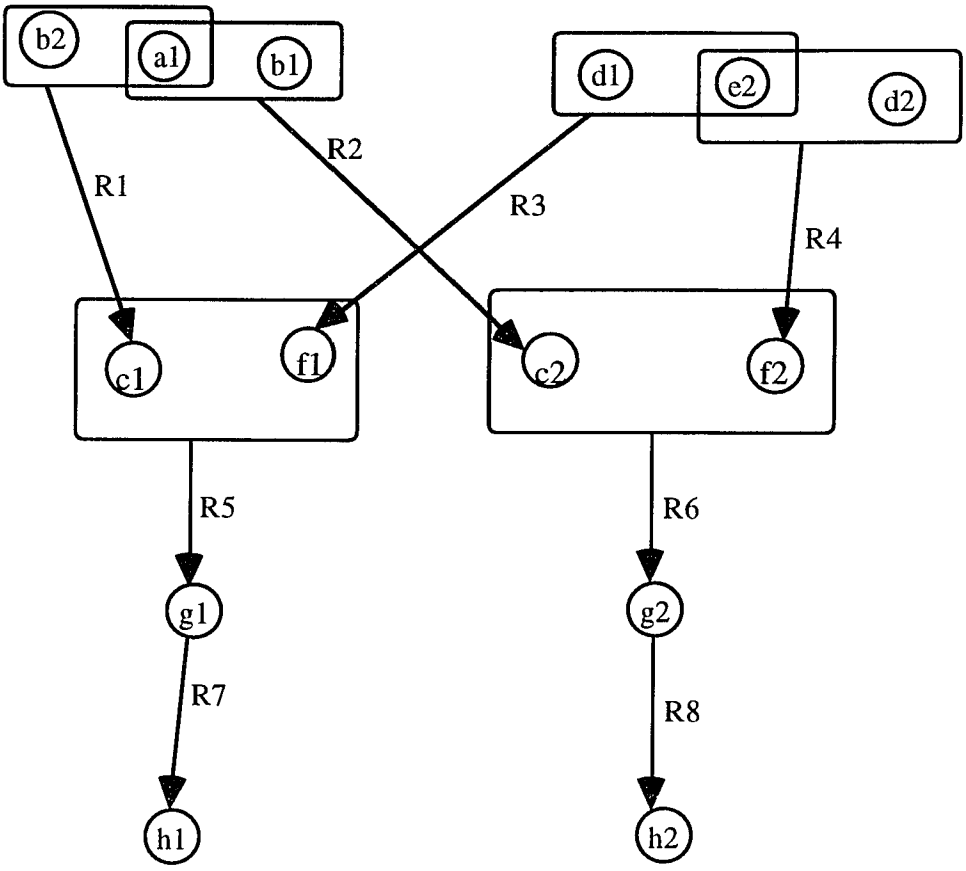

Figure 6.1 Directed Hypergraph of Rules 1 - 8

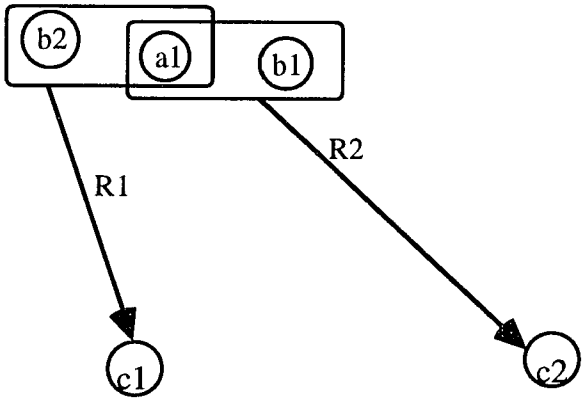

Figure 6.2 Partition P1 containing Rules 1 and 2

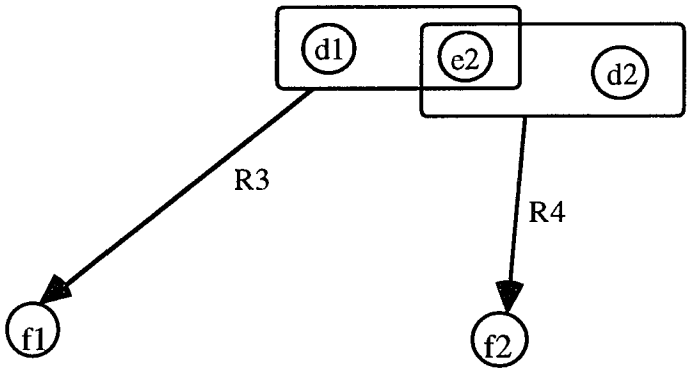

Figure 6.3 Partition P2 containing Rules 3 and 4

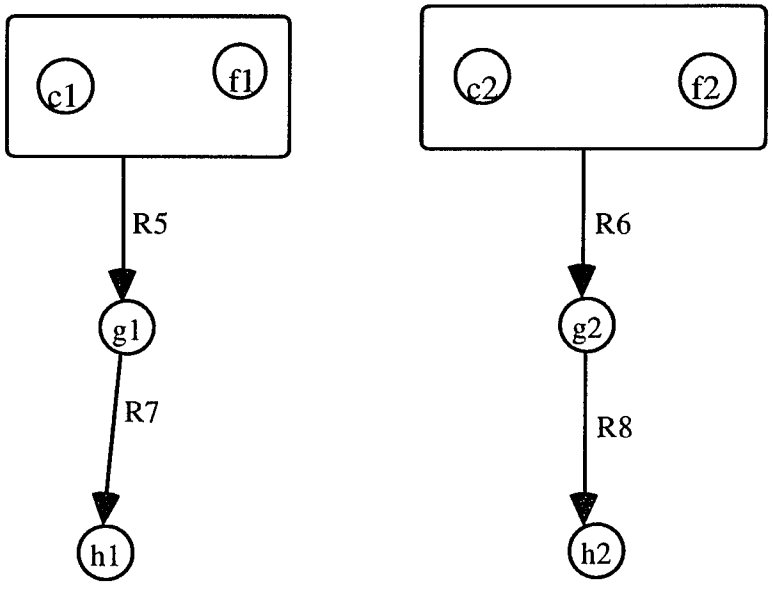

Figure 6.4 Partition P3 containing Rules 5 - 8

The partitions P1 and P3 share a single attribute C. Similarly, only one attribute F is shared between partitions P2 and P3.

6.1.1 Verification of Partitions with Single Shared Variables

If the conclusion of rules in one partition appears in the antecedents of rules of another partition, then the former partition precedes the latter. In the case being illustrated, this is necessary to identify circular paths which exist through a chain of rules across partitions. In the more general case discussed later, this ordering of partitions enables detection of errors propagated across partitions.

The precedence ordering between adjacent partitions is determined in the following way: the partition in which the common attribute is a conclusion precedes the partition in which the common attribute is part of the antecedent. For example, referring to the rule base in Figure 6.1, P1 precedes P3 because attribute C is a conclusion in P1, and an antecedent for a rule in P3. Precedence ordering across partitions is identified before starting the verification of the different partitions.

Based on the precedence ordering, partitions are checked for errors. When a common attribute between partitions leads back to a partition which has been verified earlier, then it indicates circularity. The verification procedure stops after all partitions have been verified. Lemma 6.1 and 6.2 show that the entire rule base can be accurately verified by verifying the separate partitions as discussed.

Lemma 6.1: If all the partitions done under Scheme 1 are error free, then the composite rule base is also error free.

Proof: Let P1 and P2 be two error free partitions of a composite rule base which share a single attribute A that has values a1 and a2. Assume that a contradiction has occured in the

following way. Assertion of a hypernode U in partition P1 results in inferring v1 and v2 in partition P2 and v1, v2 are mutually exclusive. This is possible iff any one of the following cases are true: Case (i) P{U;v1} and P{U;v2} traverse through the same node corresponding to a particular value of the shared attribute or Case (ii) P{U;v1} and P{U;v2} traverse through different nodes corresponding to different values of the shared attribute.

Case (i) cannot be true since P2 is error free. Case (ii) cannot be true since P1 is error free. Therefore, paths from U in P1 to v1 and v2 in P2 cannot exist.

In a similar way, it can be shown that redundant paths cannot exist across error free partitions. From this, it follows that if the partitions are error free, then the whole rule base is also error free. The same logic holds good for multiple partitions of the composite rule base.

Lemma 6.2: If the composite rule base is error free, then all the partitions done under Scheme 1 are also error free.

Proof: Partitions are subsets of the whole rule base. Therefore, if the entire rule base is error free, then the partitions are also error free.

6.2 Partitions with Mutiple Shared Variables

In this is general case, adjacent partitions of rules share nodes that correspond to different values of several attributes. Consider two adjacent partitions P1 and P2 that share a common attribute set Z. Such partitions can be verified separately if the following conditions hold:

(i) If a value of a of a variable is in some partition, then all other values of that attribute must also be in that partition. However, values of the shared attribute will be present in both the adjacent partitions.

(ii) Any path from $\underline{U}$ in P1 to v in P2 must go through $\underline{W} \subset \underline{Z}$.

(iii) $P\{\underline{U}; \underline{W}\}$ must lie entirely in P1; and $P\{\underline{W}; v\}$ must lie entirely in P2.

When a path consists of rules in three or more partitions, then the above conditions must be satisfied for each pair of adjacent partitions. In addition, if there is a path from a hypernode $\underline{U}$ in partition P_1 to a simple node y in partition P_r, that traverses through partitions P_2,..., P_r, then all paths from $\underline{U}$ to y must also traverse through partitions P_2,..., P_r.

For example, consider a rule base containing the following rules:

R1. a1 ---> e1

R2. a1 + b1 ---> g1

R3. c1 ---> f2

R4. d1 ---> g2

R5. e1 ---> f1

R6. f1 ---> i1

R7. g1 ---> i2

R8. f2 ---> h1

R9. g2 ---> j1

R10. h1 + j1 ---> k1

R11. j1 ---> l1.

The directed hypergraph of the above rule base is shown in Figure 6.5. Partitions P1 and P2 sharing multiple variables are shown in Figures 6.6 and 6.7 respectively.

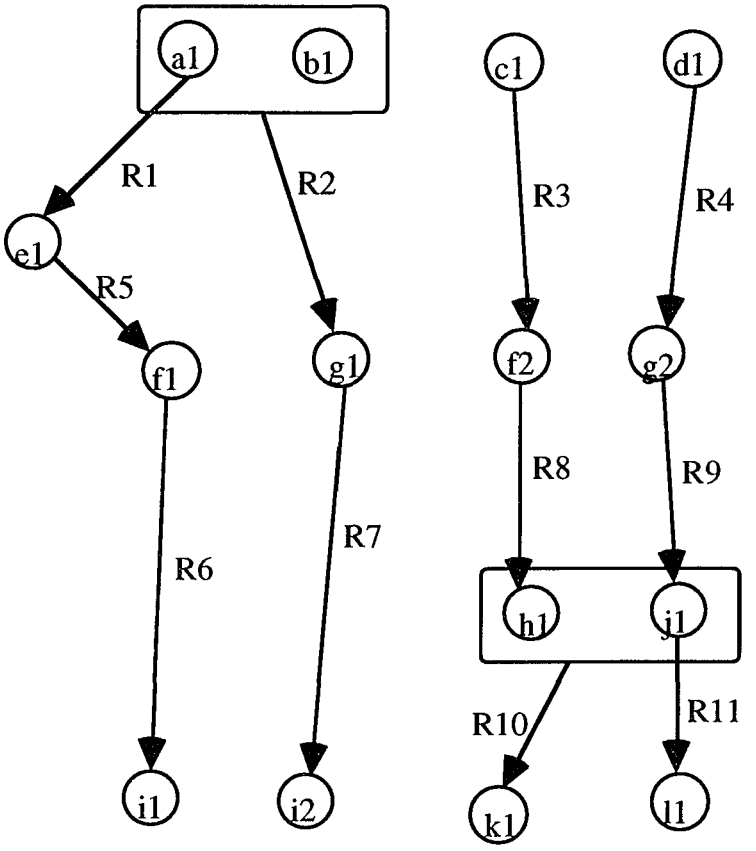

Figure 6.5 Directed Hypergraph of Rules 1 -11

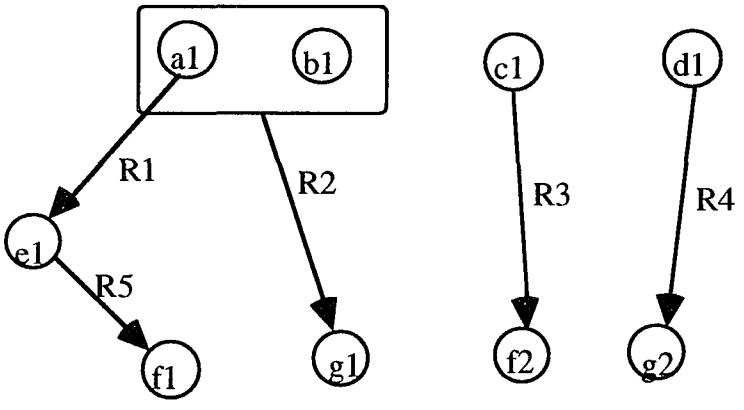

Figure 6.6 Directed Hypergraph of Partition P1

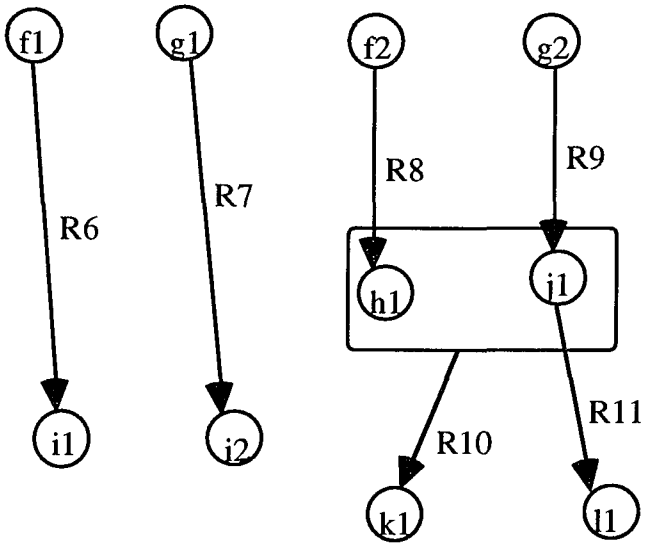

Figure 6.7 Directed Hypergraph of Partition P2

6.2.2 Verification of Partitions with Multiple Shared Variables

To start with, precedence ordering of partitions is done similar to the verification procedure for partitions with single shared variables. Based on the precedence ordering, the partitions are verified as follows:

Step 1. Rank all partitions fom 1 to i based on precedence.
Initialize k =1.

Step 2. While k ≤ i do steps 3 - 6.

Step 3. Verify Partition k.

Step 4. If k = i stop. Otherwise, identify the compound conclusion nodes (indicated in $\mathbf{C}_i$) in the shared attribute set between partitions k and k+1.

Step 5. Modify the hypergraph of Partition k+1 to indicate the compound conclusion nodes.

Step 6. Set k = k + 1.

Referring to the composite rule base in Figure 6.5, partition P1 is verified first. Matrix $\mathbf{C}_i$ will indicate (f1, g1) as a compound node, since (a1, b1) can reach both f1 and g1. Therefore, the directed hypergraph for partition P2 is modified to indicate that compound node. The modified directed hypergraph is shown in Figure 6.8. When partition P2 is verified, the contradiction in the rule base is detected due to (f1, g1) reaching both i1 and i2 that are mutually exclusive.

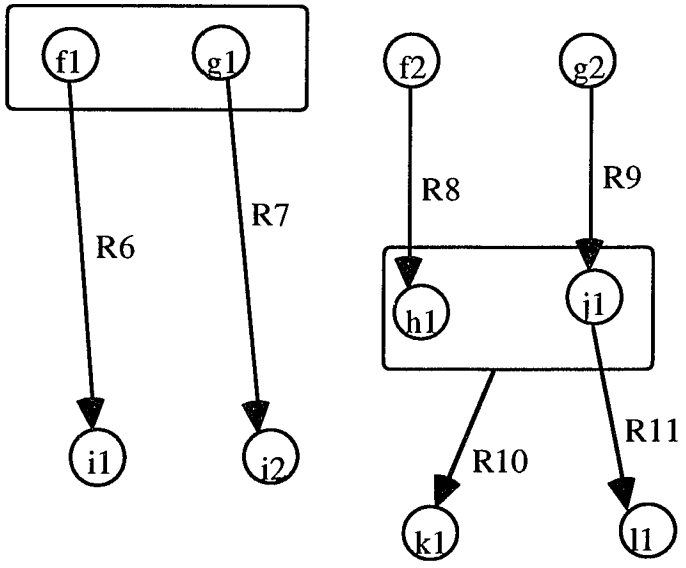

Figure 6.8 Modified Hypergraph of Partition P2

Lemma 6.3: If all partitions sharing multiple variables are error free, then the composite rule base is also error free.

Proof: Let P1 and P2 be two error free partitions of a composite rule base which share a set of attributes Z. Assume that a contradiction has occured in the following way.

Assertion of a hypernode U in partition P1 results in inferring v1 and v2 in partition P2 and v1, v2 are mutually exclusive. This is possible iff any one of the following sets of paths exist: (i) paths {U; W}, {W; v1}, and {W; v2} where $W \subset Z$; or (ii) paths {U; W}, {U; X}, {W; v1}, {X; w2} where $W, X \subset Z$.

Set of paths in (i) cannot exist since P2 is error free. Set of paths in (ii) cannot exist since the modification step in the procedure ensures that (W, X) is a compound node in P2. Therefore, paths from U in P1 to v1 and v2 in P2 cannot exist.

In a similar way, it can be shown that redundant paths cannot exist across error free partitions. From this, it follows that if the partitions are error free, then the composite rule base is also error free.

Lemma 6.4: If the composite rule base is error free, then all the partitions done under Scheme 2 are also error free.

Proof: Partitions are subsets of the whole rule base. Therefore, if the entire rule base is error free, then the partitions are also error free.

6.3 Reduction in Computational and Storage Resources

When a large rule base is partitioned into smaller sets of rules, the adjacency matrices for the smaller sets of rules are much smaller than that for the original rule base. Since the computational and resource storage requirements are directly dependent on the size of the adjacency matrices used in the verification process, the decomposed rule base can be verified much more efficiently.

For example, consider a rule base having 1000 rules (n), 10 levels (m), and a value of r (maximum number of attribute-value pairs in a hypernode) equal to 5. As shown earlier, the computational complexity of verifying this composite rule base is of the order

$O(mn^3 + mn^2r^2) = O(10^{10})$. By decomposing the rule base into say 10 partitions each containing 100 rules and 10 levels, the computational complexity is reduced to $O(10^7)$. Similar efficiencies would hold for storage requirements as well.

CHAPTER 7
CONCLUSION

Rule-based representation techniques have become popular for storing and manipulation of domain knowledge in expert systems. It is important that systems using such a representation are verified for accuracy before implementation. In this research, it has been shown how hypergraphs provide an intuitively appealing and theoretically rigorous framework to perform verification checks for rule bases. An algorithm is presented that uses the adjacency matrix for the hypergraph to detect the different types of errors. The algorithm is shown to accurately detect errors that manifest themselves over a chain of rules. At the same time, it does not indicate any instance spuriously. The algorithm is shown to be reasonably efficient.

The algorithm presented can be used to detect occurrences of all the errors once the final $\mathbf{B}_i$ matrix is obtained. However, in practice, it is recommended that errors that are identified at each iteration are corrected before continuing the detection process over longer inference paths. For example, consider the case where a contradiction is detected over two inference paths of length 3. If these conflicting paths are subpaths of longer inference paths, then the longer paths will also indicate contradictions. However, once the cause for conflict in the original paths of length 3 is eliminated, all the other paths will no longer be in conflict. The checks for conflict, redundancy and circularity can therefore be performed on the $\mathbf{B}_i$ matrix after each iteration, before proceeding any further. The nature of these checks will be the same as discussed for the final $\mathbf{B}_i$ matrix.

The algorithm has been extended to include cases where reachabilities are considered from a collection of hypernodes that correspond to antecedent clauses of

different rules. A practical difficulty associated with this problem is that there are an exponentially large number of different combinations of such hypernodes that may need to be considered. Thus, a completely automated procedure is not feasible for the verification process. A more pragmatic approach is to obtain information from experts that can help to identify the feasible combinations of hypernodes for which the reachabilities are important. It is shown that if the feasible sets of assertions are indicated beforehand, the verification process can identify errors with very little increase in computational effort.

When rule bases are large, it is not possible to verify all the rules simultaneously. In such situations, it is necessary to decompose them into smaller sets of rules in order to perform verification checks.. The directed hypergraph approach helps in characterizing the partitions of composite rule bases that can to be verified separately. The verification procedure based on the hypergraph has been enhanced to detect errors in large rule bases which can be decomposed into smaller groups.

Elicitation of valid sets of rule antecedents that are instantiated simultaneously is not addressed in this dissertation. While this research has focused on verification, given the decomposed set of rules, an important related issue would be to identify the decomposition process. Another limitation is the inability to identify the exact rule that causes the error.

An issue for future research is extension of the hypergraph verification technique to identify the paths that lead to the error. Identification of the inference paths will lead to locating the erroneous rule in the rule base. Another issue for further study is enhancement of the hypergraph technique to cover incremental addition of rules.

REFERENCES

Agarwal, R. and Tanniru, M. (1992), "A Petri-Net Based Approach for Verifying the Integrity of Production Systems," *International Journal of Man-Machine Systems,* 36, 447-468.

Agarwal, R and Tanniru, M (1992a), "A Structured Methodology for Developing Production Systems," *Decision Support Systems,* 8, 483-499.

Andert, E. P.(1992), "Integrated Knowledge-based System Design and Validation for solving Problems in Uncertain Environments," *International Journal of Man-Machine Studies,* 36, 357-373.

Bundy, A. (1988), "How to Improve the Reliability of Expert Systems," In D.S. Moralee, Ed. *Research and Development in Expert Systems IV,* 3-17, Cambridge University Press.

Boehm, B. W. (1984), "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software Journal,* January 1984.

Botten, N., Kusiak, A. and Raz, T. (1989), "Knowledge Bases: Integration, Verification, and Partitioning," *European Journal of Operational Research,* 42, 111-128.

Chung, Y (1989), "Verification of Completeness and Consistency in Knowledge Based Systems : An E R Approach," Unpublished M.S.Thesis, Louisiana State University.

Cragun, B. and Steudel, H. (1987), "A Decision-table based Processor for Checking Completeness and Consistency in Rule-based Expert Systems," *International Journal of Man-Machine Studies,* 26, 223-227.

Geissman, J. R. and Schultz, R. D. (1988), "Verification and Validation of Expert Systems," *AI Expert,* Vol. 3, No. 2, 26-33.

Ginsberg, A. (1987), "A New Approach to Checking Knowledge Bases for Inconsistency and Redundancy," *Proceedings of the Third Annual Expert Systems in Government Conference,* 102-111, Washington, D.C.

Ginsberg, A. (1988), "Knowledge-Base Reduction: A New Approach to Checking Knowledge Bases for Inconsistency and Redundancy," *Proceedings of the National Conference on Artificial Intelligence,* 589-595.

Hayes-Roth, F (1985), "Rule Based Systems," *Communications of the ACM 28,* No. 9, 921-932.

Ignizio, J. P. (1991), *An Introduction to Expert Systems: The Development and Implementation of Rule-Based Expert Systems,* McGraw-Hill, Inc.

Jacob, R. and Froscher, J. (1986), "Developing a Software Engineering Methodology for Rule-Basd Systems," Proceedings of the Fall Joint Computer Conference, Dallas, TX, 179 - 183.

Kang, Y. and Bahill, T. (1990), "A Tool for Detecting Expert System Errors," *AI Expert,* Vol. 5, No.2, 42-51.

Marcot, B. (1987), "Testing Your Knowledge Base," *AI Expert,* Vol. 2, No. 8, 42-47.

Marek, W. (1987), "Completeness and Consistency in Knowledge Base Systems," *Proceedings of the First International Conference onExpert Database Systems,* 119-126.

Mettery, W. (1987), "An Assessment of Tools for Building Large Knowledge-Based Systems," *AI Magazine,* 8(4), 81-89.

Nazareth, D. L. (1989), "Issues in the Verification of Knowledge in Rule-based Systems," *International Journal of Man-Machine Studies,* 30, 255-271.

Nazareth, D. L. and Kennedy, M. H. (1991), "Verification of Rule-Based Knowledge using Directed Graphs," *Knowledge Acquisition,* 3, 339 - 360.

Nazareth, D. L. (1993), "Investigating the applicability of Petri Nets for Rule-Based System Verification," *IEEE Transactions on Knowledge and Data Engineering,* Vol. 4, No. 3, 402 - 415.

Nguyen, T. A., Perkins, W. A., Laffey, T. A., and Pecora, D. (1985), "Checking an Expert System's Knowledge Base for Consistency and Completeness," *Proceedings of the 9th IJCAI,* 374-378.

Nguyen, T (1987), "Verifying Consistency of Production Systems," *Proceedings of the IEEE Conference on Artificial Intelligence Applications,* 4-8.

Nguyen, T.A., Perkins, W.A., Laffey, T.J., and Pecora, D. (1987), "Knowledge Base Verification," *AI Magazine,* 69-75.

Pederson, K. (1989), "Well-Structured Knowledge Bases," *AI Expert,* 4, 44-55.

Preece, A. D. (1993), "A New Approach to Detecting Missing Knowledge in Expert System Rule Bases," *International Journal of Man-Machine Systems,* 38, 661-688.

Radwan, A. E., Goul, M., O'Leary, T.J., and Moffitt, K.E. (1989), "A Verification Approach for Knowledge-Based Systems," *Transportation Research-A,* Vol. 23A, No. 4, 287 - 300.

Segev, A. and Zhao, L.Z. (1994), "Rule Management in Expert Database Systems," Management SCience, Vol. 40, No. 6, 685-707.

Simon, H. A. (1981), *The sciences of the Artificial*, MIT Press, Cambridge, MA.

Stachowitz, R. A. and Combs, J. B. (1986), "Validation of Expert Systems," *Proceedings of the Hawaii International Conference on Systems Sciences,* Kona, Hawaii, January 6 - 9, 1987.

Suwa, M., Scott, A., and Shortliffe, E. (1982), "An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System," *AI Magazine*, 16-21.

Trice, A and Davis, R (1993), "Heuristics for Reconciling Independent Knowledge Bases," *Information Systems Research,* Vol. 4, No. 3, 262-288.

Zahedi, F. (1993), Intelligent Systems for Business, Wadsworth, Belmont, CA.

APPENDIX

PROGRAM HYPERGRAPH

```
(* This program detects the following errors that can occur in rule bases: deadend
premises, unreachable goals, redundancy, contradiction and circularity. *)
PROGRAM HYPERGRAPH (INPUT,OUTPUT,NODEINFO,DOMINFO,OBSINFO,
                                        GOALINFO, MATINFO, RESULT);
CONST
(* RL and CL refer to row limit and column limit respectively. *)
  RL=100;
  CL=100;
TYPE
(* Attribute-value pairs are represented as integers from 0 to 100. Each node in the
hypergraph is considered as a set of ontegers. *)
  NODESET = SET OF 0..100;
  NODEARY = ARRAY(.1..100.) OF NODESET;
  DOMARY = ARRAY(.1..100.) OF NODESET;
(* Matrices are declared as two dimensional arrays of integers. *)
  DHARY = ARRAY(.1..RL, 1..CL.) OF INTEGER;
VAR
 I, J, IA, IB, OI, OA,GI, GA, IV, GV, AA,BB,CC,DD  : INTEGER;
 NODELABL, OBSLABL, GOALABL : NODEARY;
 DOMLABL : DOMARY;
(* Information about the rule base is given through the following text files;
        NODEINFO: set of all nodes,
        OBSINFO: set of all observable variables,
        GOALINFO: set of all goal variables,
        DOMINFO: set of all mutually exclusive variables, and
        MATINFO: adjacency matrix A. *)
 NODEINFO, RESULT : TEXT;
 DOMINFO, OBSINFO, GOALINFO, MATINFO : TEXT;
```

```
 NI, DI, ML, MI, MJ, CMP : INTEGER;
 INSET, GOALSET, CMPSET, RCH, RRSET, CIRCSET : NODESET;
 ASUM, SUM1, SUM2, CD, K : INTEGER;
 AARY, ASARY, ASPARY, AHARY, BARY, BPARY,
 CARY, CPARY, DARY : DHARY;

(* The following procedure computes matrix B from matrix A after performing column
revision. *)
PROCEDURE COLREV( AARY:DHARY; VAR BARY:DHARY);
 VAR
    IA, IB, J : INTEGER;
 BEGIN
    FOR J:= 1 TO NI DO
       BEGIN
         FOR IA:= 1 TO NI DO
            BEGIN
               BARY(.IA,J.) := AARY(.IA,J.);
              FOR IB:= 1 TO NI DO
                 BEGIN
                    IF (NODELABL(.IB.) <= NODELABL(.IA.))
                    AND (IB <> IA) THEN
                     BARY(.IA,J.) := BARY(.IA,J.) + AARY(.IB,J.);
                 END;
             END;
        END;
 END;

(* The following procedure computes matrix C from matrix B after performing row
revision. *)
PROCEDURE ROWREV (BARY:DHARY; VAR CARY:DHARY);
VAR
  I, JA, JB, CN, PR: INTEGER;
```

```
BEGIN
   FOR I:= 1TO NI DO
      BEGIN
         FOR JA:= 1 TO NI DO
            BEGIN
               CN:= 0;
                  FOR JB:= 1 TO NI DO
                     BEGIN
                        IF(NODELABL(.JA.) >= NODELABL(.JB.))
                        AND (JA <> JB) THEN
                        CN:= CN + 1;
                     END;
                     IF (CN>=2) THEN  PR:= 1
                     ELSE   PR:= 0;
                  FOR JB:= 1 TO NI DO
                     BEGIN
                        IF (NODELABL(.JA.) * NODELABL(.JB.) <> (..))
                        AND (JA <> JB) THEN
                        PR:= PR * BARY(.I,JB.);
                     END;
                  IF (CN>=2) THEN
                  CARY(.I,JA.) := PR
                  ELSE CARY(.I,JA.) := BARY(.I,JA.);
            END;
      END;
END;

(*  The following proceduretransforms matrix A to matrix AS  *)
PROCEDURE ATOAS(AARY: DHARY; VAR ASARY:DHARY);
VAR
I, J: INTEGER;
```

```
BEGIN
   FOR I:= 1 TO NI DO
      BEGIN
         FOR J:= 1 TO NI DO
            BEGIN
               ASARY(.I,J.):= AARY(.I,J.);
            END;
      END;
END;

(* The following proceduretransforms matrix AS to matrix ASP *)
PROCEDURE ASPREV(ASARY: DHARY; VAR  ASPARY: DHARY);
VAR
 I, J: INTEGER;

BEGIN
   FOR I:= 1 TO NI DO
      BEGIN
         FOR J:= 1 TO NI DO
            BEGIN
               ASPARY(.I,J.) := ASARY(.I,J.);
            END;
      END;
END;

(* The following proceduretransforms matrix B to matrix BP *)
PROCEDURE BPREV(BARY: DHARY; VAR  BPARY: DHARY);
VAR
 I, J : INTEGER;

BEGIN
   FOR I:= 1 TO NI DO
```

```
        BEGIN
           FOR J:= 1 TO NI DO
              BEGIN
                 BPARY(.I,J.):= BARY(.I,J.);
              END;
        END;
END;

(* The following proceduretransforms matrix C to matrix CP *)
PROCEDURE CPREV(CARY:DHARY; VAR CPARY: DHARY);
VAR
  I,J : INTEGER;

BEGIN
    FOR I:= 1 TO NI DO
        BEGIN
           FOR J:= 1 TO NI DO
              BEGIN
                 CPARY(.I,J.) := CARY(.I,J.);
              END;
        END;
END;

(* The followoing procedure computes matrix D1 *)
PROCEDURE FIRSTD(CARY, AARY : DHARY; VAR DARY:DHARY);
VAR
  I,J : INTEGER;

BEGIN
    FOR I:= 1 TO NI DO
        BEGIN
           FOR J:= 1 TO NI DO
```

```
            BEGIN
                DARY(.I,J.):= CARY(.I,J.) - AARY(.I,J.);
            END;
        END;
END;

(* The followoing procedure computes matrix A^i *)
PROCEDURE NEXTAS(AARY, ASPARY: DHARY; VAR ASARY:DHARY);
VAR
 I, J, K, CN, SUM : INTEGER;

BEGIN
   FOR I:= 1 TO NI DO
       BEGIN
          FOR J:= 1 TO NI DO
             BEGIN
                SUM:= 0;
                FOR K:= 1 TO NI DO
                    BEGIN
                        SUM:= SUM + ASPARY(.I,K.)*AARY(.K,J.);
                    END;
                ASARY(.I,J.):= SUM;

                IF (NODELABL(.I.) >= NODELABL(.J.))
                AND (ASARY(.I,J.) >= 1) THEN
                BEGIN
                    ASARY(.I,J.) := 0;
                    CIRCSET:= CIRCSET + NODELABL(.I.) + NODELABL(.J.);
                END;
          END;
    END;
END;
```

```
(* The followoing procedure computes matrix A_i *)
PROCEDURE NEXTAH(DARY, AARY: DHARY; VAR AHARY: DHARY);
VAR
 I, J, K, CN, SUM : INTEGER;

BEGIN
   FOR I:= 1 TO NI DO
      BEGIN
        FOR J:= 1 TO NI DO
           BEGIN
              SUM:= 0;
              FOR K:= 1 TO NI DO
                 BEGIN
                    SUM:= SUM + DARY(.I,K.) * AARY(.K,J.);
                 END;
                 AHARY(.I,J.):= SUM;
                 IF(NODELABL(.I.) >= NODELABL(.J.))
                 AND (AHARY(.I,J.) >= 1) THEN
                 BEGIN
                    CIRCSET:= CIRCSET + NODELABL(.I.) + NODELABL(.J.);
                    AHARY(.I,J.) := 0;
                 END;
           END;
     END;
END;

(* The following procedure checks whether matrices A^i and A_i are null matrices *)
PROCEDURE CHECKSUM(ASARY, AHARY: DHARY; VAR ASUM: INTEGER);
VAR
 I, J : INTEGER;
```

```
BEGIN
   ASUM:= 0;
   FOR I:= 1 TO NI DO
      BEGIN
         FOR J:= 1 TO NI DO
            BEGIN
               ASUM:= ASUM + ASARY(.I,J.) + AHARY(.I,J.);
            END;
      END;
END;

(* The following procedure computes matrix Bi *)
PROCEDURE NEXTB(ASARY, AHARY, BPARY : DHARY; VAR BARY: DHARY);
VAR
 I, J : INTEGER;

BEGIN
   FOR I:= 1 TO NI DO
      BEGIN
         FOR J:= 1 TO NI DO
            BEGIN
               BARY(.I,J.):= ASARY(.I,J.) + AHARY(.I,J.) + BPARY(.I,J.);
            END;
      END;
END;

(* The following procedure computes matrix Di *)
PROCEDURE NEXTD(CARY, CPARY, ASARY : DHARY; VAR  DARY: DHARY);
VAR
 I, J : INTEGER;

BEGIN
```

```
    FOR I:= 1 TO NI DO
        BEGIN
          FOR J:= 1 TO NI DO
             BEGIN
                DARY(.I,J.) := CARY(.I,J.) - CPARY(.I,J.) - ASARY(.I,J.);
             END;
        END;
END;

(*  Main Program  *)
BEGIN
  RESET(NODEINFO);
  RESET(DOMINFO);
  RESET(MATINFO);
  RESET(OBSINFO);
  RESET(GOALINFO);
  REWRITE(RESULT);

(*  Information from file NODEINFO is read into array NODELABL  *0
  NI:= 0;
  WHILE NOT EOF(NODEINFO) DO
   BEGIN
      NI := NI + 1;
        NODELABL(.NI.):= (. .);
        WHILE NOT EOLN(NODEINFO) DO
         BEGIN
           READ(NODEINFO,AA);
            NODELABL(.NI.) := NODELABL(.NI.) + (.AA.);
         END;
         READLN(NODEINFO);
   END;
```

```
(*  CMPSET, set consisting of all compound nodes is computed  *)
   CMPSET:= (..);
   FOR I:= 1 TO NI DO
      BEGIN
         CMP := 0;
         FOR J := 1 TO NI DO
            BEGIN
               IF (NODELABL(.I.) >= NODELABL(.J.))
               AND (I<>J) THEN
               CMP := CMP + 1;
            END;
         IF (CMP >= 1) THEN
         CMPSET := CMPSET + NODELABL(.I.);
   END;

(*  INSET, set consisting of all input variables is computed  *)
   INSET:= (..);
   WHILE NOT EOF(OBSINFO) DO
      BEGIN
         WHILE NOT EOLN(OBSINFO) DO
              BEGIN
                 READ(OBSINFO, OA);
                 INSET:= INSET + (.OA.);
              END;
     READLN(OBSINFO);
     END;

(*  GOALSET, set consisting of goal variables is computed  *)
   GOALSET:= (..);
   WHILE NOT EOF(GOALINFO) DO
        BEGIN
          WHILE NOT EOLN(GOALINFO) DO
```

```
            BEGIN
               READ(GOALINFO, GA);
               GOALSET := GOALSET + (.GA.);
            END;
            READLN(GOALINFO);
        END;

(*  Information in file DOMIMFO is read into array DOMLABL  *)
   DI:=0;
   WHILE NOT EOF(DOMINFO) DO
        BEGIN
           DI:= DI + 1;
           DOMLABL(.DI.):= (. .);
           WHILE NOT EOLN(DOMINFO) DO
              BEGIN
                 READ(DOMINFO, BB);
                 DOMLABL(.DI.):= DOMLABL(.DI.) + (.BB.);
              END;
              READLN(DOMINFO);
   END;

(*  Information in file MATINFO is read into array AARY  *)
  MI := 1;
  MJ := 1;
     WHILE NOT EOF(MATINFO) DO
         BEGIN
            AARY(. MI, MJ.) :=  0 ;
            WHILE NOT EOLN(MATINFO) DO
                 BEGIN
                    READ(MATINFO, CC);
                    AARY(. MI, MJ .) :=  CC ;
                    MJ := MJ + 1;
```

```
            END;
            READLN(MATINFO);
            MI := MI + 1;
            MJ := 1;
        END;

  (* DETECTION OF DEADEND PREMISES *)
   FOR I:= 1 TO NI DO
       BEGIN
          SUM1 := 0;
          FOR J:= 1 TO NI DO
             BEGIN
                SUM1:= SUM1 + AARY(. I, J.);
             END;
             IF (SUM1= 0)
             AND (NODELABL(.I.) * GOALSET = (..))
             AND (NODELABL(.I.) * CMPSET = (..))  THEN
             BEGIN
                WRITELN('DEADEND PREMISE FOR NODE:');
                FOR DD:= 0 TO 100 DO
                    BEGIN
                       IF ((.DD.) * NODELABL(.I.) <> (..)) THEN
                      WRITE ( DD);
                    END;
                   WRITELN;
                  WRITELN;
        END;
     END;

(* DETECTION OF UNREACHABLE GOALS *)
   FOR J:= 1 TO NI DO
       BEGIN
```

```
        SUM2 := 0;
        FOR I:= 1 TO NI DO
            BEGIN
                SUM2 := SUM2 + AARY(.I,J.);
            END;
        CD:= 0;
        FOR I:= 1 TO NI DO
            BEGIN
                IF (NODELABL(.J.) >= NODELABL(.I.))
                AND (J <> I) THEN
                CD:= CD + 1;
            END;

        IF (SUM2 = 0) AND (CD = 0)
        AND (INSET * NODELABL(.J.) = (..)) THEN
            BEGIN
                WRITELN('UNREACHABLE GOAL FOR NODE:');
                FOR DD:= 0 TO 100 DO
                  BEGIN
                      IF((.DD.) * NODELABL(.J.) <> (..)) THEN
                    WRITE ( DD);
                  END;
                  WRITELN;
                 WRITELN;
        END;
    END;

(*  DETECTION OF CONSISTENCY ERRORS  *)
  BEGIN
    CIRCSET:= (..);
    ATOAS(AARY, ASARY);
    WRITELN('A ARRAY');
```

```
 WRITELN;
 FOR I:= 1 TO NI DO
     BEGIN
        FOR J:= 1 TO NI DO
           BEGIN
              WRITE(ASARY(.I,J.));
           END;
          WRITELN;
     END;
     WRITELN;

COLREV(AARY, BARY);
   WRITELN('B1 ARRAY');
   WRITELN;
   FOR I:= 1 TO NI DO
       BEGIN
       FOR J:= 1 TO NI DO
          BEGIN
             WRITE(BARY(.I,J.));
          END;
         WRITELN;
       END;
      WRITELN;

ROWREV(BARY, CARY);
 WRITELN('C1 ARRAY');
 WRITELN;
     FOR I:= 1 TO NI DO
        BEGIN
          FOR J:= 1 TO NI DO
             BEGIN
               WRITE(CARY(.I,J.));
```

```
            END;
            WRITELN;
        END;
        WRITELN;

FIRSTD(CARY, AARY,DARY);
WRITELN('D1 ARRAY');
WRITELN;
    FOR I:= 1 TO NI DO
        BEGIN
            FOR J:= 1 TO NI DO
                BEGIN
                    WRITE(DARY(.I,J.));
                END;
                WRITELN;
        END;
        WRITELN;

ASPREV(ASARY, ASPARY);
NEXTAS(ASPARY, AARY, ASARY);
WRITELN('AS2 ARRAY');
WRITELN;
    FOR I:= 1 TO NI DO
    BEGIN
        FOR J:= 1 TO NI DO
            BEGIN
                WRITE(ASARY(.I,J.));
            END;
            WRITELN;
    END;
    WRITELN;
```

```
NEXTAH(DARY, AARY, AHARY);
     WRITELN('AH2 ARRAY');
     WRITELN;
     FOR I:= 1 TO NI DO
      BEGIN
         FOR J:= 1 TO NI DO
         BEGIN
              WRITE(AHARY(.I,J.));
         END;
        WRITELN;
       END;
       WRITELN;

CHECKSUM(ASARY, AHARY, ASUM);
    WHILE ASUM <> 0 DO
    BEGIN
    BPREV(BARY, BPARY);
    NEXTB(ASARY, AHARY, BPARY, BARY);
     WRITELN('NEXT B ARRAY');
     WRITELN;
     FOR I:= 1 TO NI DO
         BEGIN
            FOR J:= 1 TO NI DO
                BEGIN
                   WRITE(BARY(.I,J.));
                END;
               WRITELN;
         END;
         WRITELN;

CPREV(CARY, CPARY);
ROWREV(BARY, CARY);
```

```
    WRITELN('NEXT C ARRAY');
    FOR I:= 1 TO NI DO
     BEGIN
        FOR J:= 1 TO NI DO
            BEGIN
               WRITE(CARY(.I,J.));
            END;
            WRITELN;
     END;
     WRITELN;

NEXTD(CARY, CPARY, ASARY, DARY);
     WRITELN('NEXT D ARRAY');
     WRITELN;
     FOR I:= 1 TO NI DO
     BEGIN
        FOR J:= 1 TO NI DO
            BEGIN
               WRITE(DARY(.I,J.));
            END;
            WRITELN;
     END;
     WRITELN;

ASPREV(ASARY, ASPARY);
NEXTAS(ASPARY, AARY, ASARY);
    WRITELN('NEXT AS ARRAY');
    WRITELN;
    FOR  I:= 1 TO NI DO
        BEGIN
        FOR J:= 1 TO NI DO
            BEGIN
```

```
                WRITE(ASARY(.I,J.));
              END;
              WRITELN;
          END;
          WRITELN;

NEXTAH(DARY, AARY, AHARY);
     WRITELN('NEXT AH ARRAY');
     WRITELN;
     FOR I:= 1 TO NI DO
          BEGIN
             FOR J:= 1 TO NI DO
                 BEGIN
                   WRITE(AHARY(.I,J.));
                 END;
                WRITELN;
          END;
          WRITELN;
          CHECKSUM(ASARY, AHARY, ASUM);
     WRITELN('NEXT CYCLE');
     WRITELN;
     END;
 END;
   WRITELN;
   WRITELN;
   WRITELN(ASUM);
   WRITELN;
   WRITELN;

(* DETECTION OF CIRCULARITY *)
     IF (CIRCSET <> (..)) THEN
       BEGIN
```

```
        WRITELN('CIRCULAR RULES WITH NODES:');
          FOR DD:= 0 TO 100 DO
            BEGIN
              IF((.DD.) * CIRCSET <> (..)) THEN
             WRITE( DD);
            END;
           WRITELN;
          WRITELN;
   END;

   (*  DETECTION OF REDUNDANCY *)
FOR I:= 1 TO NI DO
BEGIN
   FOR J:= 1 TO NI DO
     BEGIN
       IF (BARY(.I,J.) >= 2) THEN
        BEGIN
        WRITELN('POTENTIAL REDUNDANCY WHILE INFERRING NODE:');
        FOR DD:= 0 TO 100 DO
         BEGIN
           IF((.DD.) * NODELABL(.J.) <> (..)) THEN
          WRITE(DD);
         END;
        WRITELN;
        WRITELN('    FROM NODE:');
        FOR CC:= 0 TO 100 DO
        BEGIN
           IF((.CC.) * NODELABL(.I.) <> (..)) THEN
          WRITE(CC);
        END;
        WRITELN;
        WRITELN;
```

```
      END;
    END;
  END;

   (* DETECTION OF CONTRADICTION *)

FOR I := 1 TO NI DO
 BEGIN
  RCH:= (..);
   FOR J:= 1 TO NI DO
    BEGIN
      IF (BARY(.I,J.) >= 1) THEN
      RCH:= RCH + NODELABL(.J.);
    END;
   FOR J:= 1 TO NI DO
    BEGIN
      FOR K:= 1 TO DI DO
        BEGIN
         IF (BARY(.I,J.) >= 1)
         AND (RCH*DOMLABL(.K.) <> NODELABL(.J.))
         AND ((RCH*DOMLABL(.K.))* NODELABL(.J.) <> (..))
               THEN
         BEGIN
           WRITELN('CONTRADICTION WHILE INFERRING NODE:');
           FOR DD:= 0 TO 100 DO
            BEGIN
                IF((.DD.) * NODELABL(.J.) <> (..)) THEN
              WRITE (DD);
            END;
           WRITELN;
           WRITELN('   FROM NODE:');
           FOR CC:= 0 TO 100 DO
```

```
                BEGIN
                    IF ((.CC.) * NODELABL(.I.) <> (..)) THEN
                    WRITE(CC);
                END;
            WRITELN;
            WRITELN;
            WRITELN;
            END;
          END;
        END;
      END;
  CLOSE(NODEINFO);
  CLOSE(DOMINFO);
  CLOSE(MATINFO);
  CLOSE(OBSINFO);
  CLOSE(GOALINFO);
  CLOSE(RESULT);
END. (*  End of Main Program  *)
```

VITA

Mysore Ramaswamy obtained a Bachelor of Engineering degree in Electrical Engineering from Bangalore University in 1970 and worked in industry till 1986. He completed a Master of Science degree in Engineering Management at the University of Southwestern Louisiana in 1988. His research interests include Knowledge Base Verification, Integrated Services Digital Network, and End User Computng.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Mysore Ramaswamy

Major Field: Business Administration (Quantitative Business Analysis)

Title of Dissertation: A Directed Hypergraph Approach for the Verification of Rule-Based Expert Systems

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

October 28, 1994