

KNOWLEDGE BASE VERIFICATION IN AN EXPERT SYSTEM SHELL

A THESIS

Submitted To The Department Of Computer
Engineering And
Information Sciences
And The Institute Of Engineering And Science
Of Bilkent University
In Partial Fulfillment Of The Requirements
For The Degree Of Master Of Science

By
Faruk POLAT
June 1989

QA
76.9
E96
3763

1989

KNOWLEDGE BASE VERIFICATION IN AN EXPERT SYSTEM SHELL

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND
INFORMATION SCIENCES
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

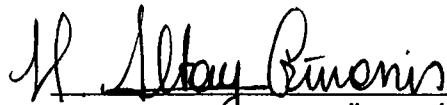
By
Faruk Polat
June 1989

Faruk Polat
tarafından bağışlanmıştır.

QA
76.9
.E96
P 763
1989

B 1863

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



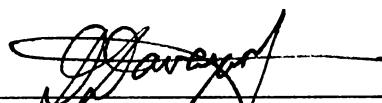
Asst. Prof. Dr. Halil Altay GÜVENİR (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Prof. Dr. Mehmet Baray

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Asst. Prof. Dr. David Davenport

Approved for the Institute of Engineering and Science:



Prof. Dr. Mehmet Baray, Director of Institute of Engineering and Science

ABSTRACT

KNOWLEDGE BASE VERIFICATION IN AN EXPERT SYSTEM SHELL

Faruk Polat

M.S. in Computer Engineering and
Information Sciences

Supervisor: Asst. Prof. Dr. Halil Altay GÜVENİR
June 1989

An important part of an expert system is its knowledge base which contains domain dependent knowledge. Knowledge base verification is one of the important problems of knowledge acquisition. It is the process of checking that a knowledge base is complete and consistent. An analysis of the rules can detect many potential problems that may exist in a knowledge base. The knowledge base may be incomplete, inconsistent, or even partly erroneous. Those problems unless identified and corrected may cause the inference engine to produce inconsistent results such as conflicting conclusions and sometimes to enter infinite loops. In order to be general, rules with certainty factors are preferred for knowledge representation. This is partly because rules are used in many applications and certainty factors are necessary when knowledge has probabilistic characteristics. Our approach is to develop a knowledge base verification tool that can be used as a part of a rule-based expert system shell.

Keywords : expert system, expert system shell, certainty factor, inference engine, knowledge acquisition, knowledge base, knowledge base verification.

ÖZET

UZMAN SİSTEM KABUĞUNDA BİLGİ TABANI DOĞRULANMASI

Faruk Polat

Bilgisayar Mühendisliği ve Enformatik Bilimleri Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Dr. Halil Altay GÜVENİR
Haziran 1989

Uzman sistemlerin önemli bir parçası da uygulamaya bağımlı bilgilerin saklandığı bilgi tabanıdır. Bilgi tabanının doğruluğunu kontrol edilmesi de bilgi toplama işleminin önemli bir bölümünü oluşturmaktadır. En basit deyişimle bilgi tabanının bütünlük ve tutarlılık içinde olması kontrol edilmesi işlemidir. Verilen kuralların analizi bilgi tabanında bulunabilecek bir çok hatalın önceden belirlenmesine yardımcı olacaktır. Uzman bilgi tabanı eksik, tutarsız, ve hatta yanlış olabilir. Bu problemler çıkarım makinasının tutarsız sonuçlar, örneğin çelişkili çıkarımlar, üretmesine ve hatta sonsuz döngülere girmesine neden olabilmektedir. Bu işlemin genel olması için bilgi kurallar şeklinde olup, her kural belli bir kesinlik değeri taşıyabilmektedir. Bunun nedeni ise bir çok uygulamada kurallar kullanılması ve bilginin doğruluğunu kesin olmayıp belli bir olasılık taşımasıdır. Doğruluk kontrolü sırasında sistem tarafından türetilen kurallar da gözönüne alınmaktadır. Bizim bu konuya yaklaşımımız uzman sistem kabuğuna herhangi bir bilgi tabanının doğruluk kontrolünü yapabilecek bir alt sistem eklemektir.

Anahtar Kelimeler : bilgi tabanı, bilgi tabanı doğrulanması, bilgi toplama, çıkarım makinası, kesinlik değeri, uzman sistem, uzman sistem kabuğu.

ACKNOWLEDGEMENT

I would like to thank my thesis advisor, Asst. Prof. Dr. H. Altay Güvenir for his guidance and support during the development of this study.

I would also like to thank Prof. Dr. Mehmet Baray, Asst. Prof. Dr. David Davenport and the research assistant Ahmet Coşar for their valuable discussions and comments.

TABLE OF CONTENTS

1	INTRODUCTION	1
1.1	Previous Work	2
2	EXPERT SYSTEMS	4
2.1	Components of an Expert System	4
2.1.1	Knowledge base	5
2.1.2	Inference Engine	6
2.1.3	User Interface	7
2.2	Types of Expert Systems	8
2.3	Advantages and Limitations of Expert Systems	8
3	KNOWLEDGE BASE CONSTRUCTION	10
3.1	Knowledge Acquisition	10
3.2	Knowledge Representation	13
3.2.1	Semantic Networks	14
3.2.2	Frames	15
3.2.3	Production Rules	17
3.2.4	Predicate Calculus	18

4 KNOWLEDGE BASE VERIFICATION TOOL	20
4.1 Knowledge Base Verification	20
4.1.1 Rules for Knowledge Representation	21
4.1.2 Unification	22
4.1.3 Inferred Rules	26
4.2 The Knowledge Base Problems Detectable by our Tool	30
4.2.1 Redundant Rules	30
4.2.2 Conflicting Rules	32
4.2.3 Subsumed Rules	33
4.2.4 Redundant If Conditions	34
4.2.5 Circular Rules	36
4.2.6 Dead-End Rules	36
4.2.7 Cycles and Contradictions in a Rule	37
4.3 Dependencies Between Rules	38
4.4 Implementation of the Verification Tool	40
5 CONCLUSION	45
REFERENCES	47
APPENDICES	50
A BNF DESCRIPTION OF RULES AND FACTS	50
B A SAMPLE KNOWLEDGE BASE VERIFICATION	52

LIST OF FIGURES

2.1 Components of an expert system.	5
3.1 Knowledge acquisition in an expert system.	11
3.2 The stages in the development of a typical expert system. . .	12
3.3 Examples for semantic nets.	14
4.1 Data structure of the rules to be used by the algorithm to find inferred rules.	27
4.2 Rules in the knowledge base after adding the inferred ones. .	30
4.3 Dependencies among rules.	40

1. INTRODUCTION

As a subarea of Artificial Intelligence, expert systems offer a new opportunity in computing and lead to the development of high-performance programs in some specialized professional domains and to the use of domain dependent methods for problem solving [4,13].

One of the important components of an expert system is its knowledge base which contains domain dependent knowledge. The knowledge base content is built by the process called knowledge acquisition whose purpose is to extract knowledge from an expert and to transform it into a form that can be processed by a computer. This iterative process may cause inconsistencies and gaps in the knowledge base [13,22].

Expert systems are supposed to give its users accurate advice or correct solutions to the problems. This requirement brings the concept of *validation* in expert systems. Expert systems are said to be valid if [22]

- Their judgments are free from the contradictions (consistency),
- They can handle any problem within their domains (completeness),
- They can deliver the right answers (soundness),
- The strength of their convictions are commensurate with the data (precision) and knowledge provided, and
- Finally they can be used with reasonable facility by those for whom they were designed (usability).

Knowledge base verification, a part of *validation* process, is one of the important problems of knowledge acquisition. It is the process of checking that a knowledge base is complete and consistent. The issue of verification

of the knowledge base in expert systems has been largely ignored which led to experts systems with knowledge base errors and no safety factors for correctness. The knowledge base may be incomplete, inconsistent, or partly erroneous. Those problems unless identified and corrected may cause the inference engine to produce inconsistent results such as conflicting conclusions and sometimes to enter infinite loops [5]. An overall analysis of the rules can detect many potential problems that may exist in a knowledge base.

The *consistency checking* means testing to see whether the system produces similar answers to similar questions. This is necessary because an expert system's conclusions must not vary according to some circumstances unless one of its components, knowledge base content or inference mechanism has been changed. It includes the checking for discrepancies, ambiguities, and redundancies in the rules of the knowledge base.

Completeness means that a knowledge base is prepared to answer all possible situations that could arise within its domain. The purpose of completeness checking is to find the knowledge gaps, in other words missing rules [18,19,20,23].

In our study, the aim is to develop a knowledge base verification tool that can be used as a part of a rule-based expert system shell. Rules with certainty factors are preferred for knowledge representation. This is partly because rules are suitable for use in many applications and certainty factors are necessary when knowledge has probabilistic characteristics. In that case, a threshold which may be set depending on the application is used as a limiting criteria during verification.

1.1 Previous Work

There have been studies on the verification of knowledge base in expert systems previously. In the context of MYCIN [4], an infectious disease consultation system, TEIRESIAS program was developed to automate the knowledge base debugging process [19]. It did not check the rules as they were initially entered into knowledge base. Rather, it assumed the knowledge transfer occurred in the setting of a problem solving session. In other words, TEIRESIAS allows an expert to judge whether or not MYCIN's diagnosis is correct, to track down the errors in the knowledge base that led to inconsistent conclusions, and to alter, delete, or add rules in the order to fix these errors.

Suwa, Scott and Shortliffe [23] wrote a program for verifying consistency and completeness for use with ONCOCIN, a rule-based system for clinical oncology. ONCOCIN requires each parameter to be designated with a context initially. It determines completeness and missing rules through combinatorial enumeration. If all context are complete, then the overall knowledge base is complete. ONCOCIN uses both data-driven and goal-driven inferencing. Although, the rule checker is written for use with ONCOCIN system, its design is general so that it can be adapted to other rule-based systems. It first considers rules related by “context,” second within context, creating a table displaying all possible combinations for condition parameters; and third displaying a table with conflicts, redundancies, subsumptions and missing rules.

Nguyen, Perkins [19] developed a knowledge base verification tool, CHECK which works with Lockheed Expert System (LES). CHECK assumes that rules are naturally separated by subject categories, a group of related rules kept together for documentation. CHECK combines logical principles as well as specific information about the knowledge representation formalism of LES. It checks the rules against all others in the same subject category and all others have the same goal for consistency and completeness. This check is done by enumeration.

Expert System Checker (ESC) [5], written by Cragun, is a decision-table based checker for rule-based expert systems. ESC first constructs a master decision table for the entire knowledge base, then automatically splits it into subtables, checks each subtable for completeness and consistency and reports any missing rules. It uses numerical checking methods facilitated by decision tables to speed completeness checking within context.

2. EXPERT SYSTEMS

An expert system is a knowledge-intensive program that emulates expert thought to solve significant problems in a particular domain of expertise. Expertise consists of knowledge about a particular domain, understanding of domain problems and skills at solving some of these problems [22,25].

Expert systems differ from the broad class of AI in several respects. First, they perform difficult tasks at expert levels of performance. Second, they emphasize domain specific problem solving strategies over the more general weaker models of AI. Third, they employ self-knowledge to reason about their own inference processes and provide explanations or justifications for conclusions reached [13]. Furthermore, they can say something about reliability.

Expert systems are also different from the conventional software programs. They handle the problems requiring human expertise by using domain dependent knowledge and reasoning techniques. In other words, they attempt to use not only the computational power of the computer, but also typical human reasoning techniques such as rules of thumb and shortcuts to solve problems [25].

2.1 Components of an Expert System

Expert systems have three basic components (Fig. 2.1):

1. Knowledge base
2. Inference engine
3. User interface

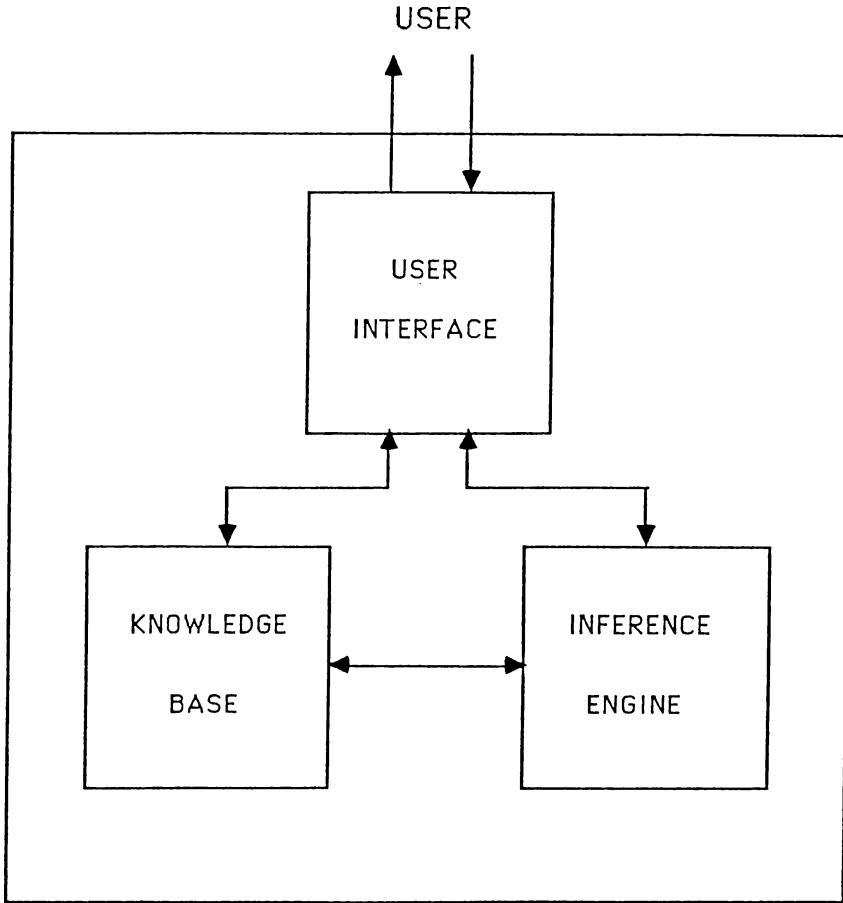


Figure 2.1: Components of an expert system.

An expert system resembles a conventional software program in its structure. The knowledge base of an expert system that contains facts and relationships between them matches the program code (written in a high-level programming language) of a software program. In the same way, inference engine in an expert system which executes the statements in the knowledge base by using its control and reasoning mechanisms matches the interpreter, or compiler in a software program [7,8,14].

2.1.1 Knowledge base

The knowledge base contains all the information necessary for solving problems on any chosen domain. This information which is specific to the particular application holds the domain facts and heuristics representing human expert domain knowledge. Facts encompass the given and unchanging knowledge about the problem and domain. They actually represent data and formulas related to an application. Heuristics are the representation of the

data of the problem and domain. Production rules, the basis of most expert systems, are commonly used to denote the expert knowledge [25]. Knowledge content is built by the process called knowledge acquisition whose purpose is to extract, to render, and to record the knowledge in a symbolic form.

Uncertainty

Knowledge obtained from human experts is sometimes uncertain. Facts and rules can be described as “maybe,” “sometimes,” “often,” etc. Further, expert systems, like human experts, may have to draw inferences based on incomplete, unavailable, unknown, or uncertain knowledge. Uncertainty can be represented by the use of probability judgment such as classical and subjective probability techniques, Bayes’ rule, and sometimes statistics. The issues of certainty factors and fuzzy logic have been used in many expert systems to represent uncertain knowledge [24].

2.1.2 Inference Engine

It contains the processes that work on the knowledge base, do analyses, form hypotheses, and audit the process according to some strategy that emulates the expert’s reasoning. The inference process actually involves several different processes that must work together these can be grouped as:

- Rule Retrieval: Identification of the rule as relevant to the conditions of the problem situation.
- Conflict Resolution: Resolving the conflict among competing rules, the result being the selection of one rule.
- Execution: Reaching the conclusion implied by the premise part of the rule that had been selected.

The inference engine takes new information, combines it with the knowledge base, and proceeds to solve the problem in working memory using its established reasoning and search strategies [16,25].

There are two major reasoning strategies, namely *forward* and *backward chaining* to control the inferencing process of the expert systems.

Forward Chaining

The system begins with a set of facts and proceeds to search for a rule whose premise is verified by those facts. The new facts are then added to the working memory and process continues until the requested conclusion is reached, or no applicable rules are left. Since it starts with what is known, with facts, it is sometimes called data-driven (antecedent) reasoning. With this kind of reasoning, we talk about what we can conclude from the given data.

Backward Chaining

It is a goal directed search that starts at the goal state and proceeds backward towards the initial conditions. The task is to see whether the necessary and sufficient antecedents that satisfy the goal exist in the domain by applying inverse operations. When there are no rules to establish the current goal (or subgoal) the program asks the user for necessary facts and enters them into the knowledge base. Since it selects a goal and scans the rules backward to achieve that goal, it is sometimes called goal-driven (consequent) reasoning. Here we try to answer whether it is possible to prove the hypothesis from the given data [14,22,25].

2.1.3 User Interface

It provides the communication between the expert system and the user. Current expert systems may be equipped with templates (menus), or natural language to facilitate their use, and an explanation module to allow the user to challenge and examine the reasoning process underlying the system's answer. A natural language interface allows computer systems to accept inputs and produces outputs in a language close to a conventional language such as English.

The interface is at both the front and end of the development process. Interfaces to expert systems are usually done in two ways: development engine and the end interface. By using the development engine, knowledge engineer or the expert can construct, maintain and debug the expert system through an editor, monitor, or validator. The end interface provides communication with the user after the expert system has been constructed.

2.2 Types of Expert Systems

Expert Systems are not general because they utilize domain-dependent knowledge for their applications. According to their application areas, they can be grouped into the following categories [13].

- Predicting
Inferring likely consequences of given situations (e.g., PLAND/CD is used for predicting crop damage, developed by University of Illinois)
- Diagnosing
Inferring system malfunctions from observed data (e.g., MYCIN is used for diagnosing infectious disease, developed by Stanford University)
- Designing
Configuring objects under constraints (e.g., XCON is used for configuring computer systems, developed by DEC & Carnegie-Melon University)
- Planning
Designing actions (e.g., TATR is used to plan bombing mission)
- Monitoring
Comparing observations to plan vulnerabilities (e.g., REACTOR is developed for monitoring nuclear reactors)
- Testing and Debugging
Identifying reasons for malfunctions (e.g., developed by Texas Instruments to test electronic circuit boards)
- Interpretation
Inferring situation description from sensor data (e.g., PROSPECTOR developed for interpreting geological structures)
- Controlling
Interpreting, predicting, repairing, and monitoring system behavior (e.g., VM is developed by Stanford University to control the treatment of patients in intensive care)

2.3 Advantages and Limitations of Expert Systems

There are some limitations of expert systems [6,11]:

- Domains of expert systems are narrow. Because building and maintaining a large knowledge base is difficult, only a few expert systems cover a significant range of knowledge.
- Certain knowledge can be quite difficult to represent efficiently if the knowledge lacks immediate if-then consequences. Knowledge representation techniques may sometimes limit the user in describing facts and relationships.
- The necessity for users to describe their problems in a strictly definite formal language is also a limitation.
- Most of the expert systems do not have a knowledge acquisition tool to allow the domain expert to describe his knowledge and update the knowledge directly. Instead, a knowledge engineer must successfully take the knowledge from the domain expert and operate the system.
- Due to the characteristics of knowledge, reasoning qualitatively and reasoning causally are both important in human reasoning, but it is difficult with expert systems to capture.
- The representation and utilization problem of inexact knowledge is also a major limitation.

The advantages of expert systems come from the separation of the expert knowledge from the general reasoning mechanism and the partitioning of general knowledge into separate rules. Some of the advantages of expert systems are given below [6,8]:

- The ability for the developers to refine old rules and add new ones during the incremental development of the knowledge base.
- By changing set of rules the same general system can be used for variety of applications.
- By changing the reasoning mechanism the same knowledge can be put to use in different ways.
- The ability of explaining its reasoning through rules to the users.

3. KNOWLEDGE BASE CONSTRUCTION

An expert's knowledge must undergo a number of transformations before it can be used by a computer. First, expertise in some domain through study, research and experience must be acquired. Next, the expert attempts to formalize this expertise and expresses it in the internal representation of an expert system. Finally, knowledge is entered to the computer.

3.1 Knowledge Acquisition

Knowledge acquisition is defined as the process of extracting, structuring, and organizing knowledge from several sources, usually human experts, so that it can be used in a program. This process (Fig. 3.1) is difficult and takes long time because knowledge in the real world is expressed in a different form than that required by the machine [9,12].

Knowledge acquisition is a bottleneck in the development of expert systems; it typically involves months or years of discussion between domain expert and knowledge engineer. During the extraction and translation of the expert's knowledge, there must be feedback from domain expert, and the knowledge engineer repeatedly refining the system until it achieves something close to expert's level of performance in problem solving [2,17].

Knowledge for an expert system can be acquired in several ways, all of which involve transferring the expertise needed for high performance problem solving in a domain from a source program. The source is generally a human expert but could also be the empirical data, case studies, or other sources from which a human expert's own knowledge has been acquired [3]. There are five major classes of techniques to acquire knowledge from the domain expert [25]. These are:

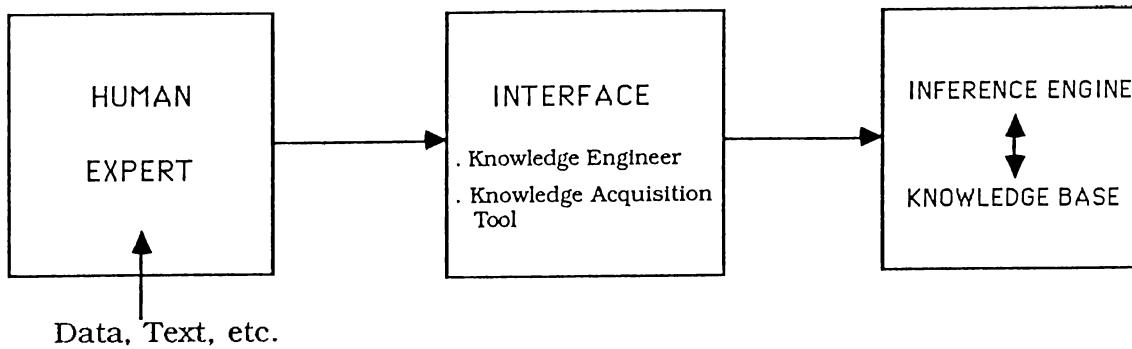


Figure 3.1: Knowledge acquisition in an expert system.

- Interviews,
- Protocols,
- Walkthroughs,
- Questionnaires,
- Expert Reports, and
- Induction.

Knowledge acquisition is carried out through several steps (Fig. 3.2):

- **Identification Stage**

This stage characterizes the important aspects of the problem by identifying the participants, the range of the problems the system must handle, the characteristics of the domain, the bounds of the domain and the user expectations. It also identifies the goals or objectives of building the expert system in the course of identifying the problem.

- **Conceptualization Stage**

The key concepts and relations are made explicit during this stage. Once the key concepts and relations are written down, much can be

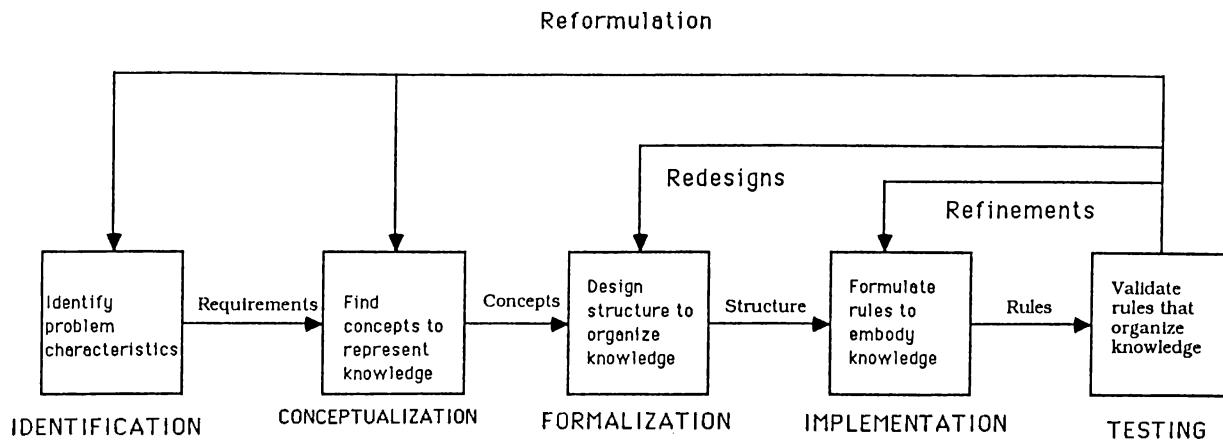


Figure 3.2: The stages in the development of a typical expert system.

gained from formalizing them and working towards an initial implementation.

- **Formalization Stage**

At this stage, the key concepts, subproblems, and information flow characteristics isolated during conceptualization are mapped into more formal representations. The result of formalizing the conceptual information flow and subproblem elements is a partial specification for building a prototype knowledge base.

- **Implementation Stage**

The formalized knowledge is mapped into the representational framework associated with the tool chosen for the problem. A useful representation for the knowledge is chosen and a prototype system is developed using it.

- **Testing Stage**

The prototype system is tested with a variety of examples to determine weaknesses in the knowledge base and inference structure [3].

A number of tools have been developed to ease the knowledge acquisition process. By using these tools, the expert interacts with the computer directly to define the knowledge base and control strategies minimizing the intermediate step of interacting with the knowledge engineer. These tools acquire

knowledge directly from the expert in two ways:

- Induction by Example

Rules are logically induced from the solutions to examples provided by the domain expert.

- Knowledge Elicitation

The tool interacts with the domain expert to elicit and structure the knowledge base without induction [25].

Various methods and tools, from structured human interviewing techniques to knowledge base editing tools, have been developed to facilitate the task of knowledge acquisition. The variety of methods in existence reflects the fact that knowledge acquisition is a multi-dimensional process; it can occur at different stages in the development of an expert system, and involve many types of knowledge [2].

3.2 Knowledge Representation

Knowledge Representation models describe the various architectures used to represent the expert's knowledge in an organized and consistent manner [25]. The representation of knowledge is a combination of data structures and interpretive procedures that, if used in the right way in a program, will lead to knowledgeable behavior. Work on knowledge representation in Artificial Intelligence has evolved the design of several classes of data structures for storing information in computer programs, and development of procedures that allow intelligent manipulation of these data structures to make inferences [1,15,17].

Techniques used for knowledge representation have undergone rapid change and development in recent years. There are several techniques for knowledge representation. Four frequently used techniques are:

- Semantic Networks
- Frames
- Production Rules
- Predicate Calculus

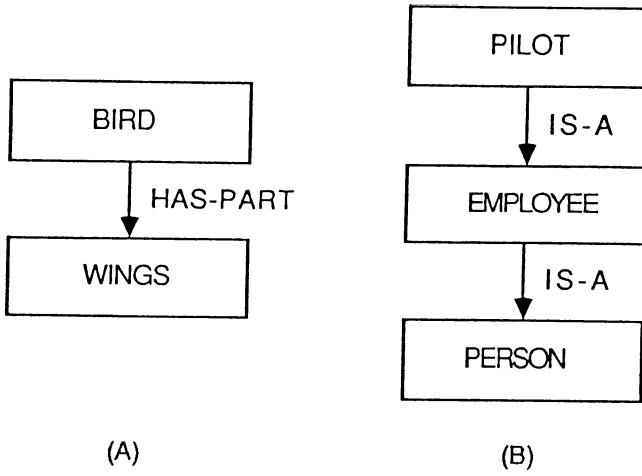


Figure 3.3: Examples for semantic nets.

Depending on the application of the expert systems, each knowledge representation model has its advantages and disadvantages. The selection of the most suitable technique in an expert system application depends on two major criteria:

- Domain of the problem
- Selection of the inference mechanism.

3.2.1 Semantic Networks

Although Semantic Networks, sometimes called Semantic Nets, are the most general representational structures and the basis for other knowledge representations, they are not directly used to model the knowledge. This is because they do not have formal definitive structural rules.

A semantic net consists of nodes and links between the nodes. Nodes are shown graphically by dots, circles, or boxes. They represent objects, concepts, and events in the domain. Links, or arcs, represent the relationships between the nodes and are shown graphically with arrows.

Consider, for example, the simple nets in Fig. 3.3. BIRD and WINGS are nodes representing sets or concepts, and HAS-PART is the name of the link specifying their possible relationship. Among the many possible interpretations of this net fragment is the statement “All birds have wings.”

Some advantages of semantic nets:

- This representation scheme is flexible. In other words, it is easy to add, delete, or modify nodes and arcs when necessary.
- The other advantage of semantic net representation is that important associations can be made explicitly and succinctly; relevant facts about an object or concept can be inferred from the nodes to which they are directly linked, without a search through a large database.

The semantic net representation provides the ability to inherit relationships from other nodes. Two kinds of inheritances are possible:

- Inheritance hierarchies where the relationship can be determined by tracing through several arcs.
- Property inheritance that describes the representation of knowledge about properties (attributes) of objects (commonly represented by IS-A arc).

As an example of the first type of inheritance, consider the example in Fig. 3.3. It does not only represent the two facts initially intended, but a third fact, “Pilot is a person” simply by following the IS-A links: “Pilot is an employee,” “An employee is a person” so “Pilot is a person.”

Disadvantages of the semantic nets:

- No formal representation structure
- Difficulties in distinguishing an individual inheritance and a class inheritance

3.2.2 Frames

A frame is a data structure that includes declarative and procedural information in predefined internal relations and consists of a collection of slots that contains attributes to describe an object, a class of objects, an action, or an event [25].

They provide a structure in which new data is interpreted in terms of concepts acquired through previous experience. This organization facilitates expectation driven processing, looking at things on the context one thinks in.

One of the characteristics of frame-based processing is its ability to determine whether it is applicable in a given situation. The idea is that a likely frame is selected to aid in the process of understanding the current situation and this frame in turn tries to match itself to the data it discovers. If it finds that it is not applicable, it could transfer control to a more appropriate frame [1].

Consider, as an example, the following frame:

```
DOG Frame
  Self : an ANIMAL ; a PET
  Breed :
  Owner : a PERSON
    (If-Needed : find a PERSON
      with pet = myself)
  Name : a PROPER NAME (DEFAULT = Rover)
```

```
DOG-NEXT-DOOR Frame
  Self : a DOG
  Breed : mutt
  Owner : Jimmy
  Name : Fido
```

“Self” slot is used to establish a property inheritance hierarchy among the frames, which allows information about the parent frame to be inherited by its children. Slots can have of their own. “If-Needed” slot in the example contains an attached procedure which can be used to determine the slot’s value when necessary. The slot “Default” suggests a value for the slot unless there is contradictory evidence.

The most important advantage of the frames is that they spend less amount of time for searching specific information, and allows for layers of abstraction to separate out low-level details from high-level abstracts.

3.2.3 Production Rules

Production Rules, which are also called rules, are conditional descriptions of given situations or context of a problem. There are two types of rule constructs, *if-then* and *if-then-else* constructs:

- *if premise(s) then action(s)*
- *if premise(s) then action(s) else action(s)*

The *if-then* construct is the most frequently used representation model. The premise part of the rule, called the condition part, states that the conditions that must occur for the production to be applicable and the action part is the appropriate action to take. Premise part of the rule is evaluated with reference to the knowledge base, and if succeeds, the action specified by the action part is performed [4]. Below is an example of a rule in MYCIN:

```
if
  1) The identity of the organism is not known
     with certainty, and
  2) The stain of the organism is grammef, and
  3) The morphology of the organism is rod, and
  4) The aerobicity of the organism is aerobic
then
  There is strongly suggestive evidence (.8) that
  the class of the organism
```

Uniformity and modularity are two important advantages of the rule-based systems. They are uniform because it is possible to add, delete, or change the rules without affecting the other rules. Production rules allow the user to model his knowledge in the way they think about solving a problem. They also replicate the reasoning statements used in human-problem solving task [25].

The organization and accessing of the rule set is also an important issue. The simple scheme is fixed, total ordering, but elaborations quickly grow more complex. Conflict resolution is used to select a rule.

The concept of the production rules comes from the production systems, also known as rule-based systems. A production system consists of a rule-base (a set of rules), a context data structure describing a specific problem

area in the knowledge base and an inference mechanism. Production systems have been found useful as a mechanism for controlling the interaction between statements of declarative and procedural knowledge. They facilitate human understanding and modification of systems with large amount of knowledge [1,4,25].

3.2.4 Predicate Calculus

As an extension of the notions of the propositional calculus, predicate calculus represent the symbols and their relationships to each other using the truth and rules of inferences such as *Modus Ponens*. Instead of looking at sentences that are of interest merely for their truth values, predicate calculus is used to represent statements about specific objects, or individuals. These statements are called *predicates*.

A predicate has a name and arguments. For example, the predicate *likes(john,kate)* has two arguments and states the fact that “John likes Kate.” Arguments can be constants (what is known), or variables (what is unknown). Upon application of a set of values on arguments, a predicate returns a value of either TRUE or FALSE.

Sentential connectives are used to make complex statements (a sequence of predicates describing a situation). Below are five most commonly used sentential(logical) connectives:

- *and*, \wedge
- *or*, \vee
- *not*, \sim
- *implies*, \rightarrow
- *equivalent*, \equiv

Truth Table for Predicate Logic						
x	y	$x \wedge y$	$x \vee y$	$x \rightarrow y$	$\sim x$	$x \equiv y$
T	T	T	T	T	F	T
T	F	F	T	F	F	F
F	T	F	T	T	T	F
F	F	F	F	T	T	T

Combining the predicates with these logical connectives, it is possible to obtain complex statements. Truth Table for Predicate Logic is given above. Consider the following example in Predicate Logic:

**grandfather(X,Y) equivalent
father(Z,Y) and
father(X,Z)**

The statement states that Person X is grand-father of person Y if and only if Person X is father of person Z and person Z is father of person Y.

The most important characteristic of the predicate calculus and related formal systems is that deductions are guaranteed to be correct to an extend that other representation schemes have not reached yet and the derivation of new facts from old can be mechanized. Predicate calculus provides modularity just in case of production rules. Additions, deletions, or modifications of statements can be made without having to worry about the context in which they will be used [1,25].

The disadvantage of the predicate logic can be seen when the number of facts become large. In that case, there is a combinatorial explosion in the possibilities of which rules to apply to which facts at each step of the proof.

4. KNOWLEDGE BASE VERIFICATION TOOL

4.1 Knowledge Base Verification

As mentioned previously, the expert systems must find correct solutions to the problems in their domain of expertise. *Knowledge Base Verification*, a part of the validation process, includes checking the knowledge base for completeness and consistency to discover a variety of errors that can arise during the process of transferring expertise from a human expert to a computer system [18,19,20].

An expert system cannot be tested, even on simple cases, until much of knowledgebase is encoded. Regardless of how an expert system is developed, its developers can profit from a systematic check of the knowledge base without gathering extensive data for test runs, even before the full reasoning mechanism is functioning. This purpose can be achieved by developing a program to check the knowledge base (assuming rules are used for knowledge representation) for consistency and completeness [18,19,20,23].

- Consistency Checking

Checking whether the system produces similar answers to similar questions. Inconsistencies in the knowledge base may appear as

- Conflict: two rules succeed in the same situation but with conflicting results.
- Redundancy: two rules succeed in the same situation but with the same results.
- Subsumption: two rules have the same results, but one contains additional restrictions

- Completeness Checking

Checking whether the system answers all reasonable situations within its domain. Whenever such completeness can be obtained, everything derivable in the domain from the given data will be derived. This can be achieved by identifying knowledge gaps in the knowledge base.

4.1.1 Rules for Knowledge Representation

During development of expert systems, it is necessary to decide on a knowledge representation scheme that is most suitable to the application. Since our aim is to develop a knowledge base verification tool for an expert system shell, which itself is a tool to develop expert systems, rules with certainty factors are used for knowledge representation. The basic advantage of the rule-based representation scheme is the modularity it provides and the simple uniform interpretive procedure that is often sufficient in rule-based systems. It is also easy to learn and use. Within the rule-based paradigm, the probabilistic approach has been commonly used for uncertain knowledge [7].

In our verification tool the knowledge base consists of rules and facts which are composed of predicates as shown below (See Appendix A).

Rule :

```
if predicate_1 &
   predicate_2 &
   .
   .
   .
   predicate_i
then
   predicate_j [ Certainty_factor ] ;
```

Fact :

```
predicate_k [ Certainty_factor ] ;
```

Predicate :

```
predicate_name_1 (arg_1, arg_2, . . . ,arg_n)
```

A predicate has a name, and finite number of arguments. Arguments can be variables, constants, or predicates. In this work, we will represent variables as strings that start with capital letters. Predicate names start with lower case letters. Constants may be of type integer, real, or string. Certainty factor is a real number between 0 and 1. It denotes the probability for occurrence of some events. The symbol “&” denotes the “logical and” operation. Below is an example of a fact in our system.

e.g.,

```
temperature (john_walker,high) [1.0] ;
```

The above fact, “temperature,” has two arguments, “john_walker” and “high.” It states that temperature of the patient “john_walker” is high with certainty 1.0, that is it is certain that he has high temperature. As an example of a rule in our system, consider the following example:

e.g.,

```
If temperature (Person,normal) &
    state (Person,in_severe_pain)
then
    ailment (Person,shingles) [0.75] ;
```

where “Person” is a variable, “normal,” “in_severe_pain” and “shingles” are constants, “temperature,” “state” and “ailment” are predicate names and 0.75 is the certainty factor associated with this rule. It states that whenever the first and second predicates in the action part hold with certainty 1.0, the predicate in the action part is asserted with certainty 0.75. In other words, if there is a person whose temperature is normal and is in the state of severe pain then it can be concluded with this rule that his ailment is shingles. If the predicates in the premise part of the rule hold with certainties 0.3 and 0.8 respectively and minimum certainty among the premises is chosen, then the certainty of the conclusion will be $0.3 * 0.75$.

4.1.2 Unification

During verification process, predicates are compared to each other to determine the relationship between them. Rules in the knowledge base may

be interrelated, if they have common predicates. These common predicates may/may not be equivalent. For deciding equivalence of these common predicates, *unification* is used.

Unification is defined as finding substitutions of terms for variables to make expressions, in our case predicates, identical. There are some rules for substitution. A variable can be replaced with a constant (this is called instantiation), with a variable, or with an expression (as long as that expression does not contain the original variable). Two clauses are said to be unifiable, if a substitution that resolves them can be found.

In our application, we use a simple unification algorithm that unifies variables with variables, or constants only. Below are some examples of predicates to be unified.

predicate-1	predicate-2	unifier	unifiable
temperature (X,Y)	temperature (A,B)	{A/X, B/Y}	Yes
temperature (john,high)	temperature (john,low)	{}	No
temperature (X,high)	temperature (Y,high)	{Y/X}	Yes
temperature (john,Y)	temperature (X,high)	{high/Y,john/X}	Yes
pre1 (X,pre2 (Y,X),12)	pre1 (Z,pre2 (A,2),12)	{2/X,A/Y,2/Z}	Yes
pre1 (X,pre2 (a,Y),10)	pre1 (b,pre2 (b,Z),10)	{}	No

Conjunctions of predicates are also needed to be compared to decide whether one is superset or subset of the other, or they are equivalent. Order of the predicates in the conjunctions may be different. There might be some restrictions imposed by a substitution list. The part of the tool to find relationships between two conjunctions of predicates utilizes the unification and takes care of the restriction imposed with a substitution list.

For example, consider the following cases:

example-1 :

```
predicate1 (X,20) & predicate2 (X)
predicate1 (janet,20) & predicate2 (janet)
```

example-2 :

```
predicate1 (X,20) & predicate2 (X)
predicate1 (Y,20) & predicate2 (Y)
```

```

example_3 :
  predicate1 (X,20) & predicate2 (X)
  predicate2 (Y) & predicate1 (Y,20)

example_4 :
  predicate1 (X,20) & predicate2 (Y)
  predicate2 (Z) & predicate1 (T,20)

```

Consider the first example. Suppose that there is no restriction given by a substitution list. That is, the variable X in the first conjunction had not been instantiated previously. It is easily seen that if variable X takes the value “janet” then these two conjunctions are equivalent. However we can not be sure whether the inference engine will instantiate the variable X to constant “janet,” or not.

In the second example, the system’s judgment about the equivalence of the conjunctions is definite. Because the variable X can be unified with variable Y assuming both of them are uninstantiated before the matching begins. In the third example, the conjunctions are not definitely equivalent because the order of predicates is different and the constants that will be provided by the inference for variables X and Y can not be equal every time.

Consider the fourth example. Although the order of predicates is different the system’s conclusion about the equivalence of the conjunctions is however definite assuming the variables X, Y in the first conjunction and variables Z, T in the second conjunction are uninstantiated before matching begins.

In order to see why the tool cannot be sure about its decision on the equivalence of two conjunctions of predicates, consider the following two different sets of facts in typical sample knowledge bases. Note that the inference engine matches the predicates with the facts in the knowledge base from top to bottom.

Facts in the first sample knowledge base :

```

predicate1(janet,20) [1.0] ;
predicate2(janet)      [1.0] ;
predicate1(john,20)   [1.0] ;
predicate2(john)     [1.0] ;

```

The relationships between the conjunctions taking into account the above facts:

Example No	Conjunction-1	Conjunction-2	Equivalent
example-1	X=janet	-	Yes
example-2	X=janet	Y=janet	Yes
example-3	X=janet	Y=janet	Yes
example-4	X=janet,Y=janet	T=janet,Z=janet	Yes

Facts in the second sample knowledge base :

```

predicate1(john,20) [1.0] ;
predicate1(janet,20) [1.0] ;
predicate2(janet) [1.0] ;
predicate2(john) [1.0] ;

```

The relationships between the conjunctions taking into account the above new facts:

Example No	Conjunction-1	Conjunction-2	Equivalent
example-1	X=john	-	No
example-2	X=john	Y=john	Yes
example-3	X=john	Y=janet	No
example-4	X=john,Y=janet	T=john,Z=janet	Yes

Our tool assumes that the inference engine uses *backward chaining*. Therefore, a rule to be fired is considered to be matched starting from its consequent. The predicate in the consequent can be asserted if and only if the predicates in the antecedent can be satisfied starting from the leftmost predicate to the rightmost one. Consider the following rule:

```

if appearance (Patient,blistery_spots) &
    temperature (Patient,feverish)
then
    ailment (Patient, chickenpox) [1.0] ;

```

if our goal is to ask whether ailment of patient “john” is chickenpox (i.e., Goal: ailment (john,chickenpox)) the inference engine will match the consequent of the rule first instantiating the variable “Patient” to the constant

“john.” It then tries to satisfy whether appearance of patient “john” is blistery spots (i.e., Sub-goal: appearance (john,blistery_spots)) If this succeeds then it tries to satisfy whether the temperature of patient “john” is feverish (i.e., Sub-goal: temperature (john,feverish)). If this also succeeds then the inference engine will be able to assert that the ailment of patient “john” is chickenpox.

4.1.3 Inferred Rules

Before going into the details of the work, it is necessary to find out those rules which are inferred by the knowledge base. This can be done by finding transitive closure of the rules in the knowledge base. However, every inferred rule may not be valid, because its certainty may be under the threshold. Finding the inferred rules is necessary because an inferred rule may contradict to another, or may cause circular chains.

For example, given the set of rules as follows:

```
R1 : If pre1 (X,john) then pre2 (X,2) [0.8] ;
R2 : if pre2 (Y,2) then pre3 (Z,Y) [1.0] ;
R3 : If pre3 (A,B) then pre4 (B,C) [0.7] ;
R4 : If pre1 (V,john) then ~pre4 (V,Z) [0.5] ;
```

In the example the symbol \sim is used to denote “logical not.” By using transitivity property, it is possible to infer the following rules (after unifying necessary predicates and applying unification list on the rest of predicates in the rules and assuming Threshold = 0.1):

Inferred Rules		Source Rules
R11 :	if pre1 (X,john) then pre3 (Z,X) [0.8] ;	R1, R2
R22 :	if pre2 (Y,2) then pre4 (Y,C) [0.7] ;	R2, R3
R33 :	if pre1 (X,john) then pre4 (X,C) [0.56] ;	R1, R2, R3

The certainty factors for inferred rules are calculated by multiplying the certainties of their source rules. For example, the certatinty factor of rule R33, 0.56, is obtained by multiplying the certainty factors of rules R1, R2 and R3, 0.8, 1.0 and 0.7 respectively.

There seems to be nothing wrong with the original set of rules if inferred

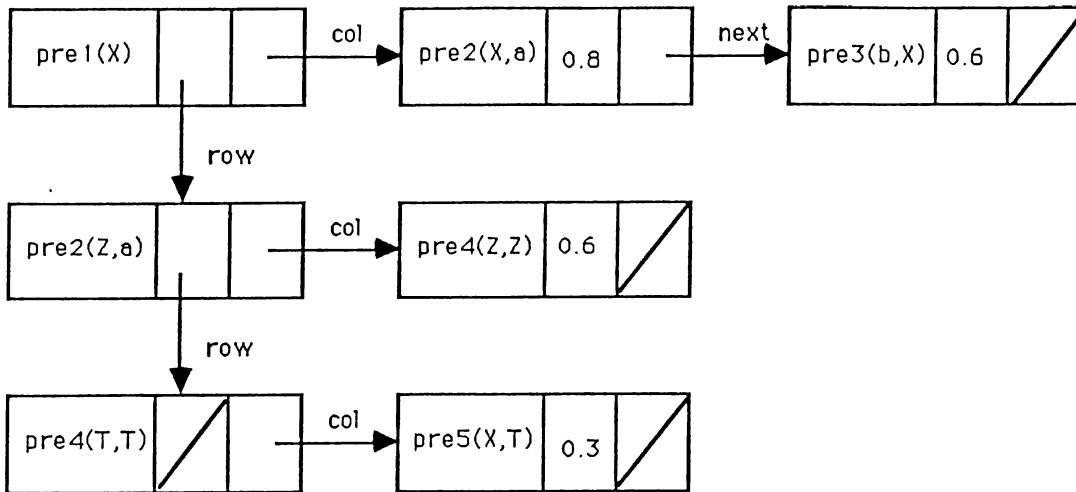


Figure 4.1: Data structure of the rules to be used by the algorithm to find inferred rules.

rules are not taken into consideration. However, it is clear that rule R4 and the inferred rule R33 conflict with each other.

In the example given above, it is seen that inferred rules may sometimes cause inconsistencies during the execution of the system. In order to ease the process of finding inferred rules, the tool converts the rules that are related to each other into a data structure that can be manipulated efficiently.

Consider the following related rules in a typical knowledge base:

```

R1 : if pre1 (X) then pre2 (X,a) [0.8] ;
R2 : if pre1 (Y) then pre3 (b,Y) [0.6] ;
R3 : if pre2 (Z,a) then pre4 (Z,Z) [0.6] ;
R4 : if pre4 (T,T) then pre5 (X,T) [0.3] ;

```

The converted data structure is shown in Fig. 4.1. The leftmost boxes, called “nodes” in the algorithm, represent the premise parts of the rules. Boxes next to these “nodes” are called “cells” and show the alternative actions to take. Each “node” and “cell” pair denotes a rule. A “node” contains a predicate, a pointer to the next “node” which is called “row” in the algorithm, a pointer to its first alternative action (i.e., “cell”) which is called “col” in

the algorithm and a flag to keep whether the closure for this “node” has been found, or not. A “cell” contains a predicate, a pointer to the next alternative action (i.e., “cell”) and a certainty factor.

Below is the algorithm used to find inferred rules using the previously constructed data structure:

```

procedure find_inferred_rules(<head>)
begin
    Initialize <node> to <head>
    while <node> is not equal to NIL do
        begin
            Insert <node> to the list of nodes pointed
            by <list_of_nodes>
            if closure of <node> had not been found then
                append_closure_of_node(<node>.col, <list_of_nodes>)
            Mark that the closure of node pointed by
            <node> has been found
            Get next <node>
        end
    end

procedure append_closure_of_node(<cell>, <list_of_nodes>)
begin
    while <cell> is not equal to NIL do
        begin
            add_cell_to_front_nodes(<cell>, <list_of_nodes>)
            Assign the pointer of the node which has the same
            predicate cell pointed by <cell> to <node>
            if <node> is not equal to NIL and
                <node>.col is not equal to NIL then
                if the closure of <node> had been found then
                    begin
                        Initialize <new_cell> to <node>.col
                        While <new_cell> is not equal to NIL do
                            begin
                                Append <node> to the list of the pointers
                                of nodes pointed by <list_of_nodes>
                                add_cell_to_front_of_nodes(<new_cell>, <list_of_nodes>)

```

```

        Delete <node> from  the list of the pointers
        of nodes pointed by <list_of_nodes>
        get new <new_cell>
        end
    end
else
begin
    Append <node> to the list of the pointers
    of nodes pointed by <list_of_nodes>
    append_closure_of_node(<node>.col, <list_of_nodes>)
    Delete <node> from  the list of the pointers
    of nodes pointed by <list_of_nodes>
    Mark that the closure of node pointed by <node>
    has been found
end
Get next <cell>
end
end

add_cell_to_front_of_nodes(<cell>,<list_of_nodes>)
begin
    Add the cell pointed by <cell> to the front
    of  cells of nodes in the list pointed
    by <list_of_nodes> if certainty is greater
    than the threshold (after unifying the
    predicates, unification list is applied on
    the predicate to be added. Variable names
    are created when variable name conflict arises.)
end

```

The above algorithm takes each node in turn to find its closure. If the closure of a node had been found previously, the next node is considered. The node taken is put into a node list (initially empty) which is used to keep the nodes for which the new inferred cells are to be appended. Then, each cell of that node is checked whether a node equivalent to it exists, or not. If a cell matches a node and the closure of that node had been found previously, then all cells of that new node are appended to the front of the nodes in the node list. If the closure of that new node had not been found, finding its closure becomes a new subproblem. In the same way, the new node is added to the node list and then its cells are checked whether they match some nodes, or

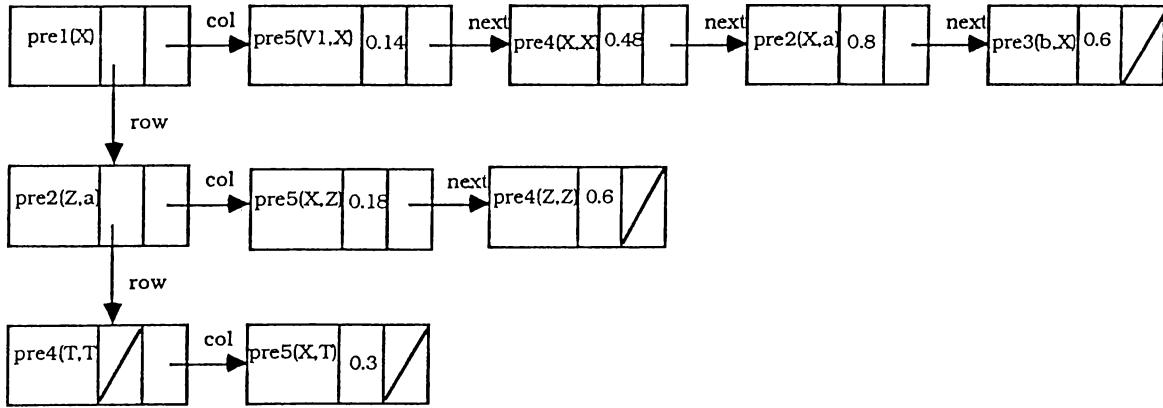


Figure 4.2: Rules in the knowledge base after adding the inferred ones.

not. This processes is repeated recursively. During the recursion, whenever the closure of a node is found (i.e., all the cells of that node is processed), it is extracted from the node list.

After executing the part of the tool to find the closure of the rules for above example, by using previously defined algorithm, the rules (including the inferred ones) in the knowledge base is shown in the Fig. 4.2.

4.2 The Knowledge Base Problems Detectable by our Tool

After finding and appending the inferred rules to the knowledge base, rules are checked against two requirements: consistency and completeness. Our tool is designed to identify knowledge base problems by performing an analysis of goal-driven rules.

In the following sections, potential problems that may occur in a knowledge base are defined and the ways our tool identifies these problems are explained in details.

4.2.1 Redundant Rules

Redundant rules are the ones that succeed in the same situation and have the same result. In other words, when the antecedents (if parts) of two rules are equivalent, their consequents (then parts) are also equivalent. Equivalence of two if parts holds when they can be unified and there are equal number of predicates in each part. Equivalence of two consequents holds if they are unifiable (Then parts of all rules have single predicate in this implementation).

For example consider the following two rules:

```
Rule 1 : if  predicate1 (X,Y) &
          predicate2 (Z,john)
        then
          predicate3 (X,Z)  [0.6] ;

Rule 2 : if  predicate2 (A,john) &
          predicate1 (C,D)
        then
          predicate3 (C,A)  [0.6] ;
```

The above two rules are redundant no matter which inference mechanism (backward or forward chaining) is used. Because there is an equal number of predicates in the antecedents of two rules and they are unifiable with substitutions { C/X, D/Y, A/Z } to the first rule.

Consider the following rules for redundancy:

```
Rule 1 : if  predicate1 (X,5) &
          predicate2 (X,Y)
        then
          predicate3 (john,Y)  [0.2] ;

Rule 2 : if  predicate1 (A,5) &
          predicate2 (B,Z)
        then
          predicate3 (john,Z)  [0.2] ;
```

In the above example, rules may be redundant. The reason for this uncertainty comes from the fact that the values that will be provided for the variable X in the first rule and the variable B in the second rule may not be same. In other words, the facts that the inference engine provide for the satisfactions of the second predicates of the rules may not be the same.

In many cases redundancies may cause serious problems. They might cause the same information to be accounted twice, leading to erroneous increases in the weight of their conclusions. Redundancies may not cause problems in the systems where certainty factors are not involved and the first successful rule is the one to succeed [18,23]. Note that if two rules are redundant, they are not required to have the same certainty factors. In other words, rules having different certainty factors may be redundant. This is valid for conflicting rules, subsumed rules and rules having redundant if conditions too.

4.2.2 Conflicting Rules

The conflicting rules are the ones that succeed in the same situation and produce conflicting results. If antecedents of two rules are unifiable and their consequents conflict with each other, we say that those rules are conflicting.

For example, consider the following two rules:

```
Rule 1 : if predicate1 (X,1) &
          predicate2 (Y,Z)
        then
          predicate3 (X,Z) [0.5] ;
  
Rule 2 : if predicate2 (A,B) &
          predicate1 (Y,1)
        then
          ~predicate3 (Y,B) [0.5] ;
```

The above two rules definitely conflict with each other because two rules are unifiable and their conclusions are conflicting. Note that although the order of predicates in the antecedents of the rules are not same there are no dependencies among predicates due to the instantiations of the variables.

Consider these two rules:

```

Rule 1 : if    predicate1 (T,1) &
          predicate2 (T,Y)
        then
          predicate3 (Z,Y)  [0.5] ;

Rule 2 : if    predicate2 (A,B) &
          predicate1 (A,1)
        then
          ~predicate3 (A,B)  [0.5] ;

```

The above two rules may conflict with each other, that is we cannot say that these rules definitely conflict with each other, because variables T and Z in the first rule and variable A in the second rule may not take same values.

4.2.3 Subsumed Rules

A clause can be defined as an expression of variables and constants. We say that a clause L_i subsumes another clause M_i if there exists a substitution s , such that the clause L_i after applying substitution s , is a subset of the clause M_i [21]. Subsumption of rules occurs when two rules have the same results, but one contains at least one additional constraint on the situation in which it will succeed. When the more restrictive rule succeeds, the less restrictive one will also succeed which is a redundancy.

In our system, this is defined as follows: if consequents of two rules are equivalent, and antecedent of one rule has some additional predicates we say that the more restrictive rule (the one having more predicates in its antecedent) is subsumed by the other one.

For example, consider the following two rules:

```

Rule 1 : if    predicate1 (X,computer) &
          predicate2 (X,Y)
        then
          predicate3 (X,printer)  [0.8] ;

Rule 2 : if    predicate1 (Y,computer)
        then
          predicate3 (Y,printer)  [0.8] ;

```

Rule 1 is subsumed by Rule 2, because Rule 2 needs less information to conclude predicate3. In other words, when Rule 1 succeeds Rule 2 also succeeds.

For example, consider the following two rules:

```
Rule 1 : if  predicate1 (X,computer) &
          predicate2 (Y,cable) &
          predicate3 (printer,Y)
    then
        predicate4 (Z,X)  [0.8] ;

Rule 2 : if  predicate1 (A,computer)
    then
        predicate3 (B,C)  [0.8] ;
```

In the above example the subsumption of the rules is only possible if the variables A and C in the second rules and variable X in the first rule are instantiated to the same value.

4.2.4 Redundant If Conditions

Sometimes rules contain unnecessary if conditions which cause the inference engine to do extra work which does not affect the result if they are extracted from the rules. Unnecessary if conditions occur when one of the predicates in the antecedent of one rule conflicts with one of the predicates in the antecedent of the other rule and all the remaining predicates in the antecedents and consequents of the rules are equivalent. Two types of subsumption can be identified by our tool.

As an example of the first type, consider the following two rules:

```
Rule 1 : if  predicate1 (X,1) & predicate2 (Y,Z)
    then predicate3 (X,Z)  [0.6] ;

Rule 2 : if  predicate1 (A,1) & ~predicate2 (B,C)
    then predicate3 (A,C)  [0.6] ;
```

The predicate “predicate2” in the first and second rules is unnecessary because it cannot affect the conclusions of Rule 1 and Rule 2. In this case, user may discard the above rules and add the following rule:

```
Rule N : if predicate1 (X,1)
          then predicate3 (X,Z) [0.6] ;
```

As an example of the second type, consider the following case:

```
Rule 1 : if predicate1 (X) & predicate2 (Y)
          then predicate3 (Z) [0.6] ;
```

```
Rule 2 : if ~predicate1 (A)
          then predicate3 (C) [0.6] ;
```

It is possible to combine these two rules with “logical or” operation after unification:

```
Rule X : if ~predicate1 (X) or
          predicate1 (X) and predicate2 (Y)
          then predicate3 (Z) [0.6] ;
```

Using the distribution property in logic, if P , Q , and R are predicates then $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$.

```
Rule X : if (~predicate1 (X) or predicate1 (X)) and
          (~predicate1 (X) or predicate2 (Y))
          then predicate3 (Z) [0.6] ;
```

This can be further simplified as:

```
Rule X : if (~predicate1 (X) or predicate2 (Y))
          then predicate3 (Z) [0.6] ;
```

If we separate this rule into two rules, we see that first predicate of Rule 1 is unnecessary:

```

Rule X1 : if    predicate2 (Y)
          then predicate3 (Z)  [0.6] ;

Rule X2 : if    ~predicate1 (X)
          then predicate3 (Z)  [0.6] ;

```

4.2.5 Circular Rules

Some of the rules in the knowledge base may cause infinite loops during the execution of the expert system. The check against the circularity ensures that there is no rule such that it requires its own action to establish its own condition, whether directly or through the exercise of other rules.

For example, consider the following two rules:

```

Rule 1 : if    predicate1 (Y)
          then  predicate2 (Y,Z)  [0.5] ;

Rule 2 : if    predicate2 (X,Y)
          then  predicate3 (X)   [0.2] ;

Rule 3 : if    predicate3 (A)
          then  predicate1 (A)  [0.5] ;

```

The above set of rules would go into an infinite loop if one attempted to backward chain with a goal matching the action part of any rule.

4.2.6 Dead-End Rules

In backward chaining, the goal(s) must match a fact, or consequent of some rule in the knowledge base. Otherwise, it is not possible to reach the goal(s). Suppose that our goal is “predicate3 (Y,john,1.170).” If there are no facts to satisfy “predicate3 (Y,john,170)” and no rules whose action part matches this predicate, then this goal cannot be satisfied.

Dead-end rules are the ones whose premise parts cannot match any fact, or consequents of any rules. Consider the following case:

```
Rule 1 : if      predicate1 (Y)
          then   predicate2 (Y,computer) [0.5] ;
```

If there are no facts to match the premise part of Rule 1, “predicate1 (Y),” and consequents of no rules match this predicate, Rule 1 will have no effect during the execution of the expert system because its conclusion is unreachable.

4.2.7 Cycles and Contradictions in a Rule

A Cycle within a rule can be detected when a predicate occurs in both antecedent and consequent parts. Below are two examples to show potential cycles within a rule. Note that in the first example cycle is unavoidable whereas in the second example there might be a cycle depending on the values provided for the variables.

Example-1 :

```
if predicate1 (A,B)    &
   predicate2 (john,D) &
   predicate3 (A,D)    &
   predicate4 (john,C)
then
   predicate2 (john,D) [0.6] ;
```

Example-2 :

```
if predicate1 (A,B)    &
   predicate2 (john,C) &
   predicate3 (B,D)    &
   predicate4 (john,D)
then
   predicate3 (A,D) [0.6] ;
```

Contradictions in a rule occur when one of the predicates conflicts with another predicate in the same rule. This may happen in two ways:

- Conflicting predicates may be both in the premise part of the rule

Below is an example of this type of contradiction. The first and third predicate of the rule might contradict with each other. Because the instantiation of variable Z which is used in the satisfaction of third predicate is done beforehand which may not be the same as the instantiation of the variable Y used in first predicate.

```
Rule 1 : if      predicate1 (Y)  &
          predicate2 (Z,Y)  &
          ~predicate1 (Z)
        then
          predicate4 (Y,Z)  [0.5] ;
```

- One of the conflicting predicates occurs in the premise part and the other one in the action part of the rule

Below is an example of this type of contradiction. The first predicate of the premise part and the action predicate definitely contradict with each other. Note that some systems may use this kinds of rules in order to switch from one situation to another.

```
Rule 1 : if      predicate1 (Y)  &
          predicate2 (Z,Y)  &
          predicate3 (Z)
        then
          ~predicate1 (Y)  [0.5] ;
```

4.3 Dependencies Between Rules

When the rules have some common predicates in their antecedents and consequents they may be naturally interrelated. As a result of these dependencies, some cycle problems may arise. Detecting this kind of cycles is very difficult with a static verification tool. These cycles can only be detected during the execution of the expert system which requires dynamic checking. Since the purpose in our approach is to find the problems in the rule set before the expert system can be used, not to write a debugger, the tool only give the possibility of independent cycles that may occur.

Consider the following rules in a rule base:

```

R1 : if predicate1 (X,high)      &
      predicate2 (Y,feverish)  &
      predicate3 (38,X,Y)
    then
      predicate5 (X,low) [0.7] ;

R2 : if predicate6 (cold,Z) &
      predicate10 (Z,take_asp)
    then
      predicate1 (Z,high) [0.3] ;

R3 : if predicate7 (hot,T) &
      predicate11 (T,high)
    then
      predicate2 (T,feverish) [0.9] ;

R4 : if predicate8 (low,20,C) &
      predicate5 (A,low)      &
      predicate9 (A,B,C)
    then
      predicate3 (38,B,C) [0.7] ;

```

In order to show how dependencies cause cycles, suppose that our goal is to satisfy predicate5 (X,low) using the above rules (See Fig. 4.3). Since the inference engine works backward, it will follow the following series of steps:

- Try to fire the rule R1 to conclude our goal
 - Try to fire the rule R2 to satisfy the first predicate of Rule R1. Suppose that first and second predicates in the antecedent of the rule R2 succeed. Therefore, first predicate of rule R1 succeeds.
 - Try to fire the rule R3 to satisfy the second predicate of Rule R1. Suppose that both predicates in the antecedent of rule R3 succeed. Therefore, second predicate of rule R1 succeeds.
 - Try to fire the rule R4 to satisfy the third predicate of Rule R1. Suppose that first predicate in the antecedent of rule R3 succeeds. Next comes the satisfaction of second predicate of rule R4.
 - * Try to fire the rule R1 to satisfy the second predicate of rule R4 which was our initial goal which is a CYCLE.

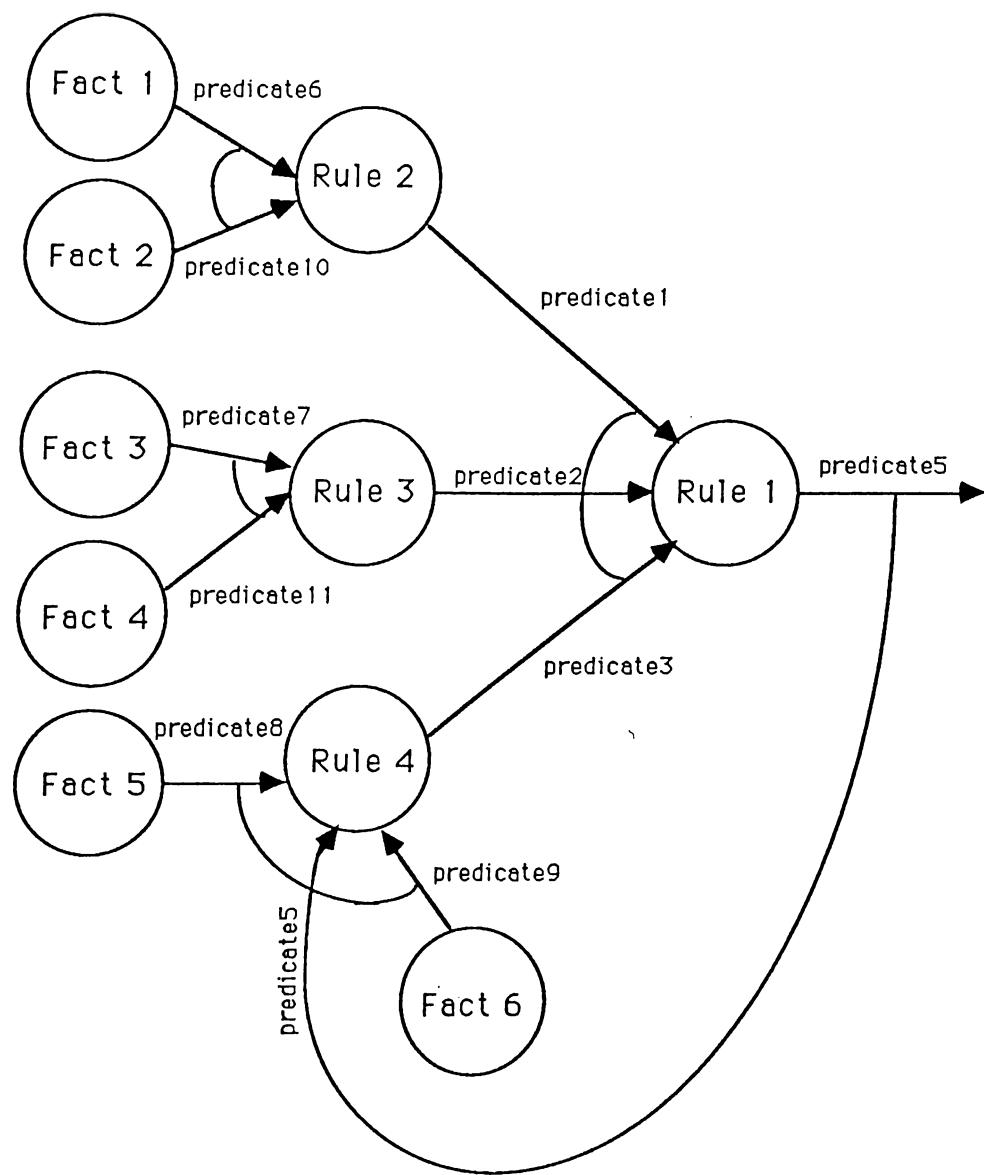


Figure 4.3: Dependencies among rules.

4.4 Implementation of the Verification Tool

Our knowledge base verification tool is designed and implemented in UNIX environment using C language. The tool first takes the rules and facts, and parses them. It stores all user defined data (variables and constants) in a symbol table and uses the pointers to that symbol table in order to access data when necessary. During the execution, only pointers are compared avoiding long string comparisons. Second, it finds all the inferred rules in order to use them during the verification process. All facts and rules that have certainty factors below a predefined threshold are ignored.

Third, relationships among rules are found and stored in two different two dimensional tables. The first table contains the relational information obtained by comparing consequents and then antecedents of the rules pairwise. The tool starts comparing consequents first, since it assumes that inference engine uses backward chaining. Comparison is done using unification. The result of comparing consequents of rules is whether they are same, conflicting, or different. In case they are same or conflicting, a substitution list is kept. Later, the antecedents of the rules are compared taking into account the substitution list coming from previous comparison of their consequents. After this comparison, number of equivalent predicates, number of conflicting predicates, whether the possible relationship is definite or not, and a substitution list are found. The tool identifies the possible relationship as definite between rules if

- all variables in the substitution list is unified with variables and
- all variable pairs in the substitution list are not instantiated until the predicates they are unified.

Each entry of this tables, Rule-relation[i,j], contains a record keeping the following information about Rule_i and Rule_j:

- Whether the predicates in their consequents are equivalent, conflicting or different
- Number of equivalent predicates in their antecedents
- Number of conflicting predicates in their antecedents
- Whether the possible relationship between them is definite, or not.

- A substitution list.

The following data structure is used for keeping the relationship between Rule_i and Rule_j:

```
Rule_relation[i,j] :
  consequent_relation (SAME, CONFLICTING, DIFFERENT)
  no_of_same          (integer)
  no_of_conflict      (integer)
  judgment            (DEFINITE, MAY_BE)
```

The second table is used for finding the potential cycles due to the dependencies among rules. Each entry of this table, If_then[i,j], keeps the information whether the consequent of Rule_i occurred in the antecedent of Rule_j, or not. The tool examines each entry of first table, Rule-relation[i,j] ; where $1 \leq i \leq j \leq N$, N is the number of rules, to find redundant rules, conflicting rules, subsumed rules and rules having redundant if conditions. If the tool identifies a problem for some Rule_i and Rule_j, it gives a message related to that problem as warning if Rule_relation[i,j].judgment = MAY_BE otherwise as a definite error.

Rule_i and Rule_j are identified as redundant if

- Rule_relation[i,j].consequent_relation = SAME and
- Number of predicates in antecedents of Rule_i = Number of predicates in antecedents of Rule_j = Rule_relation[i,j].no_of_same

Rule_i and Rule_j are identified as conflicting if

- Rule_relation[i,j].consequent_relation = CONFLICTING and
- Number of predicates in antecedents of Rule_i = Number of predicates in antecedents of Rule_j = Rule_relation[i,j].no_of_same

Rule_i and Rule_j are identified as subsumed rules if

- Rule_relation[i,j].consequent_relation = SAME and
- Number of predicates in antecedents of Rule_i ≠ Number of predicates in antecedents of Rule_j and
- Number of predicates in the antecedent of the rule having less predicates in its antecedent = Rule_relation[i,j].no_of_same

Rule_i and Rule_j are identified as having redundant (first type of redundancy) if conditions if

- Rule_relation[i,j].consequent_relation = SAME and
- Number of predicates in antecedents of Rule_i = Number of predicates in antecedents of Rule_j and
- Rule_relation[i,j].no_of_conflicting = 1 and
- Number of predicates in the antecedent of Rule_i = Rule_relation[i,j].no_of_same + 1

Rule_i and Rule_j are identified as having redundant (second type of redundancy) if conditions if

- Rule_relation[i,j].consequent_relation = SAME and
- Number of predicates in antecedents of Rule_i = 1 and
Number of predicates in antecedents of Rule_j > 1 or
Number of predicates in antecedents of Rule_j = 1 and
Number of predicates in antecedents of Rule_i > 1 and
- Rule_relation[i,j].no_of_conflicting = 1

The checks for circular rules are done during the generation of the inferred rules. Every time a new inferred rule is produced, it is checked whether the predicate in its antecedent part is same as the predicate in its consequent part. If this happens and the certainty factor of the new rule is above the threshold, the system detects the circular chain.

In order to find the problems within a rule, each rule is examined in turn. To detect self cycles within a rule, each predicate in the antecedent part is

compared to the predicate in the consequent part. If they are the same, the cycle is reported. To detect contradiction within a rule the predicates in the antecedent part are compared to each other and to the predicate in the consequent part. If the compared predicates are conflicting, the contradiction is reported. In order to detect whether a rule is a dead-end or not, all the predicates in the antecedent part are checked whether there exist facts and rules to match all of them. If this matching does not occur for a predicate, the rule is identified as dead-end.

In order to detect cycles due to the dependencies among rules, the second table is used. It contains the relationship between antecedents and consequents of the rule pairs. Using this table, the possible cycles are reported by listing the sequences of rules causing the cycles.

In a knowledge base verification using our tool, most of the time is spent to find the inferred rules. The larger the number of interrelated rules (rules having same antecedents and consequents), the longer it takes to find inferred rules. During the construction of the relationship tables, each rule is compared with others. The time spent on this depends on the average number of predicates in the antecedents of the rules. The part of the algorithm to detect problems using previously constructed tables have complexity $O(N^2)$ where N is the number of rules including the inferred ones.

In order to reach conclusions, the tool takes into account the dependencies between predicates in a rule, the relationships between rules and the way in which the unification occurs. For this reason, two types of messages are generated, the first one being warnings and the second one being errors. The word “may” is used in the warnings when the tool cannot identify problems definitely. Dependencies between rules are also found and examined to find the cycles. Our tool is designed to handle goal-driven (backward chaining) rules only.

A sample knowledge base verification using our tool is given in the Appendix B. The input knowledge base contains 29 rules and 12 facts in it. On successful completion of compilation, the tool generates 19 inferred rules together with their certainty factors using the original set of rules. In this example, the threshold is given as 0.0001. The tool ignores all the rules and facts with certainty factors under the threshold . Later, it finds the relationships and dependencies among all rules (including the inferred ones) and then reports all potential problems that exist in the knowledge base, such as redundant rules, conflicting rules, etc.

5. CONCLUSION

The verification of knowledge base becomes increasingly important as expert systems become more frequently used and their conclusions become more trusted. During the construction of knowledge bases in expert systems, many changes and additions to the rule set occur. In order to help knowledge engineers to develop expert systems rapidly and accurately, problems in the knowledge bases must be identified and corrected. This is because expert system's users expect answers to be unfailingly consistent and correct.

In this study, a knowledge base verification tool is developed to be used as a part of an expert system shell which supports rules with certainty factors as the knowledge representation scheme and backward chaining as inference mechanism. The verification tool identifies problems in a knowledge base taking into account the inferred rules as well. It checks the knowledge bases against two requirements, consistency and completeness, identifying redundant rules, conflicting rules, subsumed rules, rules having unnecessary if conditions, circular rules, cycles and contradictions within rules and the possible potential cycles due to the dependencies among rules.

Our tool can be used to detect many potential problems and gaps in the knowledge base helping knowledge engineer in the development of an expert system rapidly and accurately. Two important characteristics of our tool which do not exist in many knowledge base verification tools, such as CHECK [19] and ESC [5], are

- the generation of inferred rules and the inclusion of them in the knowledge base verification process and
- the use of certainty factors to model uncertain knowledge.

Finding and considering inferred rules in the verification process is necessary, since an inferred rule could cause serious problems. For example, it

may contradict to another rule in the knowledge base. The certainty factors are used for representing uncertain knowledge, because some expert system applications need to handle knowledge having probabilistic characteristics.

As a further research, soundness, precision and usability requirements for validation of expert systems bases [22] may be carried out. These are necessary because the expert system must produce right conclusions (soundness) and present this conclusion with a certainty appropriate to what is given (precision) and the interaction between the user and the expert system should proceed as intended by the designer (usability).

REFERENCES

- [1] Barr, A., Davidson, J., "Representation of Knowledge," *Technical Report STAN-CS-80-793*, Stanford University, 1980.
- [2] Becker S., Selman B., "An Overview of Knowledge Acquisition Methods for Expert Systems," *Technical Report CSRI-184*, 1986.
- [3] Buchanan, B.G, et al, "Constructing an Expert System," *Building Expert Systems*, F. Hayes-Roth, D. A. Waterman, D. B. Lenat, Addison-Wesley Inc., Massachusetts, pp 3-29, 1983.
- [4] Buchanan, B. G., Shortliffe E. H., *Rule-Based Expert Systems*, Addison-Wesley Inc., Massachusetts, 1985.
- [5] Cragun, B., Steudel, H. J., "A Decision-Table-Based Processor for Checking Completeness and Consistency in Rule-Based Expert Systems," *Int. J. Man-Machine Studies*, pp 633-648, 1987.
- [6] Davis, R., "Knowledge-Based Systems: The View in 1986," *AI in the 1980s and Beyond*, W.Eric, L.Grimson, R.S.Patil, MIT Press, Massachusetts, pp 13-41, 1987.
- [7] Dhar, V., Pople H. E., "Rule-Based Versus Structure-Based Models for Explaining and Generating Expert Behavior," *Communications of ACM*, Vol. 30, No. 6, pp 542-555, June 1987.
- [8] Duda, R. O., Gaschnig, J. G., "Knowledge-Based Expert Systems Come of Age," *Applications in Artificial Intelligence*, Stephen J. Andriole, Petrocelli Books Inc., Princeton, pp 45-86, 1985.
- [9] Fellers, J. W., "Key Factors in Knowledge Acquisition," *Comput. Pers.*, USA, Vol.11, no.1, pp.10-24, May 1987.
- [10] Francioni, J. M., Kandel A., "A Software Engineering Tool for Expert System Design," *IEEE Expert*, pp 33-41, 1988.

- [11] Gevarter, W. B., "Expert Systems: Limited But Powerful," *Applications in Artificial Intelligence*, Stephen J. Andriole, Petrocelli Books Inc., Princeton, pp 125-139, 1985.
- [12] Gruber, T., Chon, P., "Principles of Design for Knowledge Acquisition," *IEEE Proceeding*, pp 9-15, 1987.
- [13] Hayes-Roth F., Waterman D. A., Lenat D. B., "An Overview of Expert Systems", *Building Expert Systems*, F. Hayes-Roth, D. A. Waterman, D. B. Lenat, Addison-Wesley Inc., Massachusetts, pp 3-29, 1983.
- [14] Hu, D., *Programmer's Reference Guide to Expert Systems*, Howard W. Sams & Company, Indianapolis, 1987.
- [15] Jackson, P., "Review of Knowledge Representation Tools and Techniques," *IEE Proceedings*, Vol.134, No.4, pp 224-230, July 1987.
- [16] Keller R., *Expert System Technology, Development & Applications*, Prentice-Hall Inc., Englewood Cliffs, 1987.
- [17] Mettere, W., "An Assessment of Tools for Building Large Knowledge-Based Systems," *AI Magazine*, pp 81-89, Winter 1987.
- [18] Nguyen, T. A., "Verifying Consistency of Production Systems," *Proceedings of the Third Conference on Artificial Intelligence Applications*, pp 4-8, 1987.
- [19] Nguyen, T. A., Perkins, W. A., et al, "Knowledge Base Verification," *AI Magazine*, pp 69-75, Summer 1987.
- [20] Nguyen, T. A., Perkins, W. A., "Checking an Expert System's Knowledge Base for Consistency and Completeness," *Proceedings of the Ninth International Joint Conference on AI*, Menlo Park, CA ; American Association for AI, pp 374-378, 1985.
- [21] Nilsson, N. J., *Principles of Artificial Intelligence*, Tioga Publishing Company, 1980.
- [22] Sell, P. S., *Expert Systems-A Practical Introduction*, Macmillan, Southampton, 1985.
- [23] Suwa M., Scott A. C., Shortliffe E. H., "An Approach To Verifying Completeness and Consistency in a Rule Based Expert System," *AI Magazine* 3(4), pp 16-21, 1982.

- [24] Szolovits, P., "Expert Systems Tools and Techniques: Past , Present and Future," *AI in the 1980s and Beyond*, W.Eric, L.Grimson, R.S.Patil, MIT Press, Massachusetts, pp 43-74, 1987.
- [25] Wolfgram D., Dear T. J., Galbraith C. S., *Expert Systems for the Technical Professional*, John Wiley Sons Inc., New York, 1987.

A. BNF DESCRIPTION OF RULES AND FACTS

```
<Rule> ::= if <Condition_part>
           then <Action_part> [ <Certainty_Factor> ] ;

<Fact> ::= <Predicate> [ <Certainty_Factor> ] ;

<Condition_part> ::= <Predicate_list>

<Action_part> ::= <Predicate>

<Predicate_list> ::= [~]<Predicate> & <Predicate_list> | [~]<Predicate>

<Predicate> ::= <Predicate_name> ( <Argument_list> )

<Argument_list> ::= <Parameter_list> | <Epsilon>

<Parameter_list> ::= <Argument> , <Parameter_list> | <Argument>

<Argument> ::= <Predicate> | <Variable> | <Constant>

<Predicate_name> ::= <Lower_case_letter> {<Digit> | <Letter>}^

<Variable> ::= <Upper_case_letter> {<Digit> | <Letter>}^

<Constant> ::= <Lower_case_letter> {<Digit> | <Letter>}^ | <Digit>
               {<Digit> | <Letter>}^ | " <String> " | <Real_number>

<Certainty_Factor> ::= 0. <Digit>^+ | 1.0

<Real_number> ::= <Digit>^+ . <Digit>^

<String> ::= <Ascii_character>^

<Digit> ::= 0 | 1 | ... | 9
```

`<Lower_case_letter> ::= a | b | ... | z`

`<Upper_case_letter> ::= A | B | ... | Z`

`<Letter> ::= <Lower_case_letter> | <Upper_case_letter>`

`<Epsilon>` is used to denote empty string

where `<Condition_part>` is conjunction of predicates

`<Action_part>` is a single predicate

`<Certainty_Factor>` is a real number between 0 and 1 which indicates the probability for the occurrence of `<Action part>` if the condition part is known with certainty.

B. A SAMPLE KNOWLEDGE BASE VERIFICATION

THE INITIAL KNOWLEDGE BASE

```
Rule 1 : if predicate40 (A,predicate41 (C) ,A,"fast computer")
          then
              predicate42 (A,C) [0.9] ;

Rule 2 : if predicate42 (D,B)
          then
              predicate43 (D,a,b) [0.9] ;

Rule 3 : if predicate43 (K,a,b)
          then
              predicate44 (K,predicate45 (K,K,A) ,A) [0.9] ;

Rule 4 : if predicate40 (B,predicate41 (D) ,B,"fast computer")
          then
              ~predicate44 (B,predicate45 (B,B,Z) ,Z) [0.4] ;

Rule 5 : if predicate50 (X,C) &
          predicate51 (B,X,a)
          then
              predicate52 (X) [0.9] ;

Rule 6 : if predicate51 (A,Y,a)
          then predicate52 (Y) [0.8]
```

```

Rule 7 : if predicate1 (B) &
          predicate2 (X,N)
      then
          predicate3 (X,B) [0.9] ;

Rule 8 : if predicate2 (X,X) &
          predicate3 (X,B)
      then
          predicate1 (A) [0.8] ;

Rule 9 : if ~predicate3 (A,B) &
          predicate2 (Y,Y)
      then
          predicate1 (C) [0.5] ;

Rule 10 : if predicate3 (CC,D) &
           predicate2 (E,E)
      then
          predicate1 (C) [0.5] ;

Rule 11 : if predicate4 (X,Y) &
           predicate4 (Y,Z)
      then
          predicate5 (X,a) [0.8] ;

Rule 12 : if predicate4 (b,Y) &
           predicate4 (Y,Z)
      then
          ~predicate6 (a) [0.6] ;

Rule 13 : if predicate3 (Y,C)
      then
          predicate1 (S) [0.6] ;

Rule 14 : if predicate5 (b,Y)
      then
          predicate6 (Y) [0.5] ;

Rule 15 : if predicate5 (c,X)

```

```

        then
            predicate7 (Z) [0.6] ;

Rule 16 : if    predicate5 (b,Z)
        then
            predicate8 (Xi) [0.8] ;

Rule 17 : if    predicate6 (Y)
        then
            predicate8 (Y) [0.6] ;

Rule 18 : if    predicate6 (a)
        then
            predicate9 (b,A,Y) [0.8] ;

Rule 19 : if    predicate9 (b,a,T)
        then
            predicate10 (a,T,T,X) [0.4] ;

Rule 20 : if    predicate8 (Z)
        then
            predicate11 (Y,X,B,Z,Y) [0.6] ;
        .

Rule 21 : if    predicate25 (A,C) &
            predicate26 (a,B)
        then
            predicate27 (C,a) [0.9] ;

Rule 22 : if    predicate28 (D) &
            predicate29 (B)
        then
            predicate25 (D,B) [0.7] ;

Rule 23 : if    predicate30 (a,E) &
            .
            predicate27 (F,a)
        then
            predicate29 (E) [0.5] ;

Rule 24 : if    predicate30 (b,B)
        then

```

```

predicate26 (B) [0.4] ;

Rule 25 : if predicate31 (joe,X) &
           predicate32 (Y,X,10) &
           predicate32 (Y,20,Z)
    then
      ~predicate32 (T,20,Z) [0.9] ;

Rule 26 : if predicate31 (joe,X) &
           predicate32 (Z,Y,jane) &
           ~predicate31 (joe,Y)
    then
      predicate33 (Y,high) [0.45] ;

Rule 27 : if predicate31 (joe,X) &
           predicate31 (jane,Z) &
           predicate32 (Y,20,Z)
    then
      ~predicate32 (T,20,Z) [0.9] ;

Rule 28 : if predicate31 (joe,X) &
           predicate32 (Z,joe,Y) &
           predicate31 (joe,Y)
    then
      predicate33 (Y,T) [0.9] ;

Rule 29 : if predicate31 (joe,X) &
           predicate31 (Y,X) &
           predicate34 (Y,T)
    then predicate34 (Z,X) [0.9] ;

Fact 1 : predicate1 (a) [0.8] ;

Fact 2 : predicate5 (ali,A) [0.5] ;

Fact 3 : predicate30 (b,T) [0.8] ;

Fact 4 : predicate30 (a,T) [0.9] ;

Fact 5 : predicate28 (D) [1.0] ;

```

```
Fact 6 : predicate31 (joe,X) [1.0] ;  
  
Fact 7 : predicate32 (A,B,C) [1.0] ;  
  
Fact 8 : predicate40 (13,predicate41 (X),13,"fast computer") [1.0,1.0]  
;  
  
Fact 9 : predicate42 (a,b) [0.9] ;  
  
Fact 10 : predicate43 (c,a,b) [0.9] ;  
  
Fact 11 : predicate50 (T,c) [1.0] ;  
  
Fact 12 : predicate51 (X,c,a) [1.0] ;
```

THRESHOLD : 0.0001

START COMPILATION

Compilation OKAY

END COMPILATION

THE INFERRED RULES

```
Rule 30 : if predicate42 (D,B)  
          then  
                  predicate44 (D,predicate45 (D,D,A) ,A) [0.81] ;  
  
Rule 31 : if predicate5 (b,Y)  
          then  
                  predicate8 (Y) [0.3] ;  
  
Rule 32 : if predicate6 (Y)  
          then  
                  predicate11 (Var_1,X,B,Y,Var_1) [0.36] ;
```

```

Rule 33 : if    predicate5 (b,Y)
           then
               predicate11 (Var_1,X,B,Y,Var_1) [0.18] ;

Rule 34 : if    predicate5 (b,Y)
           then
               predicate11 (Var_2,X,B,Xi,Var_2) [0.48] ;

Rule 35 : if    predicate5 (b,a)
           then
               predicate9 (b,A,Var_3) [0.4] ;

Rule 36 : if    predicate6 (a)
           then
               predicate10 (a,Y,Y,X) [0.32] ;

Rule 37 : if    predicate40 (A,predicate41 (C) ,A,"fast computer")
           then
               predicate43 (A,a,b) [0.81]

Rule 38 : if    predicate40 (A,predicate41 (C) ,A,"fast computer")
           then
               predicate44 (A,predicate45 (A,A,Var_4) ,Var_4) [0.729]

Rule 39 : if    predicate1 (B)  &
           predicate2 (X,N)
           then
               predicate1 (S) [0.54] ;

Rule 40 : if    predicate4 (b,Y)  &
           predicate4 (Y,Z)
           then
               predicate6 (a) [0.4] ;

Rule 41 : if    predicate4 (c,Y)  &
           predicate4 (Y,Z)
           then
               predicate7 (Var_4) [0.48] ;

```

```

Rule 42 : if predicate4 (b,Y) &
           predicate4 (Y,Z)
       then
           predicate8 (Xi) [0.64] ;

Rule 43 : if predicate4 (b,Y) &
           predicate4 (Y,Z)
       then
           predicate8 (a) [0.24] ;

Rule 44 : if predicate4 (b,Y) &
           predicate4 (Y,Z)
       then
           predicate11 (Var_1,Var_6,B,a,Var_1) [0.144] ;

Rule 45 : if predicate4 (b,Y) &
           predicate4 (Y,Z)
       then
           predicate11 (Var_2,Var_7,B,Xi,Var_2) [0.384] ;

Rule 46 : if predicate4 (b,Y) &
           predicate4 (Y,Z)
       then
           predicate9 (b,A,Var_3) [0.32] ;

Rule 47 : if predicate5 (b,a)
       then
           predicate10 (a,Var_3,Var_3,X) [0.16] ;

Rule 48 : if predicate4 (b,Y) &
           predicate4 (Y,Z)
       then
           predicate10 (a,Var_10,Var_10,X) [0.128] ;

```

=====

Rules with ID numbers > 29 are inferred rules

CYCLES and CONTRADICTIONS CHECKS WITHIN RULES

The 2nd condition and then part of rule 25 may contradict to each other

The 3rd condition and then part of rule 25 may contradict to each other

The 1st and 3rd conditions of rule 26 may contradict to each other

The 3rd condition and then part of rule 27 contradict to each other

There may be a self cycle in rule 29

There is a self cycle in rule 39

REDUNDANT RULES

Rule 8 and rule 10 may be redundant

Rule 16 and rule 31 (an inferred one) may be redundant

Rule 33 (an inferred one) and rule 34 (an inferred one)
may be redundant

Rule 42 (an inferred one) and rule 43 (an inferred one)
may be redundant

Rule 44 (an inferred one) and rule 45 (an inferred one)
may be redundant

CONFLICTING RULES

Rule 4 and rule 38 (an inferred one) are conflicting

Rule 12 and rule 40 (an inferred one) are conflicting

SUBSUMED RULES

Rule 5 is sumsumed by rule 6

Rule 8 may be sumsumed by rule 13

Rule 10 is sumsumed by rule 13

THE RULES HAVING UNNECESSARY IF CONDITIONS

Rule 8 and rule 9 may have unnecessary if conditions

Rule 9 and rule 10 have unnecessary if conditions

Rule 9 and rule 13 have unnecessary if conditions

Rule 26 and rule 28 may have unnecessary if conditions

DEAD-END RULES

Rule 7 cannot be satisfied

Rule 8 cannot be satisfied

Rule 9 cannot be satisfied

Rule 10 cannot be satisfied

Rule 11 cannot be satisfied
Rule 12 cannot be satisfied
Rule 21 cannot be satisfied
Rule 27 cannot be satisfied
Rule 39 cannot be satisfied
Rule 40 cannot be satisfied
Rule 41 cannot be satisfied
Rule 42 cannot be satisfied
Rule 43 cannot be satisfied
Rule 44 cannot be satisfied
Rule 45 cannot be satisfied
Rule 46 cannot be satisfied
Rule 48 cannot be satisfied

CYCLES DUE TO DEPENDENCIES

=====

The following series of rules may cause circulation
7 8 7

The following series of rules may cause circulation
21 22 23 21

=====

END OF CONSULTATION