1-1-1992

A study of the methodologies currently available for the maintenance of the knowledge-base in an expert system

Kai Teh
Edith Cowan University

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

A STUDY OF THE METHODOLOGIES CURRENTLY AVAILABLE FOR THE MAINTENANCE OF THE KNOWLEDGE-BASE IN AN EXPERT SYSTEM

BY

KAI TEH ***B.Sc., Post-grad Dip. in Computing Science.***

A Thesis Submitted in the Partial Fulfilment of the Requirements

for the Award of

Master of Applied Science (Computer Studies)

at the School of Information Technology and Mathematics,

Edith Cowan Unversity

Name of Supervisor : Tim Roberts

Date of Submission : 29-5-92

ABSTRACT

This research studies currently available maintenance methodologies for expert system knowledge bases and taxonomically classifies them according to the functions they perform.

The classification falls into two broad categories. These are :

(1) Methodologies for building a more maintainable expert system knowledge base. This section covers techniques applicable to the development phases. Software engineering approaches as well as other approaches are discussed.

(2) Methodologies for maintaining an existing knowledge base. This section is concerned with the continued maintenance of an existing knowledge base. It is divided into three subsections. The first subsection discusses tools and techniques which aid the understanding of a knowledge base. The second looks at tools which facilitate the actual modification of the knowledge base, while the last section examines tools used for the verification or validation of the knowledge base.

Every main methodology or tool selected for this study is analysed according to the function it was designed to perform (or its objective); the concept or principles behind

the tool or methodology; and its implementation details. This is followed by a general comment at the end of the analysis.

Although expert systems as a rule contain significant amount of information related to the user interface, database interface, integration with conventional software for numerical calculations, integration with other knowledge bases through black boarding systems or network interactions, this research is confined to the maintenance of the knowledge base only and does not address the maintenance of these interfaces.

Also not included in this thesis are Truth Maintenance Systems. While a Truth Maintenance System (TMS) automatically updates a knowledge base during execution time, these update operations are not considered 'maintenance' in the sense as used in this thesis. Maintenance in the context of this thesis refers to perfective, adaptive, and corrective maintenance (see introduction to chapter 4). TMS on the other hand refers to a collection of techniques for doing belief revision (Martin, 1990) . That is, a TMS maintains a set of beliefs or facts in the knowledge base to ensure that they remain consistent during execution time. From this perspective, TMS is not regarded as a knowledge base maintenance tool for the purpose of this study.

DECLARATION

I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or a diploma in any institution of higher education; and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature

Date 29-5-92

ACKNOWLEDGMENTS

The author wishes to give special thanks to Mr. Tim Roberts for having provided some invaluable comments and guidance during the course of writing the thesis, and the librarian Jenny Renner, who went to considerable length to make sure that I obtained all the materials required. My thanks also extend to Mr. Geoff Sutcliffe for his kind advice.

CONTENTS

CHAPTER FOUR

CHAPTER FIVE

APPENDIX

BIBLIOGRAPHY

LIST OF TABLES

Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 BACKGROUND

Often the principal cost of a computer project is the maintenance cost. This is particularly evident in the case of large computer systems.

Quoting from the U.S. Department of Commerce figures of October, 1985, Carrico, Girard, and Jones (1989, p. 219) claimed that over a software project's life cycle, maintenance takes up more time, money and resources than any other aspect of the project and that "software maintenance accounts for 60 to 70 % of each software dollar allocated".

In the U.K., Lientz and Swanson, in a major survey conducted in the late seventies, found that some firms were spending up to 70% of their computing efforts on maintenance (quoted in Bennett, 1991, p. 75). Martin and McClure stated that over $20 billion per year was being spent worldwide on the maintenance of software systems (Martin and McClure, 1983). In an article published in 1988 Parikh claimed that more than $30 billion per year was spent on maintenance of software systems, and that most companies allocate 50% of their DP budget for maintenance (Parikh, 1988, p. 13).

Maintenance issues have for too long been largely ignored by computer professionals. Parikh, in a bid to highlight this gross neglect, went so far as to call maintenance a "taboo subject" (Parikh, 1988, p. 34). He pointed out that in the U.S. this " pervasive lack of attention to the subject [of maintenance] persists on a national level" (Parikh, 1988, p. 13). Gunderman lamented the fact that traditionally DP personnel had always considered maintenance as a second class activity, something for the beginners' on-the-job training or a low status assignment for the outcasts and the fallen (Gunderman, 1988, p. 55). Liu said that analysts see the maintenance function as an

inferior assignment (Liu, 1988, p. 61), and so did several others who bemoan this situation.

Paradoxically, the situation is one where on the one hand maintenance is the costliest phase of the systems life cycle, and yet on the other it is relegated to the lowest priority in that life cycle.

1.2 NEED FOR THE STUDY

From the foregoing discussion, there appears to be a need to highlight this important but neglected phase of an expert system project.

Maintenance in the context of expert systems is thought to be even more problematical and costlier than maintenance in the case of traditional systems. Among those who hold this view is Prerau. He declared that "in most instances the largest costs in the life cycle of a computer program are for program maintenance" and that this is "true to an even greater degree for an expert system program where the knowledge as well as the code must be maintained" (Prerau, 1990, p. 287). Hicks said knowledge maintenance is more dynamic than traditional data processing maintenance. He also noted that knowledge is not common, therefore not easily reusable. Besides, knowledge is often not well documented and must be acquired and tested incrementally (Hicks, 1990, p. 293).

The claim that expert system maintenance has been for the most part largely neglected is evidenced by the fact that until recent times expert systems were to be found mainly in research laboratories where maintenance issues were not a priority. Nau acknowledged the severity of this negligence by reminding us that "since expert systems have until recently been largely experimental, we have not had to consider the need for long-term maintenance ... More attention will have to be paid to these 'real world details' if expert systems are to be useful in the long run" (Nau, 1988, p. 73).

Prerau, Gunerson, Reinke, and Adler (1990) point us to the fact that making expert systems more maintainable has not been a major concern either in AI or Software engineering. Instead most work in this area has been focussed on producing a new generation of development tools (Prerau et al., 1990, p. 71). This emphasis on software development to the neglect of maintenance is seen by Parikh as "trying to fly with one wing" (Parikh, 1988, p. 22).

The legacy from this lack of attention to maintenance issues in the early days has caused many older expert systems to be rewritten because they have simply become unmaintainable.

At the Commonwealth Scientific Industrial Research Organisation (CSIRO), Jansen reported that two expert systems, the Garvan thyroid expert system and the SIRATAC cotton management expert system had to be redeveloped. In his words, "... both have a common reason for their redevelopment. They have become difficult if not impossible to maintain." (Jansen 1988, p. 101). For the same reason XCON's knowledge base, which over 7 years has grown to 6200 rules, has become so difficult to maintain that a new version, XCON-in-RIME is being written as the successor to XCON (Soloway, Bachant, & Jensen, 1987).

Signs are beginning to appear that this gross negligence of maintenance is gradually being put right. In recent times, as we witness more and more expert systems being used in the areas of industry, commerce, computer hardware and software support, aerospace, transportation, etc, computer professionals are increasingly coming to grips with the reality of maintenance issues.

As knowledge bases grow larger and become more complex, there appears to be an undercurrent of new urgency which is driving researchers in different directions in their scramble to come up with better maintenance tools, more superior methodologies, or more innovative ideas. The result of this is evidenced by the myriad of tools and techniques on the market.

When confronted with such a diversity of ideas and tools it is easy for a maintainer to be confused about where to begin. An important aim of this study is therefore to provide a taxonomic classification of these methodologies with respect to their proposed maintenance paradigms. It is hoped that such a classification will go some distance to help shorten the time developers or maintainers of expert system knowledge bases spend on searching the literature and will allow them to gain a quick insight into what is available in the market.

By classifying these tools in terms of the rationale or philosophy behind their creation, a maintainer is free to concentrate on the tools' underlying principles rather than on the specific tools themselves. This is important because while a particular tool may be out of the reach of the maintainer, the maintenance philosophy or rationale is not.

Many of the tools presented here have been used successfully on only one or two applications; nevertheless, the concepts behind them may be found to be generally applicable.

CHAPTER 2

OBJECTIVES

The objectives of this thesis are as follows :

1) to analyse currently available maintenance methodologies, and

2) to taxonomically classify them in terms of their proposed maintenance paradigms.

In this research, maintenance is seen as fundamentally involving a process which consists of the following three steps :-

i) Understanding of the knowledge base prior to making any changes.

ii) Physically modifying the existing knowledge base.

iii) Revalidating the modified knowledge base to ensure that no errors or inconsistencies have been introduced.

While a knowledge engineer will invariably go through the above three steps when maintaining an existing knowledge base, these three steps are not, however, confined to the maintenance process alone. For instance, in order to aid in the understanding of a knowledge base, good documentation, explanation facilities, and well structured and formatted code are required. This is essentially a design issue rather than a maintenance issue.

To facilitate the actual process of modifying the knowledge base, modularity of knowledge, easy-to-update documentation, knowledge structures which minimise the

effect of changes and codes that are easy to expand and easy to update are required. This again is a design issue.

To aid in revalidating the modified knowledge base in order to ensure that no inconsistencies have been introduced, knowledge base structures which facilitate selective retest and good traceability between specification requirements and code are desirable. It is obvious that this too is a design issue and should be considered during the building of the knowledge base.

In other words, to consider maintenance issues, one has to look beyond just maintenance - how the knowledge base has been built in the first place is equally as important (if not more so).

Hence this thesis is constructed along the following lines :

(1) A discussion of techniques for building more maintainable expert system knowledge bases.

(2) A discussion of techniques for maintaining existing knowledge bases.

Chapters 3 and 4 take up these two sections respectively.

CHAPTER 3

BUILDING MAINTAINABLE KNOWLEDGE BASES

This chapter is concerned with tools and techniques applicable to the knowledge base development phases.

The chapter is divided into two sections; the first section focuses on software engineering approaches while the second looks at other attempts.

3.1 SOFTWARE ENGINEERING APPROACH

Of the existing paradigms for building a more maintainable knowledge base, a vast majority to different degrees owe their existence to software engineering principles - that vast collection of methodologies and techniques for the development and management of software.

While software engineering principles are important in the construction of maintainable systems, they are not unanimously embraced by the AI fraternity. There are dissenting voices which question their relevance to expert system constructions. Among the doubters is Jansen. His line of argument is that since the main development methodology of expert systems is the knowledge acquisition phase combined with a literature search for the correct knowledge representation formalism, there is a question as to whether software engineering techniques have any use in the development of expert systems (Jansen, 1988, p. 102).

Partridge on the other hand sees some overlap between software engineering and AI problems. He declared that "Software engineering problems are a subset of AI problems : the subset of well-defined [AI] problems" (Partridge, 1986, p. 19). Unless we consider expert systems problems as well-defined problems then clearly expert systems fall outside this software engineering category according to Partridge's definition. Since it is doubtful that one would call expert system problems well-defined, by virtue of his argument one may conclude that software engineering solutions are not the best suited for solving expert system problems.

While such well meaning cautionary voices which constantly remind us to rethink our position are appreciated and their points taken, it is nonetheless undeniable that software engineering techniques do have a place in the construction of expert systems. This is evidenced by the many useful tools and strategies outlined in the following pages which are built around software engineering principles.

The current author feels software engineering to be sufficiently important to devote a section of Chapter 3 to its discussion. The software engineering principles discussed here include rigorous definition, modularity concepts, data dictionary usage, normalisation techniques and structured techniques among others.

3.1.1 RIGOROUS DEFINITION

A major pillar of software engineering is the rigorous definition of requirements. Rigorous definition in this context refers to the complete pre-specification of all logical user requirements in detail prior to the design and construction of the actual physical system. Thus the rigorous definition approach would generally cover the use of structured methodologies, data flow diagram analysis techniques, the traditional life cycle approach, conceptual modelling techniques, and others. All of these techniques fundamentally rely on the principles of rigorous definition to build some conceptual or logical model before proceeding to the construction phase.

In theory the principle of rigorous definition appears sound. If we 'get it right in the first place' through rigorous specification, then maintenance problems should be greatly reduced.

Boehm had shown that modifying a system after it has been put into operation can cost several hundreds of times more than modifying it in the early stages (Boehm, 1981, p. 40). This underlies the importance of the well-documented software engineering principle of getting it right the first time - what is to be done must be rigorously specified, how to do it is relatively insignificant.

However, in the case of expert systems this wisdom may not necessarily be true. Partridge said that since rigorous definitions "specify what the system should do rather

than how it should do it, it is here that we find our first important point of contrast with AI problems" (Partridge, 1986, p. 31).

Sacerdoti also disagrees with the rigorous definition viewpoint in the context of expert systems development. He says that "an expert system does not fit well into conventional software engineering paradigm because a detailed specification or functional definition cannot be written before coding" (Sacerdoti, 1991, p. 26).

Rolston argued that a complete understanding of the system requirements is not possible and cannot be derived at the start of a project because "iteration is inevitable in any large software development project" (Rolston, 1988 , p. 134).

Not all authors, however, hold this view. Keller is a strong advocate of structured techniques for the development of expert systems. He extolled the virtues of rigorous definition by proclaiming that "structured system development techniques offer a more appropriate approach to AI system development" (Keller, 1987, p. 2), and dedicated his book to showing how traditional system development technologies can be applied to expert systems development.

While there are many proponents of rigorous definition for both the building of conventional as well as expert systems, it should be pointed out that even amidst the ranks of conventional systems practitioners there are dissenters to the rigorous definition concept.

Apart from specifying very highly structured systems, rigorous definition does not appear to be very practicable since it fundamentally assumes that users know exactly what they want. In complex systems it is unlikely that users know precisely what they want or what is best for them.

Even if users do know what they want, it cannot be assumed that they are able to spell out their requirements precisely. Too often there is an unbridgeable communication gap between the users and the system developers.

Boar said that those who advocate rigorous specification presuppose "all requirements can be specified, ... the project team is capable of unambiguous communication ... a rigorous approach is inherently the correct approach for all life cycle phases" (Boar, 1984, p. 20). He argued that all these assumptions are flawed.

Vitalari considered structured methodologies from a cognitive and psychological perspective (Vitalari, 1984). He felt that information requirements definition is too complex for current structured methodologies to handle. Such methods are only good for documentation and writing specifications once they have been elicited, but are poor elicitors of information themselves. Hence rigorous definitions are not possible using current methodologies.

Both Boar and Vitalari offered their solutions. While Vitalari's remedy is to develop a new generation of structured methodologies, Boar advocates the use of prototyping.

In the midst of all these arguments, some developers, meanwhile, continue to build expert systems following the rigorous definition doctrine. One such example is the COLOSSUS system. The following section takes a look at COLOSSUS to try to understand why its developers supported rigorous definitions sufficiently to adopt this in the building of expert systems.

3.1.1.1 NAME OF SYSTEM : COLOSSUS

INTRODUCTION

COLOSSUS (Beinat & Smart, 1989) was developed using the 'conceptual modelling' methodology. It is an expert system to handle third party insurance claims, jointly built by G.I.O. of New South Wales, Australia, and Software Computations.

DISCUSSION

Beinat and Smart were staunch opponents of the prototyping methodology. They made this clear in their paper by putting up a strong case against prototyping while promoting the virtues of the conceptual modelling technique.

Among the downsides of the prototyping methodology claimed by Beinat and Smart is that in prototyping "The interaction is uni-directional" (Beinat & Smart, 1989, p. 76). The domain experts are not actively involved in the development of the conceptual model. (The conceptual model is the knowledge engineer's picture of the problem and its theoretical solution, in short the rigorous definition). Thus the experts have no idea of the cause of any future problems that might occur to the system and will be of little assistance in their correction.

The appeal of the prototyping methodology lies in its "political advantage in eliciting management support for the project" since by using prototyping, "a visual indication of progress can be achieved very early in the project" (Beinat & Smart, 1989, p. 76).

In contrast, conceptual modelling concentrates on the developing of two models :-

i) the strategic model - this is the domain expert's view of the model.
ii) the implementation model - this is the knowledge engineer's view of the model.

Beinat and Smart said that this dual nature of the conceptual model demands that the expert and the knowledge engineer work together to formulate and validate the representation of the problem and its solution. Due to this close involvement with the project the expert will be in a position to lead maintenance activity at a later stage.

This opinion appears to be in contradiction to the widely held view that prototyping methodology fosters greater user participation than traditional methods. Also, in the

prototyping methodology the experts can see their systems materialise into concrete systems hence no power of imagination is called for. With the conceptual modelling methodology a lot of imagination, on what a future system is going to look like and how it is going to work, is required.

Beinat and Smart cited three phases in conceptual modelling. These are the learning phase, the modelling phase and the construction phase.

During the first two phases no tangible results can be seen. He admitted that "the drawback of this methodology is that it is not possible to produce any tangible result until the third phase, well into the project" (Beinat & Smart, 1989, p. 79).

What he failed to mention is the customary shock users may receive when presented with the system for the first time. More often than not this does not coincide with their imagined system.

Beinat and Smart also conceded that "management must have confidence in the project team before this methodology is viable" (Beinat & Smart, 1989, p. 79). He did not, however, mention if that confidence was in abundant supply.

CONCLUSION

The reasons why rigorous definition still attracts a following even among expert systems developers may be that :

i) the expert systems they are developing are highly structured, rather akin to conventional data processing systems involving large databases and structured procedures,

ii) these developers may have their roots in conventional systems development, and are reluctant to abandon pre-held concepts.

In concluding, it must be conceded that the rigorous definition principle is fundamentally sound, but it should be approached with caution in the development of conventional systems, and to an even greater degree in expert systems development.

3.1.2 MODULARITY

INTRODUCTION

Among software engineering practices, the modular approach is perhaps the most useful and simple in concept. It is straightforward and easy to adapt to any system, yet effective in building maintainable systems.

The 'Chunking' phenomena (Adelson, 1990) suggests that experts generally solve problems by structuring them into clusters or chunks of information. This is evidenced by chess masters who recall game boards as functional clusters, and electronic engineers who recall clusters of circuit diagrams. In each case, they use the functional relationships which exist among the elements of the problem to structure them into chunks.

Adelson also noted that mathematicians usually transform complex equations into more modular forms by replacing them with temporary variables.

Since it is a natural phenomenon for humans to solve large problems by decomposing them into related sub-components, it seems reasonable to propose that modular representation should not just be viewed as a technique to facilitate maintenance but as a natural way to represent rules.

While there can be little dispute as to the effectiveness of modularity, the question of 'how to modularise?' appears to be a difficult one to answer.

Should we modularise according to a system's functionality or modularise to achieve structuredness and readability? Should the modules reflect the expert's knowledge or should they be structured for the convenience of the structured tool used? Should we separate implementation knowledge and domain knowledge into different modules? In a system that uses multiple knowledge representation schemes should we modularise according to the multiple knowledge representation paradigms used? These questions and their like will no doubt confront system developers who are planning to introduce software engineering practices into their design. Hence an aim of this section is to find out what researchers think should be modularised in what kind of applications.

Then there is the question of 'how to implement these modules?'. Should we implement them as multiple modules within a knowledge base, or should we have multiple knowledge bases?

For example, these are some of the ways taken by researchers in modularising their systems :-

- COLOSSUS (Beinat & Smart, 1989) was modularised on the basis of logical discrete problem solving components, (each component is called a 'focus control block'). Each block contains no more than 500 rules out of a total of some 5000 rules in Colossus.

- COMPASS (Prerau, 1990) uses the concept of 'multiple knowledge bases' to separate its knowledge base into eighteen distinct knowledge bases.

- LOOPS uses the notion of a ruleset which can be 'called' like a subroutine (Jacob & Froscher, 1990, p. 173)

- LOAN PROBE comprises 33 knowledge bases that communicate through a blackboard system (Ribar, Arcoleo, & Hollo, 1991).

- XCON's (Soloway et al., 1987) knowledge base is partitioned into 'subtasks'.

- Jacob and Froscher talked about the 'Interface specification' concept (Jacob & Froscher, 1990).

- Payne developed the 'Knowledge Flow Module' concept (Payne, 1991).

The following section looks at several modular approaches which have been chosen for their diversity of techniques. The concepts they propound may be adapted to the building of most knowledge bases. These are the 'Interface Specification' approach, the 'Knowledge Flow Module' approach and the 'multiple knowledge bases' concepts of COMPASS.

The first two approaches may be implemented in a single knowledge base, while the last approach is implemented using multiple knowledge bases.

3.1.2.1 INTERFACE SPECIFICATION APPROACH

INTRODUCTION

The interface specification approach (Jacob & Froscher, 1990), (Davis, 1990) makes the knowledge base easier to change by localising the effects of changes within the modules.

The approach may be seen as a general method since it may be used without reliance on software tools (although it is preferable to have them). It is applicable to a knowledge base whichever way it is partitioned. This is because the approach's main focus is on the information flow between different modules and their clear specification. A key aim of this approach is to reduce the amount of information that knowledge engineers have to understand before they can make a change to the module. This is achieved by :-

i) limiting the amount of information flow among the various modules, hence the effects of changes within modules are reduced,

ii) formally specifying the information flow between modules, hence making the functions of modules easy to understand.

CONCEPT BEHIND THE APPROACH

A knowledge base may contain two types of knowledge, namely control knowledge and domain knowledge. Control knowledge is that used to enable or disable the firing of rules, while domain knowledge is that which carries information between rules.

This method is applied only to the domain knowledge, hence it requires the separation of control knowledge from the domain knowledge. In some languages domain knowledge is expressed in rules while control knowledge is expressed in a different notation, so that this segregation is already made. Examples of such languages given by Jacob et al. are KES, and ORBS (Jacob et al., 1990, p. 175).

This discussion considers a knowledge base that contains only domain knowledge.

The method may be applied at the time the knowledge base is created or it may be applied after a prototype has been built. The idea is to divide rules into groups such that each group contains all the rules relevant to one specific, small area of knowledge. For example a group may contain rules for checking if an animal is a mammal, while another group contains rules to test if a mammal is a carnivore, etc.

Groups are allowed to contain subgroups, thus forming a hierarchy of groups.

Within groups there are rules. Rules are made up of facts. There are two types of facts, namely local facts, and intergroup facts. Local facts are produced and used within the group only and do not effect the rest of the system. Intergroup facts on the other hand are produced by one group and used by another. In other words intergroup facts provide the linkages between the groups.

There are two main types of intergroup facts. Those that are produced or modified by rules in a group, and those that are merely examined by rules in a group. The first type are referred to as PRODUCED facts, where the latter are the USED facts.

The interface specification method relies heavily on the clear specification of these intergroup facts. These specifications are merely documentations and do not affect the overall performance of the system. The specification in effect summarises the workings of the group that produces it.

To modify a group the knowledge engineers do not have to understand the whole knowledge base. They merely have to understand the internal workings of that group by studying these specifications, and more importantly, they must preserve the integrity of these specifications when making changes.

IMPLEMENTATION DETAILS

Starting with a knowledge base that contains only domain knowledge, the following steps need to be carried out to modularise it.

Step 1 Separate rules into groups

Rules are first separated into a hierarchy of groups. The basis of this separation is to look for rules that affect one another. Such rules are likely to be changed at the same time.

This separation may be carried out manually or it may be automated by the use of a grouping algorithm. The algorithm explored by the developers is called a 'clustering algorithm'.

The algorithm considers two rules as related if the same fact has been mentioned by both rules. Rather than making a binary valued check on whether two rules are related, the clustering algorithm uses a weighting factor to measure the extent of such relatedness.

For example, consider the following cases :

case 1

if x then y

if y then z

case 2

if x then y

if z then y

In both cases y is a shared fact. However the rules in case 1 are more related than the rules in case 2 because the rules in the former case have a greater programming dependency.

A detailed working of the clustering algorithm is given in the article by Jacob et al. (Jacob, et al. 1990, p. 184).

Step 2 Within each group pick out local and intergroup facts.

This step can also be automated since an algorithm can be applied to check if a given fact is used by rules within a single group (local fact) or whether it spans other groups (intergroup facts).

Intergroup facts whose values are produced or modified by rules in a group will be flagged by the algorithm as 'PRODUCE' facts, while those whose values are examined by rules in a group are flagged as 'USE' facts.

For example, consider the following rules of a group * :

Group A

PRODUCE A1
USE B1
USE B2

Rule 1 : if X then A1.

Rule 2 : if B1 then C.

Rule 3 : if B2 then C.

Rule 4 : if C then A2.

A1 is flagged as 'PRODUCED' ** because it is produced (ie set or modified) by this group (Group A) and it will be used by another group (not shown here).

A2 is also produced by this group (Group A) but it is not flagged because it is not used by any other group. A2 is not used by any other group because it is either a local fact or a top level output of the system.

B1 and B2 are declared as 'USE' because they are produced by other groups (not shown) and are examined by this group.

* The above discusses rules within a single group. In the case of groups within groups, other concepts, like GLOBAL, UP, DOWN, have to be considered. They are not discussed here.

** PRODUCE and USE are merely documentary and they do not affect the execution of the program.

C is also used by this group but is not declared because it is not produced by any other group. That is C is a local fact.

Step 3 Write external descriptions for intergroup facts

This is arguably the most crucial step of the whole process since upon it rest the descriptions of the group which knowledge engineers rely on for their understanding when making changes to that group. However, it is also the only step that defies automation.

In this step the developers of each group that produces intergroup facts must provide descriptions for such facts. The descriptions should specify what will remain true of that fact in the future.

The description summarises the internal workings of the group that produces it. Jacob et al. said that these descriptions should be written as "a higher level informal statement of the aspects of the output that will not change and may be considered externally visible" (Jacob, et al., 1990, p. 176). By this, it is meant, rather than writing a statement as (i) "X is true if A > B", the description could be written as (ii) "X gives the system the best estimate of whether the patient has flu".

Information about "A > B" should not be specified because such internal details may change. Besides, writing a description as statement (i) would essentially be repeating the entire group of rules as they presently are.

Writing as statement (ii) will present a higher level description of the output that will not change even when modifications are made to the details of the internal rules.

To modify such a knowledge base, the knowledge engineers need to pay particular attention to the intergroup facts since other groups can be affected by

a change in this group. They must rely on these descriptions for their understanding of the group's internal workings, and at the same time ensure that they preserve the validity of the descriptions after the change.

COMMENTS

- Although this method was proposed for used with production rules, its underlying concept should be applicable to other formalisms as well. The production system has chosen because the researchers felt that it was the most widely used type of knowledge representation in expert systems.

- While some of the steps described above can be automated, theoretically they can also be done manually if no relevant software tools are available.

- Since the statements (eg USE, PRODUCE, etc) used in this method are merely documentary, they do not affect the execution of the system. Hence they may be applied at any stage in the development of the knowledge base.

  However judgement should be used with regard to when to apply them. Applying them too early in the development stage when the system is still unstable may result in more work than benefit.

- Although the researchers claimed that this is a 'new method' (Jacob et al., 1990, p. 188), its 'group' concept is rather similar to Pascal's 'procedure' concept; the 'USE' concept resembles Pascal's 'by value' parameter concept and 'PRODUCE' resembles 'by reference' parameter.

3.1.2.2 KNOWLEDGE FLOW MODEL

INTRODUCTION

The 'knowledge flow model' approach (Payne, 1991) modularises an expert system by decomposing it into different application techniques, each application technique reflecting a unique aspect of the application.

The developers of this approach claimed that many expert systems failed because their creators had misclassified them under one of these stringent categories like diagnosis, monitoring, planning, design, etc. (Each category is referred to by this approach as an 'application technique'). In fact, most expert systems do not fit neatly into any one particular application technique. Rather, they often straddle several techniques. By failing to recognise this fact, expert system developers often run into difficulties when the system has been expanded beyond the prototyping stage.

In the Knowledge Flow Model approach an expert system is conceptualised as embracing several application techniques rather than a single technique. The system is then modularised according to these perceived categories or application techniques.

CONCEPT

Expert systems are designed to solve different types of application problems. The type of application problem solved is termed the 'application type'. Some examples of application types include claim processing, process control, component repair, crisis advice, etc.

Each application type may be decomposed or mapped into several application techniques. An application technique represents a unique aspect of the application. In other words, an application technique is a module designed to solve a particular category of problem such as diagnosis, planning, monitoring, design, corrective action, scheduling, prediction, etc.

As an example, an application type, like 'process control' may be mapped into two application techniques namely 'monitoring', and 'diagnosis'. On the other hand an application type like 'claims analysis' may mapped into 'monitoring', 'diagnosis' and 'corrective action'.

The idea is then to identify which application type an expert system belongs to. Once this is known the application techniques can be identified. Each of these application techniques is then designed as an independent module.

These modules are then combined into a larger structure called a knowledge flow model (KFM). The KFM is a structured specification which defines how these modules are linked together. In other words, the KFM describes the information flow among the modules.

The KFM may be thought of as a shell into which individual application domains may be mapped. Different application domains may share the same KFM. For example both the manufacturing production line, and insurance claim-processing application types may be decomposed to the same application techniques, namely monitoring, diagnosis, and corrective action. Hence both these different domain applications can be mapped into the same KFM.

Once the KFM has been described for a particular application domain, it can be implemented by using an expert system shell.

IMPLEMENTATION DETAILS

The KFM approach involves two major steps. These are knowledge structuring (which is concerned with the design of the application) and implementation (which is concerned with the mapping of the design into the expert system shell used).

Step 1 Knowledge structuring

i) The expert system is assessed to determine its application techniques (whether diagnosis, scheduling, monitoring , etc). The application may be made up of several of these techniques. As described earlier this may be done by determining the application type the expert system fits into. If it is found to fit a standard application type, then its application techniques are known, otherwise its application techniques have to be worked out.

ii) These application techniques are then designed as independent modules.

iii) These modules are then combined into a KFM. This is done by defining a structured specification describing the links (ie the flow of information) among the modules. The KFM thus described is a collection of stand alone modules that can be run independently or together.

iv) Since the KFM is only a general structure, the structure of a particular application domain has still to be defined.

This is done by defining the domain objects and their attributes which are required to implement each individual module found in the KFM.

Once the application domain structure is defined the KFM is ready for implementation.

Step 2 Implementation

i) This step maps the KFM into the chosen expert system's shell. For example if the production rule formalism is used, a translation of the KFM into rules and rule-control structures to control the firing of rules will have to be carried out.

ii) Each application technique module is then implemented and tested.

iii) Finally the tested application technique modules are integrated into a cohesive system. Integration is achieved by the use of instance slots.

Instance slots are locations in the knowledge base which store information that are common across modules. They can be accessed or updated by the various modules. Thus these slots may be thought of as the linkages between the various modules.

COMMENTS

- An appeal of this method is that it supports the reusability of knowledge. Once a KFM for an application has been developed, it is possible to reuse it for a different application domain. When a KFM already exists for a particular application type, then the structuring of another application type which shares the same KFM will be much simplified. It will generally be only a matter of substituting one domain object for another.

In Payne's article he described how an application type like a manufacturing production line could share the same KFM with a different application type, like an insurance claims-processing system. In such a case (where two different application types share the same KFM), once a KFM for the manufacturing production line application exists, defining the structure for the insurance claims-processing application is only a matter of substituting the components in the factory, such as sprayers and pumps with policy types in the insurance domain.

- Modifying an expert system in the way suggested by this approach appears to result in very large modules. Most of the examples given by Payne break the system down into just two or three application technique modules. For example a process control application may be decomposed into two application techniques modules, namely monitoring and diagnosis.

  To enhance maintenance it should be expected that the application type modules be further decomposed into smaller functional units.

3.1.2.3 MULTIPLE KNOWLEDGE BASES CONCEPT

INTRODUCTION

Many knowledge base systems have been implemented using the 'multiple knowledge bases' concept. Among them are PROSPECTOR (Jacob, et al., 1990, p. 173), LOAN PROBE (Ribar et al., 1991), COMPASS (Prerau, 1990), (Prerau, Gunderson, Reinke, & Alder, 1990).

In this section , the modular approach behind the building of COMPASS (Central Office Maintenance Printout and Suggestion System) will be examined.

This approach modularises an expert system knowledge base by following the modularity of the expert's knowledge, and implements the concept by using multiple knowledge bases.

COMPASS is a system that helps maintain electronic telephone exchanges. It was originally developed at GTE Laboratories as a prototype model with little attention paid to maintenance problems. Later it was re-developed from its original prototype and put into field use. The modular concept was adopted then to ensure that it was more easily comprehensible and maintainable by organisations receiving the technology.

CONCEPT BEHIND THIS APPROACH

In order to aid maintainability, the creators of COMPASS proposed to develop the system by following the modularity of the expert's knowledge. This is in contrast to defining modularity in a way that is convenient for software development. That is, the COMPASS approach structures the knowledge base so that it reflects the structure of the expert's knowledge.

The developers claimed that the use of such a functional breakdown makes it easier to split the implementation evenly among developers since such a split can be naturally done along functional boundaries.

In the rapid prototyping environment in which COMPASS was developed, the functional breakdown allowed each developer to work in relative isolation. In order to implement the system along functional boundaries the developers were prompted to ask how a human expert would perform the expert task manually. The human expert would :

i) receive a group of messages (ie input),
ii) identify, analyse and make suggestions (ie process them non-interactively), and
iii) produce a list of recommended actions (ie output).

According to the above functional breakdown, COMPASS was decomposed into five disjoint phases (input, identification, analysis, suggestion, output) plus many sub-phases. Each of the five main phases could be assigned to a developer.

IMPLEMENTATION DETAILS

It was found necessary to divide COMPASS into eighteen separate knowledge bases, seven of which analyse messages and problems; the remaining eleven are not really knowledge bases in the sense that they do not contain expert knowledge. Rather, they contain system management tools, such as utilities for the maintenance of the multiple knowledge bases, configuration management, control of inter-knowledge base data access, etc.

The seven knowledge bases are what Prerau et al. called the 'active' knowledge bases. The knowledge they contain corresponds to steps in the expert's analysis procedure.

Each knowledge base has its own name space and is treated as a single entity. That is they can be saved, loaded or displayed separately. As such, a single developer can be

assigned to each knowledge base.

The following are some of the implementation problems faced by multiple knowledge bases, and the ways COMPASS handles them :

i) Different developers tend to introduce their own individual styles into the knowledge bases giving rise to potential maintainability problems.

The solution taken by the COMPASS team is to adopt standard programming conventions to maintain uniformity of styles.

ii) To ensure that no undue multi-representational paradigms are used across the different knowledge bases, developers were required to select only rules or frames formalisms whenever possible.

iii) Multiple knowledge base systems often face the problem of not having clear access paths (ie how one knowledge base can access another knowledge base's data and what restrictions if any are required).

COMPASS adopts a set of conventions which placed restrictions on data access between knowledge bases. Also, an 'access' knowledge base is used for the purpose of providing import and export facilities for inter-knowledge base data access.

iv) In multiple knowledge base systems the control flow paths are more complex than those between routines within a single knowledge base system.

COMPASS uses a 'control' knowledge base to provide a centralised branching point for the system's control flow. This is a top level knowledge base that defines the control flow. It places constraints on inter-knowledge base control flow but does not restrict what individual developers can do within a knowledge base.

COMMENT

- The use of multiple knowledge bases undoubtedly increases the complexity of the system in terms of inter-knowledge base control and communication.

  This is evidenced by the fact that out of COMPASS's eighteen knowledge bases, only seven of them are 'active' knowledge bases (ie. contain actual expert knowledge for analysing messages and problems). The other eleven perform control and management tasks, as Prerau puts it, they are "knowledge bases only in structure - they do not contain actual expert knowledge" (Prerau et al., 1990, p. 73).

  In the LOAN PROBE multiple-knowledge base expert system, the complex communication and sharing of knowledge among the knowledge bases were handled by a blackboarding system (Ribar, Arcoleo & Hollo, 1991, p. 43).

3.1.3 DATA DICTIONARY CONCEPT

INTRODUCTION

In order to maintain a knowledge base, it is to be expected that the maintainer must be familiar with all the areas where the knowledge resides, as well as the inter-relationship among these knowledge. Such information is provided by a data dictionary.

Other helpful features provided by a data dictionary include :

- easy browsing of rules, facts, etc
- automatic documentation and cross referencing of knowledge and other concepts in the knowledge base
- ability to present knowledge in different ways

The data dictionary concept had been applied differently by different researchers on knowledge bases. Jansen and Compton's model (Jansen & Compton, 1988), (Jansen & Compton, 1989), (Jansen, 1988), uses one integrated dictionary to encompass both the knowledge base and data base. This is in contrast to Leung and Nijssen's work which uses the dictionary concept to couple expert and database systems, and stand alone dictionary systems like NEXPERT OBJECT which have interfaces to relational databases (quoted in Jansen & Compton, 1988, p. 1159).

This section discusses Jansen et al.'s version of the dictionary. This dictionary was developed at the CSIRO by Jansen and his team. Essentially what they did was to augment the use of a data dictionary to include representation for rules, and termed the tool 'knowledge dictionary'. In other words, the tool was effectively just an adaptation of conventional data dictionary technology to the area of knowledge base system.

This knowledge dictionary has already been implemented by the CSIRO team in Prolog, Hypercard, RDB (DEC's relational database package), and RALLY (a fourth generation software tool).

CONCEPT BEHIND THE TOOL

The key to this tool is that if the Relational Data Model is used as the underlying storage representation for knowledge, then the way opened for the use of the full power of relational calculus for manipulating it. In short, this means relational operators like union, difference, select, join, divide, project, etc can be used on the stored knowledge. The use of the relational operators on the data representation of rules allows for sophisticated exploring and browsing capabilities, which in turn facilitates the maintenance of the knowledge. In addition, inferencing can also be done using SQL-type data manipulation instead of resolution.

Maintenance of the knowledge will also be eased by the use of normalisation. Like data, knowledge can then also be represented in normalised form. That is, each concept is fully defined and named once only, and is found in only one place.

Often a single concept may be known by different names to different experts. Normalisation does not force the experts to settle on a single name, rather each different name given to that same concept is stored once and different expert's labels are mapped into this stored object.

IMPLEMENTATION DETAILS

Jansen et. al. claimed that the tool had been used successfully to implement production rules and semantic nets representations.

The following example illustrates the implementation of production rules. Consider the following rule to be implemented (ie. converted into relational entities) :

```
Rule 96     IF  A
                and B
                 and NOT C
            THEN
                ACTION_A
```

Step 1 : The rule is decomposed into its constituent objects, namely a rule name, a set of facts, and a set of rule actions.

In the example the rule will be decomposed thus :

rule object name	:	96
fact objects	:	A
		B
		C
action object	:	Action_A

Step 2 : The constituent components are then stored in a table as a set of relationships between the rules and each facts, and between the rule and each rule actions (see Table 3.a).

rule object name	relationship name	fact/action objects
96	presence	A
96	presence	B
96	absence	C
96	action	Action_A

Table 3.a Relationship between rule, fact and action

Step 3 : The actual implementation of the relationship can be :

i) a pointer

ii) set based (as in CODASYL database)

iii) value based (as in relational data model)

vi) function based (where the membership of a relation is dependent on the evaluation of some function which returns a true or false condition)

This example shows the function based implementation where the table of Step 2 is stored in the knowledge dictionary as Prolog declarations.

```
element(rule,'96').
element(fact,A).
element(fact,B).
element(fact,C).
element(action,Action_A).
element-relationship(presence,rule,'96',A).
element-relationship(presence,rule,'96',B).
element-relationship(absence,rule,'96',C).
element-relationship(action,rule,Action_A).
```

Step 5 : Having stored them in the desired relational data form, a number of SQL typed functions such as the following may be developed for their manipulation.

USAGE	: to determine who uses what
SHOW_RULE	: displays specified rules.
ADD_RULE	: add new rules by specifying existing facts and actions.
RUN	: carries out a forward chaining inferencing procedure.
WHY_NOT	: may be used to query why a rule did not fire.

The above functions are provided with a translation facility which converts the stored data and rebuilds them into rules in the familiar IF..THEN form for display to the expert.

COMMENTS

- As the current system only does forward chaining inferencing, its application appears rather restricted.

- Efficiency may be an issue since an interface is required for the inter-conversion between relational entities and production rule (or other formalism) format each time the user queries or accesses the knowledge base.

- An advantage of storing the rules in such neutral relational data form is that in this form it can easily be transformed into other formalisms.

Jansen et. al pointed out that a major problem in knowledge base work is the lack of integration of knowledge representation formalisms. In an extension to their work, Jansen et al. showed how the knowledge dictionary could be used as an aid to integrate some of the standard knowledge representation formalisms.

3.1.4 NORMALISATION PRINCIPLE

The benefits of normalisation have been well documented in the literature on conventional systems. Normalising a database removes from it deletion, amendment and insertion anomalies.

According to Debenham and Lindley (1991, p. 344), normalisation of rules ensures that a single item of knowledge, whether in part or whole, is represented only once in the knowledge base.

Normalisation thus results in all the rules being independent of one another. Hence if a component of the rule is modified, only one modification is required as there will not be any overlapping knowledge.

For example, consider the following unnormalised set of rules :-

Rule 1 P :- Q, R.
Rule 2 S :- R.

The above rules are unnormalised because the component of knowledge 'R' appears in more than one place, namely in Rule 1 and in Rule 2. If R is to be modified, both Rule 1 and Rule 2 will have to be changed.

On the other hand, the following example illustrates the same set of rules, but this time they have been normalised :-

Rule 3 P :- Q, S.
Rule 4 S :- R.

It is clear that if R is to be modified, only one rule (ie. Rule 4) needs to be changed.

The following section looks at a tool which relies on the normalisation principle. The tool is called Knowledge Analyst's Assistant (Debenham et al., 1991). It was developed at the CSIRO Division of Technology.

3.1.4.1 NAME OF TOOL : KNOWLEDGE ANALYST'S ASSISTANT (KAA)

INTRODUCTION

Debenham et al. identified a major contributing factor to maintenance problems as the complex relationships that exist between components of a knowledge base. They called this the 'coupling relations' between knowledge components.

There are two such kinds of relationships :-

i) the same fact has been represented, at least in part, in more than one place. (ie unnormalised)

ii) the inherent structural relationship of the representational scheme itself.

KAA is designed to handle the second form of relationship. In other words, before KAA can be applied, the first problem (ie unnormalised knowledge) must be removed.

Since normalisation removes the first kind of coupling relationship, a normalised knowledge base is the pre-requisite to the application of KAA.

OBJECTIVE OF KAA

To support each maintenance operation by automatically identifying a linked chain of modifications. In this way the maintainer can be sure that each modification has been completely effected.

CONCEPT BEHIND THE TOOL

This tool is based on the concept of a clear distinction between what is Data, Information, and Knowledge as defined by Debenham (1989).

KAA centres around four models. The data model, information model and the knowledge model together constitute what is called the three system models. The main model is the application model.

The application model is a representation of the application in question constructed in quasi natural language form. Each entity in the application model must correspond to just one element in one of the three system models.

The data model, information model and knowledge model may be seen as roughly the equivalent to the domain constants and variables and their constraints, the relationships between them and their constraints, and the rules respectively.

The four models must be normalised and their relationships (ie links between them) must be established. Being normalised these links are unique. When a maintenance operation is to be done on the application model, KAA is able to trace through these links to the system models thus ensuring that every maintenance operation can be executed completely by following a single linked chain of statements.

Four types of links are used in KAA. Two of them (TYPE 1 and TYPE 2 links) will be discussed here.

TYPE 1 links - these link every statement in the application model to one unique entry in one of the system models.

TYPE 2 links - these link the three systems models among themselves if they are related (ie if the knowledge component of one of the system models is part of the definition in another).

IMPLEMENTATION DETAILS

Pre-requisite : The four models as described above are in a normalised form.

Step 1 : When a maintenance operation is required, the analyst starts at a statement in the application model.

For example, in figure 3.a, suppose the analyst wishes to alter statement A in the application model.

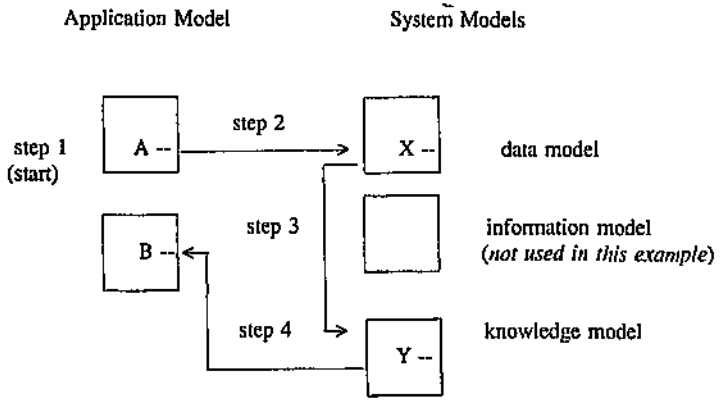


Figure 3.a Modifying an Application Model

Step 2 : Since one statement in the application model is linked to just one unique entry in one of the 3 system models (TYPE 1 link), this link is used by KAA to trace it to that system model.

In the example, KAA now automatically follows the TYPE 1 link to one of the system models (in this case entry X in the data model).

Step 3 : TYPE 2 links are used by KAA to identify all other components of the three system models which use that knowledge component in their definitions.

In the example, a TYPE 2 link is now used by KAA to trace through all system models that contain entry X in their definitions (in this case entry Y in the knowledge model).

Step 4 : Since all entries in the system model correspond to one and only one entry in the application model, TYPE 1 links are now used to trace back to all the statements in the application model which correspond to each of these unique system model entries.

In the example a TYPE 1 link traces entry Y back to statement B in the application model.

Step 5 : Each of the statements identified in the application model is then altered as required. The modification of the system models are then handed to the programmers to be implemented on the knowledge base.

In the example, the analyst will now be presented with two statements, A and B, which may be altered.

COMMENTS :

- This tool presupposes formal specification is possible (the three system models act as formal specifications while the application model acts as documentation). This, as we have seen in the earlier argument presented in 3.1.1, may be a flawed assumption (not just in the case of expert systems but in many conventional systems as well).

- The tool appears suited only for the class of very well structured systems where knowledge can be easily identified since Debenham requires at the outset a clear-cut definition of data, information and knowledge. Many experts have to work in areas where such clear-cut definitions may not be possible.

- Currently KAA is intended for rule based systems only. To use it for other formalisms, a suitable translation mechanism between the existing analytical language of KAA and these formalisms will be required.

- Opportunities exist for the application model and the three system models to become inconsistent since they are not maintained automatically in a single operation.

- Although normalisation offers many benefits to a knowledge base, as shown by Debenham's example, its value appears doubtful since unlike facts, the normalisation of rules may remove their heuristic values.

3.1.5 STRUCTURED TECHNIQUES

INTRODUCTION

Conventional structured programming techniques have long been gainfully applied to programs making them easier to understand, maintain and test. Penderson (1989) adapted such techniques to the creation of more well-structured rule bases. These techniques proposed by Penderson were designed for backward chaining rule bases only.

Penderson constantly referred to two terms, "visibility" and "transparency", in his paper. Visibility is a term which refers to the ease with which one can see the order in which statements are obeyed. Transparency refers to how easily one can grasp the meaning of a statement.

For example, conventional programming languages encourage a programming style which exhibits high visibility (i.e. it should be easy to see the order in which statements are obeyed). However conventional languages exhibit low transparency since each statement contains little information in itself. Much of its meaning depends on its position within the larger set of instructions which make up the routine.

On the other hand, the declarative style of rules in general encourages high transparency. That is the meaning of a rule is easy to grasp since it is contained within the rule itself, rather than on its location in the knowledge base. However rules in the knowledge base have low visibility. That is the order in which they are executed is hard to grasp. The order in which the rules are fired is implicit in the inference engine used.

Penderson's techniques seek to enhance the high transparency of rule bases and at the same time reduce their visibility problems.

OBJECTIVE

To apply structured methods, which are analogous to conventional structured programming techniques, to the creation of backward chaining rule bases in order to more clearly represent domain knowledge and thus ease maintenance.

CONCEPT BEHIND THE PROPOSED TECHNIQUES

The adoption of structured techniques in conventional programming brings with it the following benefits :-

i) high transparency due to self-documenting codes and modularity (since the meaning of a module is contained within the module itself)

ii) ease of maintenance since every module has a single entry and a single exit point.

iii) ease of understanding and easy detection of errors due to high visibility achieved through the elimination of 'goto' statements.

iv) portability of codes

Penderson claimed that the above benefits can be obtained if his three proposed guidelines are observed during the creation of a knowledge base. The three guidelines laid down by Penderson are :

1) keep conclusions (of rules) simple
2) keep procedural contents out of rules
3) minimise the use of ELSE statement

IMPLEMENTATION DETAILS USING THE THREE GUIDELINES

1) Keep conclusions simple

Rules must be written such that their conclusion change only a single attribute. Consider the example of the following set of rules (CF = certainty factor) :-

Rule 1	IF	season = summer	
	THEN	weather = fine	
Rule 2	IF	season = summer	
	THEN	weather = fine	CF 90
		weather = rainy	CF 15
Rule 3	IF	season = summer	
	THEN	weather = fine	CF 90
		Jane = wear sunglass	CF 75

Rule 1 concludes a single attribute and is clearly unambiguous. Rule 2, despite having two conclusions, concludes only a single attribute, namely weather, and is therefore also straightforward.

Rule 3 on the other hand concludes two attributes. Studying rule 3, one could ask if Jane's putting on sunglasses is a consequence of the weather being fine, or whether these two conclusions are independent.

Apart from the ambiguity it causes, the question of how a particular shell interprets it is also uncertain. For example should a shell with a backward chaining inference engine process this rule if EITHER of the two conclusions is the current goal, or should it process this rule only when BOTH the two conclusions are the current goal ? These possibilities reduce the portability of the rule base.

Apart from the above problems, rules that conclude more than one attribute also have a high chance of causing looping. Consider the following (backward chaining) rules :

Rule 4	IF	A > 0
	THEN	GOAL
Rule 5	IF	B
	THEN	X = C+2
		A = ...
Rule 6	IF	D
	THEN	C = A+1

Assume that :

(i) we have a backward chaining system

(ii) conditions B and D are true

(iii) GOAL (of Rule 4) is the current goal

In order to prove Rule 4, the inference engine will seek the value of A.

Rule 5 is therefore considered since it concludes A, but it concludes X as well, and in order to conclude X, it needs attribute C. Hence Rule 6 is considered since it concludes C, but to conclude C , attribute A is needed.

At this point looping may start to occur.

From the examples it is seen that rules that conclude more than one attribute make debugging of the rule base difficult since it is hard to identify which rule was responsible for a given consultation state.

Since such rules also give rise to ambiguities in the semantics of the rules (as illustrated in the example relating to Rules 1, 2 and 3), they are less transparent, hence extensive documentation is required. This is a misuse of the inherent high transparency advantage offered by rule bases.

2) Eliminate procedural content from rules

Procedural contents in rules give rise to poor transparency since it is not obvious what a procedure does. One could argue then that rules should contain knowledge rather than procedures.

Consider the rule :

```
IF sunny THEN
        call goodday-proc
```

One cannot be certain what goodday-proc does. In other words, the rule has lost its transparency; this is another way of saying goodday-proc does not state its entire meaning.

Such calling of outside programs or subroutines can produce hidden side effects and should be avoided if structured techniques are to be followed. Penderson suggested that "you may be addressing a problem that could be better solved using a conventional language" if you find that you cannot avoid including such procedural content (Penderson, 1990, p. 49), and called for a rethink on the part of the knowledge engineer regarding whether "an improper conception of the expert's knowledge or improper representation of the expert's problem solving strategy" was the cause (Penderson, 1990, p. 46).

3) Minimise the use of ELSE

The use of ELSE has many disadvantages.

The following example shows two rules (Rule 7 and Rule 8) present in a knowledge base :

```
Rule 7  IF     A  and  B
        THEN      X = 1
        ELSE      X = 2

Rule 8  IF     A and NOT B
        THEN      X = 3
```

Assume that : condition A is True, and
condition B is False

If Rule 8 has been placed before Rule 7, then X = 3 would have been concluded. However, in the example, Rule 8 has been placed after Rule 7, hence it will not be tested because ELSE would have come into effect. Hence ELSE has implicitly introduced a procedural content into the knowledge base, making it not possible to add rules anywhere into the knowledge base. This results in the reduction of transparency offered by declarative rule bases.

Also ELSE makes the meaning of the rule unclear. Consider Rule 7 for example, we know from it that X = 1 should be concluded if both A and B are true. But we are unsure when X = 2 was concluded. It could have been A was true and B was false, or A was false and B was true or both were false.

Without the use of ELSE, the inference engine can be allowed to conclude

UNKNOWN when no rules applied. This helps to identify gaps in the knowledge base and allows the possibility of introducing rules that reason about the UNKNOWN.

COMMENT

- In this proposed structured technique, we find that rules have been 'doctored' to suit the 'structuredness' of the methodology. They have lost their 'naturalness' in that they no longer reflect the way experts think, a criterion considered by many important for a knowledge base to possess (section 3.1.2.3 on 'modularity', and section 3.2.2 on 'knowledge in context strategy').

  Still one might ask if it is necessary for rules to reflect the way experts think? It is well known that the way experts report their reasoning is often different from how they actually reason in the first place. Among those who hold this opinion are Compton and Jansen (1988, p. 293) who said that experts have difficulties reporting on how they reach decisions.

- The greatest attraction of Penderson's proposals lies in the fact that the ideas he put forward are simple and can be readily adopted to any backward chaining knowledge bases without the introduction of any major tool or strategy.

3.1.6 OTHER SOFTWARE ENGINEERING TECHNIQUES

3.1.6.1 REUSABILTY

Expert system shells are examples of the reusablity concept being applied to expert systems at the implementation level. However there has not been not much progress in applying the reusabilty concept at the knowledge level (Buchanan and Smith, 1989, p. 187). Buchanan and Smith cited the example that many expert systems use facts about anatomy and physiology, yet often each encodes these facts specifically for use in a unique way.

While Buchanan and Smith did not venture to suggest any reasons for this observation, Matthews did (Matthews, 1990). He said that theoretically rules should be reusable, but in practise this is not the case because rule bases tend to be too application specific. He attributed this application specificness of rule bases to the lack of programmability of the control systems of the shells or languages in which the rules were written. This inflexibility of control structures of the shells caused programmers to include their own control mechanism into rule bases meant to express domain knowledge, thus rendering them unreusable.

Matthews suggested that host languages or shells should have control systems that allow developers to "take full charge of the control systems", a freedom, he conceded, "still relatively rare in the context of today's AI shells" (Matthews, 1990, p. 437). It should be noted that to some degree this freedom has been met by RIME (discussed in section 3.2.3) which permits the explicit expression of high level control structures.

Another obstacle to knowledge reuse is the dependency of knowledge bases on specific representation paradigms. For instance a rule-based application cannot reuse the knowledge found in a knowledge base which employs logic as its representational paradigm.

To some extent Jansen et al.'s work on knowledge dictionaries can be seen as a step in the direction of promoting reusability (see comments of 3.1.4). This is because their knowledge dictionary is stored in the relational data format (ie. a representation paradigm free format). Hence it is possible for the knowledge in the dictionary to be captured from a rule base, and converted for reuse by a knowledge base of a different paradigm.

Benn, Schiageter and Wu (1990) described how a component can be added to a KBMS to allow an inter-paradigm reuse of application objects. This new component is termed "the Conceptual Object Manager". Its purpose is to manage execution model semantics in a way that allows applications to reuse persistent information independently of their individual paradigm commitment.

3.1.6.2 DOCUMENTATION

The usefulness of external documentation may be quite restricted whether it is for conventional or expert systems since it can easily become out of date. Besides, external documentation involves an extra effort. In the opinion of the current author, documentation should be made an integral part of the code so that this duplication of effort and currency problem can be removed.

In-code documentations should be used to aid understanding of the knowledge base. The 'interface specification' strategy promoted by Jacobs et al. (discussed in section 3.1.3.2) suggested that each module should have a header describing its function. The description must be a high level one in order to remain true even though the internal details of the module have been changed.

More documentation should not be viewed as being better. Constructing knowledge bases using modular and structured techniques or use of more declarative languages like SYLLOG (section 3.2.4) should make documentation less necessary.

3.1.6.3 STANDARDISATION

In large systems there are many different programmers working on different parts of the same system. This gives rise to many different naming conventions, programming styles, etc, which in turn may lead to the development of a system which is difficult to maintain.

A case in point was the COMPASS System (discussed in section 3.1.2.3) which partitioned its knowledge into eighteen separate knowledge bases. Four programmers were engaged in the development of the individual knowledge bases, leading to differences in styles which in turn hampered maintenance (Prerau, 1990).

Prerau handled the problem of non-uniformity in programming style by adopting a set of standard naming conventions for system elements, and he imposed the use of standard representation paradigms like rules and frames wherever possible.

It should be noted that the enforcement of standards is as important as the setting of the standards themselves. Stonehocker reminded us that "standards are ineffective unless they are published, understood by all concerned, and enforced" (Stonehocker, 1988, p. 292).

3.2 OTHER APPROACHES

The current market contains many innovative tools and ideas on how to build more maintainable knowledge bases. This section selects four approaches for discussion.

The 'knowledge specification' approach is selected for the novel way in which it accommodates the concept of rigorous definition into the construction of knowledge bases.

The 'knowledge-in-context' paradigm is a radical proposal which runs counter to entrenched software engineering principles. The ideas it propounds starkly contravene the long held software engineering principles of structuredness and modularity.

The third approach looked at is founded on the belief that the root cause of maintenance problems lies in the implicity of rules and control structures. Essentially what this approach says is "make the implicit explicit, and the knowledge base will be maintainable".

The last approach contends that declarative knowledge is easier to understand and maintain than procedural knowledge.

3.2.1 KNOWLEDGE SPECIFICATION PARADIGM

INTRODUCTION

It was argued in section 3.1.1 that expert systems defied rigorous specification because of a number of reasons, among which were the incremental nature of their knowledge, their inherent unstructuredness, their possibly dynamic domains, and the difficulty of extracting knowledge from experts (because much of this knowledge is implicit).

Prototyping was seen as a more appropriate tool for the construction of the expert system knowledge base since the prototype can serve as a testing tool allowing the developer to cycle through many iterations until a satisfactory prototype of the knowledge base is obtained. However this experimental nature of the prototyping methodology also makes knowledge bases developed from prototypes hard to modify. This is because prototypes often contain many ad hoc changes, and are relatively unstructured, poorly planned, and badly documented.

This section looks at an approach described by Slagle, Gardiner, and Han (1990). This approach may be thought of as a compromise between rigorous definition and prototyping. The concept expounded by this approach attempts to get the best of both worlds by using prototyping on the one hand and producing a rigorous specification on the other.

OBJECTIVE

To produce a knowledge specification and use it as a basis for developing the expert system, and for guiding changes during the maintenance of a knowledge base.

CONCEPT

The rationale behind this approach may be summed up thus. It says that a rigorously defined knowledge specification is vital, and must be procured before design and construction of the system begins. However, current rigorous definition methodologies cannot define the specification thoroughly. Hence, instead of using prototyping to construct the system, prototyping should be used to obtain the rigorous definition of the specification, and then that specification used as a basis to construct the system.

In other words this approach employs prototyping as a vehicle to enable developers to understand the problem so that a knowledge specification can be produced.

The resulting knowledge specification has two main uses, first as a basis to construct the new system, then as a guide for making system changes during the maintenance phase.

IMPLEMENTATION DETAILS

The approach consists of five phases. These are :-

i) Requirements analysis

System objectives, scope, constraints, etc are identified at this stage. Test cases and expected results for system acceptance are also collected.

ii) Knowledge acquisition

During knowledge acquisition, an initial knowledge specification is produced. This specification contains the kinds of knowledge and reasoning processes required

to perform a task. Since it is not possible to pre-specify every requirement accurately, the specification is incomplete at this stage.

Slagle et al. suggested the use of the 'Protocol Analysis' technique for the acquisition of knowledge. (Protocols are the verbal responses that have been generated by a domain expert during a session).

iii) Knowledge specification

Protocols are analysed to identify expert problem solving strategies. The results are represented using some representation formalism. Slagle et al. used the 'Conceptual Structure' techniques of Suwa (quoted in Slagle, et al., 1990, p. 32).

A programming language is then used to convert this initial knowledge specification into a prototype.

iv) Verification

Syntactic and some semantic checks are done on the conceptual structures.

v) Validation

The knowledge specification (ie the prototype) is then validated against the requirements. This is done by running the prototype against test cases and comparing the results against the expert's analysis.

The knowledge specification is modified and expended until correct answers are obtained for all known validation cases. At this stage a stable specification would have been produced.

The prototype is now discarded, and the knowledge specification is used as a basis for implementing the production system.

COMMENTS

- Slagle et al.'s proposed method appears to go only as far as building the knowledge specification. The actual construction of the system seems to be left open so long as it adheres to the guidelines laid down in the specification.

- The specification as referred to by Slagle et al. is really more than a specification of requirements; it contains the design blueprints as well. In Slagle et al.'s case, these are represented as 'conceptual structures'.

- The strategy proposed by Slagle et al. is really just an adaptation of the familiar 'throwaway prototype' concept (Guimaraes, 1987), whereby a prototype is developed and then discarded.

- Slagle et al. justified their method by claiming that "the re-implementing of the system from a specification is likely to take less time and money than improving the (original) prototype, and will result in a system that is easier to maintain" (Slagle et al., 1990, p. 30).

  While the latter claim certainly appears plausible, the claim that re-implementing from scratch takes less time may be debatable.

3.2.2 KNOWLEDGE IN CONTEXT STRATEGY

INTRODUCTION

The developers of the knowledge-in-context strategy (Compton & Jansen, 1990) believe that experts normally explain their reasoning differently depending on who they are explaining it to, and the context in which the questions are asked. Their strategy is fundamentally based on the assumption that experts cannot report on their mental processes accurately and unequivocally, and are willing to change context at will so as to remain correct. Hence there is no occasion where experts are ever wrong, but the context in which they are right changes.

On the ground of that assumption, it is thus conceivable for the researchers of this strategy to propose that the ability to change context should be an essential component of the expert system technology.

A problem with conventional knowledge bases is that they do not reflect the thought processes of the experts. Whenever new rules are added to such knowledge bases, it is not unexpected that the new rules may conflict with existing rules. Tools are therefore required to manipulate these knowledge bases until the inconsistencies disappear. This has the effect of the rules in the knowledge bases taking on an artificial structure, thus losing the original thought processes of the experts.

In this strategy, the justification provided by experts are considered to be correct in context. Therefore, if these justifications are captured as rules in the knowledge base in that same context, then they can be used as provided without further manipulation. In other words knowledge bases should be created without engineering (ie manipulating) the rules, rather they should be captured in the context in which they are provided by the experts.

OBJECTIVE

To facilitate maintenance knowledge engineering through the development of the 'context' strategy.

CONCEPT

An assumption this strategy makes is that maintenance is normally initiated by the failure of a single case so that an expert will be called upon to provide rules to handle further cases of this type.

This happens when an inaccurate interpretation (or conclusion) is produced by the system. The expert will then be asked to provide new rules to correct the situation. The context in which these new rules are provided depends largely on the context of the wrong interpretation the expert was presented with. That is, the new rule is not a global rule, but a rule to switch interpretation from the incorrect to correct.

To capture this context, a LAST-FIRED(rule-number) condition is included for every rule in the rule base. The new rule that was added will not be allowed to fire in a case unless the old rule which produced the wrong interpretation has been fired before it.

Each rule has only one opportunity to fire. Rules have to be tested in strict order, from the oldest to the newest. Maintenance is therefore a chronological process. Each new rule, whether a correction to the old, or a rule for a previously uninterpreted case, is always added to the end of the list of rules.

IMPLEMENTATION DETAILS

i) Creating the rule base

At the time of creating the initial rule base, no test cases have yet been run, therefore the notion of whether the rules are correct or incorrect in context does not exist, the implicit context being that the expert system knew nothing of the test cases. Hence at this stage of creation, all rules should be allowed to fire. This is implemented by assigning the 'IF LAST-FIRED(0)' statement to the 'condition' part of every rule.

The rule base thus created can be viewed as a long list of rules which when run will be tested from the first rule to the last rule in sequence.

```
Rule 1   IF  LAST-FIRED(0)
             and  A= 1
             and  B = 2
         THEN  X
             .
             .
Rule 5   IF  LAST-FIRED(0)
             and  C =1
             and  D= 2
         THEN  Y
             .
             .

Rule 100  IF  LAST-FIRED(0)
              and  E = 1
          THEN  Z

Rule 101  IF  LAST-FIRED (5)
              and  E = 1
          THEN  W
```

Figure 3.b A list of rules in the knowledge base

For example in figure 3.b, rules 1 to rules 100 are the initially created rules, all of which are assigned the LAST-FIRED(0) status.

As the rule base is being tested with cases it needs to be modified. The modification is straightforward. It involves only the addition of rules to the end of this rule list, regardless of whether the new rule is a correction of an old rule or is a rule for a yet uninterpreted case.

For example in figure 3.b, Rule 5 is found to be incorrect. A new rule, Rule 101, is added to the end of the list with the condition LAST-FIRED(5) assigned to it. This means that this rule will take precedence over all the others the moment Rule 5 had been fired.

ii) Running the knowledge base

The control sequence in running the knowledge base is as follows.

Step 1 Control starts from the oldest rule to the newest down the list looking for rules which have no pre-fire conditions (ie rule with LAST-FIRED(0)).

Rule no.	LAST-FIRED(0)	Rule no.	LAST-FIRED(3)	Rule no.	LAST-FIRED(200)
1	-> xxxxx	22	-> xxxxx	7	-> xxxxx
2	-> xxxxx	200	-> xxxxx fired	99	-> xxxxx
3	-> xxxxx fired	201	xxxxx		
4	xxxxx				
5	xxxxx				
-	.				
-	.				
-	xxxxx				

Figure 3.c A conceptual representation of a set of rules in the knowledge base

Step 2 As soon as a rule has been fired, (for example rule 3 in figure 3.c) the only rules which can now be allowed to fire are those which have that rule as a pre-condition (ie rules with LAST-FIRED(3), namely rules 22, 200, 201).

Two situations may now arise :-

a) None of the rules with LAST-FIRED(3) could fire (not shown in figure 3.c).
In such an event, control would be passed back to the rule following the one which was last fired (in figure 3.c it would be rule 4 which is next checked).

b) One of the rules with LAST-FIRED(3) fired (shown as rule 200 in figure 3.c).

Since rule 200 fired, control is now transferred to check only those rule with LAST-FIRED(200) (in figure 3.c, it will be rules 7 and 99).

If none of the rules with LAST-FIRED(200) fires, control will return to check rule 201.

It should be noted that in figure 3.c, there is really only a single list, which runs from the oldest rule to the newest addition.

COMMENTS

- The control sequence of rule firing described in figure 3.c is really that of a depth first search.

- It is noted that this strategy blatantly violates the principles enshrined in the

software engineering methodology whereby rules are required to be engineered in various ways to achieved structuredness or modularity. In this strategy, rules are added only to the back of the rule list chronologically, thereby resulting in related rules being scattered over the knowledge base. However, there is method in this seemingly disorganised rule base.

- An advantage which stems from the chronological addition of rules is that a rule trace can quickly review the history of corrections and additions to the knowledge base.

- In appearance, this strategy seems to turn knowledge base maintenance into a relatively simple affair - one of merely adding rules to the back of a list ! Under the provisions of this strategy, one cannot find any occasion to delete or change any rules. Since the presumption of this strategy is that there must always be some truth in what experts say (it is only the context that is questionable), deletion and modification does not arise. To some, this assumption that the justification provided by an expert is highly accurate in context may be quite contentious.

- Although the developers of this strategy had tested it by redeveloping an existing knowledge base for the GARVAN-ES1 expert system, and then measured the knowledge engineering problems and the performance of the resulting new system against the existing system, this test appears to be rather contrived. It would be more convincing if it had been tested with a knowledge base created from scratch for a new application.

- Despite testing the strategy by rebuilding the GARVAN-ESI knowledge base, Compton et al. said that they "propose this strategy for the maintenance phase of an expert system project" (Compton, et al., 1990, p. 297). In other words, regardless of how a system had been built, theoretically the strategy can be applied to any existing knowledge base to some degree with only the relatively minor modification of including the LAST-FIRED(rule-number) conditions in front of every rule.

3.2.3 EXPLICIT HIGH-LEVEL CONTROL STRUCTURE

The proponents of this strategy believe that the implicitness of a knowledge base control structure is one of the major causes of maintenance ills, and the panacea is to make this control structure explicit.

RIME, the euphemism for "R1's Implicit Made Explicit" (Bachant, 1988, p. 205) succinctly sums up the crux of the tool which this section examines.

3.2.3.1 NAME OF TOOL : RIME

INTRODUCTION

The researchers at DEC developed RIME with the goal of facilitating the maintenance of XCON's (also called R1) knowledge base. This 'language' (Soloway, et al., 1987) was used to write XCON-IN-RIME, the successor to XCON. RIME has also been referred to as a 'methodology' (Bachant, 1988), (Hicks, 1990).

The main problems faced by XCON are its high volatility (ie. dynamism of its knowledge base) and its huge size. Hicks said that it contained 17,500 rules in September 1988 (Hicks, 1990, p. 293).

XCON was written in OPS5, hence had to rely on the implicit conflict resolution strategies of OPS5 for the control of its rule firing. As a result, programmers often had to resort to 'tricks' to explicitly change the control of the rule firing sequences. As different programmers worked on XCON at different times, the implications of these 'tricks' which were buried in the codes became increasingly difficult to comprehend.

RIME seeks to make the control of rule firing and the structuring of the rule base explicit.

OBJECTIVE OF RIME

To allow for the explicit expression of high level control structure, and to provide a framework which allows codes to be made homogeneous and predictable, hence easy to modify.

CONCEPT BEHIND RIME

RIME's builders believed that the two main factors which decide the maintainability or otherwise of a knowledge base are homogeneity and predictability.

Homogeneity is a term which is used to describe a knowledge base which uses similar solutions to achieve similar goals.

Predictability refers to the ease with which a knowledge engineer can identify or predict where codes should be changed. In order to make rules predictable, one rule should only be allowed to serve one function. Also rules that serve related functions should not be scattered over the knowledge base.

In order to enhance homogeneity and predictability, RIME focuses on two main issues. These are the control characteristics (which focus on the explicit specification of control) and the rule based characteristics (which focus on the organisation of the rule base).

(1) Control characteristics

In a typical shell or language, all types of problems share the same implicit control characteristics of the shell's or the language's control structure. This makes it impossible for programmers to change the order of the rule firing sequence without resorting to writing their own routines. This situation can be avoided by using a language which offers several different control structures. With such a language, different types of problems can be solved under different control structures.

The goal for RIME is thus to offer different types of control for solving different types of problems. To achieve this RIME introduces the concept of a problem solving method (PSM). A PSM is a programmer-defined domain-independent sequence of steps to solve a class of problems. RIME contains mechanisms and guidelines to help programmers pre-define their own PSMs. Each PSM thus explicitly spells out the sequence of rule firing. An example of a PSM is the 'propose-and-apply' method.

By using the same PSM to solve similar types of problems in a domain, homogeneity is enhanced since 'similar solutions are used to achieve similar goals'. To decide which PSM should be used, the rule base characteristics have to be considered.

(2) Rule-based characteristics

Rules with similar characteristics (ie with common properties) are grouped together and placed into a 'domain specific bucket'. These 'buckets' are also called 'problem spaces'. As each problem space uses only one PSM, the firing sequence of rules within a common problem space is known.

Another advantage of grouping similar classes of rules into a common problem space is that it ensures that as the knowledge base grows it will remain homogeneous.

To enhance predictability, RIME's language constructs are directed toward problems in the domain. This means that the composition of codes in the construct tend to reflect the composition of the problem.

To ensure that developers use these language constructs in the ways intended, an online tool called SEAR provides enforcement by using a template for each rule type to guide the creation of rules.

COMMENTS

- RIME's concept of grouping together similar types of rules under the same problem space, and assigning each problem space to only one PSM so that a single control structure is used to solve a similar kind of problem, is a laudable one, for in this way RIME has laid down strict prescribed methods for programmers to follow so that homogeneity is achieved.

  The setting of such standard prescribed methods, however, cannot be taken to mean that programmers will observe them. To ensure that programmers follow these strict language constructs in the desired way, SEAR is used to provide on-line enforcement of the coding guidelines.

  One might argue that such an overbearing approach may have the effect of curtailing the creativity and freedom of the programmer.

  RIME's approach, however, will work particularly well for large knowledge bases which are maintained by many different programmers. Rather than having programmers writing in their individual styles, such an approach will ensure that codes are standardised (or homogenised in RIME's terminology) and thus easy to maintain.

- While RIME recognises the existence of different control structures, it does not appear that it recognises the existence of different kinds of knowledge. For example, it makes no attempt to separate domain specific knowledge from problem solving knowledge, or knowledge about why one rule should be preferred over another, knowledge about how an expert system can be efficiently executed, etc. Since RIME does not have any mechanism to explicitly capture these different kinds of knowledge, they remain implicit in the knowledge base.

3.2.4 TOWARDS A MORE DECLARATIVE LANGUAGE

Among the many paradigms conceived by researchers in their quest for better ways to facilitate the maintenance of a knowledge base is the 'declarative language' concept. This concept rests on the belief that declarative knowledge is easier to maintain than procedural knowledge. Kowalski et al. said that such a concept seeks "to exploit in various ways a separation between declarative and procedural knowledge" (Walker, Kowalski, Lenat, Soloway, & Stonebraker, 1988, p. 64).

The concept is implemented by building procedural knowledge into the system, and then using this procedural knowledge to support the various declarative knowledge bases.

This section discusses an example of a shell based on this concept.

3.2.4.1 NAME OF SHELL LANGUAGE : SYLLOG

INTRODUCTION

SYLLOG (Walker, 1987) is a shell language written in Prolog. It was designed particularly for use by non-programmers. Its motivation was the desire to allow users to build their own knowledge bases and to maintain them without any knowledge of programming.

OBJECTIVE

To allow knowledge providers to code knowledge in a largely declarative form free from any control and procedural concerns.

CONCEPT

SYLLOG is based on the concept of Syllogism, the familiar

All X are Y
Z is X
---
therefore Z is Y

The idea is to be able to represent knowledge in very simple English-like sentences. To create a knowledge base in SYLLOG one is limited to syllogisms (simple declarative sentences) only. The actual language used may be an English like language, or any other language, whether natural or artificial.

SYLLOG knows little of the language concerned, unlike natural language understanding programs.

To build a SYLLOG knowledge base, knowledge in the form of facts and rule are acquired as a set of syllogisms and tables.

A syllogism (ie a sentence in SYLLOG) may contain one or more words starting with eg_, plus at least one other word not starting with eg_. Words starting with eg_ are variable names.

A table contains a group of related facts. The table resembles that of a relational database table, except that it is headed by a sentence in SYLLOG.

In the example (see table 3.b), the table stores the fact that
"an item has a number of parts ". That is, boxA has 4 cardX's,
boxB has 3 cardY's, etc.

eg_item has eg_number of eg_parts		
boxA	4	cardX
boxB	3	cardY
cardX	7	chipW
.	.	.
.	.	.

Table 3.b Item has number of parts

A rule that says "IF an item has X parts,
and each part has Y subparts
and X * Y is Z
THEN the item has Z subparts"

is represented in SYLLOG as :

eg_item has eg_X eg_parts
eg_parts has eg_Y eg_subparts
*eg_X * eg_Y = eg_Z*

---

eg_item has eg_Z eg_subparts

Once this knowledge is entered into the knowledge base, SYLLOG permits various kinds of queries, including 'what-if' type questions. SYLLOG also provides different forms of explanations (those that are obtained from instances of the rules that have been used to establish the answer, or those that are derived from deductions).

IMPLEMENTATION DETAILS

SYLLOG is implemented by seven sub-components which collectively make up the SYLLOG shell. These are :

i) A screen manager - for menu selection, and other function selections.

ii) A language file - for tailoring the system messages into English or other languages.

iii) A loader - for preparing and checking a knowledge base.

SYLLOG provides three ways of checking incoming knowledge.

a) Subject independent checking of individual rules - a syntax check.

b) Subject independent checking of the knowledge base - a limited form of consistency check, in particular a check that the rules do not contain a recursion through a negation.

c) Subject dependent checking of the knowledge base - this check ensures that the subject dependent or domain knowledge does not conflict with the real world. The experts supply constraints for the allowable situations in the knowledge base.

iv) An update component - to facilitate the process of maintenance or making changes to the knowledge base.

v) An inference engine - although SYLLOG is written in PROLOG, it has its own inference engine rather than making use of that of PROLOG. Its inference engine consists of both a backward chaining and a forward chaining component.

vi) An interface - to link to a database management system.

vii) An explanation generator - which provides automatic explanations for checking the knowledge on which answers are based. Rather than producing an execution trace which lists out all the steps the system has taken in reaching a conclusion, it recognises that there are a number of explanations as to why a conclusion follows from a knowledge base, and a very large number of explanations as to why it does not. It manages this by presenting a single explanation, then provides a way of asking for the next explanation should it be required.

COMMENTS

- As SYLLOG's inference engine is written in PROLOG, its execution speed is, at best, limited by PROLOG's speed. In fact, concerns over its operational efficiency have led its developers to admit that they have to look for "some ways to increase its speed" (Walker, 1987, p. 252).

- Since SYLLOG does not have natural language processing capabilities, the user does not have freedom of expression, but must always use the same sentence to say the same thing. To overcome this restriction, Walker suggested that the user should provide more "syllogisms to say that different sentences have the same meaning" (Walker, 1987, p. 236). This extraneous knowledge, however, may cloud the real knowledge by introducing added complications to the knowledge base.

CHAPTER 4

MAINTAINING AN EXISTING KNOWLEDGE BASE

Most authors classify maintenance into three main types. These are perfective maintenance (to make codes more easily understood, increase efficiency, etc); corrective maintenance (to correct errors); and adaptive maintenance (to adapt software to new operating environments) (Martin & McClure, 1983), (Parikh, 1988), (Gorla, 1991) and others.

Irrespective of the type of maintenance, the process of maintaining a knowledge base essentially involves three fundamental steps. These are :-

i) understanding of the knowledge base BEFORE modification may be carried out,
ii) performing the actual modification itself, and
iii) ensuring the correctness of the knowledge base AFTER modification.

This chapter is organised into three main sections to reflect the above three steps. Tools and aids targeted at minimising the difficulties found by the maintainer at each step are explored and discussed in these sections.

4.1 UNDERSTANDING THE KNOWLEDGE BASE

Before a knowledge base can be modified, the maintainer has to understand what the knowledge base does, why it is doing it, and how it does it. This understanding process is usually carried out by studying implementation level details.

A survey by Fjeldstad and Hamleu (quoted in Parikh and Zvegintzov, 1983, p. 2) showed that maintainers spent about half their time studying code. When correcting errors, over 60% of their time was spent reading code. They also revealed that this time was mainly spent trying to understand the intent and style of implementation of the original programmer(s). While Fjeldstad and Hamleu's study relates to conventional program understanding, it nevertheless indicates the magnitude of code understanding problems.

The ease of knowledge base understanding depends in part on how familiar the maintainers are with the shell or language in which the knowledge base was written; whether they were the original designers of that knowledge base; and their levels of skill and experience.

This section looks at aids which seek to enhance that level of understanding regardless of the maintainer's original background. It considers aids specifically designed to aid knowledge base understanding.

4.1.1 EXPLAINABLE EXPERT SYSTEM PARADIGM

INTRODUCTION

A typical expert system knowledge base is made up of facts and rules which are explicitly stated. But implicitly captured into these rules are many other different types of knowledge. These are the heuristics for achieving goals (ie the general problem solving principles), implementation and efficiency concerns, readability concerns and in some cases the style of the system builder. Many of these general problem solving principles and much of the rationale behind the rules and methods are 'lost' since they are not represented explicitly.

The failure to explicitly represent these different forms of knowledge which are required for the design of the system means that the expert system cannot provide explanation in terms of these sets of knowledge. Hence explanation is rigid and limited to merely a mechanical trace of the operation of the system. This makes understanding of the knowledge base difficult, and makes maintenance a daunting task.

The builders of EES (Neches, Swartout, Moore, 1988), (Swartout, Paris, Moore, 1991), (Lowry and Duran, 1989) believed that the difficulty of maintenance and understanding both stem from the same fundamental problem of not being able to capture these different types of knowledge explicitly.

By approaching this fundamental problem through the separation of knowledge into different types (ie modularising it) and explicitly capturing the system's development history, two main advantages are achieved :

i) maintenance is made easier since knowledge is modularised.

ii) better explanation can be provided since a record of the development process is

available. Better explanation in turn facilitates understanding of the expert system knowledge base.

OBJECTIVES OF EES

To facilitate understanding of the knowledge base by providing for more flexible and responsive explanation, and to make the task of maintenance easier through the separation and explicit capturing of different types of knowledge.

CONCEPT BEHIND EES

The EES paradigm is based on two fundamental principles :-

- The explicit representation of different forms of domain knowledge
- The formal recording of the system development process

(1) The explicit representation of different forms of domain knowledge

The EES knowledge base is separated into many modules. Each module explicitly represents a different type of knowledge.

These modules include :

i) The Domain Model - this contains the domain knowledge.

ii) The General Problem Solving (GPS) Principles - this component captures explicitly the set of general principles or heuristics from which the system was derived.

The GPS Principles are represented as 'Plans'. A Plan contains a 'Capability Description' and a 'Method'.

The Capability Description (also known as 'Goal') describes what the Plan is useful for. In other words it defines the Plan's Goal. For instance, a Plan's Goal may state 'for diagnosing faulty parts'.

The Method is a sequence of steps to achieve the Capability Description. In other words, it is an implementation of that Plan's Goal.

iii) Tradeoff knowledge - knowledge which indicates what benefits can be gained and what losses suffered as a result of selecting a particular GPS Principle to achieve a goal;

iv) Preferences - knowledge used for ranking GPS Principles based on the tradeoffs;

v) Terminologies - this module contains the definitions of all the terms used in the system. They are shared across GPS Principles;

vi) Integration knowledge - When EES generates the expert system code, certain sets of rules (or procedures) being generated might conflict with others. This module contains knowledge for resolving such conflicts among the various knowledge sources.

vii) Optimisation knowledge - contains cost optimising factors which indicate how the derived expert system can be efficiently executed.

The above modules collectively make up the knowledge base. Since the knowledge base is modular, maintenance is made easier. The use of a classifier (see 4.2.1) for the construction of the knowledge base further eases the task of maintenance. In the earlier version of EES (Neches, et al., 1988), the KL-ONE classifier was used

to build the knowledge base. The LOOM classifier was used in the later version (Swartout, et al. 1991).

(2) The formal recording of the system development process

The various components of the knowledge base listed above are used by EES to generate an executable expert system and a 'Development History'. The Development History is a historical record of the development process. It captures the rationale behind each specific action taken in the design of the system and hence is useful for providing explanations about why a given aspect of the system was designed or implemented in a given way. It is from the Development History that EES derives its explanatory power.

The generation of the expert system code and the Development History is done by a mechanism called the 'Program Writer'.

The Program Writer refers to the goals found in the domain model, refines them into subgoals, and carries on the refining process until the primitive levels are reached.

As it does this refinement, the Program Writer records the steps it went through, producing a 'Refinement Structure'. Since the Refinement Structure is really a historical record showing how the expert system was generated top down from the high level goals to the implementation levels, it is also referred to as the Development History. This Refinement Structure may be a lattice or a tree-type structure.

IMPLEMENTATION DETAILS

The steps involved in generating an executable expert system and the Development History are :

Step 1 : The Program Writer starts at the highest level goal.

Step 2 : It searches the GPS Principles' Plans for a match of a Plan's Capability Description with the goal.

Step 3 : If a match is found, the Program Writer uses that corresponding 'method' to implement that goal. A 'specialisation' process may be required.

For example :-

- a goal may say 'diagnose faulty XYZ-chip'

- a search of the GPS Principles Plans finds the Capability Description 'diagnose faulty component'

- 'component' is replaced with 'XYZ-chip' wherever it appears in the Plan's Method.

This specialisation process is recorded by the Program Writer so that EES can explain the relation between the specialisation and the GPS Principle from which it stems.

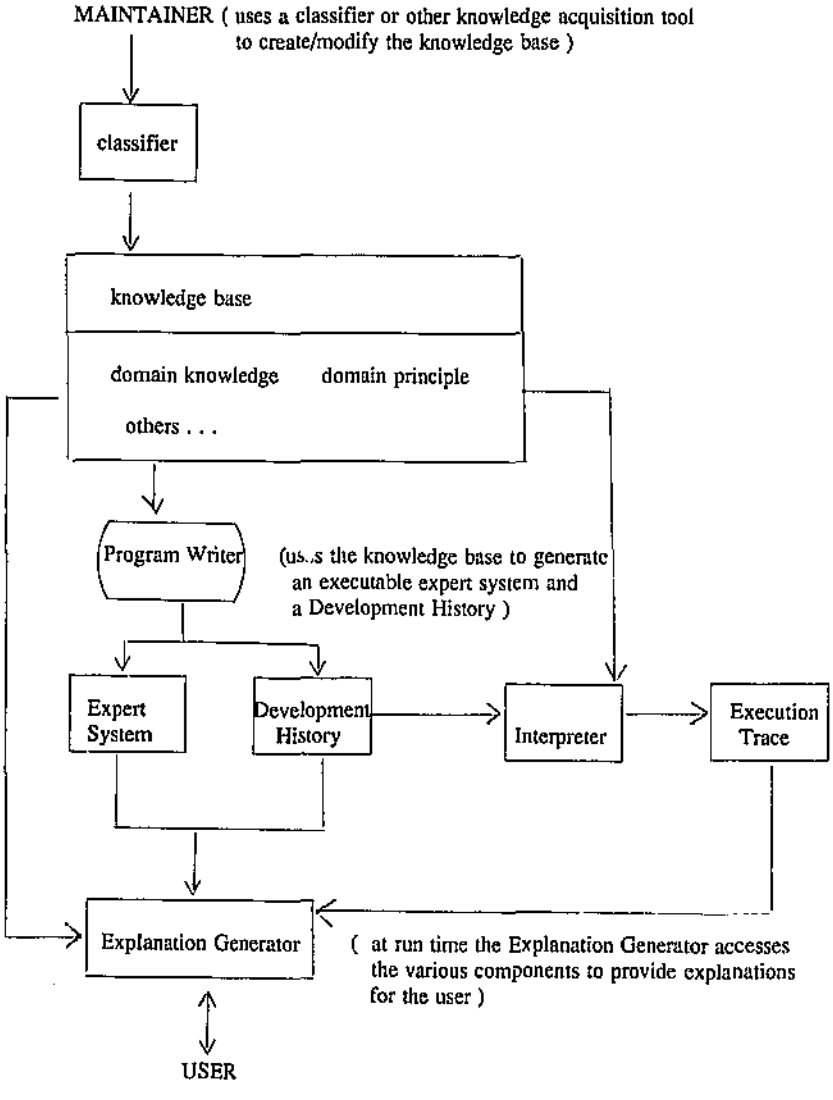
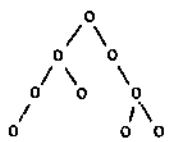


Figure 4.a : A brief outline of the EES framework

Step 4 : If no match is found, the Program Writer uses a process called 'Reformulation into cases' to reformulate the goal. There are several ways of reformulation, but generally it involves the breaking down of goals into subgoals.

The reformulation process is recorded by the Program Writer so that EES can explain how it was derived.

Step 5 : With each subgoal so generated, the Program Writer returns to re-do Step 2 until all subgoals have been implemented (ie. the leaf nodes have been reached).



Highest level goal

Reformulate into subgoals until leaf nodes are reached.

Figure 4.b : The Refinement Structure

The leaf nodes are the actual implementation. Each leaf node contains an implementation code.

The 'interiors' of the leaf node are the goals and decisions made on the way to generating that leaf (or implementation).

Once the leaf nodes (ie. executable codes) are reached, the Program Writer uses Tradeoff knowledge, Preferences, and Integration knowledge to construct the 'Control Components'.

During execution time, the Control Components select which codes to use based on the Tradeoffs and Preferences.

The Program Writer further uses the Optimisation knowledge to improve execution efficiency. This is achieved by re-organising procedures to reflect concerns for reducing costs (both computational and domain specific).

The final result of the generation is a runnable expert system and a Development History.

Two main types of reasonings are produced by the Program Writer. These are the 'Implementation' reasonings (which are of interest to the maintainers) and the 'Domain Level' reasonings (of interest to the users). These different kinds of reasonings are explicitly marked in the design record so that EES can identify them when providing explanations for different users.

During run time an Interpreter accesses the Development History and maintains an execution trace of the run.

The user interacts with an Explanation Generator. The Explanation Generator has the ability to access the Interpreter, Execution Trace, knowledge base, and the Development History in order to produce the relevant explanations.

The Explanation Generator can detect certain structures in the Development History which allows it to determine which goals are generated as a result of implementation concerns and which are problem solving goals and uses them accordingly to explain or answer users' or maintainers' queries.

Other heuristics allow the Explanation Generator to decide which level of detail is appropriate when defining terminologies to the users. When comparing concepts, it can detect similarities in the structures of different concepts and combine them into a generalised description.

COMMENTS

- This concept of explicitly distinguishing different types of knowledge and the formal recording of the system development process is not new. As noted by Neches, et al., it was used in the Xplain system. The difference is that while "Xplain recognises two forms of domain knowledge (factual vs problem solving methods) and one kind of Development knowledge ..." (Neches, et al., 1988, p. 283), EES recognises many more.

- The EES Program Writer concept appears to resemble the conventional program compiler concept whereby a program has to be recompiled each time any change is made to it. By the same token each time a modification or an error correction is done on the knowledge base, the whole expert system has to be re-generated by the Program Writer. This obviously is not a very efficient technique if there are many minor changes to be made on a regular basis.

- EES illustrates an example of a system that can 'understand' its own behaviour. As Lowry et al. said "The EES framework is a first step towards a self-aware, self-healing software" (Lowry and Duran, 1988, p. 295). This should augur well for a self-modifying system of the future since arguably self-understanding is a pre-requisite step towards self-modification.

4.1.2 OTHER KNOWLEDGE BASE UNDERSTANDING AIDS

Apart from the provision of good explanation facilities, techniques for making a knowledge base easy to understand are very much dependant on its construction.

Software engineering techniques such as modularity, structured principles, documentation, user manuals, inline comments, use of indentations, cross-reference listings, data dictionaries, standardisation, etc contribute to the ease of knowledge base understanding.

This section will outline some other methods which might be used to aid the process of understanding a knowledge base or making a knowledge base easy to understand.

4.1.2.1 AUTOMATIC PROGRAM UNDERSTANDING (APU) PARADIGM

Such a tool is invaluable to a programmer since (as mentioned in section 4.1), understanding of the code before modification can take up more than 60% of a maintainer's time.

The APU paradigm is based on the reverse engineering principle (Lowry, et al., 1989), a principle which applies compiler technology in reverse to derive a low level specification from the code. Such a principle may be seen as the inverse of the Automatic Program Synthesis (APS) concept. In the APS concept the idea is to generate codes from specifications while APU starts from the code and generates the specification.

The philosophy behind APU is to pre-store all possible implementation instances, all programming techniques and strategies, data types, data structures, and problem solving algorithms which may be used in the coding of some arbitrary programs. Once this has been done, understanding now becomes a matter of retrieving the stored patterns to match the given input patterns.

The obstacle to APU is that (apart from very limited applications) in practice, it is not possible to pre-store all such computational instances since there are an infinite number of them. Even if it were possible to pre-store them, the process of matching input patterns to the pre-stored patterns will be combinatorially explosive. Hence it would appear that such a concept may still be quite a long way from being useful as a knowledge base understanding aid.

4.1.2.2 KNOWLEDGE BASE SOFTWARE ENGINEERING (KBSE) CONCEPT

One of the aims of KBSE is to have maintenance done by performing modifications on the specifications and then rederiving the codes, rather than directly modifying the code (Lowry and Duran, 1989, p. 245).

REFINE (Lowry et al., 1989, p. 251) is a commercially available tool in which programs are specified declaratively at the level of sets and logic. Knowledge-based compilers are then used to transform them to lower level constructs.

While the KBSE concept appears to be currently directed at enhancing the development and maintenance of conventional systems, it will be interesting to see if this technology can be applied to the maintenance of knowledge bases themselves.

4.1.2.3 HOMOGENEITY AND PREDICTABILITY

To achieve homogeneity a knowledge base builder must use the same solution to solve the same type of problems. This ensures standardisation of code which in turn makes for easier understanding of the knowledge base.

Predictability demands that each rule in the knowledge base should serve only one function, and related rules should not be scattered across the knowledge base.

Knowledge bases which exhibit these two characteristics enhance understanding. RIME (see 3.2.3) is a language which promotes both these characteristics in the building of knowledge bases.

4.1.2.4 DECLARATIVE LANGUAGE

Making a language more declarative may make its code easier for the maintainer to understand. Separation of declarative knowledge from procedural knowledge is the principle behind SYLLOG (refer section 3.5.2).

4.1.2.5 PROPOSING SIMPLIFIED RULES

One of the component tools of the EVA system is the Rule Proposer (Landauer, 1990). Its function is to propose new rules from an existing set of rules. The Rule Proposer does this by analysing a given set of rules and then uses induction to form new rules. These proposed new rules represent a simplification over the old ones, thus facilitating understanding.

4.1.2.6 PRODUCING FORMAL SPECIFICATION FROM EXISTING SYSTEM COMPONENTS

The 'behaviour verifier' of EVA analyses the behaviour and interactions among system components and then reports on the collective behaviour of these components (Landauer,1990).

If formal specifications of all the sub-systems and their interactions exist, the behaviour verifier can be used to help produce a formal specification of the total system behaviour.

4.2 AIDS TO FACILITATE THE PROCESS OF MODIFICATION

This section looks at interactive aids which facilitate the process of knowledge base maintenance. These interactive aids come in many different forms.

For instance, tools like TEIRESIAS (Davis, 1984) and Knowledge Analyst's Assistant (Debeham, et al., 1991) attempt to take the place of knowledge engineers by automatically providing interactive advice and guidance to the experts during the modification of knowledge bases. When a modification or an addition is made, such tools check the modification with the existing knowledge base, then suggest further possible actions the expert might wish to take or might have overlooked.

Tools like interactive classifiers, on the other hand, use the subsumption principle to automatically determine where a newly described concept should be placed in the knowledge base and then verify their decisions with the users.

Knowledge refinement tools, like SEEK (Politakis, 1985) or SEEK2 (Ginsberg, 1988), allow the users to interactively experiment with changes by testing these changes against stored test cases, before permanently incorporating them into the knowledge bases.

The above tools are in contrast to those that are used to check the knowledge base as a separate step after changes have been made. Such tools will not be considered in this section but are discussed in section 4.3.

4.2.1 INTELLIGENT ASSISTANT CONCEPT

Among the earliest attempts at intelligent tools to interactively assist an expert to maintain an existing knowledge base was TEIRESIAS (Davis, 1984), (Davis, 1988).

This section examines TEIRESIAS, which is one of the most quoted systems. It has been cited by many authors for illustrating different concepts. It was seen as a knowledge refinement tool by Black (1986) who said that the goal of TEIRESIAS was to enable a domain expert to refine a knowledge base without the aid of a knowledge engineer (Black, 1986, p. 36). Perkins, Laffrey, Pecora, and Nguyen (1989) saw TEIRESIAS as a knowledge base debugger, claiming that TEIRESIAS was 'a first attempt to automate the knowledge base debugging process' (Perkins et al., 1989, p. 354). TEIRESIAS is considered by others as a knowledge acquisition tool which helps to automate the knowledge engineering process. Irani, Matts, Hunter, Slagle, Kain and Long (1990, p. 275) described TEIRESIAS as a knowledge editor. Davis (1985) said that one of the main goals for the creation of TEIRESIAS "has been the development of an intelligent assistant" (Davis, 1985, p. 172).

In this research TEIRESIAS is seen as an intelligent assistant to the expert. It is a useful maintenance tool which provides an expert with interactive guidance to facilitate the process of adding, deleting, or altering rules in a knowledge base.

4.2.1.1 NAME OF TOOL : TEIRESIAS

INTRODUCTION

TEIRESIAS is a program written in Interlisp at the Stanford University Computer Science department in the early eighties. The program was named after the blind seer in 'Oedipus the king' because its author likened the program to the blind prophet who has a form of 'higher order' knowledge (Davis, 1988, page, 243).

OBJECTIVE OF TEIRESIAS

To provide interactive guidance and advice to help a domain expert add, alter or delete rules from an existing knowledge base.

CONCEPT BEHIND THE TOOL

TEIRESIAS assumes that a knowledge base already exists and uses it to build 'rule models'. A rule model is a generalization of a class of rules. When a change is made to a rule or a new rule is added, TEIRESIAS verifies it with the rule model and reports to the maintainer any incompleteness or inconsistencies it detects and suggests possible remedies.

The concept of rule models used by TEIRESIAS is an example of meta-level knowledge application. Meta-knowledge enables TEIRESIAS to model its own knowledge. The rule models represented as meta-knowledge, are assembled by TEIRESIAS on the basis of the knowledge base contents. TEIRESIAS checks the knowledge base for rules which have common characteristics and uses these regularities to construct the rule models. The rule models are not static, but are assembled by TEIRESIAS as a result of its interaction with the expert.

During the modification process TEIRESIAS does not simply accept any additions or amendments of rules and add them to the knowledge base. Instead the rule models are used to evaluate the new knowledge. In so doing TEIRESIAS demonstrates the process of learning by examining what it already knows with what it is being taught.

IMPLEMENTATION DETAILS

A typical session with TEIRESIAS would involve the following steps :-

Step 1 : The domain expert indicates to TEIRESIAS that an incorrect conclusion has been detected from the knowledge base.

TEIRESIAS contains heuristics to help it select the best approach to track down the problem, and ask the expert for guidance to help it do so. This question and answer session would proceed until the problematic rules have been identified.

Step 2 : Once the incorrect rules are found the expert may modify them by changing them or by adding new ones. The expert enters this new knowledge in the form of a restricted natural language.

TEIRESIAS's task at this stage is to make sure it has 'understood' the expert correctly. It does this by matching the text entered by the expert against its own internal rule models and selecting one which characterises the text best. Since the rule models contain characteristics that the rules have in common, TEIRESIAS understands that these are the characteristics that the text entered by the expert should have. It then confirms its understanding of the new rules with the expert.

If it has been confirmed to be the case, TEIRESIAS would then generate the codes for the new rules, otherwise, the expert would use TEIRESIAS's built-in rule editor to modify the rules in a bid to help TEIRESIAS interpret them correctly.

Step 3 : Once the expert indicates to TEIRESIAS that he or she is satisfied with TEIRESIAS's interpretation, TEIRESIAS uses the rule model again, this time to see how well the new rule fits into the model. This is referred to by Davis as making a "second guess" (Davis, 1988, p. 256). TEIRESIAS does this by trying to find a complete match between the new rule and the rule model.

If there is an incomplete match TEIRESIAS points out to the expert what the differences are. For instance if the expert had modified a rule to do process A, TEIRESIAS may respond with "most rules that do process A also do process B - shall I add process B to your rule as well ? "

While the first use of the rule model (in step 2) was concerned with interpreting text and determining what the expert actually said, this second use of the rule models is to see what the expert plausibly should have said.

Step 4 : When both the expert and TEIRESIAS are satisfied, bookkeeping tasks are performed. That is, TEIRESIAS books the new rules into the knowledge base and tags them with information to aid maintenance (eg. name of author, date of change, etc).

COMMENTS

- While the generality of TEIRESIAS allows it to be applied to almost any domain, a criticism which may be levelled at it is that TEIRESIAS works only on knowledge bases built in the MYCIN architecture.

- TEIRESIAS does not make any formal assessment of the rules at the time they are initially entered. Before TEIRESIAS can be used a knowledge base must already exist.

- TEIRESIAS appears to be a foundation from which many other concepts have stemmed. It embodies concepts of knowledge refinement, machine learning, automatic debugging, knowledge editing, an example of meta-knowledge application, an intelligent assistant, etc.

4.2.2 KNOWLEDGE CLASSIFIERS

A major problem faced by a maintainer as a knowledge base grows in size and complexity is the increasing danger of introducing inconsistencies and errors every time an addition or modification is made.

This section looks at a tool which uses the contents of an existing knowledge base and knowledge about its representation to help the maintainer introduce new objects. Such a tool is referred to as a classification tool or classification system.

A classification system * is basically made up of : -

(a) a knowledge representation language -

Typically this may be a frame-based, rule-based, or other representation based language. It is used for implementing the knowledge base.

(b) a classifier -

This is an algorithm for identifying the taxonomic location of a new concept and adding it to that location in the structure.

A classifier considers the objects in a knowledge base as nodes of an ordered structure, linked together by a subsumption or an inheritance relation. When a

* Many authors appear to use the term 'classifier' or the 'language' loosely and interchangeably with the classification system itself. Brachman in clarifying what 'KL-ONE' is said "occasionally the name has been used to refer to just the language" (Brachman and Schmolze, 1989, p. 207).

node (rule) is added, the classifier determines its appropriate position in the ordering and verifies this decision with the maintainer before placing it in that location.

The main idea behind a classifier is that given two concept definitions, it is possible to determine if one subsumes the other (provided both have precise definitions).

Often, however, it is not possible to give precise definitions to every concept. A classifier has to deal with this problem.

4.2.2.1 AN EARLY CLASSIFIER

The way the KL-ONE * classifier handles this problem is by recognising two main types of concepts. The first type are those which do not have precise definitions.

These are called the Primitive Concepts (PC). The other type is the Defined Concepts. These are concepts that can be defined in terms of the PC. In other words, PCs are the basic concepts from which all concepts are built.

In a KL-ONE taxonomy, the most basic PC is the Root Concept. The Root Concept is the first concept to be defined, usually called 'THING'. THING subsumes everything and is the only concept that does not have a super-concept (ie the subsuming concept or the parent).

* The KL-ONE classification system which first appeared in 1977, consists of a representation language which is "based on the structured inheritance networks" (Brachman, et al., 1989, p. 208). Its classifier is based on subsumption inheritance.

Apart from THING every concept must have at least one super-concept plus a local internal structure. The local internal structure defines the local restriction or properties or attributes. If a concept does not have a local restriction and has only one parent (super-concept) then it is the same as the parent itself. Hence to be well-defined a concept with no local restriction must have more than one super-concept.

When a classifier is used directly by the maintainers to add a new concept the maintainers need to know the descriptive terms in use in the existing knowledge base in order to create a concept that can be accurately classified. If an error is made in the classification, the maintainers must repeatedly modify the classification until they are satisfied. This process is much more efficient if it is done interactively. An interactive classifier establishes a verification interaction with the user to ensure that a new node is subsumed correctly.

4.2.2.2 INTERACTIVE CLASSIFIERS

This section looks at a simple interactive classifier called KuBIC (Knowledge Base Interactive Classifier). KuBIC's interactive classification algorithm is implemented in Prolog. It uses a simple representation language based on the tree structure.

NAME OF INTERACTIVE CLASSIFIER : KuBIC

OBJECTIVE

KuBIC is designed for the main purpose of exploring the underlying ideas of interactive classification.

CONCEPT

KuBIC's interactive classification is based on the subsumption relation. The use of subsumption relations economises descriptions and localizes distinguishing information.

Economy of description is achieved through the inheritance of attributes and attribute values by a node's children (ie. its subsumees).

Localizing distinguishing information means that when a new node Y has been determined to be subsumed by an existing node X, then only X's children (ie Y's siblings) need to be considered in order to find a more specific subsumer of Y. In other words, the classifier may localise its questions by using only the information stored in X's children to determine this node.

IMPLEMENTATION DETAILS

There are three main steps involved in the classification of a newly introduced concept (or node). These are :

1. getting the initial description;
2. finding the most specific subsumer;
3. finding the most general subsumees.

Step 1 : Getting the initial description

The user specifies the initial description of a new node to be introduced to the knowledge base by the following steps :-

If a subsumer is known (rarely the case), the user may name the subsumer directly, then proceed to Step 2, otherwise, the interactive classifier will determine the subsumer by asking the user for attributes and attribute values about that node.

With this attribute information the interactive classifier proceeds to determine the most specific subsumer for this new node.

Step 2 : Finding the most specific subsumer

The interactive classifier searches top down for the most specific subsumer starting at the root of the tree using an appropriate search strategy.

For example, if the new node Y, has been determined to be subsumed by node X, and X1, X2, X3 are X's subsumees in the knowledge base (see figure 4.c), then only X1, X2, or X3 are possible candidates for a more specific subsumer of Y.

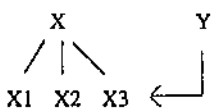


Figure 4.c Finding the most specific subsumer

The information stored in X1, X2, and X3 allows the interactive classifier to select questions which will determine which node is the more specific subsumer of Y.

This process goes on until the MOST specific subsumer of Y is found. For example, in this case, if none of X1, X2, or X3 subsumes Y, then X itself is the most specific subsumer of Y, and Y will be placed under X, alongside X1, X2 and X3.

Step 3 : Finding the most general subsumees

In order to define Y's location fully we now need to determine Y's most general subsumees.

This task is now relatively simple because the search is confined to Y's siblings only.

For example, if it was found in Step 2 that X was Y's most specific subsumer then X1, X2 and X3 are the siblings of Y (see figure 4.d)

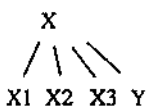


Figure 4.d Siblings of node Y

The interactive classifier now checks information obtained from X1, X2 and X3 in turn to see if Y subsumes any of them. For instance if Y subsumes X1 and X3 then X1 and X3 becomes the most general subsumees of Y (see Figure 4.e)

In this way the new node Y has found its taxonomic location in the hierarchy.

```
      X
     / \
   X2    Y
        / \
      X1   X3
```

Figure 4.e Most general subsumees of node Y

Note that the above example describes a tree classification. In a lattice classification, the process would be more complex.

COMMENTS

- To automatically determine if a new node is subsumed by another, precise definition of both nodes' attributes and attribute values are necessary, otherwise the classifier has to check these values with the user.

  Since there are no precise definitions for Primitive Concepts, the classifier needs to check every Primitive Concept in the knowledge base with the user. This hinders the functioning of the classifier since in non-trivial knowledge bases the number of Primitive Concepts are large. KL-ONE suffers from this problem, as does the above interactive classifer.

  To overcome the problem, more expressive representation languages are required. An example of a classifier that handles this problem is KLASSIC (Finin, 1988).

- Current hybrid classification systems typically contain both a frame and a rule language. The drawback of this combination is the system's inability to reason with both kinds of representation in a uniform manner.

  The KREME and the LOOM classification tools (MacGregor and Berstein, 1991) move away from this combination to support instead a description language and a rule language and use a common 'descriptive classifier' for deriving inferences between these two representations.

4.2.3 KNOWLEDGE REFINEMENT TECHNIQUES

Knowledge refinement techniques are examined in this paper because they may be used to facilitate the process of knowledge base modification on an interactive basis. However, such techniques are not suitable for making major changes to the knowledge base. They are meant to be used on knowledge bases which are already relatively accurate, and where only small changes to improve performance are required. The knowledge base to be modified is considered to be a first approximation of the final version. It is refined by adding, deleting or modifying its contents incrementally until the expert is satisfied that it can perform 'correctly'.

Traditionally, knowledge refinement techniques use learning by induction over a library of test cases. Examples of these include ID3, INDUCE (quoted in Wilkins, 1989, p. 247), and SEEK (Politakis, 1985). Learning by apprenticeship has also been used in knowledge refinement by a tool called ODYSSEUS (Wilkins, 1989).

This section examines SEEK, a knowledge refinement tool which is capable of interactively guiding a maintainer or developer during a refinement process.

4.2.3.1 NAME OF TOOL : SEEK

INTRODUCTION

SEEK (Politakis, 1985), (Ginsberg, 1988), is an acronym for System for Empirical Experimentation with Expert Knowledge. It was first developed at Rutgers University for use in the A1/RHEUM system (a system for diagnosing rheumatic diseases).

OBJECTIVE OF SEEK

To integrate the process of knowledge refinement and validation of the knowledge base into a single framework.

CONCEPT BEHIND SEEK

SEEK requires the presence of two sets of knowledge :

i) a model of an expert-derived knowledge base;

ii) a stored set of test cases. This database of stored cases is called the Case Knowledge (Ginsberg, 1988, p. 2). The Case Knowledge must be elicited from the expert. It is a set of problem scenarios for which the expert's conclusions are known.

The actual refinement process is driven by a comparison of these stored conclusions with those derived from the knowledge base. Whenever an incorrect result is detected, SEEK offers suggestions to guide the expert to revise and refine the knowledge base in order to make it reproduce the correct result.

By modifying the contents of the knowledge base in order to correct its functioning, the refinement process may be seen as performing a validation on the knowledge base.

IMPLEMENTATION DETAILS

Step 1 : A model of the knowledge base must be built by using a specialised text editor. The model is represented in a tabular form.

Step 2 : Case experiences (which forms the Case Knowledge) are collected in the form of questionnaires. They are then entered into a database forming a library of test cases whose conclusions are known. These conclusions are the 'correct' final diagnosis assigned to the test cases.

Step 3 : The refinement process. This process involves three steps :

i) obtaining the knowledge base performance summary;
ii) analysing the rules; and
iii) revising the rules.

i) Obtaining the knowledge base performance summary.

This step produces a performance summary for the knowledge base over all stored cases. The performance summary shows the number of cases in which the conclusions produced by the knowledge base matched with the stored conclusions.

The result is displayed in the form of a table from which mis-diagnosed cases may be identified and analysis carried out.

ii) Analysing the rules.

SEEK provides interactive assistance during the analysis of the rules.

Analysis may be done in two ways :

a) Analysis of the model over a single case.

This mode of analysis is used for providing the knowledge engineer with an explanation of the results produced by the knowledge base for the particular case under test.

If the conclusion obtained from the knowledge base matches the correct solution, SEEK displays the rules used to achieve the solution, otherwise, SEEK attempts to locate the partially satisfied rule for the expert's conclusion that is closest to being satisfied and advises the knowledge engineer on what sort of refinements to make. Refinements may take the form of either generalising the rule or specialising it.

Rather than testing the whole knowledge base, the knowledge engineer is also given the option of testing the performance of selected subsets of rules within the knowledge base over that single case.

b) Analysis of the model over multiple cases.

This mode is used for the global analysis of the knowledge base (ie. testing the knowledge base over a multiple or an all-case basis).

As with the previous mode, the knowledge engineer has the option of testing either the whole knowledge base or selected sections of the knowledge base. Testing selected sections allows the knowledge engineer to focus attention

on a subset of rules to be analysed.

Typically, the knowledge engineer would begin by selecting a subset of rules to be analysed. This is normally the set of rules with mis-diagnosed cases.

SEEK performs the analysis and automatically generates advice on how to refine the rules. This advice comes in the form of a report which :-

- ranks the rules which are potential candidates for generalisation and those which are potential candidates for specialisation;

- proposes experiments for carrying out specific generalisation or specialisation of the rules. As the number of possibilities that could be tried in order to correct the misdiagnosed cases is enormous, SEEK contains heuristics which allow it to narrow down the experiments to try. For example, SEEK selects only the rules that agree with the expert conclusions which are closest to being satisfied in a misdiagnosed case.

Based on this report the knowledge engineer now proceeds to the next step, that is the revision of the rules.

iii) revising the rules

The revision of the rules is carried out by trying SEEK's suggested experiments. These experiments conditionally incorporate the change into the knowledge base and test it against the case library.

SEEK, however, does not commit the knowledge engineer to make the change permanent. The knowledge engineer has the option of either accepting the change or rejecting it.

COMMENTS

- Knowledge refinement techniques suffer from the following shortcomings :

  i) only minimal changes are feasible since refinement techniques work under the assumption that the knowledge base is generally correct;

  ii) a comprehensive set of cases may be hard to collect, if not impossible;

  iii) in a multi-paradigm knowledge base many different data structures are allowed. This added complexity will make it unsuitable for knowledge refinement techniques. Kulikowski (1989, p. 171-172) claimed that "the resulting non-homogeneity blocks the application of consistent knowledge refinement heuristics."

- Further to the shortcomings of refinement tools in general, SEEK in particular, is a tool of restricted scope of applicability since :

  i) it only works on rules written in a restricted tabular format (Politakis, 1985, p. 3);

  ii) even basic refinements, which ideally should be automated, need to be done interactively, making the use of SEEK a time-consuming and tedious process.

- The above shortcomings of SEEK have been corrected by SEEK2 (Ginsberg, 1988) which works with a more general class of knowledge base formats and can perform basic refinements automatically.

  In addition SEEK2 also provides a meta-language which can be used for specifying domain-independent and domain-specific meta-knowledge about the refinement process.

4.2.4 OTHER INTERACTIVE MODIFICATION TOOLS

Knowledge base editors, debuggers, and knowledge acquisition tools are among some of the systems which allow users to elicit domain knowledge from experts, access the knowledge and manipulate or change it on an interactive basis.

4.2.4.1 KNOWLEDGE BASE EDITORS

To some degree a knowledge base editor fulfils the role of being an interactive maintenance aid to a knowledge engineer.

Features of a knowledge base editor include entry, browsing, viewing, accessing and editing facilities. The browsing facility is useful for exploring existing knowledge base before making changes.

Finin (1988) said that Schoen and Smith have described a 'display-orientated' knowledge base editor for representation language, STROBE, and Lipkis and Stallard are developing an editor for the KL-ONE representational language (Finin, 1988, p. 275).

Terveen, Wroblewski and Tighe (1991) talked about the HITS knowledge base editor, an editor which provides intelligent assistance through a process called 'collaborative manipulation' of objects in a shared workspace (an area for joint user-system problem solving).

4.2.4.2 AUTOMATED KNOWLEDGE ACQUISITION TOOLS

Marcus (1988) said that automated knowledge acquisition tools are "tools that can elicit

relevant domain knowledge from experts, maintain that knowledge in a form that makes it accessible for analysis, review or modification . . . " (Marcus, 1988, p. 1).

Examples of such tools are MORE (Kahn, 1988), an automated knowledge acquisition system that helps refine an existing knowledge base; MOLE (Eshelman, 1988), a knowledge acquisition tool for generating expert systems that perform heuristic classification; and SALT (Marcus, 1988), an automated knowledge acquisition tool that addresses synthesis (as opposed to analysis) problems.

4.3 ENSURING CORRECTNESS AFTER MODIFICATION

The previous section has been concerned with interactively maintaining the correctness of the knowledge base as it is being modified. TEIRESIAS was seen as a tool which interactively checked knowledge base consistency during a modification session, while interactive classifiers place newly described knowledge into their correct location using subsumption, and knowledge refinement tools like SEEK interactively validate the knowledge base against a set of test cases.

This section looks at a different approach, one in which a knowledge base is modified, then is checked for correctness as a separate step.

As in most skilled professions, a maintainer's skill does not lie in the ability to modify a piece of code, but in the ability to ensure that nothing goes wrong after the change. This skill can be aided by appropriate tools. This section will discuss the various verification and validation tools and techniques which help towards ensuring that nothing goes wrong as a result of making a modification.

Generally there are two levels of testing - verification and validation. However, there appears to be no agreement among authors on the use of the terms verification and validation. O'Leary, Goul, Moffit, and Radwran (1990) said that "Unfortunately, the term validation is inconsistently used ..." and that "Balci and Sargent found that a standard definition does not exist..." (O'Leary et al., 1990, p. 51).

In this thesis "verification" refer to checking that the knowledge base matches with the specification. Verification checks demonstrate consistency, completeness and correctness of the knowledge base. That is, verification is concerned with structural correctness.

"Validation" on the other hand is concerned with determining the correctness of the knowledge base with respect to the user requirements. In other words the knowledge base is 'functionally' correct and acting in accord with the user's intentions.

These definitions are in accordance with those given by Lowry et al. in the 'Handbook on Artificial Intelligence' which states that verification mathematically proves software's correctness with respect to a formal specification while validation checks whether the system satisfies the needs for which it was developed (Lowry et al., 1989, p. 248-249).

Lowry et al. further asserted that validation failure is the result of shortcomings or errors in the specification owing to miscommunication or poor understanding of initial needs while verification failure is the result of errors in the software resulting in its failure to meet specification.

This section is divided into two parts. The first part looks at verification techniques while the second deals with validation techniques.

4.3.1 KNOWLEDGE BASE VERIFICATION TECHNIQUES

This section is concerned with demonstrating the structural correctness of a knowledge base. What is meant by 'verifying for structured correctness' depends very much on the knowledge representation formalism used.

Lopez, Meseguer, and Plaza (1990) said that "if we consider production rules, the following properties would be part of the structural verification :-

- redundant rules
- subsumed rules
- circular rule chains
- unfireable rules,
- non-reachable goals, ... "

(Lopez et al., 1990, p. 59).

Most work on verification appears to have been done in the area of rule-based systems only. This paper examines some of them.

ONCOCIN will be looked at for historical reasons as it is one of the earliest attempts at knowledge base verification. This is followed by CHECK, a verification system which is an extension to ONCOCIN. More current methodologies are then commented upon.

4.3.1.1 ONCOCIN RULE CHECKER

INTRODUCTION

The ONCOCIN Rule Checker (Suwa, Scott, and Shortliffe, 1984) is a rule-based verification program written for the ONCOCIN system, an expert system for oncology protocol management.

Although specifically designed for the ONCOCIN rule base, its developers claimed that the rule checker is general and adaptable to other rule bases (Suwa et al., 1984).

OBJECTIVE OF THE ONCOCIN RULE CHECKER

To check a rule base for conflicts, redundancies, subsumptions and omissions.

CONCEPT BEHIND THE TOOL

A rule in ONCOCIN is considered to be made up of :

i) a CONDITION part which consists of one or more condition parameters

ii) an ACTION part which has a single action parameter

The basic idea behind the ONCOCIN Rule Checker is that if a rule base is partitioned into disjoint sets such that each set is made up of only those rules which conclude a value for the same action parameter, then it is possible to check these resulting disjoint rule sets independently for conflicts, redundancies, subsumptions and omissions.

IMPLEMENTATION DETAILS

Step 1 : The rule base is checked for rules which have a common action * parameter in the action part of the rules.

Step 2 : These rules (which may be scattered throughout the knowledge base) are grouped to form disjoint rule-sets.

As an example, consider a case where three rules in a given knowledge base have all been found to conclude the same action parameter, Fruit (see figure 4.f)

The rule checker groups them together to form a disjoint rule-set.

Step 3 : For each disjoint rule set thus formed, the ONCOCIN Rule Checker does the following :-

(a) identifies all parameters and parameter values used in the condition parts of these rules, and determine the total number of possible combinations of these condition parameter values.

* In selecting rules with a common action parameter, the ONCOCIN Rule Checker also considers the 'context' in which the rules apply. The 'context' is identified by having an extra slot added to the rule, hence one additional check is necessary. For simplicity of illustration 'context' is ignored in this discussion.

Rule number	Rule
34	if Color = red and Size = tennis ball then Fruit = apple
107	if Color = red and Size = marble then Fruit = grape
187	if Color = green and Size = tennis ball then Fruit = apple

Figure 4.f Rules concluding the same action parameters

In the example :

the number of condition parameters = 2 (ie. Color, Size)

the number of parameter values = 2 for Color (ie. red, green)
= 2 for Size (ie. tennis ball, marble)

the possible combinations (of condition parameter values) = 2 x 2
= 4

(b) The rule checker then creates a table consisting of all possible combinations of condition parameter values and their corresponding action parameter values, and examines the table to detect conflicts, redundancies, subsumptions and missing rules.

(c) It then produces a report which shows this table with comments or error messages (like 'redundant', 'conflict', 'subsumption' or 'missing') listed alongside entries in the table (see table 4.a).

RULE NUMBER	CONDITION PARAMETERS				ACTION PARAMETER	REMARKS
	Color		Size			
	red	green	tennis ball	marble		
34	Y		Y		apple	
107	Y			Y	grape	
187		Y	Y		apple	
		Y		Y		MISSING

Table 4.a All possible combinations of condition parameter values and their corresponding action parameter values.

Missing rules :

The above table illustrates how the rule checker detects a missing rule. The table shows the four possible condition parameter value combinations. There was no rule in the set which matches the last combination of Color = green and Size = marble. Hence the rule checker considers this rule as 'MISSING'.

Subsumption :

If there is another rule in the knowledge base which says :-

if Color = green then Fruit = apple

then this rule will get an entry into the table. The remark 'SUBSUMPTION' will appear alongside it as well as alongside Rule 187.

Conflict :

If there is a rule that says :-

if Color = red and Size = marble then Fruit = strawberry

then this rule will get an entry in the table with the remarks 'CONFLICT' appearing beside it as well as beside Rule 107.

Redundancy :

If there is a rule that says :-

if Size = marble and Color = red then Fruit = grape,

then the remark 'REDUNDANT' will appear beside this rule and also Rule 107 in the table.

COMMENTS

- Although claimed by its developers to be general and therefore adaptable to other rule-based systems, the ONCOCIN Rule Checker applies to rules which have a restricted syntax only. For instance, the action part of the rule can only conclude one parameter value. It also has no facilities to check for deadend ifs, unreachable conclusions and circular rules.

- The verified results are not always accurate. A reported missing rule may not be a real missing rule. This is because the ONCOCIN Rule Checker assumes there should be a rule for each possible combination of values of condition parameters, but some combinations may be meaningless. This false alarm serves as a distraction to a maintainer.

  Tsang, Wan, Lim, and Hioe (1988, p. 575) said that the developers of ONCOCIN "plan to apply semantic knowledge for eliminating these meaningless combinations".

4.3.1.2 NAME OF TOOL : CHECK

INTRODUCTION

CHECK (Perkins, Laffey, Pecora and Nguyen, 1989) is a rule-based verification tool designed for use with the Lockheed expert system shell. It is an extension of the ONCOCIN Rule Checker.

In addition to performing ONCOCIN Rule Checker's conflict, redundancy, subsumption, and missing rule checks, it also checks for unnecessary ifs, deadend ifs, deadend goals, unreachable conditions, unreferenced parameter values, illegal parameter values and circular rules.

It further differs from the ONCOCIN Rule Checker in that it is applied to the entire rule base rather than just subsets of rules.

OBJECTIVE

To statically verify a rule base for inconsistencies and incompleteness.

CONCEPT BEHIND CHECK

A rule in CHECK is broken down in the following manner (refer figure 4.g) :-

The rule is considered to be made up of an 'IF part' and a ' THEN part'.
A part is made up of one or more 'clauses'.
A goal is equivalent to a 'THEN part'.

```
        RULE                    GOAL
       /    \                  /    \
   IF PART   THEN PART       CLAUSE  ....
   /    \     /    \
CLAUSE  ....  CLAUSE  ....
```

Figure 4.g The rule structure of CHECK

CHECK starts from the most basic relationship and works upwards. It first establishes the relations among all the clauses in the rule base. Using this interclause-relationship it then deduces relationships among parts, and from them deduces relationships among rules.

From these three types of relationships thus obtained, it is possible to deduce how clauses in the rules (and goals) affect the other clauses to give rise to inconsistencies and incompleteness.

IMPLEMENTATION DETAILS

Step 1 : Determine how clauses are related with one another.

This is achieved by comparing the clauses of every rule against the clauses of every other rule. From the comparison a two-dimensional interclause-relationship table is drawn up showing the relations of every clause to every other clauses. These relations may be 'SAME', 'DIFFERENT', 'CONFLICT' , 'SUBSET' or 'SUPERSET'.

Step 2 : Determine how parts are related to one another.

This is achieved by looking at the interclause relationship table for the clauses that made up each part and from them deduce the overall parts relationships. Again these relationships may be 'SAME', 'DIFFERENT', 'CONFLICT', 'SUBSET' or 'SUPERSET'.

Step 3 : Determine subsumptions, redundant rules, and unnecessary clauses.

This is achieved by comparing the parts relationships of each rule against the parts relationship of every other rule in the knowledge base. The possible relationships yielded from this comparison are 'SAME' (meaning redundant)'; 'DIFFERENT'; CONFLICT'; 'SUBSET' and 'SUPERSET' (both of which indicate subsumption) ; or 'UNNECESSARY CLAUSES'.

Step 4 : Determine gaps in the knowledge base. Gaps are 'unreachable conclusions', 'deadend ifs', 'deadend goals'.

Unreachable conclusions - if a conclusion is not a goal and is not found in any IF part of a rule in the knowledge base then it cannot be reached. Hence to look for unreachable conclusions, look for THEN clauses (in the interclause-relationship table) which have DIFFERENT relations for all IF clauses and goals (note that DIFFERENT indicates there is no match).

Deadend ifs or goals - a goal or IF condition is deadend if a conclusion (ie. the THEN part of a rule) which matches it cannot be found, in addition, that goal or IF condition is not askable (ie. there is no routine to ask the user for its value).

Hence deadend ifs or goals are detected by the fact that they have the 'DIFFERENT' relationship for all conclusions in the interclause-relationship table, and the attributes they refer to are not askable.

THEN

IF

Rule	1	2	3
1		*	
2			*
3		**	

Figure 4.h : Dependency chart (Refer Rules 1,2,3 of page 126)

* indicates dependency : that is one or more IF conditions matches one or more conclusions of a rule. (eg. an IF clause of Rule 1 is concluded by Rule 2, ie. the THEN PART of Rule 2 is dependant on the IF PART of Rule 1; similarly an IF clause of Rule 2 is concluded by Rule 3)

** indicates circular rule set because a condition of Rule 3 (ie. A) is matched by a conclusion of Rule 2 and a condition of Rule 2 (ie. E) matches a conclusion of Rule 3.

Step 5 : Determine circular rule chains

Circular rule chains are determined by examining a 'dependency chart' (see figure 4.h). The dependency chart shows how a rule is dependent on another. It is generated from the interclause-relationship table.

Example :

Rule 1 IF A and B THEN C
Rule 2 IF D and E THEN A
Rule 3 IF F and A THEN E

COMMENTS

- An advantage of such static rule checkers like CHECK and ONCOCIN is that they systematically and exhaustively check every possible combination of rules in the rule base.

- The implementation overhead of CHECK, however, appears too high. Since CHECK compares every rule in the rule base with every other rule, it requires N(N-1) comparisons. Thus the number of checks performed is of the order N^2 (where N is the number of rules in the knowledge base).

- Although Certainty Factors are allowed in the LES shell, CHECK does not look at them when checking the rule base.

4.3.1.3 OTHER VERIFICATION TECHNIQUES

Although there are numerous other verification techniques using a varied mixture of concepts, almost all assume that the knowledge is stored in a rule-based format. Little work could be found in the literature which addresses the verification of knowledge bases in other than rule-based forms.

This section briefly outlines a few of these rule base verification techniques.

4.3.1.3.1 SPACE SEARCHING METHOD

Another technique for checking the consistency and correctness of a rule base is the Space Searching method of Tsang, Wan, Lim and Hioe (1988). This method was designed to fill the shortcomings of the ONCOCIN and CHECK rule checkers.

The main criticism of the ONCOCIN and CHECK rule checkers is that they detect only superficial inconsistencies; that is, inconsistencies which arise from a direct or superficial comparison of two rules. Inconsistencies that arise after a sequence of inference steps are overlooked.

In addition to this shortcoming, the ONCOCIN rule checker also over-reacts by sounding false 'missing rule' alarms on practically meaningless rule combinations.

Tsang et al. (1988, p. 575) pointed out that in spite of this "over-vigilant behaviour", the ONCOCIN rule checker "overlooks on some occasions". Since the ONCOCIN rule checker forms rules into rulesets and checks them independently, inconsistencies that arise when rules of different sets are chained together in the inference process are not detected.

The Space Search method detects inconsistencies which arise due to the interactions of rules during the inference process. It also removes some of the false warnings on improbable situations by excluding them from the search space.

However Tsang et al.'s method does not remove all false alarms as they admitted "a complete removal of these false warnings generally requires semantics knowledge of the problem domain " (Tsang et al., 1988, p. 577). This method does not include any plan for the application of such semantic knowledge.

4.3.1.3.2 PREDICATE/TRANSITION NET METHOD

The Predicate/Transition Net Method (Zhang & Nguyen, 1989) allows for the inclusion of consistency and completeness checks as part of the knowledge acquisition process, hence verification can be done in an incremental fashion as the knowledge base is being developed.

The technique is based on the use of a Predicate/Transition (Pr/T) net model representation and a syntactic pattern recognition method.

A program called 'Verifier' is used to implement this technique. This program starts off by transforming the knowledge base to be verified to the Pr/T model. The various inconsistency and incompleteness types or patterns are then defined with respect to the Pr/T model. A scanner is then used to search the Pr/T model (ie. the transformed knowledge base) for a match on these patterns. These matches are then highlighted as potential errors for the expert's confirmation.

The method is not applicable to all rule bases. The rule base must be represented in first order predicate logic before the transformation can be done. Apart from this problem the developers also admit some other shortcomings of the method; for example, an inability to handle incomplete cases.

4.3.1.3.3 ART RULE CHECKER (ARC)

ARC is a rule checker used to verify the consistency of expert system knowledge bases which utilise the Automated Reasoning Tool (ART) framework (Nguyen, 1988). ARC is basically an extension of the techniques used in CHECK (described in 4.3.1.2). The additional checks handled by ARC include checking for compound conditions, subsumed rule chains, redundant rule chains and conflicting rule chains. In addition, ARC uses a RETE-like* algorithm which is more efficient than CHECK's exhaustive checking of every possible combinations of rules in the knowledge base.

* A RETE algorithm (see Jackson, 1986, p. 128) makes use of pattern matching mechanisms to select rules from the rule-base.

4.3.2 KNOWLEDGE BASE VALIDATION TECHNIQUES

To inspire confidence in the use of the knowledge base, verification tests must be followed by validation tests.

Unlike traditional systems, expert systems face two main problems with regard to validation.

i) In an expert system there is often no absolute measure of correctness since the rules themselves are for the most part only judgemental. In many cases there is no single best answer and there may be no agreement as to what is an acceptable answer. Without an external criterion for correctness (in the form of an explicit requirement specification) it is not clear what it means for a knowledge base to be 'correct'. Hence, unlike a conventional system where correctness can be tested with a pre-determined set of results whenever changes are made to the system, in an expert system no such test is usually available.

ii) In addition to the absence of an absolute 'correctness' test, it is also impracticable (if not impossible) to fully test an expert system knowledge base. In a traditional program there are a finite number of paths which the program can take, and those paths are known in advance, hence test data can be prepared to test every known path in order to fully test the program*. The number of possible path combinations found in a non-trivial knowledge base would be combinatorially explosive, making such a test intractable.

* 'fully test' implies testing all known paths. This is different from claiming that the program is tested completely - a phenomena that is largely unattainable. Hetzel contended that testing is "not only practically but also theoretically impossible" to prove a program's correctness (Hetzel, 1988, p. 21).

EVALUATION

In the light of the above difficulties, it appears that validation as applied to an expert system knowledge base is reduced merely to 'evaluation'. This view is shared by McGraw and Harbison-Briggs (1989, p. 311) who acknowledged that the difficulty associated with validation "leads directly to the notion of evaluation rather than strict validation".

In evaluation we start with a 'valid' system and the evaluation process returns a rating. Much the same occurs here since we can only test for empirical adequacy, not absolute correctness.

The basic concept behind validation is to collect a set of test cases with known conclusions from the expert. (This is the rough equivalent of a traditional user requirements specification). This 'test cases' set is a set of scenarios in which the expert can perform 'correctly', and is itself really only a subset of all the possible cases. In other words, it is not possible to collect a full and complete 'user requirement specifications' for a non-trivial expert system knowledge base.

The performance of the knowledge base on similar test cases is then compared with the pre-determined conclusions of the set of test cases in order to determine what percentage of cases the knowledge base can perform 'correctly'.

V & V RESEARCH

Reducing validation to evaluation is clearly unacceptable. However, there appears to be no easy answer to this dilemma. Green and Keyes (1990) observed that due to the difficulties of applying verification and validation (or V&V) to expert systems, organisations refrain from requiring V&V in their expert system procurement.

This gives rise to what Green and Keyes (1990, p. 445) called the 'vicious circle' whereby "V&V is not done because nobody requires it. Nobody requires it because nobody knows how it's done. Nobody knows how because nobody has done it".

Clearly more research is urgently needed in the search for better and more formal V&V techniques in order to inspire greater confidence in the use of expert systems commercially.

Up until 1985 "there was almost no activity concerned with the testing of expert systems" (Miller, 1990, p. 249). Although the situation has changed markedly in recent years with the appearance in the market of numerous V&V approaches, the basic problems outlined above remain.

Since we cannot test for absolute correctness, could we establish a minimal level of V&V testing standards that is acceptable universally ? Can such a set of minimal testing standards be found ? Section 4.3.2.1 discusses two approaches which address this issue to some extent. They are the 'correctness principles' approach which attempts to lay down a set of acceptability principles for rule bases, and the 'validation standards' approach which aims at providing a basis for standardising the validation of a knowledge base system.

On a different note, some major projects, like EVA for instance, which was started in 1986, are continuing the efforts to research and develop V&V techniques for knowledge bases with the goal of building an integrated set of generic tools to perform V&V on any knowledge base application developed in any shell. This project is discussed in 4.3.2.2.

Other V&V techniques are also outlined in the following section.

4.3.2.1 TOWARDS VALIDATION STANDARDS

This section outlines two papers which advance the idea of taking a standard approach towards knowledge base validation. This is in line with the objectives discussed in section 4.3.2 where it was argued that in order to win user confidence in the application of expert systems, research should be directed at developing some acceptable minimal validation standards. The two approaches are in contrast to those which focus on the development of isolated validation techniques.

The first of these approaches, the 'Correctness Principle Approach', defines a set of acceptability principles for a rule base. Each of these principles is accompanied by a set of validation criteria. These criteria can be tested for by the use of analysis algorithms.

The second approach defines standards for classifying a knowledge base system. The position in the classification determines the degree and type of validation required for that knowledge base system.

4.3.2.1.1 CORRECTNESS PRINCIPLES APPROACH

This principled approach to V&V (Landauer, 1990), developed at the Aerospace Corporation in the United States, defines a set of 'correctness' principles for the V&V of a rule-based system.

These principles extend to more than just the rule base; they include the checking of connectivity between the rules and the inference engine, and other interfaces as well (for example, user interface, data interface).

Five such principles are identified in this approach. These are consistency, completeness, irredundancy, connectivity, and distribution. These principles go beyond addressing the normal correctness problems like consistency, completeness and redundancy of the rules in the knowledge base, as they include connectivity problems, (these are problems concerned with the completeness and irredundancy of the inference engine), and distribution problems (concerned with the 'esthetic' quality of the rules or the awkwardness of rule combinations). Esthetic aspects are considered because awkward rule combinations can conceal potential errors.

In order to check for the principles, validation criteria are needed. Criteria are the pre-defined standards which the quality of the knowledge base can be measured against (in other words, criteria are goals to be tested). For each principle, a set of criteria has to be determined. These criteria can then be tested by means of mathematical or computational algorithms.

Landauer advocated the use of mathematical algorithms (as opposed to algorithms based on a linguistic approach) for the testing of the criteria because "mathematical conditions can be checked effectively" (Landauer, 1990, p. 292). To test for these criteria the rule base must be viewed as a formal mathematical object (ie the rule-base has to be specified in accordance with strict conventions and vocabulary of formal logic). Only in this form can algorithms which make use of mathematical and graphical techniques be developed for analysing this mathematical structure.

Landauer's paper describes in detail several mathematical and graphical algorithms that could be used to test the rule base for some of the validation criteria. Some of the algorithms described are suitable for analysing the static structure of the rule base while others are meant for analysing its dynamic behaviour.

Static analysis involves examining rules as separate symbolic expressions without considering how they interact with other rules or procedures in the inference engine. Dynamic analysis on the other hand, involves the checking of rule interactions during inference, hence the algorithm definitions in the inference engine need to be analysed

for procedural consistency and completeness as well.

Analysis algorithms are most effective if meta-knowledge has been used to define the rule base during its construction. Meta-knowledge makes testing easier since it may replace large groups of rules during testing.

COMMENTS

- The correctness principles are suitable for use on rule bases only.

- The form of the rule base considered in this approach is not the most general. For the rule base to be acceptable for the application of these principles, it has to be specified in a form that meets a set of strict mathematical requirements.

4.3.2.1.2 VALIDATION STANDARDS

The intent of this approach (Harrison and Ratcliffe, 1991) is to provide a basis for standardising the validation of expert or knowledge-based systems. It classifies a knowledge base according to a set of conceptual standards which could define a minimal validation effort.

This approach is centred around two types of standards :

1) Standards for classifying a knowledge base

Standards are needed to classify a knowledge base so that the extent to which it can be validated and the kinds of validation techniques to be applied can be determined.

The classification standard proposed by this approach is based on the degree to which the knowledge base explicitly describe the causal relationship between its components (ie how explicit is the causal structure described).

A rule base, for example, is at the bottom end of this classification continuum because causality is implicit in a collection of rules. In contrast a system that explicitly describes structures and functions for each component as well as causal relations between the components would occupy the opposite end of this classification scale.

The degree to which a knowledge base system can be validated and the kinds of validation techniques that are applicable depend on the position the knowledge base system occupies in this classification continuum. In other words, the classification attempts to define a "minimum" set of validation effort for a given knowledge base system.

2) Standards for the decomposition of validation tasks.

While (1) determines the degree and the kind of validation required, this section is concerned with how the task of validation itself can be accomplished.

To facilitate the task of knowledge base system validation the problem-solving behaviour of the system has to be decomposed into smaller and more manageable units. These units, known as generic tasks, provide the standard for the decomposition of a knowledge base system.

A generic task is a conceptual unit which contains a sequence of conceptually distinct processes with a definable outcome (eg. assemble, plan, identify). These outcomes categorise the generic tasks. For example a generic task may fall into any one of the general categories of 'constructive', 'interpretive' and so on. Associated with the general categories are the validation criteria. The validation criteria are the testable goals which allow these units to be validated.

Thus to validate the behaviour of a knowledge base system, the system may be viewed as the application of a sequence of generic tasks. Validation can then proceed from the individual generic task modules to the complete system in an incremental fashion.

The paper illustrates the application of the approach with two systems which represent models at opposite ends of the classification scale.

The first system, VEG, was implemented as a rule based system with implicit causal relations, while the second, FIRAS, contained explicit descriptions of causality in terms of the underlying structure and functional relationships in the system.

COMMENTS

- the model works only under the assumption that a standardised hybrid shell (eg. KEE) is used to provide the inference engine for the knowledge base.

- Objects in the knowledge base must be represented in a restricted frame-like structure representation.

4.3.2.2 INTEGRATED V&V TOOL SET APPROACH

INTRODUCTION

Several major projects which have the aim of achieving an integrated environment in which the different aspects of V&V, refinement, and evaluation can be analysed and solved are currently under development.

Two such projects are the Expert System Validation Associate (EVA) project, (Chang, Combs, and Stachowitz, 1990), (Landauer, 1990) which is discussed in this section, and the European Esprit II project , VALID (Lopez, Meseguer, and Plaza, 1990) which has the rather similar aim of developing an environment which is generic in order to be applicable to different knowledge base systems.

4.3.2.2.1 THE EXPERT SYSTEM VALIDATION ASSOCIATE PROJECT

The EVA project, under development at the Lockheed Artificial Intelligence Center since 1986, covers a very comprehensive range of V&V techniques and issues. It has the long range goal of developing an integrated set of generic tools to validate any knowledge base system written in any expert system shell.

EVA is made up of many different tools which allow it to perform different verification and validation checks. The ability of these tools to be used on any knowledge base system written in any shell is made possible by EVA's unifying architecture which uses a single user interface and a single meta knowledge base for all its tools.

The basic unifying factor behind EVA is its metalanguage which is common to all its tools. The knowledge engineers use this metalanguage to specify their own validation criteria. These criteria are stored as meta knowledge. The V&V tools may then use the

information in the meta knowledge to validate the application knowledge bases.

For any new shell to use EVA a translator is required. The translator is used to translate the application knowledge and meta knowledge in the shell to the EVA database format.

The following discussion outlines some of EVA's components under the headings of verification tools and validation tools.

EVA'S VERIFICATION TOOLS

i) Structural checker

This tool checks for deadend rules, unreachable conclusions, redundancies and circular rules.

ii) Logic checker

This tool is used for checking inconsistencies in the knowledge base. Such inconsistencies occur when contradictory conclusions can be deduced from the knowledge base.

iii) Semantic checker

This is used for checking for facts which violate the semantic constraints. Such constraints have been defined by the knowledge engineer and stored as meta-knowledge.

iv) Omission checker

This tool checks if there are missing rules or facts in a knowledge base.

v) Model based verifier

This verifier makes use of a 'domain model' to check the content of a knowledge base.

The domain model of an application domain is a database which contains general knowledge derived from textbooks, government regulations, or other publications relevant to the domain. It is generally created without concern for specific expert systems.

When a particular knowledge base has been specified the domain model may be used to verify its contents.

EVA'S VALIDATION TOOLS

i) Test case generator

Test cases are required for evaluating a knowledge base's behaviour, reliability, sensitivity, etc.

Selecting such test cases is not only tedious, but is also error prone and biased. The test case generator overcomes these shortcomings by generating such test cases automatically.

ii) Uncertainty checker

Rules, slots or units may be related to one another, thus their certainty factors must also be related in some consistent way. The purpose of the uncertainty checker is to check whether such related certainty factors are compatible and consistent.

iii) Rule Satisfiability checker

This tool requires the existence of a formal specification expressed in predicate form.

It uses this predicate specification to detect rules which cannot be satisfied by the specifications, or rules which produce facts which violate the specifications. In addition it also identifies data which are satisfied by the specifications but are not covered by the existing rules.

iv) Control checker

The control checker is used for validating the inference engine.

It requires the presence of an explicit set of meta rules which specify the order constraints of the rules. Using the meta rules it is able to validate the rule firing order of the inference engine by comparing the explicit order constraints contained in the meta rules with the implicit order constraint of the rule base.

v) Behaviour verifier

This tool pre-supposes the existence of formal specifications of all the subsystems of the knowledge base system. It uses the component behaviour and interactions

of the subsystems to prove that the collective behaviour of the overall system is correct.

It also produces a formal specification of the total system from the specifications of the component parts.

vi) Rule proposer

This tool uses an existing set of rules or a set of test cases to propose a new set of rules (it does this by induction). The new rules are simplifications and can be used as an aid to help the knowledge engineer make corrections to a rule base.

vii) Rule refiner

Unlike the SEEK knowledge base refiner which unifies validation and rule refinements in the same framework, EVA's refiner is strictly a rule refiner. The refinement process either generalises or makes a rule more specific.

The EVA rule refiner relies on a set of stored test cases. The test cases set contains instantiations of the rules (ie facts).

The rule refiner automatically chooses specific instantiations from the test cases set and applies them to a rule, then interactively seeks the advice of the expert in performing the refinements.

COMMENTS

- Although EVA already contains an impressive range of V&V tools and new tools are continually being designed and implemented each year (Chang et al., 1990), it does not seem to address the problem on how to decide what constitutes an acceptable and reliable method for evaluating the results of tests of an expert system, or what should be considered as a satisfactory level of test.

  Such decisions are still being left to the discretion of the individual domain expert. Leaving such decisions to the human expert is fraught with many dangers. As Green and Keyes pointed out "the human expert may be prejudiced or parochial" or "the expert may not be independent when independent evaluation is needed" (Green & Keyes, 1990, p. 445), or worse there may be no expert available.

4.3.2.3 OTHER VALIDATION TECHNIQUES

4.3.2.3.1 TEST CASES

Test cases are useful for validating a knowledge base's functionality. Such a validation is accomplished by empirically testing the correctness of the conclusions derived from the knowledge base. Such a test, however, only partially meets the users' intentions because it is not possible to test exhaustively, and test results depend on how well the choice of test cases has been selected.

An advantage of the use of test cases is that it makes possible the automation of the validation process since test cases may be pre-stored in databases. An example of validation using test cases was seen in the SEEK system (section 4.2.3.1).

In the SEEK system, test cases have to be collected manually from the experts. Vignollet and Ayel (1991) developed a method for automatically building sets of test samples for knowledge bases. Such a generator has already been implemented for zero order propositional logic. In their paper, (Vignollet & Ayel, 1991) they discussed the implementation of this method using first order logic.

4.3.2.3.2 EXPLANATION

Apart from validating the system's derived conclusions, the system's reasoning also needs to be validated. That is, the reasons for arriving at certain conclusions have to be justified.

Explanations are used for validating the system's reasoning. They are what Hoppe called "the inspectable justification of the system behaviour" (Hoppe, 1990, p. 163).

Tsai and Zualkernan (1990, p. 133) said that explanations "can be considered analogous to inspection and walk through in conventional software testing".

Since inspection is not a formal technique, we might conclude from the above that explanation is a useful validation technique, albeit an informal one.

4.3.2.3.3 DESIGN TECHNIQUES THAT AID VALIDATION

How a knowledge base has been built affects not only how easily it can be maintained, but also how easily it can be validated. As Landauer (1990, p. 297) pointed out "it is more important to have principles to support prospective V&V (building expert systems properly in the first place) than retrospective V&V".

One way to support this concept is through the use of meta-knowledge in place of domain problem solving methods and control structures wherever possible. Meta-knowledge makes such implicit controls and problem solving methods explicit hence easier to understand and validate.

4.3,2,3,4 MAKING USE OF CONVENTIONAL SYSTEM TESTING STRATEGY

Tsai and Zualkernan (1990) proposed a unified framework for testing expert systems. The framework may be used to evaluate the applicability and effectiveness of a testing method in the context of an expert system.

Since many conventional testing methods exist, it makes sense to look at whether these methods can be adapted to knowledge base testing. Tsai and Zualkernan's framework permits the evaluation of conventional testing methods by indicating which are candidates for migration to the expert system environment.

CHAPTER 5

CONCLUSION

5.1 SUMMARY

The thesis has outlined two general sets of methodologies and tools. The first set comprises methodologies which have been designed for the construction of more maintainable knowledge bases, while the second set comprises methodologies which facilitate the process of maintaining knowledge bases (a chart of this taxonomic classification appears in the APPENDIX).

The taxonomic chart classifies maintenance tools and techniques under the headings "methodologies for building maintainable knowledge bases" and "methodologies for maintaining existing knowledge bases". Certain methods may appear at more than one places in the hierarchy. For instance, 'structured techniques' is classified under 'methodologies for building maintainable knowledge bases' and also under 'methodologies which aid the process of knowledge base understanding before modification'.

In order to facilitate quick referencing, each entry in the classification carries a parenthesised section number which corresponds to the section in the thesis in which it was described.

5.1.1 BUILDING MAINTAINABLE KNOWLEDGE BASES

Knowledge engineers contemplating building expert system knowledge bases will find the section on Software Engineering techniques (section 3.1) generally applicable as an aid for constructing more maintainable knowledge bases. The structured techniques tips given by Penderson (section 3.1.5) are simple in concept and can easily be adapted to most knowledge bases. These techniques can be used alongside the modular concept techniques (section 3.1.2) to reinforce the structuredness of the knowledge base.

With regard to the question of how best to modularise a knowledge base, the Knowledge Flow Model technique (section 3.1.2.2) provides a simple option, namely, partition the knowledge base according to the 'application techniques' which make up that application. The COMPASS solution (section 3.1.2.3) on the other hand, suggests that the knowledge base should be partitioned by following the natural modularity of the expert's knowledge.

Whichever way one may have chosen to partition the knowledge base, the Interface Specification technique (section 3.1.2.1) could still be applied to enhance the 'structuredness' of the knowledge base. Interface Specification is a rather general method applicable to a wide range of knowledge bases. Its main emphasis is on limiting the amount of information flow between the modules (hence reducing the effects of changes within the modules) and formally specifying the information flow between modules (hence making the function of modules easy to understand).

The modular concept may be implemented in one of two ways. One may implement the various modules within a single knowledge base, or alternatively, each module may be implemented as a separate knowledge base. The COMPASS system (section 3.1.2.3) uses this "multiple knowledge bases" concept to implement the various knowledge modules. However, one has to be sure that the shell supports such an implementation before embarking on it.

The ability of a system to provide automatic cross-referencing and documentation of knowledge, easy browsing and multiple views of knowledge undoubtedly aids the maintenance process. Such features are provided by a conventional data dictionary. Jansen and Compton (1988) adapted the data dictionary concept to the building of knowledge bases. They used the relational data model as the underlying storage representation for the knowledge to gain the full advantage of relational calculus for the manipulation of the knowledge.

It appears that a further benefit may be derived from the storing of knowledge in this neutral relational data model. A problem with the integration of knowledge bases is their lack of compatibility when these knowledge bases are represented in different formalisms. The storing of knowledge in this neutral intermediate relational data form would facilitate their transformation from one form to another.

A well known software engineering message is that 'if a single fact is found in only a single location, then the job of maintaining it is significantly reduced'. This is the rationale behind the principle of normalisation. A tool which is based on the principle of normalisation is the Knowledge Analyst's Assistant (KAA) (section 3.1.5). The tool interactively guides a user during the modification of the knowledge base. A prerequisite to the use of this tool is that the knowledge base must be normalised. This might alternatively be viewed as a disadvantage of this tool since normalisation of a knowledge base may be perceived to be 'unnatural'. Unlike the normalisation of facts, the normalisation of rules may remove their heuristic values.

While the concept of reusability is sound (because reusable modules not only save work but are also easier to maintain, since their functions are known), in practice, reusability does not find wide application in knowledge base constructions. The reasons on why this is the case were briefly discussed in section 3.1.6.1.

OTHER APPROACHES

Apart from software engineering, there is a host of other innovative tools and ideas on how to build a more maintainable knowledge base. Due to space and time limitations, only four different approaches were selected for discussion.

In section 3.1.1 a case was strongly put up against the software engineering concept of rigorous definition on the grounds that it is not possible to pre-define an application before its construction. Section 3.2.1 discussed a method (due to Slagle et al.) which gainfully accommodates the concept of rigorous definition into the construction of knowledge bases. This technique should therefore be of interest to knowledge engineers who find it difficult to break away from their entrenched rigorous definition view.

For intrepid knowledge engineers who wish to make a total departure from software engineering principles, the 'knowledge-in-context' strategy (section 3.2.2) may be recommended as an alternative and novel way of building knowledge bases that are easy to maintain. The knowledge base built using this method directly reflects the thought processes of a human expert. However such a knowledge base may be rather difficult to read or comprehend since it casts aside all software engineering principles of structuredness and modularity.

The third and fourth approaches are based on the use of tools and hence are not as generally applicable as the techniques described above. The former (ie third approach) is based on the belief that explicit structures are easier to maintain than implicit ones. RIME (Soloway et al., 1988) is a language based on this concept. The latter argues that knowledge bases built using declarative languages are easier to understand and therefore to maintain than those that are built using procedural languages.

5.1.2 MAINTAINING EXISTING KNOWLEDGE BASES

This section mainly discussed tools (as opposed to general techniques) which are aimed at easing the actual process of maintaining the knowledge base.

The process of maintenance is viewed by the current author as being made up of three stages. The first stage is concerned with the understanding of the knowledge base before modifications can be made. The second stage is the actual modification itself, while the third is the validation of the knowledge base to ensure it remains correct and consistent after the modification.

The various tools and methodologies were discussed in the context of these three stages.

5.1.2.1 KNOWLEDGE BASE UNDERSTANDING

To aid understanding good explanation is required. To provide good explanation a system should 'understand' itself. The Explainable Expert System (section 4.1.1) concept takes a first step at creating an expert system that can understand itself. Such a system can explain not just what it is doing, but also why it is doing it.

Apart from good explanation, another aid to knowledge base understanding is the readability of the knowledge base. This can be best achieved through the building of desirable features like structuredness, modularity, coupled with good documentation, and the adherence to the principles of standardisation during the building of the knowledge base.

In line with the discussion on techniques for building an understandable knowledge base, the current author felt that two conventional techniques may be worth consideration. The first is the 'automatic program understanding' tool (section 4.1.2.1) which was designed primarily for the deciphering of conventional programs. The second technique is based in the Knowledge Base Software Engineering (KBSE) concept (section 4.1.2.2). Since code is the obstacle to understanding an obvious solution would be to eliminate it. This is precisely what the KBSE concept sets out to do. The KBSE strategy enhances understanding by removing the code at the user level altogether. Rather than modifying the code, this strategy calls for modifications to be done on the specification. The code is then rederived from the specification.

Other aids to understanding include the use of explicit control structures to promote the homogeneity and predictability of the knowledge bases and also the use of more declarative languages.

5.1.2.2 FACILITATING THE ACTUAL MODIFICATION PROCESS

To ease the actual modification process, interactive tools which can intelligently guide or advise a maintainer are required. TEIRESIAS (section 4.2.1.1) or KAA (section 3.1.4.1) attempt to take the role of a knowledge engineer by providing guidance and advice to the expert during the modification process.

An interactive classifier aids modification by automatically determining where a newly described concept should be placed in the knowledge base, then verifying its decision with the user.

Interactive refinement tools, like SEEK (section 4.2.3.1), allow the users to interactively experiment with changes by testing these changes against stored test cases before incorporating them permanently into the knowledge base. Refinement tools, however, as the name suggests, are only useful when the knowledge base is already generally correct, and only refinement (ie fine tuning) is required. They cannot be used for making major changes, like structural changes for instance. This 'minimal change' assumption is what Lopez et al. referred to as an instance of the 'parsimony criterion' - a situation whereby if different actions are possible to achieve the same result, it is wiser to choose the most simple change (Lopez et al., 1990, p. 65).

Other interactive modification tools mentioned in the thesis included knowledge base editors and knowledge acquisition tools.

5.1.2.3 ENSURING CORRECTNESS OF THE KNOWLEDGE BASE AFTER MODIFICATION

For the sake of completeness this section (which is dedicated to V&V tools) is included. Strictly V&V tools cannot be thought of as maintenance tools. They are, nevertheless, essential for ensuring the correctness of the knowledge base after modification has been carried out.

Some tools, like the interactive tools discussed above (TEIRESIAS, KAA, Interactive Classifiers and SEEK) combine modification and validation into the same framework. In each case the newly entered knowledge is checked against the existing knowledge base in some way and then verified with the user before that knowledge is permanently added.

Section 4.3 looked at a different approach, one in which the knowledge base is modified as a distinct step (this could be done through the use of an unintelligent knowledge base editor or a knowledge acquisition tool). This is then followed by validation as another separate step to ensure that the knowledge base is correct.

Since generally two types of checking need to be carried out, namely verification and validation, this section discusses two sets of tools, verification tools (section 4.3.1) and validation tools (section 4.3.2).

Verification checks relate to checks which prove the knowledge base is structurally correct with respect to a formal specification, while validation checks are concerned with checking whether the knowledge base satisfies the need for which it was created.

(a) VERIFICATION TOOLS AND TECHNIQUES

The ONCOCIN Rule Checker (Suwa et al., 1984) was discussed since it was the basis upon which several other checkers (eg. CHECK (Perkins et al., 1989), ARC (Nguyen, 1988), SPACE SEARCH method (Tsang et al., 1988)) were modelled.

ONCOCIN (section 4.3.1.1) checks a rule base for conflicts, redundancies, subsumptions and omissions. CHECK (section 4.3.1.2) expands on ONCOCIN to include checking for unnecessary ifs, deadend ifs, deadend goals, unreachable conditions, unreferenced parameter values, illegal parameter values and circular rules. ARC is a further extension of CHECK to include checks for compound conditions, subsumed rule chains, redundant rule chains and conflicting rule chains. The SPACE SEARCH method (section 4.3.1.3.1) is an attempt to overcome ONCOCIN and CHECKs' deficiencies of only detecting superficial inconsistencies. It also removes some of the false warnings of inconsistencies produced by ONCOCIN.

Another method mentioned in this section was the Predicate/Transition Net Method (section 4.3.1.3.2). This method allows for the inclusion of consistency and completeness checks as part of the knowledge acquisition process and thus verification can be carried out in an incremental fashion as the knowledge base is being developed.

The main shortcoming of the above tools is that they all perform only static checks on the knowledge base. In other words, the control structure (ie. the inference engine) is not tested. Though it is important that one include dynamic tests (to test the inference engine) in the testing of a knowledge base, such tests are not discussed here because this thesis is concerned with the maintenance of the knowledge base rather than its control structure.

(b) VALIDATION TOOLS AND TECHNIQUES

Validation tests are necessary in order to inspire confidence in the use of the knowledge base. Normally validation tests follow verification tests.

However, current validation tests and techniques are rather inadequate because as argued in section 4.3.2, the very issue of what constitutes a 'valid knowledge base' itself is unclear. This prompts the current author to raise the question that 'if there is no such a thing as a fully valid knowledge base, can there be a sufficiently valid knowledge base ?'. In other words, is it possible to establish an acceptable minimal level of V&V testing standards ? In trying to answer this question the thesis looked at two approaches which seem to make an attempt to address this problem to some degree. These are the 'correctness principles' approach (section 4.3.2.1.1) which attempts to lay down a set of acceptability principles for rule bases, and the 'validation standards' approach (section 4.3.2.1.2) which seeks to provide a basis for standardising the validation of a knowledge base system.

Other projects like EVA (section 4.3.2.2.1) and VALID (Lopez et al., 1990) take a different path. These two projects are mainly aimed at developing an integrated environment in which the different aspects of V&V, refinement, and evaluation can be analysed and solved. They seek to develop a set of generic tools which are applicable to any knowledge base systems developed in any shells.

Although they contain an impressive array of tools, they do not seem to address the problem of what constitute an acceptable level of tests. This decision is still left to the discretion of the individual domain expert.

5.2 LAST WORD

In bringing this thesis to a close, the following section begins by reflecting on past and current methodologies. This is followed by a contemplation on future directions.

5.2.1 PAST AND CURRENT METHODOLOGIES

When confronted with a new situation, it is human nature to look back at what we already know and to try to use old knowledge to solve new problems. It is therefore not surprising that researchers tended to fall back on structured techniques, modularity, data dictionary, DBMS (and now KBMS), various verification and validation techniques etc in facing these new problems encountered in expert system knowledge base maintenance. These techniques have to various degrees been touched upon in previous chapters.

Not all researchers, however, are falling back on conventional software engineering techniques in their search for better maintenance methodologies or tools. As was seen in previous chapters, other maintenance concepts are continually being proposed by researchers.

This emphasis on methodologies and tools appears rather disturbing in the view of researchers who lie on the far end of the maintenance spectrum. The human factors researchers thought it appropriate that maintainers should be reminded that there exists another side to the maintenance coin - the human side of the maintenance equation. In this closing chapter a thought should be given to the two principles put forward by Overton, a human factors researcher :-

" - Studying maintenance means studying maintainers,

- Maintainability is not a quality of a system alone, but of a system and those who maintain it" (Overton, 1983, p. 53).

5.2.2 FUTURE MAINTENANCE DIRECTIONS

Having explored some past and current maintenance technologies, it seems in order to question what shape future maintenance technology will take. The current author sees two possible directions that such technology could take.

The first is the use of a "meta-expert system" to maintain other expert systems. Since the maintenance of a knowledge base (tracking down of errors, making amendments without upsetting existing rules etc) involves expertise, one might be tempted to ask if a 'knowledge base maintenance expert system' could be built to maintain an expert system knowledge base. Such a system could be used to diagnose the source of errors, correct them and retest the system. It might contain procedures to fix bugs, make changes, modify the knowledge base to include new enhancements or change requirements, then conduct retests of the system.

The second possible direction is the development of self-modifying expert systems. Before an expert system can be self-modifying it must be self-understanding, a capability (as we saw in section 4.1.1) that is increasingly being realised. The current author contends that it should also possess self-validating capability.

Currently many tools and methodologies for the development and maintenance of expert system knowledge bases are borrowed from conventional systems. Such methodologies contain a distinct phase whereby an expert system knowledge base has to be judged valid before it is passed on to the users to be put into operation. By transferring such a concept directly from conventional methods, expert systems are being treated in the same way as conventional systems.

In the validation of conventional systems the user's intent is often clear and can be specified, hence such a distinct validation and transfer over phase may be justified. However, human experts are not judged that way. If expert systems are to emulate the human experts then the question of 'how do experts maintain and validate their own knowledge ?' should be asked.

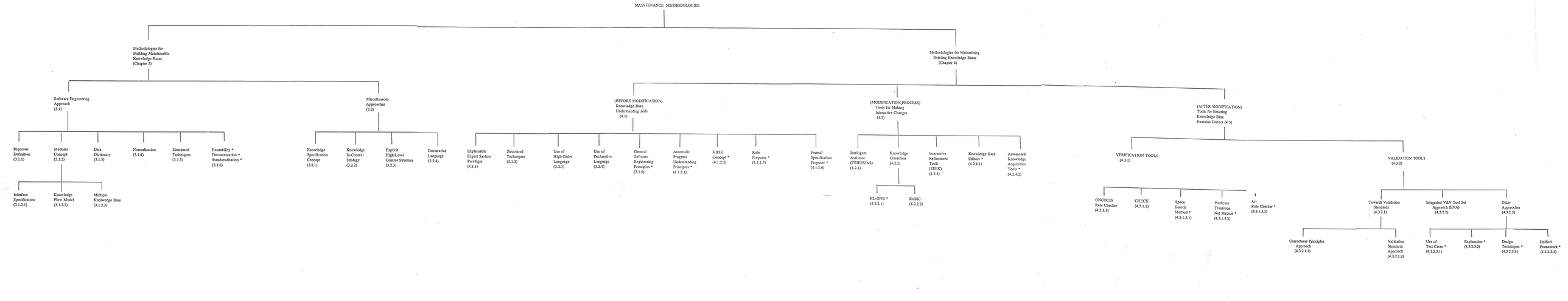
As experts improve they continually correct their own past misjudgment. Should not then validation be made a continuous routine ? Hence, unlike conventional systems, expert systems must necessarily incorporate learning. Without the ability to learn the purported expert system is not very different from a conventional program.

To some extent TEIRESIAS might be thought of as a program which demonstrates such learning capabilities. It is able to validate what it is taught with what it already knows before adding on the new knowledge permanently into its knowledge base.

The CYC project (Lenat & Guha, 1990) provides some guide to answering the question of which direction maintenance technology will head. Lenat and Guha said that "CYC will learn by discovery" and that such learning will be achieved through discussion and education rather than through the "practice of brain surgery upon Cyc's KB" (Lenat & Guha, 1990, p. 357). One would expect that validation will then be just a matter of the educators evaluating CYC (or even CYC evaluating itself since CYC's learning can go on proactively while the machine is idling (Lenat & Guha, 1990, p. 357)) on how significant or reasonable the discoveries it had made were, and correction would just be a matter of re-learning. In this sense validation would be a continuous process, rather akin to the way human experts correct their own errors and misjudgments.

It may appear far-fetched that a self-learning, self-maintaining (ie. re-learning) system could emerge out of the CYC project; but if this does occur then there may no longer be any need for maintenance technologies.

APPENDIX



NOTE

* NOT DISCUSSED IN DETAIL

A TAXONOMY OF KNOWLEDGE BASE MAINTENANCE METHODOLOGIES

BIBLIOGRAPHY

Adelson, B., (1990). Constructs and Phenomena Common to the Semantically-Rich Domains. In P. G. Raeth. (Ed). Expert Systems : A Software Methodology for Modern Applications. (pp 193-204). Los Alamitos, CA : IEEE Computer Society Press.

Arthur, L.J., (1987). Software Evolution : The Software Maintenance Challenge. New York : John Wiley and Sons, Inc.

Ayel, M., (1988). Protocols for Consistency Checking in Expert System Knowledge Bases. In Kodratoff., Y. (Ed). ECAI 88 : Proceedings of the 8th European Conference on Artificial Intelligence. (pp 220-225). London : Pitman Publishing.

Bachant, J., (1988). RIME : Preliminary Work Towards a Knowledge Acquisition Tool. In Marcus (Ed). Automating Knowledge Acquisition For Expert Systems. (pp 201-224). Massachusetts : Kluwer Academic Publishers.

Beinat, P. and Smart, R., (1989). Colossus : Expert Assessor of Third Party Claims. In Proceedings of the Fifth Australian Conference on Applications of Expert Systems. (pp 70-85). Sydney University of Technology.

Benn, W., Schiageter, G. and Wu, X., (1990). Reuse of Persistent Information Between Different Paradigms - A Knowledge Based Approach. In Proceedings : SPIE - Internatinal Society of Opt. Eng. (USA), Application of Artificail Intelligence VIII (Volume 1293). (pp 404-414). Orlando, Florida.

Bennett, K.H., (1991). Automated Support of Software Maintenance. In Information and Software Technology, Vol 33, No. 1. Durham, UK : Butterworth-Heinemann Ltd.

Black, W. J., (1986). Intelligent Knowledge Based Systems : An Introduction. Berkshire, England : Van Nostrand Reinhold (UK) Co. Ltd.

Boar, H. b., (1984). Application Prototyping : A Requirements Definition Strategy for the 80s. New York : John Wiley & Sons, Inc.

Bowerman, R. G., and Glover, D. E., (1988). Putting Expert Systems into Practice. New York : Van Nostrand Reinhold Company Inc.

Brachman, R. J. & Schmolze, J. G., (1989). An Overview of the KL-ONE Knowledge Representation System. In J. Mylopolous & M. Brodie (Eds). Readings in Artificial Intelligence and Databases. (pp. 207 - 229). San Mateo, California : Morgan Kaufmann Publishers, Inc.

Buchanan, B. G., and Smith, R. G., (1989). Fundamentals of Expert Systems. In A. Barr, P. R. Cohen and E. A. Feigenbaum (Eds). The Handbook of Artificial Intelligence (Volume IV). (pp 149-192). Reading, Massachusetts : Addison-Wesley Publishing Company, Inc.

Carrico, M.A., Girard, E.J., and Jones, J.P., (1989). Building Knowledge Systems : Developing and Managing Rule-Based Applications. New York : Intertext Publications.

Ceri, S., Gottlob, G. and Tanca, L., (1990). Surveys in Computer Science : Logic Programming and Databases. Berlin: Springer-Verlag.

Chandrasekaran, B. and Swartout, W., (1991). Explanations in Knowledge Systems : The Role of Explicit Representation of Design knowledge. In IEEE Expert, Volume 6, Number 3, June 1991. Los Alamitos, C.A. : IEEE Computer Society.

Chang, C. L., Combs, J. B., and Stachowitz, (1990). A Report on the Expert Systems Validation Associate (EVA). In Expert Systems With Applications (UK), Volume 1, Number 3, 1990. (pp 217-230). U.K. : Pergamon Press.

Compton, P., and Jansen, M., (1990). Knowledge in Context : A Strategy for Expert System Maintenance. In C. J. Barter and M. J. Brooks (Eds), AI '88 2nd Australian Joint Artificial Intelligence Conference, Adelaide, 1988 Proceedings. (pp 292-305). Berlin : Springer-Verlag.

Davis, J. S., (1990). Effect of Modularity on Maintainability of Rule-Based Systems. In International Journal Man-Machine Studies (UK), Volume 32, Number 4, April, 1990. (pp 439-447). Academic Press Limited.

Davis, R., (1984). Interactive Transfer of Expertise. In Buchanan, B. G. & Shortliffe, E. H., (Eds), Rule-based Expert Systems : The Mycin Experiments of the Stanford Heuristics Programming Project. (pp 171-205). Reading, Massachusetts : Addison-Wesley.

Davis, R., (1988). Interactive Transfer of Expertise : Acquisition of New Inference Rules. In A. Gupta & B. E. Prasad (Eds). Principles of Expert Systems. (pp 243-261). New York: IEEE Press.

Debenham, J. K., (1989). Knowledge Systems Design. Sydney : Prentice Hall.

Debenham, J. K., and Lindley, C. A., (1991). The Knowledge Analyst's Assistant : A Tool for Knowledge Systems Design. In C.P. Tsang (Ed). AI '90 : Proceedings of the 4th Australia Joint Conference on Artificial Intelligence. (pp 343-354). Singapore : World Scientific Publishing Co. Pte. Ltd.

Eshelman, L., (1988). MOLE : A Knowledge Acquisition Tool fot Cover-and-Differentiate Systems. In S. Marcus (Ed). Automating Knowledge Acquisition for Expert Systems. (pp 37-80). Boston : Kluwer Academic Publishers.

Finin, T. W., (1988). Interactive Classification : A Technique for Acquiring and Maintaining Knowledge Bases (Proceedings of the IEEE, October 1986). In A. Gupta & B. E. Prasad (Eds). Principles of Expert Systems. (pp 275-281). New York: IEEE Press.

Ginsberg, A. (1988). Automatic Refinement of Expert System Knowledge Bases. London : Pitman Publishing.

Gorla, N., (1991). Techniques for Application Software Maintenance. In Information and Software Technology. Volume 33, Number 1. Jan-Feb 1991, (pp 65-73). Butterworth Heinemann Ltd.

Green, C.J.R., and Keyes, M.M., (1990). Verification and Validation of Expert Systems. In Raeth, P.G. (Ed), Expert Systems : A Software Methodology for Modern Applications. (pp 444-449). Los Alamitos, California : IEEE Computer Society Press.

Guimaraes, T., (1987). Prototyping : Orchestrating for Success. In Datamation, Dec 1, 1987, (pp 101-106). New York : Cahners Publishing Associates.

Gunderman, R. E., (1988). A Glimpse into Program Maintenance. In G. Parikh (Ed), Techniques of Program and System Maintenance, Second Edition. (pp 55-59). Wellesley, Massachusetts : Q.E.D. Information Sciences, Inc.

Harrison, P. R., and Ratcliffe, P. A., (1991). Towards Standards for the Validation of expert Systems. In Expert Systems With Applications, Volume 2, Number 4, 1991. (pp 251-258). USA : Pergamon Press.

Hetzel, W., (1984). The Complete Guide to Software Testing. Wellesley, Massachusetts : Q.E.D. Information Sciences.

Hicks, R. C., (1990). A Composite Methodology for Low Maintenance Expert Systems Development. In Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, Volume 3. (pp 292-302). Los Alamitos, CA : IEEE Computer Society Press.

Hoppe, T., (1990). Validation of User Intention. In Current Trends in Knowledge Acquisition. (pp 161-172). Amsterdam : IOS.

Irani, E. A., Matts, J. P., Hunter, D. W., Slagle J. R., Kain, R. Y., and Long, J. M., (1990). Automated Assistance for Maintenance of Medical Expert Systems : the POSCH AI Project. In Proceedings of the Third Annual IEEE Symposium on Computer Based Medical Systems. (pp 275-281). Los Alamitos, CA : IEEE Computing Society Press.

Jackson, P., 1986, Introduction to Expert Systems. Wokingham, England : Addison-Wesley Publishing Company.

Jacob, R. J. K., and Froscher, J. N., (1990). A Software Engineering methodology for Rule-Based Systems. In IEEE Transactions on Knowledge and Data Engineering, Volume 2, Number 2, June 1990. (pp 173-189).

Jansen, B., (1988). A Data Dictionary Approach to the Software Engineering of Rule Based Expert Systems. In J.S. Gero and R.Stanton (Eds). Artificial Intelligence Developments and Applications. (pp 101-117). Amsterdam: Elsevier Science Publishers B.V. (North-Holland).

Jansen, B. and Compton, P., (1988). The Knowledge Dictionary : A Relational Tool for Maintenance of Expert Systems. In ICOT (Institute for New Generation Computer Technology) (Ed). Fifth Generation Computer Systems 1988 : Proceedings of the International Conference on Fifth Generation Computer Systems, 1988, Volume 3. (pp 1159-1167). New York : Springer-Verlag.

Jansen, B. and Compton, P., (1989). The Knowledge Dictionary : Storing Different Knowledge Representations. In Proceedings of the Fifth Australian Conference on Applicatins of Expert Systems. (pp 143-162). Sydney : University of Sydney.

Kahn, G., (1988). MORE : From Observing Knowledge Engineers to Automating Knowledge Acquisition. In S. Marcus (Ed). Automating Knowledge Acquisition for Expert Systems. (pp 7-35). Boston : Kluwer Academic Publishers.

Keller, R., (1987). Expert System Technology : Development & Application. New Jersey : Prentice-Hall, Inc.

Kulikowski, C. A., (1989). Knowledge Base Design and Construction : From Prototyping to Refinement. In G. Guida and C. Tasso (Eds). Topics in Expert System Design : Methodologies and Tools. (pp 145-178). Amsterdam : Elsevier Science Publishers S. V.

Landauer, C., (1990). Correctness Principles for Rule-Based Expert Systems. In Expert Systems With Applications (UK), Volume 1, Number 3. (pp 291-316). UK : Pergamon Press.

Lenat, D., and Guha, R., (1990). Building Large Knowledge-Based Systems : Representation and Reference in the CYC Project. Reading, Massachussetts : Addison-Wesley.

Liu, C. C., (1988). A Look at Software Maintenance. In G. Parikh (Ed). Techniques of Program and System Maintenance, Second Edition. (pp 61-71). Wellesley, Massachusetts : QED Information Sciences, Inc.

Liu, N. K. & Dillon, T. (1988). Detection of Consistency and Completeness in Expert Systems using Numerical Petri Nets. In J.S. Gero & R. Stanton (Eds). Artificial Intelligence Developments and Applications. (pp 119-134). Amsterdam : Elsevier Science Publishers B.V. (North-Holland).

Lopez, B., Meseguer, P. and Plaza, E., (1990). Knowledge based Systems validation : A State of the Art. In AI Communications (Netherlands), Volume 3, Number 2, June 1990. (pp 58-72). Netherlands.

Lowry, M. and Duran, R., (1989). Knowledge-Based Software Engineering. In A. Barr, P. R. Cohen and E. A. Feigenbaum (Eds). The Handbook of Artificial Intelligence (Volume IV). (pp 241-322). Reading, Massachusetts : Addison-Wesley Publishing Company, Inc.

MacGregor, R., and Burstein, M. H., (1991). Using a Description Classifier to Enhance Knowledge Representation. In IEEE EXPERT, Volume 6, Number 3, June 1991. (pp 41-46). Los Alamitos, CA : IEEE Computer Society Press.

Marcot, B., (1990). Testing Your Knowledge Base. In P. G. Raeth, (Ed), Expert Systems : A Software Methodology for Modern Applications. (pp 438-443). Los Alamitos, California : IEEE Computer Society Press.

Marcus, S., (1988). Introduction. In S. Marcus (Ed). Automating Knowledge Acquisition for Expert Systems. (pp 1-6). Boston: Kluwer Academic Publishers.

Marcus, S., (1988). SALT : A Knowledge-Acquisition Tool for Propose-and-Revice Systems. In S. Marcus (Ed). Automating Knowledge Acquisition for Expert Systems. (pp 81-123). Boston: Kluwer Academic Publishers.

Martin, J. and McClure, C., (1983). Software Maintenance : The Problem and its Solutions. New Jersey : Prentice-Hall Inc.

Martin, J.P., (1990). The Truth, the Whole Truth, and Nothing but the Truth : An Index Bibliography to the Literature of Truth Maintenance Systems. In AI Magazine, Special Issue, 1990. (pp 7-25)). CA : American Association for Artificial Intelligence.

Matthews, M.H., (1990). Maintenance and Language Choice. In Raeth, P.G. (Ed), Expert Systems : A Software Methodology for Modern Applications. (pp 430-437). Los Alamitos, California : IEEE Computer Society Press.

Mays, E., Lanka, S., Dionne, B., and Weida, R., (1990). A Persistant Store for Large Shared Knowledge Bases. In Proceedings : The Sixth Conference on Artificail Intelligence Applications. (Volume I). (pp 169-175). Los Alamitos, California : IEEE Computer Society Press.

McGraw, K., and Harbison-Briggs, K., (1989). Knowledge Acquisition : Principles and Guidelines. New Jersey : Prentice-Hall Inc.

Miller, L. A., (1990). Dynamic Testing of Knowledge Bases Using the Heuristic Testing Approach. In Expert Systems With Applications, Volume 1, Number 3, 1990. (pp 249-269). USA : Pergamon Press.

Nau, D. S., (1988). Expert Computer Systems. In A. Gupta & B. E. Prasad (Eds). Principles of Expert Systems. (pp 53-74). New York: IEEE Press.

Neches, R., Swartout, W. R. and Moore, J., (1988). Enhanced Maintenance and Explanation of Expert Systems through Explicit Models of their Development (IEEE Workshop on Principles of Knowledge-based Systems, December 1984). In A. Gupta & B. E. Prasad (Eds). Principles of Expert Systems. (pp 283-293). New York: IEEE Press.

Nguyen, T. A., (1988). Verifying Consistency of Production Systems (Proceedings of the IEEE Third Conference on Artificial Intelligence applications, February 1987). In A. Gupta & B. E. Prasad (Eds). Principles of Expert Systems. (pp 294-298). New York: IEEE Press.

O'Leary, D. E., (1990). Verification and Validation of Expert Systems. In Proceedings : The Sixth Conference on Artificial Intelligence Applications (Volume II). (pp 40-41). Los Alamitos, CA : IEEE Computer Society Press.

O'Leary, T. J., Goul, M., Moffitt, K. E., and Radwan, A. E., (1990). Validating Expert Systems. In IEEE Expert (USA), Volume 5, Number 3, June 1990. (pp 51-58). Los Alamitos, CA : IEEE Computer Society Press.

Overton, R. K., (1983). Research Toward Ways of Improving Software Maintenance : RICASM Final Report. In G. Parikh and N. Zvegintzov (Eds). Tutorial on Software Maintenance. (pp 47-53). Silver Spring, USA : IEEE Computer Society Press.

Parikh, G. (1983). Sex and Software Maintenance : The Taboo Topics. In G. Parikh (Ed). Techniques of Program and System Maintenance, Second Edition. (pp 33-38). Wellesley, Massachusetts : Q.E.D. Information Sciences, Inc.

Parikh, G. (1988). Software Maintenance : Penny Wise, Program Foolish. In G. Parikh (Ed). Techniques of Program and System Maintenance, Second Edition. (pp 13-19). Wellesley, Massachusetts : Q.E.D. Information Sciences, Inc.

Parikh, G. (1988). The World of Software Maintenance. In G. Parikh (Ed). Techniques of Program and System Maintenance, Second Edition. (pp 22-25). Wellesley, Massachusetts : Q.E.D. Information Sciences, Inc.

Parsaye, K., and Chignell, M., (1988). Expert Systems for Experts. New York : John Wiley and Sons, Inc.

Partridge, D., (1986). Artificial Intelligence : Applications in the Future of Software Engineering. West Sussex : Ellis Horwood Limited.

Payne, E. C., (1991). A Modular Knowledge-Flow Model. In AI Expert. Volume 6, Number 5, May, 1991. (pp 36-41). San Francisco : Miller Freeman Publications.

Penderson, K., (1989). Well-Structured Knowledge Bases. In AI Expert, Volume 4, Number 4, April 1989. (pp 44-55). San Francisco : Miller Freeman Publications.

Perkins, W.A., Laffey, T.J., Pecora, D., and Nguyen, T.A. (1989). Knowledge Base Verification. In Guida, G. and Tasso, C. (Eds). Topics in Expert System Design : Methodologies and Tools. (pp 353-376). Amsterdam : Elsevier Science Publishers S. V.

Politakis, P. G., (1985). Empirical Analysis for Expert Systems. Massachusetts : Pitman Publishing, Inc.

Politakis, P.G., and Weiss, S.M., (1988). Using Empirical Analysis to Refine Expert System Knowledge Bases. In A. Gupta and B. E. Prasad (Eds). Principles of Expert Systems. (pp 262-274). New York: IEEE Press.

Prerau, D.S., (1990). Developing and Managing Expert Systems : Proven Techniques for Business and Industry. Reading, Massachusetts : Addison-Wesley Publishing Company.

Prerau, D. S., Gunderson, A. S., Reinke, R. E., and Alder, M. R., (1990). Maintainability Techniques in Developing Large Expert Systems. In IEEE EXPERT, Volume 5, Number 3, June 1990. (pp 71-79). Los Alamitos, CA : IEEE Computer Society Press.

Ribar, G., Arcoleo, F. and Hollo, D., (1991). Loan Probe : Testing a Big Expert System. In AI Expert, Volume 6, Number 5, May 1991. (pp 43-49). San Francisco : Miller Freeman Publications.

Rolston, D.W., (1988). Principles of Artificial Intelligence and Expert Systems Development. New York : McGraw-Hill Inc.

Sacerdoti, E. D., (1991). Managing System. In AI Expert, Volume 6, Number 5, May 1991. (pp 46-33). San Francisco : Miller Freeman Publications.

Slagle, J. R., Gardiner, D. A. and Han K., (1990). Knowledge Specification of an Expert System. In IEEE EXPERT, Volume 5, Number 4, August 1990. (pp 29-37). Los Alamitos, CA : IEEE Computer Society Press.

Soloway, E. Bachant, J. and Jensen, K., (1987). Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a Very Large Rule-base. In Proceedings AAAI-87 : Sixth National Conference on Artificial Intelligence. (pp 824-829). California: Morgan Kaufmann Publishers, Inc.

Stonebraker, M. & Hearst, M. (1989). Future Trends in Expert Data Ba.e Systems. In Kerschberg, L. (Ed). Expert Database Systems : Proceedings from the Second International Conference. (pp 3-20). California: The Benjamin Cummings Publishing Company, Inc.

Stonehocker, N. M., (1988). Managing the Monster - Taking a Stand for Standards. In G. Parikh (Ed). Techniques of Program and System Maintenance, Second Edition. (pp 292-293). Wellesley, Massachusetts : QED Information Sciences, Inc.

Suwa, M., Scott, A.C., Shortliffe, E.H., (1984). In Buchanan, B. G. & Shortliffe, E. H., (Eds), Rule-based Expert Systems : the Mycin Experiments of the Stanford Heuristics Programming Project. (159-170). Reading, Massachusetts : Addison-Wesley.

Swartout, W. and Paris, C., (1991). Explanations in Knowledge Systems : Design for Explainable Expert Systems. In IEEE Expert, Volume 6, Number 3, June 1991. (pp 58-64). Los Alamitos : IEEE Computer Society Press.

Terveen, L. G., Wroblewski, D. A., and Tighe, S. N., (1991). Intelligent Assistance Through Collaborative Manipulation. (pp 9-14). In J. Mylopoulos and R. Reiter, (Eds), 12th International Joint Conference on Artificial Intelligence, Volume 1, Darling Harbour, Sydney, Australia, August 1991. Sydney : Morgan Kaufmann, Inc.

Tsai, W. T., and Zualkernan, I. A., (1990). Towards a Unified framework for testing Expert Systems. In SEKE '90 Proceedings : Software Engineering and Knowledge Engineering, 2nd International Conference, IL. , USA, June 1990. (pp 127-134). IL., USA : Skokie.

Tsang, W. W., (1988). A Space Searching Method for Checking the Consistency and Completeness of a Rulebase. In Proceedings : International computer Science Conference '88. Artificial Intelligence Theory and Applications, Hong Kong. (pp 575-579). Hong Kong : IEEE Computer Society Press.

Vignollet, L., and Ayel, M., (1991). A Model for Testing Knowledge Bases. In (pp 104-109). In SEKE '90 Proceedings : Software Engineering and Knowledge Engineering, 2nd International Conference, IL. , USA, June 1990. (pp 104-109). IL., USA : Skokie.

Vitalari, N. P., (1984). A Critical Accessment of Structured Analysis Methods : A Psychological Perspective. In T. M. A. Bemelmans (Ed). Information Systems Development for Organisational Effectiveness. (pp 421-431). Amsterdam : Elsevier Science Publishers B. V.

Walker, A., (1987). Expert Systems in Prolog. In Walker, A., (Ed), McCord, M., Sowa, J. F. & Wilson, W. G., A Logical Approach to Expert Systems and Natural Language Processing : Knowledge systems and Prolog. (pp 219-290). Massachusetts: Addison-Wesley Publishing Company, Inc.

Walker, A., Kowalski, B., Lenat, D., Soloway, E., and Stonebraker, M., (1988). In Kerschberg, L. (Ed). Expert Database Systems : Proceedings from the Second International Conference. (pp 63-69). California: The Benjamin Cummings Publishing Company, Inc.

Wilkins, D. C., (1989). Knowledge Base Refinement Using Apprenticeship Learning Techniques. In K. Morik (Ed), Knowledge Representation and Organization in Machine Learning. (pp 247-257). Berlin : Spriger-Verlag.

Wood, T. W., and Frankowski, E. N., (1990). Verification of Rule-Based Expert Systems. In Expert Systems With Applications, Volume 1, Number 3, 1990. (pp 317-322). USA : Pergamon Press.

Woods, W. A., (1990). Important Issues in Knowledge Representation. In Raeth, P.G. (Ed), Expert Systems : A Software Methodology for Modern Applications. (pp 180-192). Los Alamitos, California : IEEE Computer Society Press.

Zhang, D., and Nguyen, D., (1989). A Technique for Knowledge Base Verification. In IEEE International Workshop on Tools for Artificial Intelligence Architectures, Languages and Algorithms. (pp 399-406). Los Alamitos, CA. : IEEE Computer Society Press.