PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Knowledge Base Modification

by

Paul Hunter, B.Sc.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department Systems and Computer Engineering
Carleton University

July 1991

© copyright

1991, Paul Hunter

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-70929-4

Canada

The undersigned hereby recommend to the Faculty of Graduate

Studies and Research acceptance of the thesis,

"Knowledge Base Modification"

submitted by PAUL HUNTER, B.Sc., in partial fulfillment of

the requirements for the degree of Master of Engineering

A handwritten signature in black ink, reading "Richard Pearce", written over a horizontal line.

Thesis Co-Supervisor

A handwritten signature in black ink, reading "B. Pagnuk", written over a horizontal line.

Thesis Co-Supervisor

A handwritten signature in black ink, written over a horizontal line.

Chairman,
Department of Systems and
Computer Engineering

Department Systems and Computer Engineering.

Faculty of Engineering

Carleton University

ABSTRACT

This thesis presents an original method for automating modifications of the recommendations of a rule based expert system. This method is referred to as AMOK (Automatic Modification Of Knowledge). AMOK changes the rules to modify the expert system's recommendations. This allows a user of AMOK to tailor or correct the expert system's output without having to understand the details of the expert system's design or internal knowledge representation. Key parts of AMOK have been prototyped to demonstrate the feasibility of its implementation and to investigate factors influencing its performance.

The user of AMOK simply corrects the output produced by the expert system to the desired values corresponding to a given input. AMOK computes the changes to the knowledge base by making modifications to the rules. This improves and contrasts with previous approaches, where a user had to directly add, change, or remove rules to modify the expert system's output.

ACKNOWLEDGEMENTS

I would like to thank my supervisors, Dr. D. Peacocke and Professor B. Pagurek for their help, encouragement, and ideas. I wish to acknowledge the computing resources provided by my employer Bell-Northern Research.

I would also like to express my gratitude and love to my wife, Fiona, and my two children, Stephanie and Carolyn. I would have never been able to undertake this work without their support and understanding.

TABLE OF CONTENTS

Abstract.....	iii
Acknowledgements	iv
Table Of Contents	v
List of Figures	ix
List of Tables.....	xi
Glossary.....	xii
1 Introduction.....	1
1.1 Expert Systems	1
1.2 Motivation.....	7
1.3 Modification or Acquisition.....	9
1.4 Thesis Summary	12
1.5 Thesis Structure	13
1.6 Notation	13
1.6.1 IBM's VM-Prolog	14
1.6.2 Text Notation.....	15
2 Background.....	16
2.1 Historical Overview.....	16
2.2 TEIRESIAS.....	17
2.2.1 Meta-knowledge	19
2.3 Meta-language.....	20

2.4	Pairwise Rule Interaction.....	21
2.5	Certainty Factors.....	23
2.6	Rule Set Interactions.....	26
2.7	Theory Revision and Knowledge Base Reduction.....	28
2.8	Summary.....	30
3	System Description.....	32
3.1	Problem Statement.....	33
3.2	Terminology.....	34
3.3	Overview.....	36
3.4	Main Example.....	38
3.5	Named Nodes.....	39
3.6	Identification of the change required.....	43
3.7	Identification of impacted nodes.....	45
3.8	Reducing hypotheses to disjunctive normal form.....	47
3.9	Incorporation of the change identified.....	49
3.9.1	Deletion.....	50
3.9.2	Addition.....	50
3.10	Simplification of the modified normal form.....	51
3.11	Reconstruction of the knowledge base.....	54
3.11.1	Overview of Reconstruction.....	54
3.11.2	Reconstruction Options.....	56
3.11.3	Reformatting.....	59
3.11.4	Substitution of a Named Node.....	60
3.11.5	Syntax Simplification.....	64
3.11.6	Algebraic Simplification.....	65

3.11.7	Automatic Theorem Proving.....	66
3.11.8	Logical Equivalence.....	66
3.11.9	Graph Search.....	67
3.11.10	General Graph Search Methods.....	70
3.11.11	Search Termination.....	72
3.12	Comparison with Rule Changes.....	77
3.12.1	Generalization.....	78
3.12.2	Conflict.....	79
3.12.3	Redundancy.....	81
3.12.4	Specialization.....	82
3.13	System Summary	83
4	Implementation	84
4.1	Reversible Prolog	85
4.1.1	Predicates in Prolog.....	86
4.1.2	Ground Terms.....	87
4.1.3	Reversibility.....	88
4.1.3.1	Pure Prolog.....	91
4.1.3.2	Real Prolog	95
4.2	Identification of change required	99
4.3	Identification of impacted nodes	100
4.4	Reducing hypotheses to disjunctive normal form.....	100
4.5	Simplification & incorporation of the change.....	101
4.6	Reconstruction of the knowledge base.....	102
4.6.1	Computation of coverage and dependency	103
4.6.2	Expression Equality.....	109

4.6.3	Expression Transformation.....	114
4.6.4	Graph Search.....	117
4.7	Specimens.....	122
4.7.1	Example 1	123
4.7.2	Example 2	124
4.7.3	Example 3	124
4.7.4	Example 4	125
4.7.5	Example 5	126
4.8	Implementation Limitations.....	127
4.9	Implementation Summary	128
5	Conclusions.....	129
5.1	AMOK'S Performance.....	131
5.2	Extending AMOK.....	134
5.3	Further Applications	134
	References.....	137

LIST OF FIGURES

Figure 1 - Expert System Overview.....	3
Figure 2 - Expert System Development Cycle.....	8
Figure 3 - Implementation Overview	37
Figure 4 - Knowledge Base Example.....	39
Figure 5 - Named Nodes.....	41
Figure 6 - User Input.....	45
Figure 7 - Determining Impact.....	46
Figure 8 - Incorporating Change.....	49
Figure 9 - Simplification of Change.....	53
Figure 10 - Dependency and Coverage.....	62
Figure 11 - Expression Transformation.....	68
Figure 12 - Local Minimum.....	73
Figure 13 - Searching for Minimums.....	73
Figure 14 - Search Depth.....	75
Figure 15 - Reconstruction of Knowledge Base.....	76
Figure 16 - Rule Change Example.....	78
Figure 17 - Rule Change, Generalization.....	79
Figure 18 - Rule Change, Conflict.....	81
Figure 19 - Karnaugh MAP for h4.....	105

Figure 20 - Karnaugh MAP for h_4'	107
Figure 21 - Graph Search Example.	121

LIST OF TABLES

Table 1 - Hypothesis and Named nodes.....	41
Table 2 - Table of manual rule reconstruction.....	55
Table 3 - Table of coverage and dependency.	64
Table 4 - Table of Complexity Factor.	75
Table 4 - DNFs of simple expressions.....	101
Table 5 - Performance of SAME predicate.....	113
Table 6 - Performance of REPLACE predicate.	116
Table 7 - Performance of MOVE predicate.....	119

GLOSSARY

expert system. A computer system that achieves a high level of performance in problem areas that normally require, for humans, education, training and years of practice.

AMOK. Automatic Modification Of Knowledge, the system described in this thesis.

disjunctive normal form. A Logical expression written as the disjunction of a set of conjunctive literals. It shows clearly the smallest set of input values that can cause a hypothesis to be concluded.

DNF. See disjunctive normal form.

fact. A proposition whose validity is accepted.

forward-chaining. A control procedure used in expert systems. New facts are recursively derived from the currently believed facts.

heuristic. A procedural tip or incomplete method for performing a task.

hypothesis. A conclusion of the expert system, which follows from the consequent proposition of a rule.

inference engine. The control structure of an expert system.

knowledge acquisition. The extraction and formulation of knowledge from an expert.

knowledge modification. The adjustment of some relationships within the defined structure of the knowledge base.

knowledge base. The repository of knowledge in a expert system. In this thesis it is used to describe a structure that contains a collection of rules representing the knowledge of an expert.

knowledge base reconstruction. Converting a knowledge base written in DNF to the original rule form, preserving as much as the original knowledge contained as possible.

learning. The process of improving performance by changing knowledge or control.

meta-knowledge. Knowledge about knowledge.

meta-rule. A rule that describes how rules should be used, or derived.

minimal upper bound. The lowest common point in the rule lattice of the nodes affected by the change of a rule.

MUB. See minimal upper bound.

named nodes. Labeled logical expressions written as rules are called named nodes. They are used in rules to help define concepts, and may also be output as a hypothesis to the user.

rule. A pair that consists of an antecedent condition and a consequent proposition, which can support deductive processes such as forward-chaining.

shell. An expert system is normally constructed from several pre-designed software components, for example, an inference engine, rule language and knowledge base. These components are called the expert system shell.

unnamed nodes. The expressions used in rules that have not been defined as named nodes.

1 INTRODUCTION

1.1 Expert Systems

Expert systems are a field of Artificial Intelligence which studies methods to mimic the problem solving skills of human experts. These systems are successful in domains where it is easy to identify the bounds of the problem, and where human experts exist. A typical expert system can do all or most of the following tasks:

- Solve or help to solve problems which would otherwise require the services of a human expert.
- Integrate new knowledge into the existing knowledge base.
- Display knowledge in a form which is reasonably easy for people to read and understand.
- Provide explanations of its recommendations and inference sequence.
- Reason with inexact knowledge.

To support these functions, an expert system is normally designed out of several building blocks. These building blocks are called the expert system 'shell'. The following is a list of the typical components of a shell:

- A knowledge base to contain rules and facts.
- An inference engine for reasoning with the rules and facts.
- An explanation generator for explaining a recommendation.
- Methods for acquiring new knowledge and encoding it into the knowledge base.

The shell is the frame to which the knowledge must be added. It is much like a spread sheet, the general structure has been defined and the user simply enters the data. In a spread sheet the data consists of the numbers and formulas to be used. In an expert system the data is the knowledge to be used by the inference engine, and is expressed as rules. Determining these rules is a non-trivial job.

This thesis concentrates on the modification of knowledge which is in an expert system's knowledge base. I present a method for automating modifications of a rule based expert system's knowledge base, called **AMOK** (Automatic Modification Of Knowledge). A simplified expert system is shown in Figure 1, with some components of interest. Although a typical knowledge base is encoded in a more understandable form than conventional programs, it will still have complex notations which an untrained user could not understand.

The implementation of the inference engine can lead to further complications affecting the way the knowledge is used. For example, in some systems rules are used from the top down, whereas in others the rules are used from the bottom up. This can influence the output of the system.

Figure 1 - Expert System Overview.

Expert systems have developed passed the stage of limited deployment trial systems with small knowledge bases. Expert systems are now designed using rule bases of many hundreds of rules (e.g., Kahn87 & Ginsberg88a). In such large systems it is important to adopt sound engineering practices. Yet, this in itself does not allow modifications to be made easily. Rule interactions, such as described in Chapter 2, can cause complex side effects.

The growth in size and complexity of knowledge based systems is partly due to the increase in power of general purpose small personal computers. Since the early expert systems, such as MYCIN, were developed in 1976

(Shortliffe76), there have been large reductions in the cost of memory and processor power. Other factors increasing the size of knowledge bases are:

- The development of general expert system shells. These allow users to take advantage of a pre-designed inference engine and knowledge representation language. This allows more effort to be spent on development of the knowledge base.
- More experience in the use of expert systems. Building on this experience helps in the design of more complex problems requiring greater knowledge.

Reasons to separate the knowledge base from the inference engine include software management, design simplification and flexibility. Knowledge bases implemented using a formal logic system as the knowledge representation language, have, in principle, an added advantage. The development of tools to help maintain the knowledge base may use the already large theory base derived from the analysis of these logic systems. This may provide a sound base for tool development.

A major problem for any tool builder is that there is no consensus on the knowledge representation language or logic which should be used in the development of knowledge bases. Some systems are based on predicate calculus, Horn clauses, propositional logic or a custom designed language. This unfortunately hampers the reuse of research and software.

Research continues in the area of knowledge representation, examining ideas such as frames or objects. Various non-standard logics are also being used to investigate issues in knowledge representation. These are logics other than classical propositional logic or predicate calculus. They include multi-valued, intuitionist, modal, and temporal logics. Few actual knowledge bases have been built with these logics, so standard logic is of more interest in this thesis. Still, the problem of knowledge modification must be addressed, regardless of the knowledge representation language used. Thus some ideas investigated in this thesis may have a wider application than presented here. In this thesis I will be using propositional logic to express the rules of the knowledge base.

Propositional logic allows rules similar to the following to be written. The English version of the rule is shown first, followed by the propositional logic form:

- If you have a head pain, then you have a headache.

head_pain -> headache

- If you have a high temperature or you are sweaty, then you have a fever.

high_temperature | sweaty -> fever

- If you have a fever and a headache, then take Aspirin.

fever & headache -> Aspirin

The knowledge base will contain a set of rule like these. This would allow questions like "What medicine should be taken for a head pain and a high temperature" (*head_pain & high_temperature*), to be answered with "*Aspirin*". This could be done by trying each rule in turn to see if the antecedents of the rule are true. When the hypothesis of a rule is concluded, then the rule is said to have fired.

If *head_pain* and *high_temperature* are true and these are input by the user, the first pass of the expert system will allow the following rules to fire:

head_pain -> *headache*
high_temperature | *sweaty* -> *fever*

This concludes that *fever* and *headache* are now true. In the second pass, the rule:

fever & headache -> *Aspirin*

will fire and therefore *Aspirin* will be concluded.

This method of problem solving is called forward chaining. This is in contrast to a backward chaining approach, where the goal of the search is used to generate subgoals which are either true themselves, or can be subdivided into further subgoals. This process of reducing the goals to subgoals continues until the subgoals are either true or cannot be proven.

Current approaches to knowledge acquisition and modification can be classified in several ways. These range from the traditional software engineering approach of imposing structure on knowledge, to approaches

inspired by artificial intelligence and psychology, such as those used in machine learning. These approaches can also be directed at different system users, such as an expert human knowledge engineer or an untrained human. Several systems which have been previously developed are reviewed in Chapter 2.

1.2 Motivation

The motivation for this work comes from problems encountered by developers, maintainers and end users of expert systems. The problems of knowledge modification occur at many points during the design and deployment cycle of a knowledge based system. A simplified view of the development cycle of an expert system is given in Figure 2 to illustrate the issues.

Figure 2 - Expert System Development Cycle.

As illustrated, this is not a linear process, cycles and rewrites occur as the system develops. These types of development processes and cycles have been identified in the development of many deployed expert systems. Loberg (Loberg88) describes experience in the design of a diagnostic expert system in telecommunications.

Towards the end of the development cycle, the knowledge base is tuned and adjusted to give the desired results. AMOK can aid in this modification process by allowing an end user, who need not be the initial knowledge base designer, but may be a domain expert, to make modifications, even after the expert system has been deployed.

1.3 Modification or Acquisition

We can help to clarify the distinction between **Knowledge Acquisition** and **Knowledge Modification** by referring to Ginsberg (Ginsberg79). In this paper, Ginsberg identifies two stages in the development of knowledge bases. These stages are, in summary:

- 1 Initial rough knowledge base extraction**, where the performance is not comparable with that of the experts.
- 2 Knowledge base refinement**, where the initial knowledge base is progressively refined into a system which performs at a high level.

These stages are characterized by:

- 1 Addition of rules**, until complete sets of rules relating to various hypotheses have been included.
- 2 Tuning of these rule sets** by the addition, removal, or modification of rules in the rule base, during a testing and debugging phase.

It can normally be assumed that at least the first stage of the above activities will (at least at initially) be undertaken by someone with detailed knowledge of the rule structures, and of the inference shell. This is also the area of knowledge acquisition that research in machine learning primarily addresses.

The second stage of development, described above, may continue. After a system has been developed and tested, there will be additional reasons to change the knowledge base. This may occur for a number of reasons, some of which are listed below.

- The understanding of the domain will continue to improve. In most expert fields, the knowledge evolves over a period, and the system must be flexible enough to incorporate this revised knowledge.
- The domain itself can change over time. For example, in a system which configures computer hardware, new or improved computer hardware modules are being developed continually. Information on how to configure these new hardware modules will need to be added.
- Local methods may dictate different solutions. In local situations alternative solutions may be preferred over the normally recommended solutions, or other solutions which are not normally considered possible¹ may be considered.

¹ Local Methods are required to allow for special circumstances that may exist in the location where a particular system is being deployed. For example, in areas of the world different versions of drugs may be available. These will require changes in prescriptions and dosages.

- **Errors found in the initial system's recommendations need to be corrected. These errors are likely to be small in number, as the system was considered correct when it was released.**

This stage of knowledge base refinement is typically carried out by designers who were not the original system designers. The type of cycles identified during the development stage will also be found during the knowledge base refinement stage. It is not expected that at the knowledge base refinement stage the system design would require change or that new concepts would need to be introduced.

To summarize the distinction, knowledge acquisition is the building of structure in a knowledge base to synthesize an initial base level of knowledge. This would include the definition of the internal modules, nodes and dependencies. Knowledge modification is limited to the adjustment of some relationships within the defined structure of the knowledge base.

Machine learning is also a large research area that closely relates to knowledge acquisition. Failure-based learning in particular appears to be strongly related. However, much of this work relates to the identification of gaps in the knowledge, and in trying to determine if the knowledge base is complete. The focus of this thesis is the interactions between rules.

1.4 Thesis Summary

The development, deployment and maintenance of any computing application which uses a knowledge base can be aided by a tool which automates the process of knowledge base modification. This thesis presents a method for automating the process of modification of a knowledge base written in propositional logic. This is done without requiring the user to be aware of either the details of the knowledge base, or the implementation of the expert system. This is valuable, because up to now anyone wishing to change the output of the system, would need proficiency in knowledge base design to be able to add, modify or delete rules from the knowledge base.

The key contributions of this thesis are:

- Proposing a design of a method for automating modifications to an expert system's knowledge base, in a system called AMOK. A prototype of the key aspects of this method is implemented in Prolog.
- Showing how this can be achieved by building on two of the earlier works in the area of knowledge base management described in Chapter 2.
- Discussion of the importance of labels for concepts, and how these relate to reassembly of a knowledge base.

- Presenting a local minimum directed graph search method to perform the rule reassembly process.
- Studying the use of reversible predicates in Prolog. Showing the benefits of code reuse and the trade off in execution performance.

1.5 Thesis Structure

This chapter provided an overview of expert systems, a discussion of the difference between knowledge modification and acquisition, the motivation for developing a knowledge modification system, and a summary of the scope of the thesis. A historical overview of related work is provided in Chapter 2.

Chapter 3 describes the modification method proposed by this thesis and relates it to the work in Chapter 2. Chapter 4 discusses implementation of AMOK and the results obtained, while Chapter 5 reviews the conclusions of the thesis, and suggests further areas for continuation of this work.

1.6 Notation

The notation used in the thesis is based on VM-Prolog running under CMS on an IBM Mainframe. Selecting one representation to be used both in the Prolog code and the text of the thesis, allows it to be used consistently.

1.6.1 IBM's VM-Prolog

VM-Prolog is IBM's Mainframe implementation of the Prolog language. The following list notes details which may be different to other common versions of Prolog.

- Variables are represented by `*` as the leading character.
- Unnamed variables are represented by `*`.
- Lists are represented by curly braces `{}`.
- List elements are written as, `{a,b,c}`.
- The head and tail of lists are represented by `{*head ! *tail}`.
- The logical symbols are:
 - 'and' is `&`,
 - 'or' is `/`,
 - 'not' is `^`.
- Predicates are represented by `A <- B & C.`, meaning that *A* is implied by *B* and *C*.
- Goal queries are input as `<- B.`, meaning try and prove *B*.
- Cut is represented by the slash symbol `/`.

1.6.2 Text Notation

The text, Prolog code and logical symbols are kept separate as far as possible. The following rules apply.

- Prolog code and computer interactions are printed in a different font, which maintains equal spacing for all characters. This preserves the look of computer output.
- When either code or logical expressions are included in the body of the text they are written in italics.

2 BACKGROUND

This chapter contains a historical review of earlier research into knowledge acquisition and modification. This provides a context for the research presented in the later chapters.

2.1 Historical Overview

A historical overview shows how the key concepts in the field of knowledge modification have developed.

- 1979 Davis' system named TEIRESIAS, (Davis79).
- 1982 Bowen and Kowalski's paper looking at Meta-language in Logic Programming, (Bowen82).
- 1982 Suwa, Scott and Shortliffe's paper examining pairwise rule interactions, (Suwa82).
- 1986 Wilkin and Buchanan's paper dealing with rules with certainty factors, (Wilkins86).
- 1987 Raatz and Drastal's paper examining interactions between sets of rules, (Raatz87)

1988 Ginsberg's papers on knowledge base reduction examining inconsistency and redundancy (Ginsberg88a), and theory revision (Ginsberg88b).

The remainder of this chapter now describes each of these works in detail, concentrating on the aspects relevant to this thesis.

2.2 TEIRESIAS

The TEIRESIAS program was designed to provide assistance to the builder of knowledge-based systems (Davis79). This activity focused on transferring knowledge from the human expert to the system. It was initially developed to work with the MYCIN medical diagnostic expert system. Its main function was to enlarge and correct the knowledge base which MYCIN used. MYCIN's basic control structure, representation language and knowledge base were already in place.

Before the development of TEIRESIAS, debugging of a knowledge base was performed at several levels in different expert systems. Examples of these debuggers are:

- High level language debuggers available to the programmers of the computer. For example, if the language selected to develop the expert system is LISP, the trace feature of most LISP interpreters can be used to help debug the expert system.

- Source level tracing facilities to display the critical execution of the inference engine. These could display, for example, the rule stack which contains the list of rules waiting to be tried.

None of these methods would be of use to a domain expert attempting to modify, add, or correct a knowledge base. It is unlikely that they would have the programming expertise necessary to use these tools. This would slow down communication between the expert and the knowledge engineer trying to add the rules to the knowledge base.

In TEIRESIAS, modification of a knowledge base takes place by identifying a mis-diagnosed case. A rule trace can be examined to decide which rules fired, which ones did not, and why. It is assumed that the domain expert has a high degree of knowledge of the operation of the inference engine and the knowledge representation language used.

TEIRESIAS would then aid the expert to formulate changes to existing rules or to add new rules to the knowledge base. This is done by using meta-knowledge. TEIRESIAS does not deal with the problem of determining if new or modified rules will introduce inconsistency into the knowledge base or cause other errors in diagnostics. It only deals with a single error case at one time.

2.2.1 Meta-knowledge

A key aspect of the TEIRESIAS program is the investigation of the use of meta-level knowledge to both guide a restrictive natural language interface, and to aid the expert in rule formation. This meta-knowledge allows TEIRESIAS to look at its own internal knowledge and use this knowledge to guide the expert modifying the knowledge base.

Meta-level knowledge is gained by the system, by analysis of its own knowledge base. This meta-knowledge is at a level above the normal knowledge in the data base, and can be stored as a rule about manipulating rules. It is knowledge about knowledge. Examples of meta-knowledge are meta-rules, which provide information on typical rule forms in the knowledge base, and also determine which rules should be tested before others.

The restrictive natural language interface is based on a keyword pattern match, using statistical analysis of the rule patterns of the knowledge base. Davis argues that there are patterns to the way an expert will reason about a topic, and these patterns can be formalized into **rule models**. Rule models are generated by performing a statistical analysis to find common patterns in the existing knowledge base.

These rule models are then used to guide the keyword pattern match. They are templates which keywords are fitted to. It is still possible that this will lead to an interpretation that the expert may reject. In that case, heuristics are used to guide the search for the next rule model to be tried.

The second application of the meta-knowledge in TEIRESIAS is in rule formation. TEIRESIAS uses rule models to provide helpful hints to the expert. These hints will inform the expert that the current rule being considered for addition to the knowledge base is missing some information typical of rules fitting this model.

2.3 Meta-language

The concept of meta-level knowledge can be extended to form the concept of meta-language. Bowen and Kowalski discuss meta-language in logic programming (Bowen82), where the central motivation is that of managing a logic program's database of clauses². It gives the designer not only the ability to examine the knowledge contained in the system, but also provides an extension to the object level language which allows logic programs to express concepts at this meta-level.

In database management, meta-language descriptions can be written to perform the process of assimilation of new input into the current knowledge base. Using meta-level predicates like *DEMO* we can define new ones such as *ASSIMILATE*.

² There may be little difference in principle between a logic program's database of clauses, and an expert system's rule base. In practice the rules can be of any chosen representation.

DEMO (GOAL, PROGRAM) allows the internal interpreter to be called to decide if a *GOAL* is provable from a given *PROGRAM* (a set of clauses). It is decided that the goal is provable if it can be shown to be true by the logic program's interpreter.

ASSIMILATE (CURRENT_DB, INPUT, NEW_DB) can then be defined in terms of *DEMO* to determine if the knowledge to be added was already implied by the database or is inconsistent with the existing database.

This idea of meta-language can be extended to allow many different and flexible meta-level programs to be written. This will allow fast implementation of heuristics where the meta-level concepts are made explicit. It may be possible to execute the meta-level program using the same interpreter as the object level language.

2.4 Pairwise Rule Interaction

In Davis's, and Bowen and Kowalski's papers, described previously, it was understood that pairs of rules can interact to produce undesired results. This problem was examined in detail in two papers by Suwa, Scott and Shortliffe; and Nguyen, Perkins, Laffey, Pecora (Suwa82 & Nguyen85). In these papers two types of interactions are described relating to knowledge base design. The first type results from a conflict between the rules. The second type of interactions are due to internal limits of the particular system which is being

studied³. The second type of interactions are of less interest, as they depend on implementation details of individual programs.

There are three types of interactions found by both Suwa and Nguyen which can be identified during pairwise comparison of the rules in a knowledge base. These are:

- **Conflict**, where two rules will succeed in the same situation but give conflicting results. For example:

$p(X) \rightarrow q(X).$

$p(X) \rightarrow \text{not}(q(X)).$

Where X is a variable.

- **Redundancy**, where two rules can succeed in the same situation and give the same results. A simple example of this would be a duplicate rule, where the variables have been renamed:

$p(X) \rightarrow q(X).$

$p(Y) \rightarrow q(Y).$

³ An example of this type of rule interaction would be mutual recursion of rules in a system that is unable to handle recursive rules.

Where X & Y are variables.

- **Subsumption**, where two rules give the same result, but one contains additional restrictions on situations where it will succeed.

$p(X) \& q(X) \rightarrow r(X)$.

$p(Y) \rightarrow r(Y)$.

Where X & Y are variables. In this example, the first rule is subsumed by the second rule. This is because the first rule will only fire when the second rule fires, and they both conclude the same result.

These interactions are not all logical errors, but they may affect the effectiveness of the data base. In general, only conflicts are guaranteed to be errors. Redundancy will have no effect except to reduce the efficiency of the inferences, and subsumption may be common in knowledge bases which contain certainty factors.

2.5 Certainty Factors

Wilkins and Buchanan address the issue of rule modification for correcting conclusions in a rule base, which uses rule certainty factors (Wilkins86). The basic approach of this work differs from TEIRESIAS, in that it is based on reducing a rule base to produce the lowest total errors, when the

system is run against an input training set⁴. Wilkins also identifies that in TEIRESIAS, correcting an error in one test run can cause another to fail. This would go undetected in a TEIRESIAS type of system, which considers only a single problem at a time.

Wilkins identifies three causes of error in rule sets which contain certainty factors. These are missing rules, wrong rules, and harmful interactions between good rules. It is the harmful interactions which are examined in depth, the other cases being similar to those identified by Suwa and Nguyen, discussed above.

Wilkins' paper presents a view of what a **good rule** should look like. The **goodness** of a rule is defined in terms of four measurable quantities, these are:

Generality	measures the number of training cases that the rule covers.
Specificity	measures a rule's discrimination level.
Collinearity	measures the rule's similarity to the training cases.
Simplicity	measures number of clauses in the rule.

For a rule to be considered a good rule, we require:

- 1 High generality, where a rule must cover several training cases.

⁴ Errors are defined as false positive or false negative identified in the training set.

- 2 High specificity, where a rule must meet a given discrimination level.
- 3 Low colinearity, where rules must not be too similar to the training cases.
- 4 High simplicity, where a maximum number of clauses in the rule is specified.

The method described in the paper produces a subset of an induced good rule set generated by training instances. As this method generates a subset, it implies that the rule base is only modified by removing rules. This subset is generated to minimize the number of mis-classified cases from the training instances. The rule base is also seen has having interactions which are not simple pairwise interactions between rule pairs, so a global approach is used to compute the modifications to the knowledge base.

A heuristic method, called the Antidote Algorithm, computes the rule subset with the least errors when the test cases are run through the system. The weighting can be adjusted to give different values of false positive and false negative errors, depending on the domain. For example, in a medical system a false negative can have more serious consequences than a false positive.⁵

⁵ It may be better to falsely alert a doctor of the possibility of disease (false positive) rather than have diseases overlooked (false negative).

The improvements identified while running the training set will yield a general improvement. This is shown experimentally by running a second set of validation cases through the old and new systems. The improved performance of the modified knowledge base is demonstrated.

2.6 Rule Set Interactions

Raatz and Drastal show that pairwise rule comparison cannot detect all interactions between rules, even when a simple representation language is used (Raatz87). For certain interactions, sets of rules need to be examined when modifications are being considered. For example, consider the following rules:

$f1 \mid f2 \rightarrow h1.$

$h1 \ \& \ f3 \rightarrow h2.$

$h2 \ \& \ f4 \rightarrow h3.$

and consider adding the rule:

$f3 \ \& \ f1 \rightarrow h3.$

This rule clearly adds to the methods by which $h3$ can be proved. Evidently, in order for $h3$ to be true when $f1$ & $f3$ were true, $f4$ had to be true. With this change, the value of $f4$ is not important. This leaves some segment of the original rule set redundant, and in effect interacts with all three rules above.

Yet this interaction would be missed by a simple pairwise comparison of the rules.

These types of interactions are detected by analyzing the knowledge base using lattice theory⁶. The addition of a new rule can create a new rule chain with a frontier that interacts with existing frontiers⁷. It is shown that rule set frontiers interaction take place up to the **minimal upper bound** (MUB) of the affected nodes of the added rule. The MUB is the lowest common point in the rule lattice of the nodes affected by the change. If we take the example above the MUB of all nodes is *h3*. If we were to add a new rule:

f5 & f1 -> h9.

Then the MUB for *f5* is *h9*. The MUB for *f1* now contains two nodes, *h9* & *h3*. We can limit analysis of a rule base up to the MUBs the nodes affected. This is because the minimal upper bound identify a lowest hypothesis in the rule lattice where all the effects of the change can be observed.

Types of interactions between the rule sets are defined as specialization, generalization, conflict and redundancy. Of particular interest are interactions

⁶ Rule bases are not true lattice structures, but a number of similarities exist.

⁷ A frontier of a rule chain is a expanded formula . This represents one way in which evidence can be combined to establish the conclusion of the rule.

between the rule **sets** which closely relate to interactions described between rule **pairs**. These are:

- **Specialization** of a rule set is related to subsumption of rule pairs.
- **Generalization** of a rule set is the inverse of specialization.
- Rule set **conflict** is the similar to rule pair conflict.
- **Redundancy** in a rule set is the same as redundancy in rule pairs.

2.7 Theory Revision and Knowledge Base Reduction

Ginsberg addresses the topic of theory (knowledge base) revision, which uses a set of empirical results in the domain to guide revision together with the information implicit in the theory (Ginsberg88b). The work undertaken by Ginsberg applies to knowledge bases written in propositional logic. The implicit information in the theory is the existing rules and structure of the theory. This leads to an approach of tailoring the theory to fit the empirical results using conservation strategies. The tailoring of the theory could be the generalizing of a rule that is the closest to being satisfied in a given case.

Theory revision as described is a three step process, theory reduction, theory modification, and theory retranslation. This work is fundamentally important to the remainder of the thesis as the reduction of a knowledge base to

a disjunctive normal form is a key component of the design of AMOK, and the basic steps of AMOK are similar to those Ginsberg describes.

Theory reduction is also used to detect inconsistencies and redundancies in knowledge bases (Ginsberg88a), where the issue of interaction of sets of rules is examined. The two key steps to reduce sets of rules are:

The level of each rule is computed to determine rule dependence.

During the reduction phase, a note is kept of the cycle the rule was reduced at. Cyclic relationships, which may cause loops during rule evaluation, can therefore be determined.

The knowledge base is reduced into a minimal disjunctive normal form.

This process expands the rules into a normal form, starting with rules which contain only base facts (literals) as antecedents. For example the rule:

$f1 \mid f2 \rightarrow h1.$

These rules are at the first level of the knowledge base. This process is repeated for the next level of rules (second level) which use only first level rules and literals, and continues until the complete rule base has been reduced. It is during this phase that interaction between the disjunctives is detected and reported to the user.

In principle, complexity in such a system is of concern. Up to 3^n disjunctives may be generated (for a 4 level knowledge base), where n is the number of rules at each level. Clearly, this may make such a system unusable with a large knowledge base. In practice, Ginsberg determined that the complexity of a sample knowledge base containing up to 370 rules was almost prohibitive. A TI Explorer II⁸ running on this knowledge base took 10 hours to fully reduce it to 35,000 disjunctives.

Theory refinement is a five phase process. Massive generalization and specialization is followed by focused generalization and specialization, and if still required a final "solving all problems" phase is undertaken. These phases are all based on a statistical analysis of the knowledge base, using a key principle that when solving a generalization [specialization] problem do not introduce a new specialization [generalization] problem.

Theory retranslation is unfortunately not reported in either of these papers. It is a topic that is further investigated in this thesis.

2.8 Summary

Some important research related to the problems investigated in this thesis have been examined. It should be noted that none of the systems above attempted to allow untrained users of the expert system to make modifications to

⁸Explorer is a trademark of Texas Instruments Inc.

the output of the system. This thesis will show how some of these ideas can be combined to result in a system which supports automatic knowledge base modification.

3 SYSTEM DESCRIPTION

In Chapter 1, the concepts of an expert system shell and knowledge acquisition were discussed, and the importance of the modification phase was emphasized. In Chapter 2 we followed the evolution of knowledge modification from manual, rule based TEIRESIAS, to automated systems looking at rule sets, by empirical methods, as described by Wilkins. Finally the work of Raatz, Drastal, and Ginsberg looking at tools to analyze complex rule interactions between rule sets was reviewed.

This chapter presents an original method for modifying knowledge bases by further developing the concepts and methods documented by Raatz, Drastal, and Ginsberg. A prototype implementation of the method, called **AMOK** (Automatic Modification Of Knowledge) is discussed in Chapter 4, where design, implementation, and performance issues are also elaborated.

Raatz and Drastal outline a method in which rule interaction between **rule sets** can be determined. It is shown that the **minimal upper bound** must be examined to compute all the interactions introduced by a new rule. Ginsberg describes an approach of **knowledge base reduction**, which is used to compute rule inconsistency and redundancy. Building on these two results, this thesis shows how to change a reduced knowledge base, how the

knowledge base is rebuilt, how to restore the structure of the knowledge, and how to compute and resolve the interactions introduced. The importance of links between the nodes in the knowledge representation is illustrated and used to reassemble the knowledge base.

3.1 Problem Statement

If we have a set of input Facts F , and a Knowledge Base KB , such that

$$KB(F) = H,$$

where H is an output set of hypotheses, then there are two types of changes to consider:

- 1 A hypothesis h_j in the output set H is not correct for this given input set. This is a false positive.
- 2 A hypothesis h_j which is not in the output set H was required for this given input set. This is a false negative.

In an expert system which prescribes medicinal drugs, an example of these may be:

- 1 Recommending Valium for the symptoms of a simple headache (false positive).
- 2 Not recommending Aspirin for a simple headache (false negative).

The first problem is to identify the changes which should be made to KB resulting in a modified knowledge base KB' , which will make the correct recommendations. A second problem is to identify the effects that these changes will have on other inputs. This may be caused by interactions between rules. An example of this may occur after correcting the first problem in the previous example. If there exists an illness when Valium should be recommended and one of the symptoms is a headache, will the system still correctly recommend Valium?

The example above implies that there need to be some restrictions on the changes which will be made to the knowledge base. The changes must in some sense be minimal, and preserve the structure of the knowledge base. The idea behind having the restrictions, is to reduce the number of side effects generated by interactions.

The selection of the knowledge representation language has an important impact on the ability to analyze a knowledge base. In general, it is impossible to develop algorithms to decide the completeness of first order predicate logic, and the work of Ginsberg, Raatz and Drastal are in propositional logic. Therefore, for simplicity, the work in this thesis expresses knowledge in rules, which have an equivalent form to propositional logic, as opposed to full first order predicate logic.

3.2 Terminology

We have an initial knowledge base KB , where:

Rules are represented by r_i , where i is an integer, e.g., r_{32} .

Facts are represented by f_i , where i is an integer, e.g., f_{14} and f_{23}

Hypotheses are represented by h_i , where i is an integer, e.g., h_{12} and h_1 .

For example, rule r_{32} which concludes hypothesis h_{12} and uses facts f_{14} and f_{23} and hypothesis h_1 might be written as:

$$h_{12} \leftarrow f_{14} \ \& \ (f_{23} \mid h_1).$$

This rule expresses a relationship between h_{12} , f_{14} , f_3 , and h_1 . The rules, facts and hypothesis can all take the values of true or false. If an fact does not have a defined value, then the closed world assumption is used and its value is assumed to be false. Reducing the rule's antecedent expression into a disjunctive normal form (DNF) results in the rule being rewritten as:

$$h_{12} \leftarrow (f_{14} \ \& \ f_{23}) \mid (f_{14} \ \& \ h_1).$$

DNFs are represented internally in AMOK with inner lists representing conjunctions, and outer lists representing disjunctions:

$$h_{12} = \{\{f_{14}, f_{23}\}, \{f_{14}, h_1\}\}.$$

For a given input set of facts I , to the knowledge base KB , we obtain a set of hypotheses H , i.e., $KB(I) \Rightarrow H$.

3.3 Overview

The overview of AMOK in Figure 3 shows the main steps that are used to solve the problem. The DNF represents the smallest sets of input values that can cause a hypothesis to be concluded. So, dealing with changes to this form allows the change with the minimum impact to the rest of the rule base to be selected.

The main steps in the process which AMOK follows are:

- Identification of the change required.
- Determination of the subset of the knowledge base affected by this change.
- Reduction of this subset of the knowledge base to the disjunctive normal form.
- Incorporation of the change identified into the normal form.
- Simplification of the modified normal form.
- Reconstruction of the knowledge base.

Figure 3 - Implementation Overview

The design of AMOK is closely tied to the design of the expert system shell being used to execute the knowledge base. Both in AMOK and the expert system shell, the 'closed world assumption' is used. If a fact is unknown then it

is assumed to be false, the assumption being that if the fact was true it would have been known. This does not allow the possibility of a value being unknown.

3.4 Main Example

Throughout the remainder of this chapter a simple common main example will be used to help illustrate the ideas and methods developed. This example is introduced in this section.

The knowledge base is assumed to have been developed by an initial system designer and domain expert. We will refer to the knowledge base as *KB*. In this example, *KB* has the following rules.

$h0 \leftarrow f4.$	$r1$
$h1 \leftarrow f1 \mid f2.$	$r2$
$h2 \leftarrow h1 \ \& \ f3.$	$r3$
$h3 \leftarrow h0 \ \& \ (f5 \mid f6).$	$r4$
$h4 \leftarrow h2 \ \& \ h3.$	$r5$
$h5 \leftarrow h3 \ \& \ f7.$	$r6$

Facts:	$f1, f2, f3, f4, f5, f6, f7$
hypotheses:	$h0, h1, h2, h3, h4, h5$
rules:	$r1, r2, r3, r4, r5, r6$

The knowledge base can be represented graphically as in Figure 4 below. Facts and hypotheses are nodes, and the rules define the arcs between the nodes. The figure shows nodes and the arcs joining the nodes. When two or

more arcs join at a node, the logical operator *and* is denoted by a part circle joining the two arcs, *or* lacks this.

Figure 4 - Knowledge Base Example.

3.5 Named Nodes

As shown in Figure 4 above, hypotheses are of two types. *h4* and *h5* form end points in the graph, no rules contain these nodes as antecedents. These nodes may be output from the expert system as advice to its user. The remaining hypotheses, *h0*, *h1*, *h2*, and *h3* are intermediate nodes. They are reused in rules to help define 'higher-level' concepts, and may also be output as advice to the user. I will refer to these intermediate hypotheses as **named nodes**.

A named node is a concept the expert defined when the knowledge base was being built. Since the expert has a name for this concept, it is reasonable to assume that the concept is known, used and considered important, in the

domain of expertise. The naming of this concept, by writing a rule to describe it, provides a method for its reference and use in further rules. This is similar to the concept of subroutines in programming.

Named nodes are important, because changes to the definition of these nodes ripple up to any rules containing these nodes, so users of these nodes do not have to be aware of the detailed definition. They can use the named node as a **handle** to the concept. In contrast, **unnamed nodes** are expressions used in rules. Even if this pattern is repeated in several rules, they are treated as independent entities, so changing one has no effect on the other instances of this pattern.

For example, there may be two ways of writing a rule for *h14*, one using a node *h11*, the other directly using the facts that *h11* is composed of.

1 *h11* <- *f11* | *f12*.

h14 <- *h11* & *f16*.

2 *h14* <- (*f11* | *f12*) & *f16*.

These are shown graphically in Figure 5, below.

Figure 5 - Named Nodes.

In the right hand side of Figure 5, f_{11} / f_{12} is an unnamed node, represented by the •. There is no way to reference this node in another rule. But, if the definition of the named node h_{11} on the left hand side were to change, then this would automatically ripple up to h_{14} , and any other node that uses h_{11} . It is therefore important to keep track of the named nodes used by each hypothesis. The named nodes of the hypotheses in the main example we are using are:

Hypothesis	Named nodes
h_2	h_1
h_3	h_0
h_4	h_2, h_3
h_5	h_3

Table 1 - Hypothesis and Named nodes.

The relationships between named nodes, as defined by the original rule definitions, should remain in place, because the establishment of these links required expert guidance. AMOK defines heuristics to allow these links to be broken when the change to the knowledge base is extensive. These heuristics are based on changes to the coverage and dependency of the rules, and are discussed in further detail in section 3.11.4.

It is also feasible to identify common patterns of expressions amongst rules in the knowledge base. If these patterns represent existing named nodes, then the expression could be replaced by the already defined node. If this expression does not represent a defined named node, then AMOK could conjecture that the pattern represents a concept, which is at present not defined by a named node. For example, a section of a knowledge base contained rules:

$h91 \leftarrow (f91 \mid f92) \ \& \ f93.$

$h92 \leftarrow (f91 \mid f92) \ \& \ (f94 \mid f95 \mid f96).$

$h99 \leftarrow f91 \mid f92 \mid f97.$

Then we could assume that $(f91 \mid f92)$ is a common pattern and may represent a concept. It could then be left to the user to name this node, using a rule defining the pattern (e.g., $h100 \leftarrow (f91 \mid f92)$), which could then replace the expression in existing rules, giving:

$h91 \leftarrow h100 \ \& \ f93.$

$h92 \leftarrow h100 \ \& \ (f94 \mid f95 \mid f96).$

$h99 \leftarrow h100 \mid f97.$

$h100 \leftarrow (f91 \mid f92).$

This idea is not expanded upon further in this thesis. The remainder of this chapter discusses the details of the knowledge modification method outlined earlier.

3.6 Identification of the change required.

Analysis of the user input identifies the change required to the knowledge base. The user identifies the required output to match a given input set. This implies that either a recommendation should be added or discarded for that particular output set. The user is only allowed to add or remove **one** hypothesis in each cycle of AMOK. This avoids the possibility of two changes conflicting with each other. Multiple changes could be performed in sequence by the user, however this may produce different results, depending on the order in which they are done.

Relating back to the main example, if the input facts from the user are:

$I = f1, f3$

Then KB would be run and conclude some hypotheses, in this case:

$KB(I) \Rightarrow h1, h2. \quad \{\text{With all other hypotheses false.}\}$

Now, suppose that the user wishes to modify the knowledge base so that:

$$KB'(I) \Rightarrow h1, h2, h4$$

In other words the user now wishes the knowledge base to conclude *h4* for this given input set of facts, *f1*, *f3*. The relationship between these nodes needs to be changed and therefore will be directly affected by the modification. This is shown by the shading in Figure 6. Nodes *f2*, *h1*, and *h2* are not shaded because the user did not input any modifications relating to these nodes, and as we will see, the remaining nodes are also not shaded because they are not within the scope of the change. How the scope is determined is now discussed in the next section.

Figure 6 - User Input.

3.7 Identification of impacted nodes.

Once the nodes directly affected have been identified, the minimal upper bound (MUB) of the set containing these nodes is computed. The MUB is the highest common point in the rule lattice of the nodes affected by the change. This will define the subset of the knowledge base affected by this change (Raatz87). Any nodes outside this subset can not be altered by any rule changes inside this subset of the knowledge base. In the example, the MUB of the change is node $h4$, and the area of the knowledge base impacted is shaded in Figure 7. The impacted nodes are $f1, f2, f3, f4, f5, f6, h0, h1, h2, h3, h4$.

The MUB identifies the lowest hypothesis in the graph where all the effects of the addition of the new rule can be observed. Thus it indicates where the computation of the DNFs need to begin. Also because the nodes $h5$ and $f7$

are in the shaded part of the graph, they need not be considered any further in the discussion of this example.

Figure 7 - Determining Impact.

There are two reasons for identifying the impacted nodes:

- 1 To identify the limits of the side effects of the change AMOK is about to make. This data can later be used to inform the user about possible undesired side effects that this change may have.
- 2 To define the nodes for which the DNF will need to be generated. This allows the computation of the DNF for a subset of the hypothesis in the knowledge base. Two actions are performed for each hypothesis:
 - i Named nodes, generated by other intermediate rules, will be expanded. This is performed recursively until only input

facts are left in the rules, and identifying all the named nodes this hypothesis depends on. This reduction can be computed starting with the bottom nodes in the knowledge base, as these values may be required by higher nodes. This will save intermediary nodes from being frequently recomputed.

- ii Where more than one rule can conclude a hypothesis, rules are combined by a disjunction. For example, if there are two rules defining *h101*:

h101 <- *f101*.

h101 <- *f102*.

These two rules can be combined into a single rule,

h101 <- *f101* | *f102*.

3.8 Reducing hypotheses to disjunctive normal form.

Two actions may be required to reduce the antecedent of a rule, defining a hypothesis, to the disjunctive normal form (DNF). These are:

- 1 Move all negation inwards in the expression, using logical equivalences such as:

$$\neg(a \& b) = \neg a \mid \neg b$$

$$\neg(a \mid c) = \neg a \& \neg c$$

$$\neg(\neg a) = a$$

- 2 Distribute the disjunctions ('or') over the conjunctions ('and'), using logical equivalences such as:

$$a \& (b \mid c) = (a \& b) \mid (a \& c)$$

This results in an expression containing all the conjunctions inside the disjunctions, for example $((a \& b \& c) \mid (a \& b \& d))$ is in DNF.

In our example, we can now reduce each of the hypotheses to DNF form, giving:

```
h0 = {{f4}}
h1 = {{f1},{f2}}
h2 = {{f1,f3},{f2,f3}}
h3 = {{f4,f5},{f4,f6}}
h4 = {{f1,f3,f4,f5},{f1,f3,f4,f6},{f2,f3,f4,f5},{f2,f3,f4,f6}}
h5 = {{f4,f5,f7},{f4,f6,f7}}
```

Recall that the DNF represents the smallest sets of input values that can cause a hypothesis to be concluded. *h0* will be concluded whenever *f4* is true. *h5* will be concluded when one of the two input conditions are met, either *f4*, *f5*

and f_7 are true or f_4 , f_6 and f_7 are true, and the values of other inputs do not matter.

3.9 Incorporation of the change identified.

The disjunctive form of the hypothesis to be changed is now modified to incorporate the change. This changed hypothesis will either be an addition or a deletion of a hypothesis that is concluded for the given input set. In our example, we wish to increase input cases where h_4 can be concluded. This is shown graphically in Figure 8. This new rule has not yet been fully integrated into the knowledge base.

Figure 8 - Incorporating Change.

As the change is incorporated into the knowledge base we are forced to make some assumptions as to the meaning of the input that was given. This relates to the desire to make the changes to the rules, to be in some sense

minimal, to preserve the structure of the knowledge base. The interpretation of minimal is linked to the assumptions made as a deletion or addition is performed to the knowledge base, these are now detailed.

3.9.1 Deletion

In general the deletion of a hypothesis from the output set is simpler. As the hypothesis was concluded from the input, at least one disjunction in the DNF for this hypothesis must be true in the input set. All such disjunctions are removed. It is possible that the set of input facts given by the user is a superset, containing a disjunction in a rule's DNF. Consider for example, if the user does **not** wish h_4 to be concluded when (f_1, f_2, f_3) are input, and h_4 's DNF is defined as:

$$h_4 = \{\{f_1, f_3\}, \{f_1, f_3, f_4, f_6\}, \{f_2, f_3, f_4, f_5\}, \{f_2, f_3, f_4, f_6\}\}$$

Then the input set is a superset of the disjunction $\{f_1, f_3\}$, and so this disjunction needs to be removed. Because f_2 is not included in the disjunction, its value is of no interest. In other words, $\{f_1, f_3\}$ does not depend on the value of f_2 , nor on any other value not specified in the DNF.

3.9.2 Addition

For addition of a hypothesis, we wish to add a new set of disjunctions to a rule's DNF. Because the DNF represents the smallest sets of input values that can cause a hypothesis to be concluded, the smallest change is to add the input facts as a new disjunction. During the addition of this disjunction, we may find

some conflicts and opportunities to simplify the DNF. The following section explains how these are resolved.

In the example we wish to add a new disjunction to the DNF for $h4$. This disjunction is the set of input facts as provided by the user, here $\{f1, f3\}$.

In our example:

$$h4 = \{\{f1, f3, f4, f5\}, \{f1, f3, f4, f6\}, \{f2, f3, f4, f5\}, \{f2, f3, f4, f6\}\}.$$

Then the DNF for $h4'$ (the modified version of hypothesis $h4$) must contain the disjunctions from the original form, together with the new disjunction. The DNF of $h4'$ will be:

$$h4' = \{\{f1, f3, f4, f5\}, \{f1, f3, f4, f6\}, \{f2, f3, f4, f5\}, \{f2, f3, f4, f6\}, \{f1, f3\}\}$$

3.10 Simplification of the modified normal form.

The disjunctive normal form can now be examined and simplified. This leads to the detection of interactions between disjunctions. At this point, redundant terms can be removed. Information on these interaction may be useful to help explain to a user the action that AMOK took, this has yet to be determined.

Remove has two possibilities:

- 1 To remove $\{f1, f2\}$ from a DNF containing $\{f1, f2, f3\}$ would not change this term, because of the closed world assumption. If $f3$ were true, this would have been known, so the input $\{f1, f2\}$ is

equivalent to $\{f1, f2, \neg f3\}$, and therefore $\{f1, f2, f3\}$ will not be removed from the DNF.

- 2 To remove $\{f1, f2\}$ from $\{\{f1, f3\}, \{f1, f2\}\}$ would result in simplification to $\{\{f1, f3\}\}$. In this case the two terms match exactly and we can safely remove the term from the DNF.

Addition also has two possibilities:

- 1 To add an unrelated disjunction like $\{f1, f2\}$ to a DNF like $\{\{f9, f10, f25\}, \{f52, f22\}\}$, the disjunction is simply appended to the list of disjunctions, e.g., $\{\{f9, f10, f25\}, \{f52, f22\}, \{f1, f2\}\}$.
- 2 To add a generalized disjunction like $\{\{f1, f2\}\}$ to a DNF like $\{\{f1, f2, f3\}, \{f3, f7, f8\}\}$ allows the replacement of the $\{f1, f2, f3\}$ term. This is because if the hypothesis is true due to the value of $\{f1, f2\}$, the old term $\{f1, f2, f3\}$ is redundant.

The effect of addition and removal to the hypothesis' DNF are similar. Both actions can cause the removal of a term or terms, removal directly, and addition by redundancy. Now, going back to our main example, we can see the action we made in the previous section to simply add the term was correct. The DNF of $h4'$ will look like:

$$h4' = \{\{f1, f3, f4, f5\}, \{f1, f3, f4, f6\}, \{f2, f3, f4, f5\}, \{f2, f3, f4, f6\}, \{f1, f3\}\}$$

In this case, the term we were adding, $\{f1, f3\}$ is more generalized than two of the terms in the DNF, $\{f1, f3, f4, f5\}$ and $\{f1, f3, f4, f6\}$. Following the second case of addition shown above, we can simplify the DNF of $h4'$ to give:

$$h4' = \{\{f1, f3\}, \{f2, f3, f4, f5\}, \{f2, f3, f4, f6\}\}$$

It is interesting to consider how this will appear in the figure we have been using to illustrate our main example. We now know what the logical expression is for $h4'$, but how can this be represented? We do not yet know how to express $h4'$ in terms of the named nodes $h2$ and $h3$. We can't, therefore, link up the nodes in the figure.

Figure 9 - Simplification of Change.

3.11 Reconstruction of the knowledge base.

From the disjunctive normal form the knowledge base can be rebuilt, allowing the effect of the change to be examined as a rule change in the knowledge base. This is the most complex task to be undertaken. We now have a modified DNF for the hypothesis, and wish to rewrite it as a rule as **similar as possible to the original rule**.

3.11.1 Overview of Reconstruction

To complete the task of modifying the knowledge base, the DNF of the changed hypothesis must be reconstructed into rule form. When this is done, the knowledge base is in a usable format for the expert system, and the modification to the rule can be communicated to the person making the change. It would be then possible to communicate the side effects of the change by showing the modified rule and comparing it to the old rule.

To examine how a reconstruction method may work, I will first show how a rule could be rewritten manually. This manual process will be the basis for the automatic method developed, and can then be used for comparison with the automatic method's implementation described in Chapter 4. Going back to the main example:

Rewriting $h4' = \{f1, f3\}, \{f2, f3, f4, f5\}, \{f2, f3, f4, f6\}$ gives,

```

=>      h4' <- ((f1 & f3) | (f2 & f3 & f4 & f5) | (f2 & f3 & f4 & f6)).
=>      h4' <- (f1 & f3) | ((f2 & f3 & f4) & (f5 | f6)).
=>      h4' <- f3 & (f1 | (f2 & h3)).
=>      h4' <- f3 & (f1 | f2) & (f1 | h3).
=>      h4' <- h2 & (f1 | h3).

```

Expression for h4'	Function
$\{\{f1, f3\}\{f2, f3, f4, f5\}, \{f2, f3, f4, f6\}\}$	Reformat DNF to rule format
$(f1 \ \& \ f3) \mid ((f2 \ \& \ f3 \ \& \ f4) \ \& \ (f5 \mid f6))$	Replace f4 by h0
$(f1 \ \& \ f3) \mid ((f2 \ \& \ f3 \ \& \ h0) \ \& \ (f5 \mid f6))$	Remove surplus brackets
$(f1 \ \& \ f3) \mid (f2 \ \& \ f3 \ \& \ h0 \ \& \ (f5 \mid f6))$	Replace $(*a \ \& \ *b) \mid (*a \ \& \ *c)$ by $(*a \ \& \ (*b \mid *c))$
$f3 \ \& \ (f1 \mid (f2 \ \& \ f3 \ \& \ h0 \ \& \ (f5 \mid f6)))$	Replace $h0 \ \& \ (f5 \mid f6)$ by h3
$f3 \ \& \ (f1 \mid (f2 \ \& \ h3))$	Replace $(*a \mid (*b \ \& \ *c))$ by $(*a \mid *b) \ \& \ (*a \mid *c)$
$f3 \ \& \ (f1 \mid f2) \ \& \ (f1 \mid h3)$	Replace $f1 \mid f2$ by h1
$f3 \ \& \ h1 \ \& \ (f1 \mid h3)$	Replace $h1 \ \& \ f3$ by h2
$h2 \ \& \ (f1 \mid h3)$	End

Table 2 - Table of manual rule reconstruction.

The table above illustrates how this was accomplished. It shows the expression and the legal moves from one form to the next, but does not explain how the moves were selected. It does not prove the result is the simplest expression possible, nor that it is as similar as possible to the original rule. However, experience and intuition suggests that it is.

Comparing the resultant rule $h4' \leftarrow h2 \ \& \ (f1 \ / \ h3)$ with the original rule $h4 \leftarrow h2 \ \& \ h3$, which could be rewritten as $h4 \leftarrow h2 \ \& \ (f1/h3)$, clearly shows the extent of the rule change in a form which can be communicated to the user. This is the only change to be made to the knowledge base, the other rules remain unchanged.

The remainder of this section explains how this process can be automated. For any given DNF of a hypothesis, the rule form is not unique. The form of the expression may differ in minor detail (e.g. the order of the terms in the expression), which is relatively unimportant. Deciding if the new form of the rule should still contain all, some or none of the named nodes identified during the reduction phase is an important determination. This is then a key phase, where the semantic interpretation of the modified knowledge base takes place. This is because if the relationship between the named nodes and the hypotheses are modified then the meaning of the knowledge base has been changed, and later changes will have different effects.

3.11.2 Reconstruction Options

A number of options were examined to attempt to select the best method of rule reconstruction. They can be categorized:

- 1 **Reversal.** This method would use the trace of the operations that were applied to reduce the rules to the DNF, to guide the reconstruction. These operations would be applied in reverse sequence to reconstruct the rule. This method could be sufficient

if the change to the DNF was minor, like the addition of an unrelated term that did not interact with other terms in the DNF. For example, if the DNF for $h4'$ had a new term $\{h99\}$, then the trace back could be effective in finding the new rule for $h4'$. It is hard to see how it could be made to work effectively if a major change to the DNF had been made.

- 2 **Generate and Test.** This method would use a function to generate possible solutions for the new rule, based on the original rule. This function would try adding and deleting conjunctions and disjunctions to the rule form. The generated solutions would therefore be similar to the original form on which they were based. They would then be tested against the DNF for equivalence. The major disadvantage of this method is that it can be a very indiscriminate approach, and there is no guarantee that a solution would ever be found, unless good functions can be found to generate possible solutions.

- 3 **Rule Expansion.** This is a variation of the "Generate and Test" method. It would use an expanded form of the original rule, containing variables as a template, that could then be solved against the DNF to find values for variables added to the rule. So, for example, the rule for $h4$ is:

$$h4 \leftarrow h2 \ \& \ h3.$$

Therefore, we could add variables $\Omega1$, $\Omega2$, and $\Omega3$ to give:

$$h4 \leftarrow ((h2|\Omega1) \ \& \ (h3|\Omega2)) \ | \ \Omega3.$$

We can then solve to find values for the variables, by comparing with the DNF, to give:

$\Omega_1 = \text{nil.}$
 $\Omega_2 = \text{fl.}$
 $\Omega_3 = \text{nil.}$

giving the solution of the rule for h_4' .

Although this method would overcome some limitations of the Generate and Test method outlined, it would still require a spectrum of different rule expansions to be tried. This would lead back to the problem that there would be almost an endless number of variations of the original rule form to try. Some of the possible options for the given example above are:

$h_4 \leftarrow ((h_2|\Omega_1) | (h_3|\Omega_2)) | \Omega_3.$

$h_4 \leftarrow (\Omega_1 \& (h_3|\Omega_2)) | \Omega_3.$

$h_4 \leftarrow ((h_2|\Omega_1) \& \Omega_2) | \Omega_3.$

$h_4 \leftarrow ((h_2\&\Omega_1) | (h_3\&\Omega_2)) | \Omega_3.$

- 4 **Graph Search.** The final method of the rule reconstruction process examined is Graph Search. Each node in the graph represents a logical expression, for example h_1 / f_3 . The arcs in the graph represent logical equivalency substitution, or replacement of expressions by named nodes. The goal of the search is to then find the expression that is closest to the original rule.

The Graph Search method was selected for implementation for two reasons:

- 1 This method is the only one that is capable of locating solutions to all problems of reassembly. Some of the limitations of the method can be overcome if good search heuristics are found.
- 2 There is an added enhancement possible based on the Reversal method. The named nodes that were identified at this stage can be added to the search criteria to guide the search path.

The implementation of "Graph Search" is further expanded in the following sections.

3.11.3 Reformatting

The internal notation of the list representation of the DNF needs to be changed into the logical representation used in the knowledge base by a simple reformatting operation. This results in an expression which consists of disjunctions of conjunctives.

For example:

$$h21' = \{\{f21, f22\}, \{f21, f23\}\}$$

would be rewritten as:

$$h21' \leftarrow (f21 \ \& \ f22) \mid (f21 \ \& \ f23).$$

The rule is now in a form which could be used in the knowledge base. However, it is not written in a simple compact form, nor in terms of the named nodes which were, perhaps, identified when the DNF was computed for *h21*.

For example:

$$h21' \leftarrow (f21 \& f22) \mid (f21 \& f23).$$

can be simplified into:

$$h21' \leftarrow f21 \& (f22 \mid f23).$$

and assuming that:

$$h20 = f22 \mid f23,$$

and

$h20$ is one of original $h21$'s named nodes

then $h21'$ could be rewritten as:

$$h21' \leftarrow f21 \& h20.$$

Before using a named node which the original rule form was dependent on, AMOK must establish if the modified hypothesis should still be dependent on this named node. For example, should $h21'$ above be written in terms of $h20$ or is the relationship between $h20$ and $h21$ different from that between $h20$ and $h21'$? The importance of this is discussed in Section 3.5. on named nodes. If it is decided to attempt to replace expressions in the rule for a named node this will be attempted during the rule rewriting phase.

3.11.4 Substitution of a Named Node

Deciding if the hypothesis is still related to the named nodes identified during rule reduction, has no one correct answer. Two values have been defined that measure the relationship between named nodes and the hypothesis, determining if a rule defining a hypothesis should still be dependent on a named node. These two values are called the **dependency** and **coverage** of the named node. This is computed both before and after the changes are made to the DNF to reveal if the relationship has strengthened or weakened. These values are computed over the set of input values for the

hypothesis, which will be a superset of the input values of the named node, an example is in section 4.6.1.

Dependency is a measure of how strong the relationship is between the value of the named node and the value of the hypothesis over the range of input values. Its value is the ratio of the number of times the named node is true when the hypothesis is true, divided by the number of times the named node is true for the input set.

Coverage is almost the inverse of this. It measures the relationship between the value of the hypothesis and the value of the named node. Its value is the ratio of the number of times the hypothesis is true when the named node is true, divided by the number of times the hypothesis is true for the given input set.

Figure 10 - Dependency and Coverage.

It may be possible to substitute other named nodes for expressions in the DNF, but we limit substitutions to previously identified name nodes.

Dependency and coverage are **measures** used to decide if the previously identified name nodes are still related to the hypothesis. This work continues to examine the case where a **positive** relationship exists. If the negation of a named node is used in a rule then these relationships would be defined differently.

It is always possible to add a named node to the expression. This could be achieved by including both it and its negation, so as to cancel each other out (e.g. $exp / (h9 \ \& \ ^h9)$ is the same as exp .), or using other, perhaps less obvious, means. However what we are trying to measure is the degree to which the

named nodes are relevant to the hypothesis, and exploring the change to this relationship after the modification.

All previously identified named nodes are tested for coverage and dependency. If they have either the same or greater dependency and coverage after the DNF has been modified, then it is concluded that the relationship should be maintained, and they are added to a list of nodes to be substituted back into the rule form. **The critical factor is the dependency.** If this ratio has decreased it becomes likely that the link between the hypothesis and the named node may have to be broken. The user will be given the option to decide.

Even if the named nodes fail to meet this requirement, it may still be possible to substitute them back into the rule. Substitution of the named nodes can still be attempted, but the user should be warned that the coverage or dependency has decreased. This will give the user the option to remove them from the list of nodes to be substituted. The actual values can be represented in a table of data for each named node under consideration, namely, in the example we are investigating, *h2* and *h3*.

Named Node	Original dependency	New dependency	Original coverage	New coverage
h2	9/24	19/24	9/9	19/19
ratio	0.38	0.79	1	1
h3	9/24	9/24	9/9	9/19
ratio	0.38	0.38	1	0.47

Table 3 - Table of coverage and dependency.

The Table 3 can be used to decide if the named node should be substituted when rewriting the rule. It is shown how this table is computed in section 4.6.1. Here it shows that both the coverage and dependency have either increased or remained the same for *h2*, so we deduce that *h4'* should contain *h2* as a sub-expression. For *h3* the dependency has remained the same, but the coverage has decreased, so the user should be informed.

3.11.5 Syntax Simplification

Simplification is used to change the form of the representation of the rules. The user of AMOK wants to use expressions and rules which can be readily comprehended, whereas inside the system the rules are reduced to disjunctive normal form to ease the computation. As explained above, the method selected to reconstruct the rules is based on graph searching. The basic approach to this problem was designed by looking at some examples of problems commonly found in high school mathematics text books, such as '*simplify the following expression*'. There appear to be three main skills used to solve these types of problems.

- 1 Recognition of applicable rules, which, when applied, will directly simplify the expression.
- 2 Use of a rule that will initially make the expression more complex, but will allow further progress to be made.
- 3 Recognition of when the correct solution has been found.

All three of these aspects have to be addressed in an automated system to solve related problems. "Algebraic Simplification" and "Automatic Theorem Proving" have been studied, and the next sections outline some reported results in these areas.

3.11.6 Algebraic Simplification

Moses (Moses71) summarized some of the work in algebraic simplification. A unique problem is connected with algebraic expression simplification, because there are numbers of equivalent forms of expressions, and it is not obvious which one the user would like to have output. For example, there are equivalence relationships between trigonometric functions and exponential functions with complex arguments. The solution recommended is to reflect the input form of the user.

An additional problem addressed in this thesis is the number of possible simplification rules that could be applied to the graph search at any one point. It is not clear if a sequence started with a particular simplification rule will lead to the best result. In Moses' paper this problem is only indirectly addressed by

allowing the user to select the simplification rules to be invoked and applying local heuristics when expressions match certain forms. No reference is made to the selection strategies necessary if numerous simplification methods are applicable.

3.11.7 Automatic Theorem Proving

Another related field of study is Automatic Theorem Proving (ATP). It has become apparent for some time that as well as using logical inference to combine known theorems and axioms, heuristics also must be employed to guide the search towards the solution. ATP has a known search goal, which can help the system to select rules to apply, or can lead directly to a proof via a method like "proof by contradiction"⁹. Also, because the goal of the search is known, it is easy to decide to halt, as either a proof has been found in a fixed time or it has not.

3.11.8 Logical Equivalence

As well as trying to rewrite the rule using the named nodes, we also try to simplify the rule. This can be done by using many logical equivalency substitutions. These are expressions like:

⁹ Proof by Contradiction assumes the negation of the result and then shows that this leads to a contradiction. This implies the assumption was invalid, and therefore proves the validity of the theorem.

1. $a \& a$ is identical to a .
2. a / a is identical to a .
3. $\text{not}(\text{not}(a))$ is identical to a .
4. $(a \& b) / (a \& c)$ is identical to $a \& (b / c)$.
5. $a / (b \& c)$ is identical to $(a / b) \& (a / c)$.

It should be noted, that as in 5 above, not all equivalencies simplify the expression. In fact, the reverse of each relationship may be specified, so some expressions can be broken up allowing the simplification process to continue. This will cause problems as we search for a solution, as these substitutions develop the search space to expand and become very large. This point is explored further in Section 3.11.11.

3.11.9 Graph Search

The final step of the rule rewriting process is a directed graph search problem. Each node in the graph represents a logical expression, for example $h1 / f3$. The directed arcs in the graph represent logical equivalency substitutions or replacement of expressions by named nodes, both actions are called **transformations**.

Figure 11 shows an example of a segment of a graph. The nodes in the graph are represented by circles with their value shown beside the circle. The

transformations (arcs) are shown by arrows, with the transformation indicated beside the arrow.

Figure 11 - Expression Transformation.

The arcs are directed because the transformations between the nodes are one way operations. In general, the inverse transformation exists for each arc because all nodes are logically equivalent. If the inverse transformation is defined in the program, then this is represented by an arrow with two heads. However, because the goal is to simplify the expression not all the possible inverse transformations are defined.

The object of the search is to reduce the expression to the simplest possible terms. There is no unique solution, two different people could possibly come up with two different solutions to the same problem. It is often subjective which expression is the simplest. For example, is $a \& (b/c)$ simpler than $(a/b)\&(a/c)$?

To measure the complexity of an expression, a metric is defined for each node in the graph. This metric is called the **complexity factor** of the expression. The goal of the search is to locate the node with the smallest

complexity factor, where the named nodes have been replaced. This metric provides a consistent measure to determine the better solution.

The complexity factor chosen must be simple, as it is computed for every node visited while searching the graph, and the graph is typically quite large. One option for the complexity factor of an expression is the **number of terms in the expression**. So $h1 / f3$ would have a complexity factor of 2. There are several other possible measures, which could be used, such as the number of characters, the number of conjunctions, the number of brackets, etc.

However for ease of computation and because it is at least one factor a human would use, the number of terms in the expression was selected as the metric to guide the graph search. The goal is to locate the node in the graph with the minimum value. An example of a simple search tree is shown below, with the complexity factors shown inside the circles representing the nodes.

In this example, the required solution for the graph search is the node where the complexity factor is a minimum. In the partial graph shown, this is found at the node $h1 / f3$, where the value is 2.

In all but trivial cases, the search space will be very large or even infinite because there are transformations included which expand the expressions. It is therefore either impossible or undesirable to implement an exhaustive search of the complete space. We now fall back on some general techniques to search the graph.

3.11.10 General Graph Search Methods

There are many search procedures that can be used to find satisfactory paths to a known goal, for example Winston in his book Artificial Intelligence (Winston84), lists five methods which are outlined below.

1. In a **Depth First** search, each path in the graph is expanded until it either fails or succeeds to find the goal. At each level the child node to be expanded first is chosen at random and if that path fails to solve the problem then the next node to expand is selected.

Depth First searches are the default method in a Prolog program. In an infinite graph they may never locate the solution.

2. In **Hill Climbing**, the child nodes generated at each level in a depth first search are sorted so that the first child to be tried is the one nearest to the goal. This requires the ability to measure the remaining distance to the goal.

Hill Climbing requires the measure to be monotonic, otherwise false "foothills" may be found which may cause the search to terminate prematurely. This means that a measure should always increase as you approach the goal, and decreases as you move further away.

3. **Breadth First** searches the graph by expanding each node in each level in sequence, so that every path through the graph is investigated to the same depth before the next level is explored.

If the graph in a Breadth First search contains many branches at each level, the amount of store required to save them all can grow exponentially with the depth of the search.

4. **Beam Search** is comparable to Breadth First search, except that only the best n nodes are selected at each level, and the other nodes are purged. This again, assumes a measure is available of the remaining distance to the goal, but it ensures that rapid growth of the search tree is confined.

In this search method it is also possible that the solution will not be found as it was located down a path which was trimmed from the search. But, it allows the store required to be linear to the depth of search.

5. **Best First**, is a modification of Hill Climbing. Recall that Hill Climbing required the search moved on from the best child node

under consideration. In Best First search, **all** nodes are sorted and the best path which has not terminated is selected for further expansion.

The search method chosen is best first. This is because the time for the search is limited, and looking from the current best solution may well lead to an even better solution. As well if the search is terminated before completion (perhaps due to a time limitation), then a sorted list is readily available to allow the current best solution to be given.

3.11.11 Search Termination

When a point is reached that has a complexity value of one, it will be known that a **global** minimum has been reached, and the search can be halted as the best solution has been found. But, usually the value of the global minimum is not equal to one and is, most importantly, unknown. In this case, how do we know when to stop the search? We can use some heuristics to help to produce at least a reasonable answer, so that the search is constrained by time or distance.

An initial method of obtaining solutions is to detect **local minimum**. These are defined as nodes whose neighbors in the search path all have higher values. The **neighbors** of a node are nodes that lead into it and nodes that follow it in the search path. In the example below, the node with the value of 4 is a local minimum.

Figure 12 - Local Minimum.

The starting point for the search for the local minimums is the initial expression. The goal state is unknown at the beginning of the search. Starting from this point a best first search of the space is performed locating and stopping at the local minimums on each branch of the search tree. This is shown in Figure 13.

Figure 13 - Searching for Minimums.

It is possible that cycles can exist in the graph, such as between the nodes with values 6, 2 and 9 in the graph above. To avoid the search looping, a search path history can be stored, and each time an expression is generated it is checked to ensure that the search has not visited this node before.

The table below shows a serious limitation with this approach. It is very common, when simplifying an expression, to expand a term in order to continue making further simplifications. The search should not stop at each initial local minimum. Instead the search will be allowed to continue past this point to a depth of one, and if further simplifications are possible the search will be allowed to continue. This will detect if further simplifications are possible which may lead to a better solution. This is shown graphically for a single path through the graph in Figure 14.

Expression	Action	Complexity Factor
$(f1 \ \& \ f2) \mid f3$	Expand disjunction	3.
$(f1 \mid f3) \ \& \ (f2 \mid f3)$	Substitute in h1	4.
$h1 \ \& \ (f2 \mid f3)$	Substitute in h2	3.
$h1 \ \& \ h2$	Substitute in h0	2.
$h0$	End	1.

Table 4 - Table of Complexity Factor.

Figure 14 - Search Depth.

The basic limitations of this approach are:

- The speed of the basic replacement operation will be critical.
- Speed of the search through the space.
- It does not guarantee to locate the best solution.

Applying the search to the main example, we will arrive at the result derived by the manual method. This is shown in Figure 15. The lines shown in bold have been changed to reflect the modified rule. The named nodes remain in the rule, and so any future changes to either $h2$ or $h3$ will still be reflected in $h4$. Had the rewriting process not taken place, then future changes to $h2$ or $h3$ would not be rippled up to $h4$.

Figure 15 - Reconstruction of Knowledge Base.

3.12 Comparison with Rule Changes

The approach taken in this thesis differs from most of the earlier work by allowing the end user to specify an input, and a matching output, and getting the changes to the rule base computed by AMOK. This contrasts with the method described by Raatz and Drastal, for example, where the user specifies a rule change in the hope of causing the correct behavior of the system, and attempting to identify interactions.

Four types of interactions are specified by Raatz and Drastal (Raatz87). They can be compared with the method used by AMOK. AMOK is able to handle all the types of interactions specified and resolves the conflicts. This also allows the changes made to the knowledge base to be communicated to the user of the system with some meaning. For example, if a rule has been generalized then the user can be informed that this rule will now apply in more general cases. For illustration consider the following simple knowledge base, shown in Figure 16 below.

```
h51 <- f51 | f52.  
h52 <- h51 & f53.  
h53 <- h52 & f54.
```

Figure 16 - Rule Change Example.

This reduces to a DNF of:

$$\begin{aligned}
 h51 &= \{\{f51\}, \{f52\}\} \\
 h52 &= \{\{f51, f53\}, \{f52, f53\}\} \\
 h53 &= \{\{f51, f53, f54\}, \{f52, f53, f54\}\}
 \end{aligned}$$

3.12.1 Generalization

Given the input set of:

$$I = f53,$$

and requiring that

$$KB'(I) \Rightarrow h52.$$

This implies that the DNF $h52'$ can be computed as:

$$\begin{aligned}
 h52' &= \{\{f53\}, \{f51, f53\}, \{f52, f53\}\} \\
 \Rightarrow h52' &= \{\{f53\}\}
 \end{aligned}$$

Figure 17 - Rule Change, Generalization.

The input $h50$ has now been generalized to a new input set. The knowledge base has been split into two distinct sections. Now, $h52$ and $h53$ no longer depend on the input values of $f51$ or $f52$.

3.12.2 Conflict

Consider, given a null input set:

$I = ()$, and requiring that $KB'(I) \Rightarrow h51$.

Because of the closed world assumption, we can assume that the input is equivalent to:

$I = \wedge f51, \wedge f52, \wedge f53, \wedge f54$

$h51' = \{\{f51\}, \{f52\}, \{\wedge f51\}, \{\wedge f52\}\}$.

This is clearly a conflict as $h51$ can be deduced from both $\wedge f51$ and $f51$, or $\wedge f52$ and $f52$.¹⁰ In this case there are two ways to consider this apparent conflict:

- 1 Specify that the new input overrides the old knowledge base, and define the DNF for $h51'$ as:

$$h51' = \{\{\wedge f51\}, \{\wedge f52\}\}.$$

- 2 Due to the closed world assumption and because the f s are by definition ground, then this is in fact the same as:

$$h51' = \{\}.$$

This implies that $h1'$ will always be true. This is shown in graphic form in Figure 18, where T is always true.

¹⁰ This is not a conflict in a logical sense as this is consistent with saying that $h1$ is independent of $f1$. However it is normally assumed that rules of this form are potential source of errors in a knowledge base.

Figure 18 - Rule Change, Conflict.

3.12.3 Redundancy

Given an input of:

$I = f51$, and requiring that $KB'(I) \Rightarrow h51$.

This clearly can already be reduced from the existing knowledge base, and so adds no new information.

$$h51' = \{\{f51\}, \{f51\}\}$$

$$\Rightarrow h51' = \{\{f51\}\}$$

$h51'$ remains unchanged. Adding any new rule would add a redundancy into the knowledge base.

2

of/de

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

3.12.4 Specialization

This is the case where the rules for a hypothesis become more restrictive, so that they apply to fewer cases. Consider, that given input set containing a new input of *f55* with:

$I = f51, f52, f55$, and requiring that $KB'(I) \Rightarrow h51$.

In this case the system would conclude *h51*, and the output is exactly as required for the test run. However, it is likely to be the case that the user is saying that *h51* can **only** be concluded when *f51*, *f52*, **and** *f55* are input, because *h51* would have been output if just *f51* and *f52* were input. Another question to be decided is if **both** *f51* & *f52* are required. There are 3 distinct and reasonable options for the DNF of *h51*:

- 1 No change
 $h51 = \{\{f51\}, \{f52\}\}$
- 2 All of *f51*, *f52* & *f55* are required
 $h51 = \{\{f51, f52, f55\}\}$
- 3 Either of *f52* or *f53* are required together with *f55*.
 $h51 = \{\{f51, f5\}, \{f52, f55\}\}$

In this case, as before, we assume that the user is specifying the minimum requirements for concluding *h51*, so we will assume that case 2 is the intended result. If case 3 was intended, then all is not lost. It is just that this would have to be entered as two separate training cases.

3.13 System Summary

This chapter has presented a six step method to perform rule base modification. It focused on manipulation of DNF to cause the required changes, and to minimize the impact on the rest of the knowledge base. The user did not have to intervene, except to provide guidance when the dependency or the coverage of the named nodes lessened, and then only to confirm or overrule the system's action. The method was shown for one main example and other examples were used at each stage to explain the action to be taken for other cases. The Chapter 4 contains further examples of the results obtained using AMOK. The key design problem to be addressed is the rule reconstruction, which will reduce to a graph search problem. The next chapter reviews the progress made in implementing AMOK.

4 IMPLEMENTATION

This chapter explains some of the implementation details of AMOK, the proposed design having been reviewed in the previous chapter. It explains the implementation method selected, and discusses the issues raised by the techniques used. A key decision, made early in the implementation of this thesis, was to select Prolog as the implementation language. The reasons for this were:

- 1 Prolog is commonly used in Artificial Intelligence applications programming.
- 2 It is possible to implement a knowledge base and expert system shell easily in Prolog.
- 3 VM-Prolog was available on the computer I had access to.
- 4 I wished to investigate the **declarative** programming style Prolog supports, and determine the benefits and costs.

The implementation started with an expert system shell and example knowledge base, so that ideas could be tested. The implementation was then

focused on key modules of the system, such as the graph search performed during the reconstruction phase.

AMOK is programmed in VM-Prolog running on an IBM Mainframe computer. VM-Prolog is IBM's version of Prolog designed to run on mainframe computers. It provides a large number of built-in predicates, and will accept several forms of syntax, for this thesis the syntax used was chosen to be similar to Edinburgh syntax. In some areas of the implementation, the real-time performance of the code is an issue. In these cases, data on performance is provided. These performance measurements are based on the built in ability of VM-Prolog to measure the amount of CPU cycles which each Prolog goal consumes. These measurements are provided in CPU milliseconds on an IBM 3084 mainframe.

It became clear during the module testing phase of the Prolog that AMOK was performing too slowly. This appeared to be a consequence of the use of the declarative programming style, and the initial graph search method used, which was simpler than the one described in Chapter 3. The graph search algorithm was improved, and then a study was undertaken on the major benefit, code reversibility of a declarative programming style.

4.1 Reversible Prolog

This section examines a feature of the Prolog programming language which helps to make it unique, at least compared to other well known programming languages, such as FORTRAN, FASCAL, and LISP. Reversibility

has been taken advantage of in the implementation of this thesis to allow code reuse, but this has an impact on the execution speed of the implementation.

In simple terms **reversibility** is the ability of a predicate to work in both directions. For example, if there is a predicate which is called *FATHER(*dad,*child)*, which when given the name of a child will return the name of the father. It could also be used in reverse to provide the name of a child of a given father. Some examples of practical uses of reversibility are discussed. However not all predicates are reversible and the remainder of the section examines what can effect reversibility.

It is evident that reversible predicates are, in general, harder to write than non-reversible ones. They require a better understanding of the two key tools available to the Prolog programmer, namely unifications and backwards depth first search to ensure that they will work correctly in all directions. It is interesting that although the reversibility of Prolog predicates is often discussed as a benefit, there appears to be little documented on the methods or limitations of reversibility.

4.1.1 Predicates in Prolog.

Predicates in Prolog can be considered much like functions in a more conventional language. Some of the differences are that:

- 1 They return true or false, rather than a value.

- 2 They bind variables in non-ground terms during unification as required during the goal search. This is how values are 'returned' to the calling predicate.
- 3 Top down, depth first searching is used to try to satisfy a goal, so that if the first predicate fails, then the next one will be tried in the search order until either the goal is satisfied, or there are no more predicates left to try.

4.1.2 Ground Terms

Ground terms in Prolog are terms which do not contain **any** variables.

Examples of ground terms are:

- the atomic element *a*.
- the atomic element 1.
- a predicate with 3 ground arguments.
- the logical expression *a & b*.
- the list containing 3 items 1, 2 and 3.

In contrast, any term which contains a variable is non-ground. Examples are:

- the variable **a*.
- an unnamed variable ***.

- $A(*a, *b, *c)$ - A predicate with 3 variables as arguments.
- $\{*a, *b, *c\}$ - A list with 3 variables.
- $\{a, b, *x\}$ - A list with 1 variable, $*x$.

4.1.3 Reversibility

The simple case of a reversible predicate is a one which has two arguments. This predicate can then use the value of the first argument to compute the value of the second, or in reverse to use the value of the second argument to compute the value of the first.

However this is not the only way in which Prolog predicates can be used. Other uses are based on the complex pattern matching which can take place, called **unification**, and because predicates can have more (or less) than 2 arguments.

The definition of **reversibility** which will be used is:

A predicate is reversible if it will work as it is logically defined when called with any combination of ground and non-ground arguments, in any position.

A simple example of a reversible predicate is a list predicate *MEMBER* defined as:

```
member(*x, {*x!*tail}).  
member(*x, {*head!*tail}) <- member(*x, *tail).
```

There are at least 3 distinct ways that this member predicate can be used to compute member relationships.

member(a,{b,d,c,a}).

In this usage, the member goal is true if the first argument is a member (element) of the second argument (a list). True is returned in this case.

*member(*a,{b,d,a}).*

This goal can be used to generate all members of a list. The first argument is a variable, and it will be bound to each element of the list in turn as more solutions are requested. In this example, **a* will be bound to *b*, then to *d* and lastly to *a*.

*member(a,*list).*

In this form we are asking the question "what list is *a* a member of", to which there are an infinite number of answers. When called for the first time **list* is bound to *{a!*1}* i.e. a list with *a* as the first element. The second time a list of the form *{*1,a!*2}* is returned, which represents a list with a variable **1* as the first member, *a* as the second member and variable **2* as the tail. This pattern continues for ever.

In the simple example of the *MEMBER* predicate, reversibly can increase the number of uses of a predicate. Whereas, normally in LISP, all 3 cases of *member* would have to be coded explicitly.

A second example is the predicate *APPEND*, which provides the ability to determine if a list is the result of joining two sub-lists together. *APPEND* is defined as:

```
append({},*list,*list).
append(*head!*tail,*list,{*head!*tail2})
  <- append(*tail,*list,*tail2).
```

This can be used in a number of ways, such as:-

Appending two lists

```
GOAL : <- APPEND({1},{2},*1) .
<- APPEND({1},{2},{1,2}) .
```

To split a list into 2 parts which are the same.

```
GOAL : <- APPEND(*1,*1,{1,2,3,1,2,3}) .
<- APPEND({1,2,3},{1,2,3},{1,2,3,1,2,3}) .
```

To generate a list which will split into 2 equal parts.

```
GOAL : <- APPEND(*1,*1,*2) .
<- APPEND({}, {}, {}) .
MORE ?
<- APPEND(*1,*1,*1) .
MORE ?
<- APPEND(*1,*2,*1,*2,*1,*2,*1,*2) .
```

4.1.3.1 Pure Prolog

Pure Prolog is defined as predicates which are defined within the scope of symbolic logic (e.g. the cut is not used). However even limiting ourselves to this subset of Prolog, not all predicates will be reversible.

Typical of most built in predicates in Prolog¹¹, the built in arithmetic predicates are optimized to take advantage of the native machine instructions of the underlying processor. In VM-Prolog the *SUM(*X,*Y,*Z)* predicate demonstrates the problems. If *X and *Y are bound variables and *Z is a free variable then *Z is bound to *X + *Y. If one of *X or *Y are free and *Z is bound, then the free variable is bound to the difference. However if two of the variables are left free and *Z is bound, for example, to 3, an error message is generated.

What we really desire Prolog to do in this case is to generate all the solutions to the equation,

$$*z = *y + 3.$$

¹¹All built in predicates are not strictly speaking 'Pure PROLOG'. However they are used as base building blocks for arithmetic and lists, and we will see that they need to be defined logically for other predicates that use them, to function reversibly.

This can be achieved if we either define our own version of the *SUM* predicate which tests for free and bound variables or we implement our own base arithmetic¹².

Rather than rely on the built in arithmetic of Prolog we can go back to first principles and define arithmetic from pure logic. We can then add relationships such as addition to the base. The following shows how this can be done.

```
/* First what is a number - naturals only here
*/
/* A number is represented as the successor
relationship */
/* so that 3 is s(s(s(0)))
*/

num(0).
num(s(*n)) <- num(*n).

/* Now plus */

plus(0,*n,*n) <- num(*n).
plus(s(*l),*m,s(*n)) <- plus(*l,*m,*n).
```

Now *PLUS* is a reversible version of *SUM*. This version will take any combination of free and ground terms as arguments, and continue to provide solutions to the relationship as required. This is demonstrated below, with the numbers in 'successor form'.

```
GOAL : <- PLUS(0,0,0) .
OMS SUCCESS
<- PLUS(0,0,0) .

GOAL : <- PLUS(0,0,*1) .
```

¹²The first option is discussed in further details later.

```

1MS SUCCESS
  <- PLUS(0,0,0) .

    GOAL : <- PLUS(S(0),0,*1) .
0MS SUCCESS
  <- PLUS(S(0),0,S(0)) .

    GOAL : <- PLUS(S(0),*1,*2) .
0MS SUCCESS
  <- PLUS(S(0),0,S(0)) .
MORE ?
0MS SUCCESS
  <- PLUS(S(0),S(0),S(S(0))) .
MORE ?
0MS SUCCESS
  <- PLUS(S(0),S(S(0)),S(S(S(0)))) .

    GOAL : <- PLUS(*1,*2,*3) .
0MS SUCCESS
  <- PLUS(0,0,0) .
MORE ?
0MS SUCCESS
  <- PLUS(0,S(0),S(0)) .

    GOAL : <- PLUS(0,S(0),*1) .
0MS SUCCESS
  <- PLUS(0,S(0),S(0)) .
MORE ?
0MS FAIL

```

We can now use *PLUS* to write other reversible predicates such as *TOTAL*¹³. This predicate totals numbers stored as elements in a list.

```

total({},0) .

total({*head!*tail},*t) <- total(*tail,*tt) &
plus(*head,*tt,*t) .

```

¹³This predicate fails to be reversible when the built in *SUM* predicate is used in place of *PLUS*.

This version will work in both directions, the reverse direction is shown below.

```

GOAL : <- TOTAL(*1,S(0)) .
OMS SUCCESS
<- TOTAL({S(0)},S(0)) .
MORE ?
OMS SUCCESS
<- TOTAL({S(0),0},S(0)) .
MORE ?
OMS SUCCESS
<- TOTAL({0,S(0)},S(0)) .

```

However even the reversibility of this form of *TOTAL* has limitations. In Prolog the search for solutions is guided by a top down depth first method. Because of this it is possible for a search tree to be finite in one direction, but be infinite in the other. An example of this can be seen using the *TOTAL* predicate defined above, if we keep asking for more solutions.

```

GOAL : <- TOTAL(*1,S(0)) .
OMS SUCCESS
<- TOTAL({S(0)},S(0)) .
MORE ?
OMS SUCCESS
<- TOTAL({S(0),0},S(0)) .
MORE ?
OMS SUCCESS
<- TOTAL({0,S(0)},S(0)) .
MORE ?
garbage collector called by EVAL
      ls      gs      trail      rules
1023868    1016    15576    308592      0
0          0
1023868    1016    20464    308592      0
0          0
time 175 ms pass 0
*** not enough memory gained by garbage collector
***
581MS ERROR

```


To explain this, we have to look at what has happened as we looped on the *PLUS* predicate. *PLUS* is trying to find a solution to *PLUS(*,0,*)* which allows the goal *TOTAL(*,0)* to be solved. However *PLUS(*,0,*)* starts generating solutions which are non-zero, and so will fail when unified with the *TOTAL* goal, this repeats until the stack runs out.

TOT is a second alternative for implementing this predicate to try to solve the looping problem, writing the *PLUS* relation first. But, because *TOT* generates free variables for both **tail* and **tt*, it also becomes stuck in an infinite loop.

```
tot({},0).

tot(*head!*tail,*t) <- plus(*head,*tt,*t) &
tot(*tail,*tt).
```

When this is run the following output is generated.

```
GOAL : <- TOT(*1,S(0)) .
garbage collector called by EVAL
      ls      gs      trail      rules
894216  274756    1020    308592      0
0      0
894216  274756    1020    308592      0
0      0
time 83 ms pass 0
*** not enough memory gained by garbage collector
***
106MS ERROR
```

4.1.3.2 Real Prolog

In addition to the problems discussed above with search in Pure Prolog, there are also problems introduced by features of Prolog which are outside the

scope of logic. The prime example of this is the CUT, of which there are two flavors: the 'green cut' and the 'red cut'.

Green cuts are defined as cuts which prune the search tree examined by Prolog, without changing the 'logical meaning' of the program. An example of this green cutting using logical arithmetic, can be seen in *MERGE*.

```
merge({*x!*xs},{*y!*ys},{*x!*zs}) <-
  gtt(*x,*y) & / &
  merge(*xs,*y!*ys,*zs) .

merge({*x!*xs},{*y!*ys},{*x,*y!*zs}) <-
  eqq(*x,*y) & / &
  merge(*xs,*ys,*zs) .

merge({*x!*xs},{*y!*ys},{*y!*zs}) <-
  gtt(*y,*x) & / &
  merge({*x!*xs},*ys,*zs) .

merge(*xs,{},*xs) .

merge({},*ys,*ys) .

/* Uses the 'logical numbers' defined above with
   EQQ and GTT added */
```

The first test run shows the results **without** the shown cuts in the program, and shows how this program can be run in reverse to generate lists which merge to produce the output list.

```
GOAL : <- MERGE(*1,*2,{S(0),0}) .
2MS SUCCESS
<- MERGE({S(0)},{0},{S(0),0}) .
MORE ?
3MS SUCCESS
<- MERGE({0},{S(0)},{S(0),0}) .
MORE ?
2MS SUCCESS
<- MERGE({S(0),0},{},{S(0),0}) .
MORE ?
1MS SUCCESS
<- MERGE({}, {S(0),0},{S(0),0}) .
```

```

MORE ?
2MS FAIL

```

In this second run, **with** the cuts in, it will fail to produce all the correct solutions to the goal query. These have been trimmed by the cuts.

```

GOAL : <- MERGE(*1,*2,{S(0),0}) .
OMS SUCCESS
<- MERGE({S(0)},{0},{S(0),0}) .
MORE ?
OMS FAIL

```

Although we have kept the logical correctness of the program, and it generates one correct solution, it fails to find all solutions to the goal, whereas the uncut one does. The cut version is quicker and requires less storage to save backtracking points.

Cuts are referred to as Red, when they result in the writing down of rules which are not logically correct. A major use is to modify the flow of control of Prolog so that explicit conditions governing the use of the rule are omitted. An example of this is in a version of *MINIMUM* written to reduce the number of tests made.

```

minimum(*x,*y,*x) <- lt(*x,*y) & !.
minimum(*x,*y,*y).

```

In this example, we have removed the test on the second rule, assuming that if the first rule fails then the second case will succeed. However this can lead to errors. A goal like

```

minimum(1,3,3).

```

will succeed when it clearly should fail. The correct coding should include an explicit test on the second rule as shown below.

```
minimum(*x,*y,*x) <- !t(*x,*y) & !.  
minimum(*x,*y,*y) <- !t(*y,*x) & !.
```

As could be predicted, the use of red cuts leads to all the problems associated with the use of green cuts, with additional problems caused by the logically incorrect rules as shown in the example above.

Negation, as implemented in Prolog, can also be expressed as a cut relation.

```
^ *x <- *x & / & fail.  
^ *x.
```

This implementation of negation is referred to as 'the closed world assumption', because the negation of the goal is true whenever Prolog has failed to prove the goal. This does not allow the possibility of a value being unknown. This implementation of negation is a red cut, and may lead to the problems described above in the section on red cuts.

In conclusion, when using pure Prolog the only issues to consider for reversibility are that of (1) will the search terminate in all directions, (2) will the built-in predicates work as required, and (3) will the program run fast enough?

When considering cut or negation the program may well perform as required when given goals of one form. When a reverse form is provided there is no guarantee that all the solutions will be correct or generated. If green cuts are used, then all the solutions may not be generated as the search tree has been pruned. However for red cuts and negation, faulty results can be produced when the predicate is not used as expected. It follows that, in Prolog, it is

necessary to not only document what a predicate does, but also how it should be used.

It is also possible in most PROLOGs to work around some problems described above. For example, infinite search may be avoided in some cases by testing to see if arguments to a predicate are ground or free variables. This requires some META-LEVEL predicates outside first order logic. These could be used to encode a number of different versions of predicates, so that they will work transparently to the user regardless of application. This, however, is hardly an elegant solution, and would only work in some of the simpler cases. In the case of the *SAME* predicate, described later, which is used as the basis for rule reconstruction, it is speed which is lost by using a reversible predicate. The gain is in code compactness and reuse. The following sections discuss the implementation of the design of the stages of AMOK in the order that they are executed in.

4.2 Identification of change required

The implementation of a simple, template matching, Human to Machine Interface (HMI) is a text book application of Prolog, using unification. After all, some of the original applications written in Prolog were in the area of natural language processing, and the built in pattern matching capabilities of Prolog form an ideal base of this type of HMI.

The identification of the change required comes directly from the user interface. The task is to recognize if a hypothesis should be added or dropped from the output set associated to the given input facts.

4.3 Identification of impacted nodes

The minimal upper bound (MUB) function is computed by searching the knowledge base structure. First, all nodes which are specified in either the user input or the initial hypothesis of the system are identified. This list is used as the seed nodes of AMOK to identify all nodes which may be effected by this run. A recursive exhaustive depth first search of the knowledge base is used to identify all nodes relating to these initial nodes. The search is made from the bottom up from the input facts, and top down from the hypothesis. The impacted nodes are a superset of the nodes identified in these searches.

4.4 Reducing hypotheses to disjunctive normal form.

Standard Prolog techniques can be used to reduce a hypothesis to the disjunctive normal form (DNF). For a hypothesis written in propositional logic this consists of two steps:

- 1 Move all negation inwards in the expression. Cancel out all double negations.
- 2 Distribute the disjunctions ('or') over the conjunctions ('and').

However a full version which supports Predicate Calculus was implemented, which may aid in possible future expansion of AMOK. It is based on the version in Clocksin & Mellish's book (Clocksin84). This predicate also performs the rule rewrite to put the DNF into list notation. To provide some idea of the operation of this predicate, we can examine some typical results. In all of these tests the execution time was too small to be accurately measured.

INITIAL EXPRESSION	DNF
$a \ \& \ b$	$\{\{a,b\}\}$
$a \ \ b$	$\{\{a\},\{b\}\}$
$(a \ \& \ b) \ \ (c \ \& \ d)$	$\{\{a,b\},\{c,d\}\}$
$(a \ \ b) \ \& \ ^{(c \ \ d)}$	$\{\{^a c, ^a d, a\},\{^a c, ^a d, b\}\}$

Table 4 - DNFs of simple expressions.

4.5 Simplification & incorporation of the change.

The incorporation and simplification of the identified change are simple list operations, the exact detail of which depend on the change. The different cases were discussed in the previous section. These do not present any great implementation issues. Most of the desired list operations exist in VM-Prolog. The remainder are easily coded, using the standard list techniques of operating

on the head and the tail of the list recursively until the list is exhausted, or the problem is solved.

4.6 Reconstruction of the knowledge base

This is the most complex task to be implemented. It tries to mimic the methodology of a human in simplifying an expression. The task is broken down into four basic operations to be implemented, these are:

- 1 Computation of named node coverage and dependency. This is used to guide the use of named nodes in the reconstruction.
- 2 Expression Equality. This provides a logical unification of two Prolog expressions. These expressions are allowed to contain free variables and this predicate is reversible. As well as testing for equality, it also generates expressions that are logically equal. **This predicate provides a logical unification.**
- 3 Expression Transformation. This uses the logical transformations and the named node substitutions to transform one logical expression into another. This uses the *Expression Equality* test to determine if an expression matches the definition of a named node.
- 4 Graph Search. This implements the graph search method discussed in the previous chapter. It uses the expression transformations to determine the nodes in the graph.

4.6.1 Computation of coverage and dependency

A Karnaugh map¹⁴ can be used to represent these concepts as it helps to visualize the problem¹⁵. This is a representation of the truth values of the logical expressions for given input values. The points in the space where the expressions are true are denoted with dots (•) in the figures below.

Referring back to the main example in Chapter 3, we will now examine the Karnaugh Map for the expression for the DNF, both before and after the modifications.

The original DNF is:

$$h4 = \{\{f1, f3, f4, f5\}, \{f1, f3, f4, f6\}, \{f2, f3, f4, f5\}, \{f2, f3, f4, f6\}\}$$

and named node h2 DNF is:

$$h2 = \{\{f1, f3\}, \{f2, f3\}\}$$

also named node h3 DNF is:

$$h3 = \{\{f4, f5\}, \{f4, f6\}\}$$

¹⁴ Karnaugh maps are logic truth tables written in such a way that only one bit changes from one adjacent position to the next. So rather than plotting values in a row as 00, 01, 10, 11, they would be plotted as 00, 01, 11, 10.

¹⁵ I originally hoped that Karnaugh maps could be used to solve the rule simplification problem. This turned out to not be feasible because these maps are designed to solve problems using basic logic circuits of 'and' and 'or' gates rather than using complex expressions such as, named nodes as a basic form.

These are shown in Figure 19, which uses a Karnaugh map to display the relationships. As shown in the map, there are 9 points in this input space, denoted by the dots, out of a total of 64, where the hypothesis, h_4 , will be concluded. The overlay of the expressions for the named nodes, h_2 & h_3 , are also shown.

Figure 19 - Karnaugh MAP for h_4 .

There are 9 points in this space where h_4 is concluded. It is now possible to examine the relationship between these points and the named nodes, h_3 and h_2 . The first relationship is the coverage. This is the number of points where the

value of the hypothesis is included (or covered) by the named nodes. As shown in the Karnaugh Map, all 9 points which conclude $h4$ are inside both $h2$'s and $h3$'s boxes, so whenever $h4$ is true so are $h2$ and $h3$. Therefore both $h2$ and $h3$ have a coverage of 9 out of 9 points.

Next we can look at the dependency of $h4$ on the named nodes $h2$ and $h3$. We can see that $h2$ contains a total of $(6 \times 4) = 24$ points of which 9 have the same value as $h4$. For $h3$ we have a total of $(8 \times 3) = 24$ points of which again 9 have the same value as $h4$. So, the dependency is $9/24$ for both $h2$ and $h3$.

The map of the modified version of the hypothesis derived earlier, is shown in Figure 20.

Recall that the modified hypothesis $h4'$ is.

$$h4' = \{\{f1, f3\}\{f2, f3, f4, f5\}, \{f2, f3, f4, f6\}\}$$

The shaded area in the bottom right hand corner should be noted in the figure, as this denotes the area that has been changed from the previous map. The DNF of the named nodes $h2$ and $h3$ have not changed, as no changes have been made to their definitions.

Figure 20 - Karnaugh MAP for h_4' .

There are now 19 solution points in the input space for $h4'$, and, based on the two Karnaugh Maps, we can examine the coverage and dependency of each named node in the old and new form.

In the coverage of the 19 points which conclude $h4'$, all are inside $h2$'s area, but $h3$'s area only covers the original 9 points, and therefore its coverage has decreased. Next we can look at the dependency of $h4'$. We can see that $h2$ still contains a total of 24 points of which 19 have the same value as $h4$. For $h3$ we have 24 points of which again only 9 have the same value as $h4$. The complete results are shown in the Table 3.

The number of input facts here is 6 ($f1, f2, f3, f4, f5, f6$). $f7$ is not included as it is not in the set of input values for the hypothesis $h4$, so its value has no impact on the problem in hand. Six input facts implies that there are 64 possible distinct combinations of input values (each one of the input values will be true or false). In general the size of this solution space is 2^n , where n is the total number of f 's in the space. It is not practical to draw a Karnaugh map for the space when n is greater than 6.

The coverage and dependency can be calculated by first computing an extended version of the DNFs. The extended version expands the DNF to explicitly list all the input facts for both the true and false cases, whereas in the DNF we have been dealing with up to now the values that were of no interest were removed from the expression. For example, the original DNF is:

$$h4 = \{ \{f1, f3, f4, f5\}, \\ \{f1, f3, f4, f6\}, \\ \{f2, f3, f4, f5\},$$

 $\{f2, f3, f4, f6\}$.

This can be rewritten by expanding all the "don't care" values in each term of the DNF, and removing the duplicate terms, as follows:

$$h4 = \{ \{f1, f2, f3, f4, f5, f6\}, \\ \{f1, f2, f3, f4, f5, ^f6\}, \\ \{f1, ^f2, f3, f4, f5, f6\}, \\ \{f1, ^f2, f3, f4, f5, ^f6\}, \\ \{f1, f2, f3, f4, ^f5, f6\}, \\ \{f1, ^f2, f3, f4, ^f5, f6\}, \\ \{^f1, f2, f3, f4, f5, f6\}, \\ \{^f1, f2, f3, f4, f5, ^f6\}, \\ \{^f1, f2, f3, f4, ^f5, f6\} \} .$$

This can also be done for the named nodes *h2* and *h3*. It then becomes a simple, but fairly slow process to compare the expressions and compute the coverage and dependency.

4.6.2 Expression Equality

SAME is a basic primitive logical operation. It has a complex task to test or generate two expressions, which possibly contain free variables, to see if they represent the same **logical** form. This test for equality will be able to match expressions which are logically the same, so for example:

$(a \& b)$ can be logically unified with $(b \& a)$

$(a \& *)$ can be logically unified with $(c \& a)$

$(a \& *1 \& *2)$ can be logically unified with $(*1 \& a \& *2)$

$(a \& b \& c)$ can be logically unified with $(b \& c \& a)$ and $(c \& *)$.

In effect this operation knows very simple logical operations (e.g. commutative law) which preserve equality. If these laws were defined as separate operations, each expression would have so many different forms, that the graph search would become impossible, because the number of nodes would be huge.

```

/* SAME TEST to see if two logical forms can be
logically unified */
/* Considers things like swapping around of parts.
*/

same(*a,*a) .

same(^*a,^*b) <-
  same(*a,*b) .

same(*a | *b,*c | *d) <-
  same(*a,*c) &
  same(*b,*d) .

same(*a | *b,*c | *d) <-
  same(*a,*d) &
  same(*b,*c) .

same(*a|*b,*c|*d) <-
  perm(*c|*d,*e | *f) &
  same(*a,*e) &
  same(*b,*f) .

same(*a & *b,*c & *d) <-
  same(*a,*c) &
  same(*b,*d) .

same(*a & *b,*c & *d) <-
  same(*a,*d) &
  same(*b,*c) .

same(*a&*b,*c&*d) <-
  perm(*c&*d,*e & *f) &
  same(*a,*e) &
  same(*b,*f) .

/* A permutation function. This perm ONLY swaps
around 1st level terms. */

```

```

/*
*/

perm(*a|*b,*c) <- perm_dis(*a|*b,*c).

perm(*a&*b,*c) <- perm_con(*a&*b,*c).

perm_dis(*x | *xs,*z | *zs) <- / &
  select_dis(*z,*x | *xs,*ys) &
  perm_dis(*ys,*zs).

perm_dis(*x,*x) <- /.

perm_con(*x & *xs,*z & *zs) <- / &
  select_con(*z,*x & *xs,*ys) &
  perm_con(*ys,*zs).

perm_con(*x,*x) <- /.

select_con(*x,*x & *xs,*xs).

select_con(*y,*x & *y,*x) <- ^(*y = (* & *)).

select_con(*x,*y & *ys,*y & *zs) <-
  select_con(*x,*ys,*zs).

select_dis(*x,*x | *xs,*xs).

select_dis(*y,*x | *y,*x) <- ^(*y = (* | *)).

select_dis(*x,*y | *ys,*y | *zs) <-
  select_dis(*x,*ys,*zs).

```

It can be seen that the code contains built in knowledge about the logical operations that do not effect the value of the expression. It also contains a predicate to compute permutations because this may be used to generate expressions that are permutations of an input expression.

The performance of this operation has been measured on a test case where the length of the expression increases from 1 to 10. The results here show the time taken to attempt to match to ground expressions.

Length The number terms in the expression. The expression is of a form $(a \& b \& c \& \dots)$. For example $(a \& b \& c)$ has a length of 3

Match A case where there is a match in the two string tested. An example would be $SAME(a \& b \& c, a \& b \& c)$ for a match in the same order, and $SAME(a \& b \& c, c \& b \& a)$ for a match in reverse order.

No Match Where all but one term is the same. This is the worst case. An example would be $SAME(a \& b \& c, x \& b \& c)$ for an error at the start and $SAME(a \& b \& c, a \& b \& z)$ for errors at the end of the expression. The two cases are examined as they give quite different results.

Length	MATCH		NO MATCH	
	Same order	Reversed order	Error at start	Error at end
1	0	0	0	0
2	0	1	1	1
3	0	1	1	2
4	0	2	3	11
5	0	11	11	260
6	0	64	64	30258
7	0	448	442	*
8	0	3578	3551	-
9	0	32094	32357	-
10	0	*	*	-
* indicates that the calculation was halted before a result was obtained.				

Table 5 - Performance of *SAME* predicate.

The reason why failures take so long relates to the flexibility of the code to allow free variables. However, the advantage of this implementation can be seen in the following sections where the *SAME* operation forms a building block

to test for expression equality. As can be seen from the code, *SAME* is fully reversible, and although *PERM* does contain cuts, these are green cuts used to remove duplicate results.

4.6.3 Expression Transformation

The named node substitutions and rule simplifications are computed together using a generic **replace** function. This will either replace an expression for a named node or apply the logical equivalence relationship. The generic **replace** function is defined as:

```
replace(*exp1, *exp2, *exp3, *exp3)
```

This function will replace *exp1* in *exp2* with *exp3* to give *exp4*. This uses the same predicate explained above, and therefore has the similar performance issues. A table summarizing the performance is presented after the code.

The real advantage of this code is the flexibility. It will handle simple cases as well as very complex cases where there are many free variables, allowing the state changes to be coded as 'meta-rules', as shown below. This flexibility to write meta-rules allows rapid investigation of the trade off between providing many replacement operations and the search time. This would benefit future investigations trying to tune the performance of AMOK.

```
/* Replace exp1 in exp2 with exp3 to give exp4
*/
/*
*/
/* Considers things like swapping around of parts.
*/
```

```

/* simple case of perfect match */
replace(*exp1,*exp2,*exp3,*exp3) <-
  same(*exp1,*exp2).

/* replacing through negation */
replace(*exp1,^*exp2,*exp3,^*exp3) <-
  same(*exp1,*exp2).

/* case when exp1 is in this level of exp2. */
replace(*exp1a | *exp1b,*exp2a |
*exp2b,*exp3,*exp4) <-
  select_dis(*exp2ax,*exp2a | *exp2b,*exp2arest) &
  same(*exp1a,*exp2ax) &
  select_dis(*exp2bx,*exp2arest,*exp2brest) &
  same(*exp1b,*exp2bx) & / &
  *exp4 =(*exp2brest | *exp3).

/* case when exp1 is level down of exp2. */
replace(*exp1,*exp2a | *exp2b,*exp3,*exp4) <-
  replace(*exp1,*exp2a,*exp3,*exp4a) & / &
  *exp4 = (*exp4a | *exp2b).

/* case when exp1 is level down of exp2. */
replace(*exp1,*exp2a | *exp2b,*exp3,*exp4) <-
  replace(*exp1,*exp2b,*exp3,*exp4b) & / &
  *exp4 = (*exp2a | *exp4b).

/* case when exp1 is in this level of exp2. */
replace(*exp1a & *exp1b,*exp2a &
*exp2b,*exp3,*exp4) <-
  select_con(*exp2ax,*exp2a & *exp2b,*exp2arest) &
  same(*exp1a,*exp2ax) &
  select_con(*exp2bx,*exp2arest,*exp2brest) &
  same(*exp1b,*exp2bx) & / &
  *exp4 =(*exp2brest & *exp3).

/* case when exp1 is level down of exp2. */
replace(*exp1,*exp2a & *exp2b,*exp3,*exp4) <-
  replace(*exp1,*exp2a,*exp3,*exp4a) & / &
  *exp4 = (*exp4a & *exp2b).

/* case when exp1 is level down of exp2. */
replace(*exp1,*exp2a & *exp2b,*exp3,*exp4) <-
  replace(*exp1,*exp2b,*exp3,*exp4b) & / &
  *exp4 = (*exp2a & *exp4b).

```

Table of test runs for the REPLACE predicate				
exp1	exp2	exp3	Result	Time
a	a & b	c	b & c	0
a & b	a & b & c	d	c & d	1
a & b	a & c & b	d	c & d	1
a & b	i (a&b)	x	i x	1
*x & *x	a & b & a	*x	b & a	1
*x & (*y *z)	a & (b c)	(*x&*y) (*x&*z)	(a&b) (a&c)	0
*x & *x	a&b&c&d&e	*x	b&c&d&a	125
*x & *x	a b c d e	*x	FAIL	1
*x & *x	a&b&c&d&e	*x	FAIL	135
*x & (*y *z)	(b c) & a	(*x&*y) (*x&*z)	(a&b) (a&c)	0
(*x&*y) (*x&*z)	a (a&b) (b&c)	*x & (*y *z)	a (b & (a c))	3

Table 6 - Performance of REPLACE predicate.

Some examples are provided of the types of expressions that can be used with the results and time taken. This table shows the replacement operation requested, and the results. The CPU time measured for this operation is also shown. It is intriguing that it may take longer to attempt the replacement of a close match, which fails, than apply a replacement that works. This is because every possibility has to be tried before failure is reported, and, while the basic operands match, different permutations of the variables also must be tried. *REPLACE* is not fully reversible, in order to speed it up, and to remove many duplicate solutions, green cuts were added. However, it is still flexible enough to allow meta-rules be defined to generate the moves in the search.

4.6.4 Graph Search

There are two factors in the graph search problem. The first is the rules that will be used to transform the expressions, these are:

```
/* A library of Meta Rules to be applied to
simplify the */
/* rule form.
*/
/* Uses a 'Move' function to show what are moves
that can be made */
/* This is the form of these functions :-
*/
/* MOVE(*exp1,*move,*exp2)
*/

/* First some rules to try and replace expressions
for named nodes */

/* Named Nodes substitutions are stored in a global
*nn_subs as a */
/* list. e.g. *nn_subs(*sub,*nn) will return, in
order, the */
/* substitutions to try.
*/
```

```

move(*exp1,named_node=*nn,*exp2) <-
  nn_subs(*sub,*nn) &
  replace(*sub,*exp1,*nn,*exp2).

/* Next rules to Simplify the expression.
*/

move(*exp1,compress_conjunction,*exp2) <-
  replace(*a&*a,*exp1,*a,*exp2).

move(*exp1,compress_disjunction,*exp2) <-
  replace(*a|*a,*exp1,*a,*exp2).

move(*exp1,double_negation,*exp2) <-
  replace(^(*a),*exp1,*a,*exp2).

move(*exp1,common_disjun,*exp2) <-
  replace((*a|*b)&(*a|*c),*exp1,*a&(*b|*c),*exp2).

move(*exp1,common_conjun,*exp2) <-
  replace((*a&*b)|(*a&*c),*exp1,*a|(*b&*c),*exp2).

/* Next rules which just move negation inwards.
*/

move(*exp1,neg_in_and,*exp2) <-
  replace(^(*a&*b),*exp1,^*a|^*b,*exp2).

move(*exp1,neg_in_or,*exp2) <-
  replace(^(*a|*b),*exp1,^*a|^*b,*exp2).

/* Next rules that now move it outwards.
*/

move(*exp1,neg_out_and,*exp2) <-
  replace(^*a&^*b,*exp1,^(*a|*b),*exp2).

move(*exp1,neg_out_or,*exp2) <-
  replace(^*a|^*b,*exp1,^(*a&*b),*exp2).

/* Now rules that make everything more complex.
*/

move(*exp1,exp_conjun,*exp2) <-
  replace((*a&*b)|(*c&*d),*exp1,
    (*a|*c)&(*a|*d)&(*b|*c)&(*b|*d),*exp2).

```


MOVE is used to generate moves in the graph search. Given an expression it will attempt to apply the logical transformations to the expression. For explanation reasons the transformation applied is noted. Again the performance of this predicate is shown:

Old Expression	New Expression	How	Time
a & a	a	compression conjunction	3
$\wedge(b \& c)$	$\wedge c \mid \wedge b$	move negative in over and	3
$(f1 \& f3) \mid (f2 \& f3 \& h0 \& (f5 \mid f6))$	$f3 \mid (f1 \& f2 \& h0 \& (f5 \mid f6))$	remove a common conjunction	3825
Repeat for expression above	?	?	*
* indicated that there were no more results generated in a reasonable time.			

Table 7 - Performance of *MOVE* predicate.

Next we have the Prolog implementation of the Best First Search Method.

This can simply be shown as:

```
/* Solves the graph by expanding the most promising
node */
/*
*/
```

```

/* Passes around a list of lists to represent
current state */
/*****
*****/

bag_of(*v,*g,*l) <-
  compute(list,*v,*g,nil,*l) & /.

bag_of(*v,*g,nil).

search(((end.*rest).dontcare),(*tonode.*end.*rest
)) <-
  move(*end,*tonode) &
  satisfactory(*tonode).

search(*first.*frontier,*good_path) <-
  extend(*first,*new_paths) &
  append(*frontier,*new_paths,*new_frontier1) &
  sort(*new_frontier1,*new_frontier2) &
  search(*new_frontier2,*good_path).

extend((end.*rest),*paths) <-
  bag_of((other.*end.*rest),
    move(*end,*other) &
  ^member(*other,*rest),
    *paths).

/* Clause to sort a list (using insertion sort); */

sort(code $ {*X!*L},*M) <-
  sort(*L,*N) &
  sortx(*X,*N,*M).
sort({},{}).

sortx(*X,code $ {*A!*L},{*A!*M}) <-
  (*a = {*a1!*}) &
  (*x = {*x1!*}) &
  value(*A1,*va) & value(*X1,*vx) & lt(*va,*vx) &
  / &
  sortx(*X,*L,*M).
sortx(*X,*L,{*X!*L}).

```

In order to show how this code performs, it is easiest to use an example graph to be searched. This is shown in Figure 21 below.

Figure 21 - Graph Search Example.

The output from this run is:

```

<-search({{a}},*).
GOAL : <- SEARCH({{A}},*1)
1 : CALL ==> SEARCH({{A}},*1)
2 : CALL ==> SEARCH({{G,A},{F,A},{B,A},{C,A}},*1)
3 : CALL ==>
  SEARCH({{F,A},{B,A},{B,G,A},{C,A}},*1)
4 : CALL ==>
  SEARCH({{B,A},{B,G,A},{C,A},{C,F,A}},*1)
5 : CALL ==>
  SEARCH({{G,B,A},{D,B,A},{B,G,A},{C,A},{C,
F,A},{H,B,A}},*1) .
6 : CALL ==>
  SEARCH({{D,B,A},{B,G,A},{C,A},{C,F,A},{H,
B,A}},*1) .
6 : EXIT ==>
  SEARCH({{D,B,A},{B,G,A},{C,A},{C,F,A},{H,
B,A},{E,D,B,A}}) .
5 : EXIT ==>
  SEARCH({{G,B,A},{D,B,A},{B,G,A},{C,A},{C,
F,A},{H,B,A},{E,D,B,
A}}) .

```

```

4 : EXIT ==>
    SEARCH({{B,A},{B,G,A},{C,A},{C,F,A}},{E,D
,B,A}) .
3 : EXIT ==>
    SEARCH({{F,A},{B,A},{B,G,A},{C,A}},{E,D,B
,A}) .
2 : EXIT ==>
    SEARCH({{G,A},{F,A},{B,A},{C,A}},{E,D,B,A
}) .
1 : EXIT ==> SEARCH({{A}},{E,D,B,A}) .
12MS SUCCESS
<- SEARCH({{A}},{E,D,B,A}) .

```

4.7 Specimens

This section contains a summary of further examples of the results that would be produced by running AMOK. These examples are based on the knowledge base described in Chapter 3. They were computed partly by hand and partly by the Prolog code described.

h0 <- f4.	r1
h1 <- f1 f2.	r2
h2 <- h1 & f3.	r3
h3 <- h0 & (f5 f6).	r4
h4 <- h2 & h3.	r5
h5 <- h3 & f7.	r6

Facts:

f1,f2,f3,f4,f5,f6,f7

hypotheses:

h0,h1,h2,h3,h4,h5

rules:

r1,r2,r3,r4,r5,r6

Hypotheses DNF form

Named Nodes

$h_0 = \{\{f_4\}\}$	
$h_1 = \{\{f_1\}, \{f_2\}\}$	
$h_2 = \{\{f_1, f_3\}, \{f_2, f_3\}\}$	h_1
$h_3 = \{\{f_4, f_5\}, \{f_4, f_6\}\}$	h_0
$h_4 = \{\{f_1, f_3, f_4, f_5\}, \{f_1, f_3, f_4, f_6\},$	h_2, h_3
$\{f_2, f_3, f_4, f_5\}, \{f_2, f_3, f_4, f_6\}\}$	
$h_5 = \{\{f_4, f_5, f_7\}, \{f_4, f_6, f_7\}\}$	h_3

4.7.1 Example 1

This is the example in the main body of the text. This example added to the cases when a hypothesis should be concluded, and the dependency of the named nodes remained constant or increased.

Input: f_1, f_3

Original Output: h_1, h_2

Modification Required: Add h_4

Modified DNF: $h_4 = \{\{f_1, f_3\}, \{f_2, f_3, f_4, f_5\}, \{f_2, f_3, f_4, f_6\}\}$

Named Nodes Dependency:

	<u>Old</u>	<u>New</u>
h2	0.38	0.79
h3	0.38	0.38

Modified Rule: $h4' \leftarrow h2 \& (f1 \mid h3).$

4.7.2 Example 2

This example is based on a removal of a term hypothesis. It shows that the dependency between the named node and the hypothesis has decreased and so the link has been broken.

Input: f1,f3

Original Output: h1,h2

Modification Required: Delete h2

Modified DNF: $h2' = \{\{f2,f3\}\}$

Named Nodes Dependency:

	<u>Old</u>	<u>New</u>
h1	0.5	0.33

Modified Rule: $h2' \leftarrow f2 \& f3.$

4.7.3 Example 3

This adds to the cases when a hypothesis should be concluded. In contrast with example 1, the coverage between the named node and the

hypothesis has decreased and so the link has been broken. The dependency remained constant.

Input: f3

Original Output: nil

Modification Required: Add h2

Modified DNF: $h2' = \{\{f3\}\}$ (Terms have been removed)

Named Nodes Dependency:

	<u>Old</u>	<u>New</u>
h1	0.5	0.5

Named Nodes Coverage:

	<u>Old</u>	<u>New</u>
h1	1	0.75

Modified Rule: $h2' \leftarrow f3$

4.7.4 Example 4

This adds to the cases when a hypothesis should be concluded. In this case the addition has increased the relationship to the existing named node.

Input: f4

Original Output: h0

Modification Required: Add h2

Modified DNF: $h2' = \{\{f1, f3\}, \{f2, f3\}, \{f4\}\}$

Named Nodes Dependency:

	<u>Old</u>	<u>New</u>
h1	0.5	0.75

Modified Rule: $h2' \leftarrow (h1 \ \& \ f3) \mid f4.$

4.7.5 Example 5

This shows the cases when a hypothesis should be concluded under more restrictive conditions. In this case the user is saying that the output can be given only when all of the inputs are present. There are no named nodes to be considered.

Input: f1, f2

Original Output: h1

Modification Required: Restrict h1

Modified DNF: $h1' = \{\{f1, f2\}\}$

Modified Rule: $h1' \leftarrow f1 \ \& \ f2.$

Finally, in this chapter, the additional work to complete a full implementation of AMOK is discussed. This work is required to take AMOK from a feasibility investigation to a completed fast system which could be used in a production expert system shell to make changes in a reasonable time.

4.8 Implementation Limitations

The major implementation issue is speed. This was expected due to the exponential nature of some of the problems. However, the problem was worse than anticipated, due to the speed of some underlying primitives. The key one is *SAME*, which tests to see if two logical expressions can be unified. This is written using reversible code, and it was shown that when an expression is longer than 5 or 6 terms the performance may be too slow in many cases. This is used as a building block to define *REPLACE* which although not being fully reversible, does allow a complex logical pattern matching to take place. This allowed meta-rules to be defined to generate the moves for the search.

However, because of the performance problems, the running of the complete system was never possible. The feasibility of the system was demonstrated by implementation of each section in a stand alone fashion. The basic graph search method was shown to work, and the methods to replace terms in expressions was also demonstrated. It was in the integration of these blocks that the performance degraded to a point where AMOK is not usable. In order to progress further in this endeavor, a re-implementation of the Prolog predicates needs to take place to speed up the graph search. The use of predicates designed for performance rather than code reuse is recommended.

In summary the limits identified in the system design and the implementation section are:

- 1 Restrictive propositional logic was used for the knowledge base.

- 2 Changes are limited to the addition or removal of one hypothesis at a time.
- 3 Only fairly basic examples have been tried, because of the speed of the implementation of AMOK.
- 4 The dependency and coverage were only discussed for positive relationships.

4.9 Implementation Summary

In this chapter we have seen how the major sections of AMOK have been implemented. The procedure for verifying that the overall approach is feasible has been that of taking the major functions, and showing that they each perform as intended. This has demonstrated that this system works on the knowledge base in the example, and is capable of providing the type of automatic modification of a knowledge base sought.

5 CONCLUSIONS

This thesis investigated methods of automating modifications for recommendations from a rule based expert system. The investigation, based on the modification of a reduced form of the knowledge base, commenced before the papers by Ginsberg relating to this approach were published. Ginsberg's paper (Ginsberg88b) helped to confirm that this direction held possibilities. In particular, the key attribute of this work in common with Ginsberg's approach is reduction of a knowledge base written in propositional logic to a disjunctive normal form.

This work also departs from Ginsberg's work, and the major areas of difference are:

- 1 Modification of the knowledge base results from individual changes requested by a user, rather than being driven by a statistical analysis of empirical results. I believe that this is a more realistic situation for local end user modification, where users are likely to require one or two modifications, rather than massive changes. However, having a set of empirical results to verify that the knowledge base functions as required, could be useful as

regression testing to ensure that unexpected side effects have not accrued over time.

- 2 AMOK uses the minimal upper bound to limit the area of the knowledge base to be examined. This is partly possible because individual changes are considered. However, Ginsberg realized that this type of technique was required, because of the time that reduction takes.
- 3 This thesis focused on rule base reassembly which Ginsberg does not describe. Reassembly using graph search and reversible code for flexibility was found to be a slow process that was not usable on a real knowledge base.

Knowledge base reconstruction was the most complex problem studied in this thesis, many options were examined. Approaches to the problem that were studied included, using Karnaugh Maps, variations of generate and test, and graph search based methods. A best first graph search was chosen and implemented, this used local minimum detection to help decide when the search should be terminated. Dependency of the named nodes was found to be important in guiding the reconstruction process. At the end of this work there are two key questions that remain to be answered:

- 1 The current design and implementation of AMOK does not perform fast enough for it to be used with a reasonably sized knowledge base. Can this be corrected, or is this approach fundamentally flawed?

- 2 AMOK and other related approaches use a very limited knowledge representation language, propositional logic. This does not reflect the languages used in many expert systems. Could the approach of knowledge base reduction be extended to other representations, for example, predicate calculus?

5.1 AMOK'S Performance

The performance problems described in Chapter 4 can be categorized as as either intrinsic due to the nature of the problem, or correctable because they are due to the implementation method selected. The nature of the intrinsic problems can be recognized by looking at the key areas studied, this will place some bounds around the size of problem that AMOK can handle regardless of the implementation chosen. The problems that can be corrected are in some ways self imposed because they can be rectified in a later implementation of AMOK. This Chapter makes some recommendations on what areas of AMOK should be rewritten.

In order to estimate the time taken to reduce a knowledge base it would help to understand the intrinsic nature of the problem. In a hypothetical knowledge base with n inputs, we can put a bound on the total number of possible disjunctives that can be generated as being being less than every possible combination of inputs. This can be written as:

$$\sum_{i=1}^n \binom{n}{i}$$

$$= 2^n - 1$$

However this is an unrealistically high estimate. It is more reasonable to assume that only a subset of the possible expressions will be generated, let us say l percent. Based on work with AMOK, it is also reasonable to assume that most of these disjunctions will have a similar size, say k . In this case the expression simplifies to:

$$l \cdot {}^nC_k$$

Ginsberg estimated that the time to compute the reduction of a knowledge base was exponential in size. We can estimate the size of the exponential problem by using Ginsberg's data; with 50, 150, and 370 rules, reduced in 40 cpu seconds, 5 cpu minutes, and 10 cpu hours, respectively. Applying a best fit exponential curve, where t is the time taken to reduce the knowledge base, and r is the number of rules to be reduced, gives:

$$t = 0.21 \cdot e^{(0.022 \cdot r)}$$

Using this equation, an estimation of the full reduction of a 1,000 rule knowledge base is 940 years, which is clearly unpractical. The solution adopted in AMOK is to narrow down the section of the knowledge base examined to the area effected by the modification.

Reducing the effective size of the knowledge based that is examined is a typical approach to performance problems. However, search problems tend to be inherently exponential. Methods for increasing the performance of a search,

like reducing the search space, reduce the size of the exponent associated with the search, but do not change the exponential character.

The implementation of AMOK also suffered because the use of reversible code produced unacceptable performance of some very basic predicates. Some of these sections of AMOK should be rewritten for speed. Predicates like the low level predicate *SAME*, can be made to run faster, without sacrificing the flexibility required. In this case the code produced an exponential performance curve, where a better solution must exist. This is because *SAME* can be considered to be a special type of unification function, and the performance of these functions has been examined in detail, see for example: (Baxter76, Jaffar84).

Based on the above factors, I believe that undertaking research to speed up AMOK might result in a version that could begin to support a realistically sized knowledge base written in propositional logic. This belief is supported by experience in similar fields like Automatic theorem Proving and Algebraic Simplification, where successful systems have now been implemented. At that point, further study could be undertaken on some of the further ideas suggested by this thesis, such as what is the best method for the user interface to convey the rule modifications, and are coverage and dependency the best way to determine if a named node should be used in the rule reconstruction?

5.2 Extending AMOK

Because AMOK uses a very limited knowledge representation language, propositional logic, it is not possible to apply it to many real expert systems.

Ginsberg (Ginsberg88b) recognizing this issue, stated:

"While no general algorithm can exist that solves the problems of inconsistency and redundancy for knowledge-based systems having the expressive power equivalent to first-order logic, we hope to develop an efficient implementation of KB-reduction that will handle occurrences of individual variables in rules and other features of predicate logic not currently accounted for in KB-Reducer."

There are a number of individual enhancements that it may be possible to add to AMOK. Items like variables, predicates, and functions would all add to the power and usability of AMOK. Based on the performance issues in Ginsberg's work and AMOK, it is not possible to say if these enhancements could be made. However, it is still clear that systems like AMOK need to be developed to improve the ability to deploy expert systems.

5.3 Further Applications

This work could also be continued to investigate issues that are related to knowledge modification. This however depends on the successful implementation of a system that supports a richer knowledge representation language, as well as addressing the performance issues.

Application in machine learning.

As this system can automatically make modifications to a knowledge base, it would be useful to incorporate it into some machine learning systems. In general, machine learning systems are addressing the problem of acquiring new knowledge, not absorbing this knowledge into the knowledge base. A possible exception is in systems that 'learn by being told', where the focus of research tends to be on incorporating new rules. Current machine learning systems, can be broken into two groups, by the methods they use to incorporate new knowledge.

- 1 Assuming that the addition of new rules will not impact on the performance of existing knowledge base, therefore the knowledge is incremental. This is probably not a very realistic assumption, and so a system like AMOK could be used to help add this knowledge.
- 2 Knowing the rules all interact, then the learning problem is examined at a statistical level, and test runs are performed on the expert system to maximize its performance.

In both these cases the only method to determine the impact of any changes is to run a regression test suite against the modified knowledge base. This work offers a significant improvement by:

- 1 Identifying the section of the knowledge base which can be affected by the change.

- 2 Identifying the rules that were modified, so that the user will be able to understand the effect of the changes.

Compiled Expert Systems.

A limiting factor on the size of current expert systems is the speed they take to execute. This could be improved by compiling them into a reduced form, such as disjunctive normal form, and then use a compression scheme to allow quicker execution.

Automated establishment of concepts.

As discussed in Chapter 3, the establishment of named nodes representing concepts is normally assumed to require expert guidance. It would be possible to scan a knowledge base for patterns which frequently occur, and deduce that these are hidden concepts. This is similar to the rule models used in TEIRESIAS.

REFERENCES

- Baxter, L. *The Complexity of Unification*. PhD. thesis, University of Waterloo, 1976.
- Bowen, K. A. ; Kowalski, R. A. *Amalgamating Language and Metalanguage in Logic Programming*. Logic Programming (editors Clark, K. L.; Tarnlund, S. A.), 1982; pp 153-172.
- Clocksin, W. F. ; Mellish, C. S. *Programming in Prolog*. Springer-Verlag, 1984.
- Davis. *Computability and Unsolvability*. McGraw-Hill Book Company, New York, 1958.
- Davis, R. *Interactive Transfer of Expertise: Acquisition of New Inference Rules*. Artificial Intelligence; No 12, 1979; pp 121-157.
- Davis, R. *Meta-Rules: Reasoning about Control*. Artificial Intelligence; No 15, 1980; pp 179-222.
- Ginsberg, A.; Weiss, S.M.; Politakis, P. *Automatic Knowledge Base Refinement for Classification Systems*. Artificial Intelligence; No 12, 1979; pp 121-157.

- Ginsberg, A. *Knowledge-Base Reduction: A New Approach to Checking Knowledge Bases for Inconsistency & Redundancy*. Proceedings of AAAI88; Vol 2, 1988; pp 585-589.
- Ginsberg, A. *Theory Revision via Prior Operationalization*. Proceedings of AAAI88; Vol 2, 1988; pp 590-595.
- Jaffer, J. *Efficient unification over infinite terms*. New Generation Computing, 2, 1984; pp 207-219.
- Kahn, G. S.; Kepner, A.; Pepper, J. *TEST: A Model-Driven Application Shell*. Proceedings of AAAI87; 1987; pp 814-818.
- Loberg, G. M. *Expert Systems*. IEEE Globecom'88, Workshop on the Application of Emerging Technologies in Network Operations and Management.
- Moses, J. *Algebraic Simplification: A Guide for the Perplexed*. Communications of the ACM, August 1971, Vol 14, Number 8; pp 527-537.
- Nguyen, T.A.; Perkins, W.A.; Laffey, T.J.; Pecora, D. *Checking an Expert System Knowledge Base for Consistency and Completeness*. Proceedings of Ninth International Joint Conference on Artificial Intelligence, 1985; pp 375-378.

- Oppacher, F. *An Expert System Tool for Coping with Inconsistent Information*. International Conference on Databases in the Humanities and Social Sciences, 11-13 July, 1987.
- Peacocke, D.; Rabie, S. *Knowledge Based Maintenance in Networks*. IEEE Journal on Selected Areas in Communications; June 1988, Vol 6, Number 5; pp 813-818.
- Raatz, S.; Drastal, G. *Pattern of Interaction in Rule-Based Expert System Programming*. Computational Intelligence; No 3, 1987; pp 107-116.
- Shortliffe, E.H. *MYCIN: Computer-based Consultations in Medical Therapeutics*. American Elsevier, New York, 1976.
- Suwa, M.; Scott, A.C.; Shortliffe, E.H. *An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System*. The AI Magazine; Fall 1982; pp 16-21.
- Wilkins, D.C.; Buchanan, B.C. *On Debugging Rule Sets When Reasoning Under Uncertainty*. Proceedings AAAI-86; Fifth National Conference on Artificial Intelligence; vol.1; pp 448-454.
- Winston, P.H. *Artificial Intelligence*. Second Edition; Addison-Wesley Publishing Company. 1984.

END

0/2·0/6·92

FIN