INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700  800/521-0600
A tool for interactive verification and validation of rule-based expert systems

Jafar, Musa Jafar, Ph.D.
The University of Arizona, 1989
A TOOL FOR INTERACTIVE VERIFICATION AND VALIDATION
OF
RULE BASED EXPERT SYSTEMS

by
Musa Jafar Jafar

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF SYSTEMS AND INDUSTRIAL ENGINEERING
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA
1989
As members of the Final Examination Committee, we certify that we have read
the dissertation prepared by Musa Jafar Jafar entitled A Tool For Interactive Verification and Validation of Rule Based
Expert Systems

and recommend that it be accepted as fulfilling the dissertation requirement
for the Degree of Doctor of Philosophy.

Marcel F. Neuts
William R. Ferrell
A. Terry Bahill

Final approval and acceptance of this dissertation is contingent upon the
candidate's submission of the final copy of the dissertation to the Graduate
College.

I hereby certify that I have read this dissertation prepared under my
direction and recommend that it be accepted as fulfilling the dissertation
requirement.

Dissertation Director
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgement the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: ____________________________
ACKNOWLEDGMENTS

I would like to thank my advisor Dr. A. Terry Bahill for his help, patience guidance and support during my education here at the University of Arizona. I would also like to thank the members of my committee: Dr. Marcel F. Neuts for his encouragement, teaching philosophy and friendly conversations and Dr. William R. Ferrell to be remembered as the first Professor that I had a class from in the United States.

I thank my Mom and Dad who I missed a lot and I will give everything for their happiness. Last but not least I thank my wife Valerie for the spiritual support and her Dad, Charles Alexander, for his help in breaking up my long sentences.
# TABLE OF CONTENTS

LIST OF FIGURES ........................................................................................................ 8

LIST OF TABLES ........................................................................................................... 9

ABSTRACT ....................................................................................................................... 10

1. INTRODUCTION ....................................................................................................... 12

1.0 Background ........................................................................................................... 12

1.1 AI Objective ......................................................................................................... 12

1.2 Expert Systems .................................................................................................... 14

1.2.1 Inference Engine ............................................................................................. 15

1.2.2 Knowledge Base ............................................................................................. 16

1.2.3 Explanatory Interface ..................................................................................... 17

1.2.4 Knowledge Acquisition ................................................................................ 18

1.3 Knowledge Engineering ..................................................................................... 19

1.3.1 Expert's Knowledge ...................................................................................... 21

1.3.2 The Role of a Knowledge Engineer .............................................................. 21

1.4 Verification and Validation .................................................................................. 22

1.4.1 Verification and Validation Activities ........................................................... 23

1.5 Sources of Errors ............................................................................................... 26

1.6 Dissertation Organization .................................................................................... 26

2. LITERATURE SURVEY ......................................................................................... 28

2.0 Background ........................................................................................................... 28

2.1 Automatic Knowledge Acquisition Tools .......................................................... 29

2.1.1 TEIRESIAS/EMYCIN .................................................................................. 30

2.1.2 MORE/MOLE ............................................................................................... 34

2.1.3 ACQUINAS .................................................................................................... 37
2.2 Automatic Knowledge Verification and Validation Tools ......................... 39
  2.2.1 ONCOCIN Knowledge Base Checker ................................................. 39
  2.2.2 CHECK/ARC ....................................................................................... 40
  2.2.3 Expert Systems Validation Associate ................................................... 41
  2.2.4 Expert Systems Checker ....................................................................... 43
  2.2.5 Other Verification and Validation Techniques ................................ ..... 43
  2.3 Summary ................................................................................................. 43

3. PROGRAM ORGANIZATION AND KNOWLEDGE BASE PREPROCESSING .................................................. 45
  3.0 Background .............................................................................................. 45
  3.1 Definitions ............................................................................................... 46
  3.2 Why Automatic Verification and Validation .............................................. 48
  3.3 The Program Structure ............................................................................. 49
  3.4 The Preprocessor ..................................................................................... 50
    3.4.1 The Special Character Handler ......................................................... 51
    3.4.2 The Unclosed Comments Handler ..................................................... 52
  3.5 The Syntax Analyzer ............................................................................... 53
    3.5.1 Knowledge Trees .............................................................................. 55
    3.5.2 The Data Base .................................................................................. 56
  3.6 Summary .................................................................................................. 59

4. KNOWLEDGE BASE VERIFICATION ......................................................... 60
  4.0 Background ............................................................................................ 60
  4.1 The Syntactic Error Checker .................................................................... 62
    4.1.1 Use of Illegal Values ....................................................................... 64
      4.1.1.1 Utilized Versus Legal Values .................................................... 65
      4.1.1.2 Utilized Versus Concluded Values ......................................... 67
    4.1.2 Unused Legal Values and Questions .................................................. 70
    4.1.3 Errors in the Premises of a Rule ....................................................... 72
    4.1.4 Errors in the Conclusions of Rules .................................................. 75
  4.2 The Debugger .......................................................................................... 80
  4.3 Summary .................................................................................................. 82
LIST OF FIGURES

1.1 The Different Components of an Expert System ........................................ 15
1.2 The Development Life Cycle of an Expert System ................................. 20
1.3 Expert Systems VS Conventional Software Problem Type ...................... 23
1.4 The Development Life Cycle of Conventional Software ......................... 24
2.1 An Example of EMYCIN English and Machine Representation .................. 33
2.2 An OPS5 Production Rule ........................................................................ 34
2.3 An Example of Heuristics Used by MOLE .............................................. 36
2.5 An Example of Heuristics Used by ACQUINAS ...................................... 38
5.1 A Small Knowledge Base With Typical Errors ....................................... 88
7.1 A Knowledge Base That Asks Inconsistent Questions ............................ 112
LIST OF TABLES

4.1 M.1 Metapropositions and Their Usage .......................................................... 74

6.1 Errors Detected in Masters Projects and Commercial Systems ....................... 103

6.2 Errors Detected in Students Systems (Fall 1988) .......................................... 104

6.3 Errors Detected in Students Systems (Fall 1987) .......................................... 105

6.4 Errors Detected in Students Systems (Fall 1986) .......................................... 106

6.5 Errors Detected in Students Systems (Fall 1985) .......................................... 107
ABSTRACT

In this dissertation I address the problem of verification and validation of rule based expert systems. It is a problem knowledge engineers have to deal with while building their expert systems to ensure the reliability, accuracy, and completeness of their knowledge bases.

I also present the structure of a program that automates verification and validation. Interactive as well as Automatic Verification and Validation is valuable, especially when the size of a knowledge base grows and manual techniques are not feasible. It ensures the stability of the system and raises the confidence in its level of performance.

My program's name is Validator. While many of the techniques are generic, the code and implementation is specific to rule based systems and M.1. It checks knowledge bases written in the M.1 shell from Teknowledge.

M.1 is a PC-based software tool that can be learned easily even by individuals with little programming experience. It is primarily a backward chaining shell but it can be customized to do forward chaining. It is written in the C programming language. It can be interfaced with other languages such as Assembly language and with existing software like DBASE, graphics packages and spread sheets. It uses an English-like representation which is easy to read, understand and to prototype with. It is available for me and has been used in a senior level class for the last four
years. Over 80 student projects, five M.S. degree theses and reports and a funded expert system were written using M.I. They are available for developmental and final testing of the program.

The objective of this research is to make it easy for expert systems developers to build the right system right by proposing practical and simple methods for building verification and validation programs to insure the integrity and performance of large scale knowledge based systems.

In this dissertation I also present the structure and development of SRPRES. A residential energy savings advisor expert system that teaches utility customers how to effectively use their energy sources.
CHAPTER 1

INTRODUCTION

1.0 Background

In this dissertation I present a program that interactively verifies and validates an expert system's knowledge base, the program's name is Validator. Expert systems have recently emerged as successful applications of artificial intelligence (AI). An expert system is a computer program that interacts with a human in a consultation dialogue. Explaining his problem, providing suggested tests, and asking questions about the program's inferencing process, an end user receives an expert level advice concerning his problem. The knowledge base of an expert system is usually built through the laborious interaction with a domain expert who may not be able to communicate all of his knowledge. Building a correct, complete and reliable knowledge base is the big bottleneck in the construction and development of expert systems. In this dissertation I present some practical techniques that provide consistent methods toward solving that problem.

1.1 AI Objective

AI is a branch of computer science concerned with developing computer systems that exhibit the characteristics associated with human intelligence. One of
The objectives of AI research is to find problem solving techniques for ill structured problems. First let us define a well structured problem.

**Definition 1.1**: A well structured problem satisfies the following criteria: (1) The problem can be stated in terms of numerical variables, (2) the problem goals can be specified in terms of well defined objective functions and (3) the problem has an algorithmic solution (Newell, 1969).

A problem that does not satisfy one or more of the above conditions is classified as ill structured. Well structured problems with large search space such as chess playing, are also considered to be ill structured.

In the early stages of AI, researchers used conventional programs to implement their problem solving techniques. The knowledge about the problem domain and solution were mixed with the procedures used to solve the problem. The resulting code was hard to read, modify, or even test and debug. The General Problem Solver (Newell, 1980) is a classical example of that. One of the major breakthroughs in AI techniques was the separation of the knowledge about the problem domain from the inferencing procedures used to manipulate that knowledge. This separation helped programmers as well as experts to easily build, examine, modify, update, test and debug a knowledge base without mangling with the source code of the inferencing procedures.

AI applications include areas such as Computer vision, pattern matching, robotics, game playing, theorem proving and natural language understanding.
1.2 Expert Systems

Expert systems (a by-product of AI research in the area of knowledge based systems) are knowledge-rich computer programs that operate effectively in a specific and narrow domain. They enhance productivity by making expertise available where it is scarce. They use heuristics, judgemental, and common sense knowledge (which might be fuzzy or incomplete) to infer new facts, to explain their decision making process, and to draw conclusions about a domain specific problem. Fuzzyness or uncertainty in a knowledge base occurs either because of the randomness of the process or because of the lack of evidence or support for a certain conclusion at the time of inferencing (Henkind and Harrison, 1988). An expert system (figure 1.1) consists mainly of the following four modules:

1- An inference engine (shell) that contains the methods of plausible reasoning that controls the manipulation of facts and rules of the knowledge base.

2- A knowledge base that contains the facts rules and heuristics that captures an experts domain knowledge in a format that is plausible to the inference engine.

3- An explanatory interface that contains the explanation, trace and graphical facilities to support smooth interaction with the user.

4- A knowledge acquisition module that contains the strategies to be used to elicit the basic knowledge.
1.2.1 Inference Engine (Shell)

The shell is a computer program that reads, interprets and knows how to actively use a knowledge base. Inference engines are characterized by the match and
select rules methods they use, the rules execution methods they use and by which knowledge representation scheme they use.

The match and select strategy depends on the type of inferencing strategy, backward chaining strategy where rules are invoked based on their conclusions versus forward chaining strategy where rules are invoked on the basis of their premises. In backward chaining systems it is easy to find which rule is going to fire first, while in forward chaining shells the inference engine might try to make a pass over all the rules to decide which one is going to succeed. Some inference engines, such as 1st-Class, use optimization techniques to insure minimum time access and parsing of the knowledge base. The rule execution could be a stack (M.1), a queue with or without user assigned weights (CLIPS, OPS5), metarules (Teiresias) or best-first search (KAS). The knowledge representation scheme could be rule-based (M.1), frame-based (KEE) or object-oriented (Smalltalk).

1.2.2 Knowledge Base

A knowledge base is domain specific. It includes basic facts and inferencing rules. The knowledge base captures an expert's or group of experts' knowledge about the domain of the problem and the methods used to solve that problem.

Basic facts identify the basic states of a system, the rest of the knowledge come through basic facts. They are beliefs about the values of the attributes of the physical objects of the system. We arrive at other states of the system by reasoning about facts. So we acquire the picture that reflects the state of the system by form-
ing an inverted tree with basic states and facts forming the foundation and other states and facts being supported by reasoning that traces back to the basic states and facts of the system.

A rule consists of facts, demons and an action part. The facts and demons are called premises; the action part is called conclusion. Rules are used to capture the reasoning process and describe relations and methods of the knowledge domain. Demons are shell specific, they are used to modify non-basic facts that were attained during the inferencing process, or the inferencing process itself. Each rule contains information that captures a chunk of knowledge about the subject domain or the new state of the system if that rule happens to fire (Intermediate rules, which are created by knowledge engineers to speed the inferencing process, are excluded).

1.2.3 Explanatory Interface

The interface deals with the man-machine communication. It is usually built on top of a knowledge base. It provides the explanatory facilities and the vocabulary that expresses the knowledge to the end user. Researchers (Kopec and Michie, 1982) have concluded that the interface is difficult to build and needs a breakthrough in AI research in communication. Problems with user acceptance originate from lack of understanding of an integrated human-machine system design. The rejection of a computer system by an end user is attributed to a variety of factors such as, (1) the use of computers to perform tasks that were used to be performed by humans, (2) unfamiliarity with the computer technology, (3) fear of
expert systems replacing humans, "human-out-of the loop design" and (4) the belief that humans do better than machines. Knowledge engineers seem to forget or underestimate the users satisfaction aspect as well as the man-machine interaction aspect. Human factors methods should play a more important role in humanizing the relationship between the man and the machine rather than computerizing it. For example, when a user asks for more explanation in response to a question, M.1 will list all the relevant rules. This procedure is confusing and of little benefit to an inexperienced user. The knowledge engineer has to override that option by providing tailored explanations. Hence, to build a more effective knowledge-based system, the knowledge engineer has to develop a system that meets the needs and requirements of an end user; i.e., one that is easy to use, helpful and that creates a comfortable environment and reduces the mental workload of the user.

1.2.4 Knowledge Acquisition

Knowledge acquisition is the process of transferring knowledge from the expert, or source of knowledge, to the knowledge base. In other words, it is the process of capturing an expert's knowledge about a subject domain into structures that can be understood and interpreted by the control part (shell). These structures are called the knowledge base. Knowledge acquisition is still considered to be the bottle-neck in expert systems development, and it is likely to lose information during the knowledge acquisition process. Scientists (Brownston et al., 1985) described the goal of a knowledge acquisition system as to allow a domain expert to enhance
domain knowledge in a program with only a little help from a knowledge engineer.

1.3 Knowledge Engineering

The performance of an expert system and the resources consumed in its construction are highly affected by the knowledge engineer and his communication capabilities. The knowledge base is incrementally built through the collaboration of the knowledge engineer and the domain expert. Communicating with an expert is time consuming and highly subjective. In addition, important concepts might not be expressed in a given session. Another concern of the knowledge engineer is to ensure that all the requirements for building an expert system are satisfied. Some of these requirements are choosing the appropriate shell, proper inferencing strategy, correct knowledge representation scheme and the right user interface.

The development cycle of a knowledge base is iterative and incremental. It was described by a spiral model (Boehm, 1986) (figure 1.2) which starts with a set of initial requirements (R0) that describe the problem, followed by the initial design (D0) and knowledge base (K0) (the knowledge base is ready to run and be verified and validated (V0), then the specification phase that represent the initial prototype (P0) of the expert's knowledge. The prototype Pi+1 is an improved expanded version of the current prototype Pi and reflects the operational capabilities of the system at the stage i+1. The cycle of iterative prototyping is repeated and the knowledge base keeps expanding and being revised until the final prototype has been obtained,
verified, validated and tested against certain performance indices or figures of merit defined by the system engineers. The final prototype will then produce the final system with the final operational capabilities.

Because of the separation of the declaration and control parts of an expert system, the initial prototype PO is easy and fast to build and execute. It usually covers a subset of the system and demonstrates the adequacy of the expert system technology to provide a solution for the current problem.

Figure 1.2 The Development Life Cycle of an Expert System
1.3.1 Expert’s Knowledge

An expert’s knowledge has its foundations in the world around him, supplied by his senses, observations, and contact with that world. Experts build their knowledge through sensory contact, education and reasoning. Simple beliefs help experts gain more knowledge, generalize and build more complicated models of their area of expertise. An expert’s knowledge is usually unstructured, complicated and sometimes incomplete. Knowledge may not even be directly accessible by the expert, not only because he cannot verbally express it, but because he might not be aware of its effect on the knowledge domain of the problem. In constructing an expert system’s knowledge base, the knowledge engineer must know the states of the problem: which ones are basic and what suits them to be basic. He should know more about the non-basic states and how are they supported by reasoning from the basic ones.

1.3.2 The Role Of A Knowledge Engineer

In building an expert system the knowledge engineer has to answer four main questions.

1- How can the information given by the domain expert be refined?

2- How consistent is the expert?

3- Is the system being built right?

4- Is the right system being built?
Possible answers for the second question were published in a paper (Bahill et al., 1987). In this research I will explore verification and validation (V&V) techniques to answer the third and fourth questions.

1.4 Verification and Validation (V&V)

V&V is a method of testing and reasoning about the correctness of a knowledge base. It is designed to increase appropriate confidence in the level of performance of an expert system. The goal of a knowledge engineer is to build a system that satisfies all kinds of users. Usually we relax our objective by trying to show that the system satisfies its documented specifications. This is hard to accomplish, so we usually try to increase confidence in the system's level of performance. Verification determines if we built the system right, while validation determines if we built the right system. In other words, V&V is performed inside each phase and between phases of the development process to insure the consistency of correct interaction between the different components of the system and to insure the completeness and correctness of the product.

V&V of software is an expensive and time consuming process. It has been estimated that testing and debugging comprised about 80 percent of the cost of the NASA APOLLO project (Yourdon, 1975), 44 percent of the Saturn 1 project, and 50 percent of the Naval Tactical Data System (Boehm, 1970). Systems constraints such as criticality, error tolerance and budget determine how much V&V has
1.4.1 Verification and Validation Activities

Verification and validation of an expert system has its roots in conventional software validation and testing (Quirk, 1985; Deutsch, 1982; DeMillo et al., 1987; Andriole, 1986) such as validation of simulation models. Although conventional software problem type (figure 1.3) and development life cycle (figure 1.4) differ from those of an expert system (figure 1.3 and figure 1.2), both share the same verification and validation techniques and goals in term of reliability and acceptability of the end user system. The separation of logic and control, the metaknowledge embedded in the shell as well as the development cycle (spiral model) provide an excellent environment for easy V&V of a knowledge based system. The V&V activity can be divided into three separate activities or techniques, Formal, Dynamic and Static.

<table>
<thead>
<tr>
<th>Expert Systems</th>
<th>Conventional Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem is ill structured</td>
<td>Problem is well structured</td>
</tr>
<tr>
<td>Requirements are vague</td>
<td>Requirements are well defined</td>
</tr>
<tr>
<td>Problem solution is heuristic</td>
<td>Problem solution is algorithmic</td>
</tr>
<tr>
<td>Code is unstructured and declarative</td>
<td>Code is structured and sequential</td>
</tr>
<tr>
<td>or functional</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.3 Expert Systems VS Conventional Software Problem Type
Figure 1.4 The Development Life Cycle of Conventional Software

In the formal technique, the knowledge base is analyzed to ensure efficiency, stability and low order of complexity. Rules are checked for:
1- Proper ordering of premises to stop expanding the search tree if a premise is already guaranteed to fail.

2- Proper ordering of rules so that the rule with the smallest search tree will be spanned first.

3- Properly analyze the knowledge base structure and try to create dummy intermediate rules that might not have any physical meaning or relevance to the domain but are used as premises. This helps cut the search space and speed the inferencing process.

Proper ordering of premises and rules and creating intermediate states helps cut the search space and speed the inferencing process.

Dynamic techniques include running as many test cases as possible. Where an expert, a group of experts, or an independent organization that was not involved in the development of the project evaluate the system.

Static techniques are used to ensure that certain types of errors are not present in the system. These techniques are used to measure the adequacy and design of the system and to address the form and structure of a knowledge base without really checking the functionality of the system. They also check for referencing and instantiations and are effective in finding faulty logic design and coding errors. In this research, I attempt to develop static techniques and methods to automate the V&V process of rule-based expert systems during the development process.
1.5 Sources Of Errors

Testing a system merely tells whether or not a system has errors. It is not possible to find the cause of errors through testing alone. Testing usually detects two types of error: (1) errors related to the knowledge base and (2) syntax errors.

Knowledge base errors are caused by missing knowledge base items, dead end constructs, or syntactic and semantic errors. Three main reasons cause the missing items: (1) either the knowledge engineer did not ask the right questions or (2) the expert forgot to talk about the missing knowledge base items or (3) the expert underestimated the importance and relevance of a certain item to the decision making process. Syntactic errors are expert system shell specific. They occur because of misspellings, wrong instantiations of expressions values and missing metafacts.

Interactive verification and validation can efficiently solve all these problems except the missing knowledge base items. Scientists attempted to solve the missing knowledge base items by creating rules for every possible combination of expressions (Suwa et al., 1982).

1.6 Dissertation Organization

In chapter two I present a survey of the literature and techniques used to build correct expert systems. This includes automatic knowledge acquisition and automatic verification and validation. Chapter three describes the basic structure of Validator and presents low level syntax checking that Validator can perform on a
knowledge base. Chapter four discusses the techniques that Validator uses for the verification of the syntax and semantics of a knowledge base. Chapter five discusses the techniques that Validator uses for the validation of a knowledge base and its rules, facts and other constructs. In chapter six I present the results of testing Validator on the various available expert systems. Chapter seven is a summary of the work done and what can be done in the future concerning interactive verification and validation of rule based expert systems. Appendix A presents SRPRES, a residential energy savings expert system that I built for Salt River Project (Phoenix Arizona) and Western Area power Administration (Boulder City Nevada) through a $40,000 contract.
CHAPTER 2

LITERATURE SURVEY

2.0 Background

The builders of man/machine systems have the responsibility of ensuring the quality, performance and acceptability of their system by the end user as well as the future maintainability of the system, they must also insure that all the system requirements are satisfied through the system’s life cycle. For traditional software the builders usually have enough knowledge about the problem domain and constraints to do this, while in knowledge based systems the builders have to deal with the knowledge transfer problem, a subset of the larger knowledge building problem, and the mismatch between the machine representation of the problem and the human concept. Knowledge building is still considered to be the bottleneck that is slowing the progress of expert systems. The problems and shortcomings of the knowledge building process can be summarized in four major categories:

1- inability of the knowledge engineer to ask the right questions,

2- lack of knowledge from the knowledge engineer’s side about the domain and structure of the problem,

3- inability of the domain expert to correctly express his input-output behavior,
4- inability to guarantee the correctness and completeness of the knowledge base built from the extracted knowledge.

With the above problems in mind, knowledge base building and refining took two different directions; automatic knowledge acquisition (AKA) and automatic knowledge verification and validation (AV&V). Different tools were developed to aid in the knowledge extraction process and close the gap between the machine representation of the problem and the human concept. In this chapter I am going to survey existing tools in the two categories, the field is only twelve years old but much work has already been done.

**2.1 Automatic Knowledge Acquisition Tools**

The main objective of the researchers in this area is to get the knowledge engineer out of the loop and allow the expert system to communicate directly with the expert. These tools are limited in power and capability. They are not good for problems other than classification-type problems.

AKA tools are designed primarily for classification type problems. Techniques from cognitive psychology are used to elicit information and to understand how humans represent and operate on information about the external world. The main objective of these tools is to provide mechanisms for interviewing domain experts and building knowledge based system without the intervention of the knowledge engineer. The two main goals for the AKA process are to design comput-
able formalisms that can capture the kinds of knowledge needed to solve expert tasks and to make it easy for people to use these formalisms to represent and operationalize their expertise.

A lot of work has been done in the AKA area: TEIRESIAS (Davis and Lenat, 1982), EMYCIN (Melle, 1980), MORE (Kahn et al., 1985), MOLE (Eshelman et al., 1987; Eshelman and McDermott, 1986), SEAR (Brug et al., 1985), SEEK and SEEK2 (Politakis, 1985), Autointelligence ( Parsaye, 1988), ONCOCIN (Shortliffe et al., 1981; Suwa et al., 1982), ETS (Boose and Bradshaw, 1987) and ACQUINAS (Boose and Bradshaw, 1987; Kitto and Boose, 1987; Boose, 1986). In this survey I am going to cover the TEIRESIAS and EMYCIN, MORE and MOLE, and ACQUINAS systems.

2.1.1 TEIRESIAS/EMYCIN

TEIRESIAS (Davis 1976) evolved from the MYCIN (Buchanon and Shortliffe, 1984) experiment. Its goal was to develop an automatic debugging tool for the MYCIN system, so that experts can communicate entirely in English with MYCIN through that tool. MYCIN started as a joint project between Stanford medical and computer science researchers. With its How/Why explanation facilities TEIRESIAS helped experts to judge, incrementally build and refine a MYCIN knowledge base. It also allowed them to track down errors and inconsistencies in the knowledge base that led to false conclusions and to modify, delete or add appropriate rules to fix those errors. TEIRESIAS used its knowledge about the problem
structure and presentation to tell if a rule fits into a knowledge base or not. It suggested modifications to rules provided by an expert after comparing them to rules that were already in the knowledge base.

Another system that emerged from the MYCIN and TEIRESIAS experiments was EMYCIN or Empty MYCIN. It is an expert system shell that has most of TEIRESIAS's capabilities. The EMYCIN AKA module consists of four main facilities:

1- An explanation facility to make the system more intelligible and clear by displaying the reasoning that led to each result and by helping to uncover incompletenesses and deficiencies in the knowledge base. The explanation module is invoked through the why and how functions during a consultation session with an expert. To examine the chain of reasoning, it traverses the nonexplored branches of the production rules goal tree (using a depth first search strategy) to provide explanations about its inferencing strategy and to display rules that are in the path to the current goal.

2- A debugger that checks for spelling errors and for consistency of syntax and semantics of new rules. It checks for erroneous interactions among rules as well as duplications and sub-rules. The debugger is activated when an error or a disagreement between the expert and the performance program is detected. It asks the expert's opinion on the validity of some of the currently invoked rules. It does not create new rules on its own, rather it listens and comments on the
correctness and consistency of the expert's new added rule.

3- A knowledge acquisition module that asks the user to insert a rule in Abbreviated Rule Language (ARL), then uses its knowledge to understand the rule by deciphering the English text (figure 2.1) of the rule and asks for questions and requests explanations if the terms are unfamiliar to the system. It rephrases the rule and displays the way it was understood by the system. If the knowledge acquisition module was invoked during the debugging session it checks for the validity of each premise in the current context. It also links the rule to a goal path, puts it in a category of rules and asks information about the rule (date of creation, author, cause of creation and comments). This module allows for multiple experts to work on the same knowledge base while keeping their differences.

4- A verification module that compares the system's results with a data base of stored correct results so that an expert can trace the reasoning process that led to a misdiagnosed case and find out why the performance system MYCIN, was unable to find the correct result and make the right recommendation.

† ARL constitutes an intermediate communication language between English and LISP.
### English form

**RULE50**

if 1) the infection is primary-bacteremia, and
   2) the site of the culture is one of the sterile sites, and
   3) the suspected portal of entry of the organism is the gastrointestinal tract

then there is suggestive evidence (0.7) that the identity of the organism is bacteroides.

### LISP form

**PREMISE**

\[-33-\]

\[
\text{($\text{AND} \ (\text{SAME CNTXT INFECT PRIMARY-BACTEREMIA}) \ \\
\text{(MEMBF CNTXT SITE STERILESITES}) \ \\
\text{(SAME CNTXT PORTAL GI)})}
\]

**ACTION**

\[-33-\]

\[
\text{(CONCLUDE CNTXT IDENT BACTEROIDES TALLY 0.7)}
\]

---

**Figure 2.1 An Example of EMYCIN English and Machine Representation**

In summary EMYCIN understands three types of knowledge:

1- **Object-level knowledge** which is domain specific. It is knowledge about the problem domain and its objects.

2- **Conceptual knowledge** about the objects (their attributes and values), the rules and the functions involved.

3- **Metalevel knowledge**, which is knowledge about specific representations. It is totally independent of the problem domain.

**TEIRESIAS** aids experts and knowledge-engineers in refining and adding to an existing knowledge base, making it feasible for the program to understand English. It stresses the incremental approach to knowledge building, what R. Davis calls transfer of expertise. **EMYCIN** builds a knowledge base from scratch, hence
the domain vocabulary is unknown to the system, making ARL an easier and faster form of communication between the system and the expert.

2.1.2 MORE/MOLE

An oil drilling expert system MUD (Kahn and McDermott, 1984), was built at Carnegie-Mellon University for NL Boroid. It takes OPS5 types of diagnostic productions to represent the knowledge and reason about it (figure 2.2). The handicrafting of MUD led the developers to MORE, a knowledge-acquisition tool on top of MUD. It provided mechanisms for interviewing an expert, extracting knowledge, refining the extracted knowledge and building an initial knowledge base.

(P name
  (object *name dte-host *transfer-type <var1>)
  (object *name dte-peripheral *transfer-type (<var2> <> <var1>))
->
  (object *transfer-problem *type mismatch))

explanation:
Data transfer problems occur if a data transfer host and a data transfer peripheral have different types of transfers.

Figure 2.2 An OPS5 Production Rule.

In summary, MORE acted as an information solicitor to build a knowledge base and evaluate its strengths, weaknesses and potential inconsistencies. It used numeric supports for associations provided by the expert to calculate support values for hypotheses. MORE used these support values to accept or reject a given
hypothesis.

The experience gained from the MUD and MORE systems led the researchers to build MOLE (a more general performance and automatic knowledge-acquisition system). MOLE is an expert system shell with AKA capabilities for heuristic classification problems. MOLE is a smart successor of MORE. It gets its power from paying close attention to the problem solving methods used by the performance systems. It tries to build a knowledge base with a minimal amount of information elicited from the expert by using simple heuristics such as: (1) every symptom is assumed to have a cause, (2) every symptom has an explanation, (3) explanations should not multiply beyond necessity, (4) every abnormal finding has an explanation and (5) two hypotheses should not be accepted if one is enough. Such heuristics allow MOLE to interpret the cause of every piece of evidence by assuming that a hypothesis exist that accounts for it. MOLE looks for the best candidate hypothesis by raising the confidence factor of the most likely to be true. This heuristic also allows MOLE to build a knowledge base with as little intervention from the expert as possible.

MOLE builds a knowledge base in two phases. First MOLE, the AKA system, gathers information by asking the expert to provide (1) lists of events to be classified, (2) type of each event (observed or inferred), (3) direction of association between events and (4) associations between these events. If the expert cannot decide the direction of links between events MOLE tries to draw these links by
using two simple heuristics (figure 2.3). MOLE also uses its capabilities to provide positive and negative supports for each hypothesis. This made MOLE less dependent on user-provided support values and more dependent on reasoning about evidence.

HEUR1
If event E1 implies event E2
and whenever E2 is false rules out E1
then E2 is a symptom of E1.

HEUR2
If E2 is a symptom of E1
and E3 is a sibling of E1
then E3 is a symptom of E1

Figure 2.3 An Example of Heuristics Used by MOLE

In the second phase the expert invokes the performance program and provides test cases with known results for the system to analyze. If a mismatch occurs, the performance program tries to find if the diagnosis is reachable and looks for missing differentiations or missing covering hypotheses. MOLE tries to fix the problem by lowering or raising the likelihood of the hypothesis, combining two or more hypotheses, or covering the missing hypothesis.

In summary, MOLE understands types of knowledge that associate symptoms with hypotheses. It differentiates among hypotheses covering a certain symptom by asking for more explanations about each hypothesis and for conditions that strengthen the best link and weakens the rest.
2.1.3 ACQUINAS

ACQUINAS (Boose 1986) is an expanded version of Expert Transfer System (ETS). It incorporates ideas from Psychology (Personal Construct Theory) (Kelly, 1955) and knowledge based systems to assist in the knowledge acquisition process. Knowledge bases are created for KS300/EMYCIN, S.1 and M.1 from Teknowledge, Personal Consultant from Texas Instruments, LOOPS from Xerox, MRS from Stanford, KEE from Intellicorp and OPS5 from Carnegie Mellon University. It also provides the functions necessary for the specific shell (functions to handle certainty factors from the grid ratings). ACQUINAS is a group of knowledge acquisition, representation and reasoning tools sharing a common user interface (Dialogue Manager) but performing different tasks such as similarity analysis of traits, suggestion of additional triad formations and test review of a consultation. ACQUINAS interviews an expert and analyzes and refines the elicited knowledge. It builds the initial rating grid of traits by grouping traits into triads and comparing them to each other, builds an initial knowledge base (identification phase) and helps the expert explore the main concepts of the problem and discriminate among different traits. In addition it builds a tree from the knowledge base (conceptualization phase) and generate rules based on the rating grids and relationships in the tree (formalization phase). The knowledge base is built in hierarchies such as solution, trait, experts and case hierarchies. The grids from each hierarchy are attached by nodes.

The Dialogue Manager operates in three different modes: automatic, assis-
tant and off. When not in the off mode, it uses heuristics (figure 2.4) for the expertise elicitation phase to control the interaction between the system and the expert. It asks the expert for more explanations when needed to fill the grids and analyzes similarities between objects. In the assistant mode the Dialogue manager has a learning capability, it keeps a file of the expert’s responses to different questions, compares them with other experts’ history files and tries to extract heuristics by matching similarities in responses.

```
if fill-in-rating-grid-complete
and number-of-elements < 6
and number-of-traits > 3
and element-similarity > 0.9
then recommendation = (analyze similarities elements)
because "there is a high degree of similarity between the two elements in the knowledge base Analyze similarities to distinguish them."
```

Figure 2.4 An Example of Heuristics Used by ACQUINAS

In summary, the model of problem solving currently used in ACQUINAS is that of multiple knowledge sources (experts) that work in a common problem-solving context (case) by selecting the best alternatives for each of a sequential set of decisions (solutions). Alternatives at each step are selected by combining relevant information about preferences, constraints and evidence (reasoning).

ACQUINAS is effective if used with other AKA tools such as TEIRESIAS. It is also effective in demonstrating the feasibility of using an expert
system to solve the problem in the early stages of prototyping.

2.2 Automatic Knowledge Verification and Validation Tools

The other direction the research in knowledge base building and refining took was to keep the knowledge engineer in charge of the knowledge transfer process and provide him with the necessary tools to build complete and high quality knowledge based systems. These tools act as an aid to the knowledge engineer. They help the knowledge engineer debug and check the syntax and semantics of a knowledge base to insure its correctness and completeness. In this section I will survey most of the tools that have been developed, or are under development, in the area of AV&V.

2.2.1 ONCOCIN Knowledge Base Checker

ONCOCIN (Shortliffe 1979) is an Oncology protocol management system designed to describe appropriate treatment for cancer patients. It is a forward chaining system equipped with special hardware and software interface to ensure ease and simplicity when used. The ONCOCIN knowledge base checker (Suwa et al., 1982) is a program that has been developed, devised and tested within the ONCOCIN system to ensure consistency, effectiveness and completeness. It also insures that the set of rules comprehensively spans the ONCOCIN knowledge base.

The checker examines a set of rules and partitions them into disjoint sets based upon their conclusions. It also makes a table of all possible combinations of
attributes used in the condition parts of rules, assuming a rule for each possible com-

bination that apparently led to semantic inconsistencies in some of the rules. The system also checks for redundant rules, conflicting rules and subsumed rules. The context and conditions of two rules are also examined. If both succeed in the same situations and have the same conclusion, they are considered to be redundant; if they conclude different values for the same set of parameters they are considered to be in conflict. When two rules have the same conclusion, but one of them contains extra condition clauses, one rule subsumes the other.

Runs of the rule checker on the ONCOCIN system suggested missing rules and detected conflicts, redundancies and inconsistencies between the rules. That was considered valuable to the knowledge engineers and experts that were involved with the ONCOCIN project. The system also produced tabular display of rule sets that made it easier for the developers to read and interpret the knowledge base.

2.2.2 CHECK/ARC

Check (Nguyen et al., 1985; Nguyen, 1987) is a program that verifies the consistency and completeness of expert systems built with Lockheed Expert System Shell (LES). It checks for redundant and conflicting rules, it also checks for unreachable and missing clauses, and dead end rules.

The system also produces dependency charts that show the rules interactions. It produces if-if tables that captures all the results from the comparisons
between the premises of the different rules to check for conflicts, then-if tables that captures all the results of the comparisons of the conclusions and the premises of the different rules to check for circular chains and then-then tables that captures all the results of the comparisons of the conclusions of the different rules to check for subsumed rules and unnecessary clauses. The dependency charts are helpful in showing rule interactions during the development process when knowledge engineers build on and update a knowledge base.

The same authors implemented the same techniques into another program named ARC (Nguyen et al., 1987) that is used to validate knowledge bases written using Automatic Reasoning Tool (ART) from Inference Corporation.

2.2.3 Expert Systems Validation Associate (EVA)

EVA (Stachowitz et al., 1987) is a metaknowledge based system shell that provides verification and validation capabilities for knowledge based systems in use at the Lockheed Artificial Intelligence Center at Austin Texas. It is written in QUANTAS Prolog and covers ART and LISP knowledge based systems. EVA has two major components or modules:

1- The analyzer module, which uses conversion algorithms to translate applications that are specific to a certain object shell (currently ART and LISP) into structures and schemas that can be stored, understood and compared in the metashell itself so they can be used by the validation modules.

2- The validation module, which has three checkers:
a- A logic checker that looks for inconsistencies (direct and indirect), numeric completeness of values, inconsistencies under generalization hierarchies and compatibility and conflicts.

b- A structure checker that checks for reachability, redundancy, duplications, subsumptions and relevance (direct and indirect) between rules. It also checks for cyclic rules.

c- A semantic checker that checks for violations of the semantic constraints of the object shell and consistency and agreement of these constraints with the metaconstraints of the metashell (EVA) itself.

Future work on EVA involves extending its capabilities in two different directions:

1- To handle applications written in other shells such as KEE and OPS5.

2- To develop the following modules: (1) an omitted rules checker to check the set of facts and rules for semantic information, (2) a rule proposer to cut the search space by proposing intermediate rules on conditions that are used in more than one rule, (3) a behavior verifier that analyzes the interactions between the different system components and analyzes the outputs and the behavior of the system with respect to inputs and the external interface and (4) a rule firing control checker to enable the user to have more control on the order in which rules fire.
2.2.4 Expert Systems Checker (ESC)

ESC (Cragun, 1987) is a decision-table based processor for checking completeness and consistency in a rule based expert system. The authors use decision tables to validate a knowledge base. Creation of the decision tables is based on similarity in logic between the rules. The tables are then checked for ambiguities and redundancies between the rules and for the completeness of the knowledge base.

2.2.5 Other Verification and Validation Techniques

Other ideas for verification and validation knowledge based system were also presented by (O'Keefe et al., 1987; Balci and Sargent, 1981). They suggest using simulation models validation techniques to evaluate the performance and consistency of a knowledge based system. Unless the knowledge engineer is able to come up with quantification techniques to compare the system's and human's performance, the effectiveness of such techniques and the extent of their application is domain dependent.

2.3 Summary

In searching for a general purpose expert system shell, several shells with limited AKA and AV&V capabilities were constructed. Several other AKA and AV&V tools were also constructed to support existing Shells. With these tools, the builders of an Expert System can ascertain that the system is bug-free. They can also guarantee the internal consistency of the system, its modules and structures. Yet the
acceptability of the System in a certain environment is far from achievable; testing the performance and acceptability of the delivery system is still an ad hoc process, and quantitative methods are yet to be developed.

In this dissertation I will present **Validator**. A program that automatically verifies and validates the internal consistency of a rule based expert system. The implementation is specific to M.1 and the ideas are based on the ONCOCIN Knowledge-Base Checker (Suwa et al., 1982).
CHAPTER 3

PROGRAM ORGANIZATION AND KNOWLEDGE BASE PREPROCESSING

3.0 Background

The most difficult tasks in expert systems design are verification, validation and testing. These activities are usually carried out to check the correctness of syntax and semantics of a knowledge base system. They are also done to ensure the following three system requirements that are usually developed during the initial requirement development phase of a project (Wymore, 1976, 1988).

1- The compliance of the system with all the requirements established for it during the requirements development phase at the beginning of each prototyping phase of the development cycle of the system (figure 1.2). These requirements include input/output functional requirements, input/output performance requirements and system/software compliance requirements.

2- The conformance of the system to the design from which it was built. The design used to build the system is usually picked from the set of feasible designs proposed for it. It also conforms to software, hardware, bioware, environmental and allocation of resources requirements.

3- The evaluation of the system by an expert, a group of experts or an independent
organization to determine the acceptance of the final system by the customer.

3.1 Definitions

Different researchers use different definitions for verification, validation and testing that share the same final objective "production of quality end user systems". The first group of researchers considered debugging as the main activity with verification; validation and testing as subactivities of debugging. The second group considered validation to be the main activity and rest of the activities to be part of validation, dynamic validation includes testing and static validation includes checking the correctness of the syntax and the semantics of the system. The third group considered testing to be the main activity, black box testing includes running test cases and white box testing includes syntactic and semantic checking of the system. Examples of these philosophies can be found in (Quirk, 1985; Deutsch, 1982; DeMillo et al., 1987; Andriole, 1986).

In this dissertation I will not follow the definitions of any of these groups. First, I am going to define what I mean by verification, validation and testing. That will help in defining the framework of this research and it will also draw the guidelines for the verification and validation routines that were developed to complete and fulfill the requirements of this research. In other words, these definitions will serve as the guidelines or the requirements to be fulfilled by this research.

Definition 3.1: Verification means building the system right: that is ensuring that the system correctly implements the specifications. It determines the conformance of the
knowledge base to its design requirements and with the software syntax from which it was built. It also guarantees the consistency of product at the end of each phase with itself and with the previous prototypes. In other words, it guarantees the honest and smooth transition from one prototype to another.

**Definition 3.2:** Validation means building the right system: that is writing specifications and checking performance to make sure that the system does what it is supposed to do. It determines the correctness of an end product, conformance of the output with the customer's established requirements and completeness of the system.

**Definition 3.3:** Developmental testing means running test cases to explore the system and expose errors.

**Definition 3.4:** Final testing or evaluation means running as many test cases as possible and watching the input-output behavior of the system to evaluate its performance. The performance evaluation of a system is a complicated process that has to take many factors such as input-output behavior of the system, interface, environment and cost of operation and maintenance of the system into consideration. It is a decision making process that might include statistical hypotheses testing and building confidence intervals. Some researchers suggest final testing be handled by independent organizations that were not involved in the project. This type of testing is helpful in evaluating the final product if it was handled within the guide lines of the system requirements that were established during the initial requirements development process of the system.
In a typical expert system verification is done first, then validation, then developmental testing and finally final testing and evaluation. Verification, validation and developmental testing are carried out during and at the end of each prototyping phase. A verified and validated system is far from being accepted. Verification and validation usually determine the compliance of the system with all the requirements and conformance of the system to the design from which it was built. Early or developmental testing strengthens the confidence of the knowledge engineer and the expert in the system and its performance. Final testing and evaluation of the system has to be done to determine the acceptance of the system.

3.2 Why Automatic Verification And Validation

In traditional verification, validation and testing the knowledge engineer reads the knowledge base looking for syntactic and semantic errors, then he runs as many test cases as possible and watches for mismatch between the system’s output and expert’s (expected) concepts or opinion. This is time consuming, error prone and does not guarantee finding all the mistakes. Development testing cannot tell what went wrong in the system. It can only tell if the system has errors or mistakes. It does not tell where an error occurred or what caused it. It is a hit or miss activity because it is almost impossible to find test cases that cover every possible situation of the system especially for large systems.
3.3 The Program Structure

In this chapter I will describe a program that automatically verifies and validates a rule based expert system. The ideas are generic and general, they can be applied for verification and validation of any rule based expert system. The implementation is specific to the M.1 shell from TEKNOWLEDGE. The program's name is *Validator*. It reads a knowledge base and automatically checks it's syntax and semantics for possible errors and bring them to the attention of the knowledge engineer. The program acts as an aid to the knowledge engineer. It does not have the capability of fixing the errors. The task of fixing errors is left to the knowledge engineer. The program consists of 6 different modules or components:

1- a preprocessor module,
2- a syntax analyzer module,
3- a syntactic error checker
4- a debugger,
5- a chaining thread tracer
6- a knowledge base completeness module

In this chapter I will describe the general framework of the program and the first two modules.

The knowledge base is first run through a spelling checker, then through the preprocessor where some low level syntax checking is carried out. The processed knowledge base is then sent to another new file where comments are checked and deleted. In the new file the knowledge base words are separated by
blanks. The new file is then processed one word at a time and information about the knowledge base is organized and stored into trees and link lists. The trees and link lists are then checked by the different verification and validation modules (modules 2, 3, 4, 5 and 6) for syntax compliance, completeness, redundancies and other aberrant behavior of the knowledge base.

3.4 The Preprocessor

The preprocessor is a program that scans a knowledge base one character at a time. The stream of characters forming the knowledge base are read from left to right.

Definition 3.5: The set of separators \( S = \{ , [, ], >, <, =, +, -, /, \text{comma} \} \) are the characters that can be used to separate two strings or words in addition to the blank character and the end of line symbol (EOL).

For each character \( c \) in \( S \), the preprocessor replaces \( c \) by the string " \( c \) " i.e. adding a space before and after each character of \( S \). This speeds the reading of the knowledge base by the syntax analyzer program, which in turn groups the words of the knowledge base into suitable expressions in parse trees and link lists.

The preprocessor has two main features: (1) it handles special characters such as end of word and end of line, and (2) it flags potential unclosed comments.

\[ \uparrow \text{ An expression consists of an object and its set of attributes } \]
3.4.1 The Special Character Handler

Different special characters are used by different word processors to signal the end of line or end of word. WordStar for example, treats the last character of every word as a special character. It signals its most significant bit to one by setting the last bit of the character one. Whereas UNIX based systems append the \0 character to the end of each word. Such signaling of special characters might get in the way of the expressions comparison process used in the syntax analyzer module and string matching routines used in the verification and validation modules of the program.

Using bitwise operators, the preprocessor strips the special characters before performing any object grouping in the knowledge base. This treatment of special characters prevents unexpected behavior of the program, such as storing expressions in the wrong branches of the parse trees or even the wrong parse tree.

Another potential problem the preprocessor handles is the end of line character, or characters. WordStar signals a new line as a sequence of two characters, a carriage return (CR) followed by a linefeed (LF). Whereas UNIX based systems (on which the program was originally developed) require only the LF as an end of line character. After the most significant bit of each character is reset, the Preprocessor also checks for an LF character after each CR character. If an LF is encountered then CR is ignored and LF is processed. Otherwise the character is put back into the knowledge base file so that it will be read again.
3.4.2 The Unclosed Comments Handler

Comments are used to make a knowledge base more readable and self explanatory. Commenting a knowledge base is a good practice in structured programming, it helps in future updating and maintenance of a knowledge base, especially when the builders and the maintainers are not the same. Comments are also the source of a simple but highly damaging error in a knowledge base. Compilers as well as expert systems shells usually flag as error an unclosed comment, if it was the last comment in a program or a knowledge base. However if the unclosed comment was followed by another comment all the information between the two comments will be ignored and the two comments are wrapped on to be one long comment. In other words, a typographical error such as replacing */ by a *? or 8/ will cause the part of the knowledge base between the current unclosed comment and the next comment to be ignored. Such an error might go undetected if none of the developmental test cases exercise that part of the knowledge base. The Helper expert system of the SIE department was the only finished system where such an error was detected. Due to a typographical error the end of comment string */ was replaced by *?. This error (since the open comment was not the last comment of the knowledge base) escaped detection by the shell and caused a whole block of the knowledge base to be ignored. The preprocessor performs simple syntax checking on the knowledge base characters to delete this type of error. Potential unclosed comments are flagged and the knowledge engineer is warned.
After the comments are processed and checked, they are deleted from the knowledge base file. Deleting comments cuts the size of the knowledge base to be read by the syntax analyzer which produces faster processing.

3.5 The Syntax Analyzer

The knowledge base is then read one word at a time, words are then grouped into objects which are the basic structures of a knowledge base. The process of grouping words into objects is done in an object processing routine. Word grouping to form objects is not obvious because the M.I system allows words to be treated as operators. For example, declaring \textit{of} as an infix operator will allow the knowledge engineer to say \textit{color of car} instead of \textit{color-of-car} In other words infixing allows a word to glue the previous and next word to form the new object. Similarly prefixing glues the next word to the operator, and postfixing glues the previous word to the operator to form the new object. The routine takes a word and returns the object and the next word. Returning the next word helps in checking the syntax of the knowledge base. Each word is then checked to see if it is an M.I reserved word, a shell specific statement or an operator declaration.

Usage of reserved words for purposes other than what the system intended might be damaging and have serious side effects on the knowledge base. In the following example taken from a Home Appraisal Advisory System (HAA) the knowledge engineer declared the reserved word \textit{or} to be an infix operator.
the infixing of or changed the structure of all the disjunctive rules in the knowledge base. The above rule now has two premises and a conclusion. It will succeed if a value Casas Adobes Park was found for the expression name of subdivision and medium or old was found for the expression age of home-cap.

Such usage of reserved words might have serious side effects; it modifies the whole structure of the knowledge base. The knowledge engineer can never use a disjunctive rule through the whole system without the rule being misinterpreted by the expert system shell. Undoubtedly such usage of reserved words will draw errors in updated versions especially when the maintainers are not the builders of the system.

After the knowledge base is checked for the reserved words, objects their attributes and the corresponding values are then collected into trees and link lists. These trees and link lists preserve the declarations and the knowledge contents of each expression and the knowledge base. For example, different trees are created for different items of the knowledge base. These trees and link lists are then to be displayed for the knowledge engineer and be used by the other modules of Validator during the analysis of the knowledge base.
3.5.1 Knowledge Trees

Each node of each tree consists of (1) an object string (*object) to carry the object's name, (2) a structure (*attribute_values) of two link lists to carry the attributes (*attributes) and the list of corresponding values (*values), and (3) the links to the next siblings (*left and *right) of the tree. The following example† shows a tree declaration:

```c
struct ATTRIBUTES {
    char *attribute;
    struct ATTRIBUTES *next;
};

struct ATTRIBUTES_VALUES{
    struct ATTRIBUTES *values;
    struct ATTRIBUTES *attributes;
    struct ATTRIBUTES_VALUES *next
};

structure TREE{
    char *object;
    struct ATTRIBUTES_VALUES *attribute_values;
    structure TREE *left,*right;
};
```

The *attribute_values structure of the TREE structure is a link list that is used to accommodate the different attributes and values of the same object. Such a structure enabled the program to store an object with different attribute structures in the same node. Trees were created to handle the facts and their values, the messages, the

† The example as well as the program were written in the C programming language.
expressions that appear in premises, the goals or initial data statements, the legal values statements of expressions and their values, and the whenfound and when-cached statements (left and right side). The tree representation of the knowledge base items is easy to parse for the needed comparisons. Also, using the same representation helped in keeping the size of the code small, since different routines were used to do parsing, comparisons, string matching and building the trees. Note that the whenfound and whencached statements were used by the shell to force forward chaining or alter the inferencing procedure whenever a certain type of knowledge was concluded. I treated these statements exactly like rules, with the left side of the statement as the conclusion and the right sides as the premises.

3.5.2 The Data Base

Spelling errors and some of the syntactic and semantic inconsistencies are easy to detect by a human if these expressions are displayed systematically. Manual detection of inconsistencies and errors depends on the knowledge engineer's experience and skills. However inconsistent items are easier to detect by a human if they are close to each other. Humans can detect direct inconsistencies between pairs better than collections. Machines are better at detecting global and indirect inconsistencies. So, the trees of goals and initial data statements, the lists of the infix, prefix and postfix declared operator, rules' premises and rules' conclusions, left sides of whenfound and whencached statements, right sides of whenfound and whencached statements, the lists of all possible questions, the lists of facts and displayed
text messages, and lists of all declared legal values and the expressions to which they correspond to are displayed to the knowledge engineer for checking. These displays are shown in alphabetic order, they are easy to read and compare. For example, during this type of manual debugging, it is easier for a knowledge engineer to find a misspelled word. In the following example (from the list of all possible conclusions of the Advice system) it was easy to see that *demester* is a misspelled word and the rule related to it was not going to fire when it was listed in the form:

```
demester(SEM)
semester(SEM) = checked
semester(SEM) = ignored
semester(SEM)
```

It is also easy to find strange or odd instantiations of expressions or an illegal usage of a certain expression. In the following example (from the list of premises tree of the Helper System) it was easy to see that the value *yes* that was given to the expressions *restart* and *with-what* is a mistake.
Another important aspect of this data display is that it is easier to read and compare than reading the whole knowledge base or portions of it. If the knowledge engineer were to read the whole knowledge base it would be hard for him to remember his syntax, word abbreviations and the legal values provided for a certain expression. For example, it would be hard for him to remember if he had used a hyphen or an underscore when he abbreviated a couple of words. In the following example from the list of facts of a Car Troubleshooting System a misspelled control statement is listed with facts and is easily detected.
intake-type(dual-one-barrel,dual-port) = type-five-intake.
intake-type(dual-one-barrel,single-port) = type-five-intake.
intake-type(single-two-barrel,single-port) = type-seven-intake.
intake-type(dual-two-barrel,single-port) = type-seven-intake.
nonautomaticquestion(intake-type(carburetion,head-port))

The knowledge engineer wrote nonautomaticquestion(intake-type(carburetion,head-port)) instead of noautomaticquestion(intake-type(carburetion,head-port)) in order to prevent the system from asking a question if values for the expression intake-type could not be concluded internally by the system. The misspelling was easy to find because the misspelling of noautomaticquestion caused it to be listed as a fact.

3.6 Summary

Errors are characterized by the time of their detection, the type of faults they cause and their effect on the knowledge base. I found that fixing low level syntactic errors is valuable to the knowledge engineer. Low level syntactic errors and misspelling might cause the expert system shell to misinterpret a knowledge base and consequently alter its syntactic structure. The detection of such errors saves the knowledge engineer and his expert a lot of time frustration and grief. It also saves them time otherwise to be consumed in debugging and looking for typographical and spelling mistakes. These types of mistakes were committed a lot by the builders of the expert systems that I tested. These errors are diversified and range from the simple mistake that has no effect on the knowledge base to the ones that can alter the whole system structure.
CHAPTER 4

KNOWLEDGE BASE VERIFICATION

4.0 Background

In this chapter I present general techniques for the automatic verification of an expert system knowledge base. These techniques help a knowledge engineer refine and debug a knowledge base and aid him in finding syntactic and semantic errors in it before validating the knowledge base and running test cases on the system. This chapter describes the Syntactic Error Checker (the third module of Validator) and the debugger (the fourth module). The techniques described in this chapter will rapidly produce tabulated results of the vocabulary (e.g. terms, goals, legal values etc) used by the knowledge engineer to represent the expert's knowledge. Such results make it easier for the knowledge engineer to understand, analyze and debug the knowledge base, they also speed the development of a system by showing errors that might not be found by developmental testing only.

If developmental testing is to be the only means of verification and validation of a knowledge base, the knowledge engineer has to run as many test cases as possible on the system, then look for symptoms of faults in the knowledge base by observing the output of the system. Such symptoms are summarized in the following seven categories:
1- Too much output or output overflow, such as flooding the user with redundant or correct, sets of recommendations and actions.

2- Inconsistent output, such as telling the user to perform a set of conflicting actions, or producing a set of contradicting recommendations.

3- Incorrect output, such as providing the wrong outputs for a given set of inputs.

4- Incorrect processing such as inferencing the wrong branches of a knowledge tree or asking for inconsistent or redundant information.

5- Less than expected output.

6- Premature exit from the system due to a run time error that caused the system to halt.

7- An infinite loop in the system.

Although developmental testing does not find the location and the reason for an error, the techniques that are presented in this chapter and the next are not intended to decrease the importance of developmental testing of a knowledge base. They are rather tools to speed the development of a system and give the knowledge engineer and his expert(s) more time to concentrate their work and efforts on areas that need real human attention during the development process of an expert system. Furthermore, automatic verification increases the confidence in the final product. In theory a verified system is a system without logical errors, a factor to be taken into consideration during final evaluation the system.
4.1 The Syntactic Error Checker

The first verification component of Validator is a "smart" syntactic error checker. Syntactic errors are common in expert systems knowledge bases, they are usually caused by misspellings, typographical errors or ill formed structures of the knowledge base. The syntactic error checker looks for syntax that, although legal (illegal usage of reserved words for example), produces unspecified behavior by the shell. Such errors usually escape detection by the shell and the knowledge engineer. Validator detects these errors in the context in which they occur. Syntactic errors also occur when the knowledge engineer lacks knowledge about the shell, its environment and the ability to recover from knowledge base errors.

Before I start describing the components of the syntactic error checker, I need to make some generalizations concerning the shell (M.l). Knowledge base systems built with M.l are designed around a knowledge base of facts, metafacts, rules and statements. Facts provide values for expressions, they are usually used to represent static knowledge that is not appropriate to represent in rules. Metafacts provide information useful in determining the values for an expression such as questions and legal values. Rules (example 4.1) describe the domain judgmental and heuristic knowledge and the control knowledge of the system, they are knowledge base entries of the form
Example 4.1

Statements are special types of metafacts that are used to interrupt the backward chaining process of the system. They are used to carry out extra tasks during the inferencing process. The inference engine usually checks the validity of these statements whenever an entry is added to the knowledge base or value is to be sought by the system. Proceeding Example 4.2 shows typical statements, where LIST is a list of one or more propositions or expressions.

```
if premise
   and premise
   and ....
then conclusion.
```

Example 4.2

M.1 primarily uses a backward chaining reasoning process to reach conclusions. The reasoning process starts with a goal statement whose value(s) are to be sought by the inference engine. The inference engine starts by searching the cache, then sequentially searching the knowledge base for relevant facts, metafacts and conclusions of rules in the order of their occurrence in the knowledge base. M.1 can be programmed to have forward chaining capability via the 'whenfound or
whencached statements. These statements form a set of high priority goals waiting for their left sides to be valid so that they can interrupt the backward chaining process by forcing the system to seek the truth value of their right side premises.

In this chapter I am not going to differentiate between a rule, a whenfound or a whencached statement. Whenever I mention premises of a rule, I mean the condition part of a rule, the right side of a whenfound or the right side of a whencached. Whenever I mention the conclusion of a rule I mean the conclusion of a rule, the left side of a whenfound or the left side of a whencached statement.

4.1.1 Use Of Illegal Values

Expressions get their values from facts, from a user’s response to a question or from the conclusion of a rule. The order of firing of these constructs depend on their position in the knowledge base and the type of the expression (multi- versus single-valued expressions) they use. The problem related to the use of illegal values is due to the mismatch between the utilized and legal values or the utilized and concluded values of expressions of the knowledge base. The legal values are the values that appear in a legal values (legalvals) metafact of an expression. They are attached to the question metafact of the expression and are the set of responses defined by the knowledge engineer to be acceptable for a question from the system. The utilized values are the values that appear in premises of rules. They are the values of expressions that allow premises that use them to be evaluated to true. The concluded values are the values that appear in the conclusions of rules. They are the
values that are set for an expression when a rule that uses that expression succeeds.

4.1.1.1 Utilized Versus Legal Values

Providing legal values for an expression ameliorates typing errors in response to a question. If no legal values were provided, the system would take any user's response as an answer. So typographical errors as well as wrong responses might escape detection. Even if detected, they are hard for a user to recover from without restarting the whole system. Providing effective error recovery procedures is time consuming and sharply increases the size of the knowledge base. The knowledge engineer has to build procedures that will backtrack virtually to every branch of the knowledge base tree that has been affected by the wrong response, allow the user to recover from his error, and then return him to the same place in the knowledge base before the error was discovered. Partial solutions for the error recovery problem were provided in SRPRES (APPENDIX A) via an "oops" construct that enables the user to recover from errors. Providing legal values will also allow an end user to abbreviate his response; for example, he can type y instead of yes as an answer. So using legal values provides a safeguard against typographical errors and helps in abbreviating long answers. Providing legal values will not prevent the knowledge engineer from erroneously providing an illegal value in the premise of a rule as a utilized value or in a fact. M.1 does not check the knowledge base to ensure that only legal values have been used. M.1 only checks user responses to questions to see if they match legal values specified by the knowledge
engineer. Legal values are related with questions and not with rules premises or facts. Such usage of illegal values is very common and often found in a knowledge base. It usually forces a conjunctive rule that uses such a value to fail. Example 4.3 from the Common Pediatric Rashes Diagnosis System, shows that the two rules concluding values for rash-identity will fail because illegal utilized values were used.

question(location) = 'Where on the body is the rash located?'.

legalvals(location) = [around-mouth, between-fingers, follows-nerve-root, genitals, gingiva, lower-extremities, palms, pharynx, scalp, soles, site-of-previous-injury, trunk, upper-extremities].

if type = scaly
   and location = feet
   and ..
then rash-identity = athletes-foot cf 90.

if type = vesicular
   and location = posterior-pharynx
then rash-identity = viral-stomatitis cf 95.

Example 4.3

The second premise location = feet of the first rule and the second premise location = posterior-pharynx of the second rule have illegal utilized values. The expression location will be tested against the utilized values feet and posterior-pharynx. Since feet and posterior-pharynx are not members of the set of legal values provided for the expression through the M.1 legalvals statement, such a construct will cause the premise of the expression location to be evaluated to false, that
will cause the rule to fail. Consequently, all the rules that use the conclusion of the first rule rash-identity = athletes-foot as a premise will fail. Such errors will also stop the system from seeking the rest of the premises that come after the false premise of the rule.

4.1.1.2 Utilized Versus Concluded Values

The second type of error related to legal values that might cause a conjunctive rule to fail is due to the mismatch between the utilized values in premises and concluded values of an expression in a rule. Example 4.4 taken from The Poker System, shows that the first rule will fail due to the mismatch between the utilized and concluded values of the expression open-advice.

```
if my-hand = pair-queens
and open-advice = pass
then stay-advice = get-out.

if my-hand = three-of-a-kind
and ... 
then open-advice = sandbag.

if my-hand = straight-flush
and ....
then open-advice = open.

if my-hand = garbage
and ...
then open-advice = cannot-open.
```

Example 4.4

The set of concluded values for the expression open-advice are open, and cannot-open. The set of utilized values for the same expression is pass. For premise three
open-advice = pass of the first rule to be set to true and consequently for the rule to succeed, the expression open-advice has to be assigned the utilized value pass, which is not a member of the set of concluded values for the expression; hence the rule will fail.

Example 4.5 from the Wine Advisory System (a demonstration program provided by Teknowledge, the developer of M.1), also shows a set of rules that can never succeed because of the mismatch between the utilized and concluded values of an expression.

| rule-12: if has-sauce = yes and sauce = sweet then best-sweetness = sweet cf 90 and best-sweetness = medium cf 40. |
| rule-26: if best-sweetness = dry then recommended-sweetness = dry. |
| rule-27: if best-sweetness = medium then recommended-sweetness = medium. |
| rule-28: if best-sweetness = sweet then recommended-sweetness = sweet. |

Example 4.5

From rule-12, the expression best-sweetness will be instantiated either to the value sweet with certainty factor 90 or to the value medium with certainty factor 40 (sweet and medium constitute the concluded values for the expression). For rule-26 to succeed, the expression best-sweetness has to be instantiated to the value dry of the
set of utilized values. This can not happen, because the conclusion of rule-12 is the only place in the knowledge base where best-sweetness can get its values, so rule-26 fails.

In example 4.6 from the Advice System, flagging the utilized value yes of the expression semdel(SEM-M) of the second rule reveals a serious error in the system.

```plaintext
if semester(SEM)-course(M) = [C,U]
and do(set course_to_delete = C)
and delete_course = deleted
and acculmulate(SEM-M)
then semdel(SEM-M) = done.

if N-1 = M
and not(M=0)
and cached(attribute-semester(SEM) = [T, Y, N])
and do(set attribute-semester(SEM) = [T, Y, M])
and semdel(SEM-M)
then acculmulate(SEM-N) = yes.

accumlate(SEM-N) = yes. /* When that semester is empty. M=0.*/
```

Example 4.6

The knowledge engineer intended to give the expression acculmulate(SEM-N) a yes value if the second rule ever failed because the semester is empty (look at the comment provided by the knowledge engineer in the knowledge base next to the fact). But the rule will always fail because of the mismatch between the utilized value
(second rule) and the concluded value (first rule) of the expression \textit{semdel(SEM-M)}. Consequently the expression \textit{accumulaste(SEM-N)} will take the value \textit{yes} from the provided fact no matter what happens to the second rule.

4.1.2 Unused Legal Values And Questions

The syntactic error checker module also prepares a list of all unused legal values and unasked questions for the knowledge engineer to review. This helps in detecting misspelled words and items that have never been used. For example, if a question was never been used, it is better to delete it from the knowledge base. That will also cut the size of the knowledge base and consequently the search space and time. It will also speed the inferencing process during a consultation.

The \textit{syntactic error checker} searches the premises of rules looking for the declared but unused legal values. It flags them as potential errors. Flagging unused legal values might reveal a flawed structure in the knowledge base. Flagging meaningless or unused legal values might lead to constructs that were misunderstood by the system or mistakenly put into the knowledge base by the knowledge engineer.

In (example 4.8) from the Helper System, the knowledge engineer used \textit{integers} instead of the M.1 reserved word \textit{integer} to inform the system of the set of a user's legal responses for the question that concerns the expression \textit{restart}. A typographical error from the knowledge engineer's side caused the \textit{Syntactic Error Checker} to flag every used value of the expression \textit{restart} to be an undeclared legal value and the string \textit{integers(0,9)} to be a never been used legal value for \textit{restart}. 
question(restart) = [nl,'Which of the following.....see again ?
1. print your consultation,
8. restart consultation completely,
9. quit this consultation.
**** Choose one or more of the above .....',nl].

legalvals(restart)=integers(0,9).

Example 4.7

Unused legal values are common in knowledge bases. They might be a problem or might reflect the presence of an error or a bad knowledge representation scheme if more than one of the legal values is unused. In the following example from the Rashdec System, the Syntactic Error Checker flagged the legal values normal and subnormal as unused legal values. Pointing out such a potential error can often prompt a knowledge engineer to change the structure. The knowledge engineer could have asked the same question in a different form (example 4.8). The suggested question is direct and provides fewer options for the user to choose from.

<table>
<thead>
<tr>
<th>Original Structure</th>
<th>Suggested structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>question(temperature)=&quot;What is the temperature of your patient? 1. elevated (greater than 100 degrees Fahrenheit) 2. normal (between 98.2 and 99.9 degrees Fahrenheit) 3. subnormal (less than 98.2 degrees Fahrenheit)&quot;. legalvals(temperature)=(elevated,normal,subnormal).</td>
<td>question(temperature)=&quot;Is the temperature of your patient more than 100 degrees Fahrenheit?&quot;. legalvals(temperature)=(yes,no).</td>
</tr>
</tbody>
</table>

Example 4.8
The expressions of the questions tree are also compared with the expressions of the legal values tree of the knowledge base. Questions that have no legal values provided for them are also flagged as unused questions.

4.1.3 Errors In The Premises Of A Rule

Errors that appear in the premises of a rule usually cause two types of potential problem in the knowledge base. (1) They cause that rule to fail, so no value will be concluded for the expression in the conclusion proposition of that rule. Consequently, all the rules that use the conclusion proposition of the rule as a premise will fail. (2) The system will stop testing the remaining premises of the rule, and consequently the system will stop seeking values for the rest of the expressions of the rule, displaying messages or even firing other rules.

In this section I will discuss three types of errors in the premises of rules: (1) errors due to the negation of the expression in a premise, (2) errors due to the negation of the value of the expression in a premise, and (3) errors due to using status expressions as values in a premise. M.I uses the not operator to negate the value of an expression. For example the premise not(eolor(ear) = white) will be evaluated to true if the expression eolor(ear) has values other than white. The wrong usage of the not operator usually leads to different types of errors in the knowledge base.

1- Errors due to the negation of the expression on the left side of a premise. If a premise of the form: not(type(air-conditioner)) = central, appeared in a rule, the
shell will treat the left side of the premise as a new expression whose value is to be sought by the system.

2- Errors due to the negation of the value of an expression on the right side of the premise. If a premise of the form: \( \text{type(air-conditioner) = not(central)} \) appeared in a rule, then \( \text{not(central)} \) is the utilized value for the expression \( \text{type(air-conditioner)} \).

So if the user’s response is different from that value, or if the system was not able to conclude that value (through another rule or a fact) for the expression, the premise is going to be evaluated to false and the rule related to that premise will fail.

3- Errors due to instantiating an expression to a status expression or adjective.

Assertion about an expression status is usually done via the adjectives \textit{definite}, \textit{known}, \textit{unknown}, \textit{sought} and \textit{unique}. Usages of metapropositions is provided in Table 4.1. Facts and fact propositions (special types of premises) use the = sign to attach them with their assigned values, as in the premise \( \text{best-color = red} \). Metapropositions use the \textit{is} operator to assert the status or conditions of the values of the expression rather than testing the value(s) of the expression. They make statements about the consultation, rather than about the problem domain.
Table 4.1 M.1 Metapropositions and Their Usage

<table>
<thead>
<tr>
<th>Metaproposition</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression is definite</td>
<td>is evaluated to true if the expression has a value that is known with certainty</td>
</tr>
<tr>
<td>expression is known</td>
<td>evaluated to true if the expression has a value with certainty more than 20</td>
</tr>
<tr>
<td>expression is sought</td>
<td>always evaluated to true</td>
</tr>
<tr>
<td>expression is unique</td>
<td>evaluated to true if exactly one value has been concluded for expression</td>
</tr>
<tr>
<td>expression is unknown</td>
<td>evaluated to true if the expression has no values</td>
</tr>
</tbody>
</table>

A problem that is related to the usage of metapropositions and metafacts is the usage of the = sign instead of the is operator with these adjectives, i.e. assigning any of the above adjectives as a value for a certain expression. In the following example from the VW Motor Troubleshooting System, the rule will not succeed because of the third premise of the rule. The declaration of unknown to be one of the legal values for the expression exhaust-tubing will cause the proposition exhaust-tubing = unknown to be evaluated to false, so the display message will not be provided for the user of the system because the knowledge engineer's representation was unable to override the system's usage of the adjective unknown. Defining unknown to be a member of the set of legal values of an expression does not override its shell usage.
question(exhaust-tubing) = ['What size of exhaust ....?'].
legalvals(exhaust-tubing) = [one-and-quarter-inch, one-and-half-inch, unknown].

if peak-power = low-end-power and
   exhaust-tubing = one-and-half-inch or
   exhaust-tubing = unknown and
   display([nl,'The tubing of your exhaust ....'])
then advice = small-tubing.

Example 4.9

Interestingly enough, the above rule is legal and the success of the rule is guaranteed if the system was able to conclude the disputed premise from the conclusion of another rule without asking the question concerning the expression.

4.1.4 Errors In The Conclusions Of Rules

Two problems are related to errors in the conclusion propositions of a rule: (1) The rule will not fire, this happens by taking the rule out of the search tree. (2) Other rules that use the conclusion propositions of the rule as a premise will fail.

Errors in the conclusions are usually due to the wrong instantiations of the expressions in the conclusion of the rule or a misunderstanding of the capabilities of the shell. Six different types of errors and their effect on a knowledge base will be presented in this section.

1- Errors due to the negation of the conclusion of a rule. If a metaproposition of the form not(type(air-conditioner) = central) appeared in the conclusion of a rule, the whole rule will escape firing even if the expression type(air-conditioner)
is in the search path of a goal or an initial data statement. The rule escaped firing because the shell tried to seek a value for the expression in the conclusion of a rule, in this case the expression is not going to have any value to be instantiated to.

2- Errors due to the negation of the expression or the left side of a conclusion. If a proposition of the form: \( \text{not(type(air-conditioner)) = central} \) appeared in the conclusion of a rule, the shell will take the expression \( \text{not(type(air-conditioner))} \) as the conclusion expression to be sought. The rule will never fire because the conclusion expression is not going to be in any search path of a goal.

3- Errors due to the negation of the value of the conclusion expression. If a conclusion proposition of the form: \( \text{type(air-conditioner) = not(central)} \) appeared in the conclusion of a rule, the rule will fire if it is in the search tree of a goal or an initial data statement. If the rule succeeded, then the value of the expression \( \text{type(air-conditioner)} \) will be set to the string \( \text{not(central)} \); that is not what the knowledge engineer had in mind for a value.

4- Errors due to the usage of metapropositions as conclusions of rules. The shell usually uses the metaproposition \( \text{expression is unknown} \) as a premise of a rule to indicate that no value has been set for the expression within a given certainty. If the same metaproposition ever appeared in the conclusion of a rule then the shell will take the expression to be the string \( \text{expression is unknown} \) since the operator \( \text{is} \) is an M.1 reserved word. It is defined as an infix operator that glues words to
it's right and left. M.1 also uses the metaproposition *expression is unknown* to indicate that all the rules that conclude values for that expression had failed and no value was concluded for that expression. So if the rule succeeded, the conclusion would be of the form: *expression is unknown = yes* otherwise it will be of the form: *(expression is unknown) is unknown*. Things will look even worse if the knowledge engineer used *expression is known* as a conclusion of a rule. Then he might find metapropositions of the form *(expression is known) is unknown* in the file of saved consultation.

5- Errors related to assigning status adjectives as values for an expression in the conclusion of a rule. Such usage in a conclusion will create an interface problem if the expression was ever declared to be multivalued and a question was provided to the expression too. In such case the question is going to be asked even if the rule succeeded. If the user's answer was *unknown* then the expression has a value which is equal to the string *known* and has the metaproposition *expression is unknown* connected to it. Such errors in the conclusions of rule will lead to different unspecified behavior by the system. In the following example from the Home Appraisal System, the knowledge engineer used *unknown* as a concluded value for the expression *value*. 
6- The last type of error that usually appears in the conclusion of a rule is displaying a message in the conclusion. Forward chaining systems fire rules based on their conditions while backward systems fire rules based on their conclusions. So, it is legal to have display message functions only in the conclusions of forward chaining systems (OPS5) and in the conditions of backward chaining systems (M.1). M.1 uses the built in display function as a premise to type messages to the screen, print them to a printer or save them to a file. The function takes quoted strings and print control characters as arguments. If the display function ever appeared in the conclusion of a rule then the whole rule is just a dead rule because M.1 is a goal driven rather than a data driven shell. Conclusions have to be attached to the inferencing tree via a certain construct (the conclusion expression appears in a premise of another rule), and rules fire based on their conclusions. In the following example from the Dell System, the knowledge engineer used a display message in the conclusion of the rule. If you look into the rule you will find another error due to the usage of unknown as a value in the second premise of the rule.
if want-printer
   and printer-type = unknown
then display('Please talk to a Dell salesperson to choose a printer.').

question(want-printer) = [nl,'Do you want a printer?'].
legalvals(want-printer) = [yes,no].
explanation(want-printer) = ["You have the option of purchasing .....'"].

question(printer-type) = [nl,'Do you want a laser or dot matrix printer?'].
legalvals(printer-type) = [dot,laser].

Example 4.11

M.1 is a robust system that tries to recover from knowledge base errors. By doing so, the system creates knowledge base structures that were never intended. This also leads to screening errors that are better brought to the attention of the knowledge engineer instead of making them invisible. Systems that recover from errors should keep a record of these errors for inspection. Otherwise, there is no easy way for the operators or users to detect odd behavior of the system. Another odd behavior of the system is how it deals with metapropositions. If a metaproposition appears as a premise of a rule, it will be treated as a metaproposition. On the other hand, if it appears as a conclusion, the whole metaproposition will be treated as an expression with a value yes to be assigned to it if the rule ever succeeded, a behavior by the system that the knowledge engineer might not be aware of.
4.2 The Debugger

Debugging is the tedious and difficult process of finding and correcting errors in the knowledge base. These errors are usually discovered during developmental testing and rely heavily on the knowledge engineer's intuition, his accumulated experience and common sense. Debugging is also time consuming and costly. On the other hand, automatic debugging tools are more reliable because they reduce the possibility of human errors. The debugger (Module 4 of Validator) looks for the usage of variables in rules. Such an error usually halts the whole system with an ambiguous error message such as *What is the value of expression?*

M.1 allows knowledge engineers to use variables. The shell identifies a variable if it starts with a capital letter. Variables can be used in facts, metafacts and rules. They act as symbolic place holders. Each construct that uses variables is logically equivalent to an infinite set of constructs that could be obtained by replacing the variable with the suitable terms. In the following rule, the variable X will be instantiated whenever the conclusion of the rule falls in the search path of a goal or a subgoal. However, the knowledge engineer mistakenly typed a Y instead of X in the third premise. If this rule ever fires during a consultation the computer will halt the whole session because the variable Y can not be instantiated to any value during the consultation.
if air-conditioner = air-conditioner-X
and type(air-conditioner-X) = central
and not(maintained(air-conditioner-Y))
then maintain(air-conditioner-X).

Example 4.12

In the next example from the Advice System, the debugger flagged the usage of the two variables P1 and P2 as attributes in premises ten and eleven respectively as errors that will crash the system.

if course_to_delete = DEPNUM
/* and attribute(DEPNUM) = [T,P1,P2,U,SG,G,AREA] */
and cached(semester(SEM)-course(COURSE) = [DEPNUM,U])
and resetting-SEM-COURSE = done
and attribute-semester(SEM) = [TIME,YEAR,TOTAL]
and TOTAL - COURSE = NUM1
and setting-semester(SEM)-course(COURSE)-NUM1 = checked
and do(reset setting-semester(X)-course(Y)-Z)
and TOTAL - 1 = NTOTAL
and do(set attribute-semester(SEM) = [TIME,YEAR,NTOTAL])
and permdela(DEPNUM-P1)
and permdelb(DEPNUM-P2)
and display([nl,
DEPNUM,,' was deleted from semester number ',SEM,nl])
then delete_course = deleted.

Example 4.13

Further investigation of the rule showed that it is not a dead rule. It is in fact in the search path of more than one goal tree. A careful look at the second line of the example shows that the knowledge engineer has commented out a premise that is
necessary for the successful instantiation of the variables P1 and P2, and for the normal progress of a consultation.

4.3 Summary

In this chapter I presented the different types of errors the verification section of Validator can detect. The errors are brought to the attention of the knowledge engineer as output messages with the subsequent explanation listing the expression its value, and the reason for suspecting an error. Some of these errors have serious effects on the knowledge base, such as halting a consultation or preventing a subset of the knowledge base from being active. Other potential errors, such as screening the usage of some clauses, have no effect on the knowledge base as long as the knowledge engineer knows exactly what he is doing. Validator has no capability of correcting errors; it just flags them. It is up to the knowledge engineer to decide if they are actual errors.
CHAPTER 5

KNOWLEDGE BASE VALIDATION

5.0 Background

In this Chapter I present general techniques for the validation of a knowledge base to help guarantee its completeness and consistency of the knowledge base. These techniques are: (1) checking the validity of the premises and conclusions of each rule of the knowledge base, (2) checking the validity of every goal and initialdata statement of the knowledge base, (3) checking the validity of the left and right sides of the whenfound† and whencached statements, (4) searching the knowledge base for the existence of multiple methods of seeking values for expressions, this includes having questions and facts, questions and rules, or facts and rules to seek values for the same expression, (5) checking the validity of each fact and question of the knowledge base and (6) checking multiple declarations of the knowledge base items such as multiple questions.

This chapter describes the Chaining Thread Tracer (the fifth module of Validator) and the Knowledge Base Completeness Module (the sixth module). These two independent modules are usually executed after the knowledge base has been

† For the rest of this chapter, I will use the statement whenfound to mean either a whenfound or a whencached.
checked for correct spelling, the data base created by Validator has been manually inspected, and the verification modules have already checked the system. The activities of these two modules are carried out before developmental testing.

5.1 The Chaining Thread Tracer

When syntactic error checking and debugging is complete and all the syntactic errors are removed from the knowledge base, rule chaining can begin checking the knowledge base. This module determines whether goal driven rules (rules that have premises and conclusion) as well as data driven rules (rules constructed via the use of whenfound metafact constructs) will fire by tracing their connectivity back to the goal tree. Rules that do not fire are flagged as dead rules and are brought to the attention of the knowledge engineer. A rule is flagged as dead if it is the root of a dead tree. Flagging a dead rule might uncover a whole set of dead rules.

Premises are also checked for validity. Invalid premises lead to the failure of the rules that use them and prevent premises of the same rule that come after them from being sought for testing.

5.1.1 The Backward Chaining Thread Tracer

Backward chaining systems start their search with a goal as the root of an inverted AND/OR goal tree. Conclusions of rules that conclude the goal, facts that set values for the goal and questions concerning the goal constitute the child nodes of the tree. Facts and questions are terminal nodes, or leaves, while the premises of
each rule constitute the child nodes of each rule node. A goal tree is considered to be valid if its terminal nodes are either metapositions that are always true, questions or facts.

Backward chaining systems use different strategies to search the knowledge base seeking value(s) for a goal. Expressions of nonterminal nodes constitute subgoals to be sought by the system. M.1 puts rules in the search tree of a goal based on their conclusion only. It does not check the value of the conclusion expression to make sure that it is the right rule to fire. This will lead to searching branches of a tree although they are not going to contribute to the solution of the problem. Such processing confuses the end user and might lead to asking him contradicting or irrelevant questions. It is the knowledge engineer’s responsibility to provide control rules to ensure the smooth transition of questions from one state to another and to cause only the right branches of a goal tree to be searched.

I called this component a Backward Chaining Thread Tracer because it traces connectivity of the premises and the conclusions of each rule through the backward chaining system. It checks the validity of each rule by tracing its premises and its conclusion to see if they are properly interconnected. Rules that are not connected are flagged as dead rules. Such rules should be deleted from the knowledge base or have their conclusion premise linked to other knowledge base constructs. For example, the goal identity of animal of figure 5.1 (page *) brings us

† Expressions that get their values from the conclusions of rules are the nonterminal nodes expressions.
to the conclusion of rule 3. From here the expression subtype of animal links the first premise of rule 3 to the conclusion of rule 2. Next the expression type of animal links the first premise of rule 2 to the conclusion of rule 1. The conclusion of rule 1 is then linked to the premise of rule 1, coat of animal = hair. There is a question that can provide a value for the expression, coat of animal, so this chain of linking is valid.

To detect logical errors in backward chaining systems, the Backward Chaining Thread Tracer checks every premise of every rule to ensure that they lead to valid ends. A premise has a valid end if it appears in the conclusion of another rule, or its expression has a question provided for it by the knowledge engineer with the utilized value of the expression as a member of the set of legal values provided for the expression. For example, in figure 5.1 the first premise of rule 2 has a valid end since it appears as a conclusion in rule 1. The second premise of rule 2 has a valid end too, because the expression extremities of animal has a question provided for it.

The Backward Chaining Thread Tracer also checks the conclusion part of every rule for valid ends. A rule’s conclusion is considered to have a valid end if it is either a goal an initialdata statement, or it appears as a premise of another rule. For example, in figure 5.1 the conclusion of rule 1 appears as a premise in rule 2, in turn, the conclusion of rule 2 appears as a premise in rule 3, the conclusion of rule 3 is a goal. Hence rules 1, 2, and 3 all have valid ends. An added complexity in the
conclusion validation checking that we have to deal with is caused by the fact that, conclusions of rules can be linked with two types of premises. A rule with a conclusion of the form \textit{animal} = \textit{mammal}, has to be linked with the premise \textit{animal} = \textit{mammal} and also with \textit{not(animal = ANY)}, where \textit{ANY} can take any value other than mammal, since both premises are going to be in the search tree of a goal statement.

Figure 5.1 shows a typical knowledge base that has various errors that can be detected by this module of \textit{Validator}. The first three rules are correct. If you have a zebra in mind, it will correctly identify this animal. The rest of the rules illustrate various mistakes that might result when a knowledge base is expanded. The backward chaining thread tracer will flag rule 6 as extraneous because \textit{habitat of animal} appears nowhere else, this rule will never be reached during inferencing, so it can never fire. Rule 5 shows another mistake that will be detected, \textit{feed of animal} will be flagged as a potential error because the expression has no question and it does not appear in the conclusion of any other rule.

The \textit{Backward Chaining Thread Tracer} also checks every goal and initial-data statement for validity. A goal or an initialdata statement is considered to be valid if it appears as an expression in the conclusion of a rule or a question metafact. Goals and initialdata statements that are not valid are brought to the attention of the knowledge engineer.
infix of.
goal = identity of animal.

rule1:  if coat of animal = hair
        then type of animal = mammal.

rule2:  if type of animal = mammal and
        extremities of animal = hooves
        then subtype of animal = ungulate.

rule3:  if subtype of animal = ungulate and
        marking of animal = black-stripes
        then identity of animal = zebra.

rule4:  if coat of animal = scales
        then identity of animal = fish.

rule5:  if feed of animal = meat and
        marking of animal = black-stripes
        then identity of animal = tiger.

rule6:  if coat of animal = feathers and
        animal-swims
        then habitat of animal = antarctic.

question(coat of animal)=('What is the coat of animal?').
legalvals(coat of animal)=[hair, feathers].

question(extremities of animal) = 'What is type of the extremities of the animal?').
legalvals(extremities of animal) = [hooves, claws].

question(marking of animal) =
('What is type of the marking of the animal?').

Figure 5.1. A Small Knowledge Base With Typical Errors.
5.1.2 The Forward Chaining Thread Tracer

This module detects forward chaining rules that will never fire. M.1 uses the whenfound statement to force the system to do forward chaining. The left side premise of a whenfound statement has a valid end if it appears as a premise of at least one rule, on the right side of another whenfound or in a goal statement. So each left side premise of a whenfound is checked for validity. Nonvalid whenfound statements will create knowledge base items that never fire. They have to be either connected to other active knowledge base constructs or deleted from the knowledge base. The right side premises are considered to have valid ends if they satisfy the same conditions as the premises of goal driven rules. The following example from the Stutter System shows a usage of a whenfound as well as an error in the usage of the set of legal values for *accessory*. Since the conclusion expression (*accessory-type*) of the first rule lies in the search tree of a goal statement (not shown in the example) the rule will trigger the question concerning the expression *accessory*. The set of legal values does not include the value 9, apparently the whenfound construct will never fire.
if accessory = 1
   and accessory = 2
   and accessory = 3
then accessory-type = yes cf 98.

multivalued(accessory).

question(accessory)=[ nl,nl, 'Because you indicated that the parents have expressed concern about Associated Struggle behaviors we need to check the specific kinds of behaviors.
1 - Changes in loudness/pitch during speech disruptions.
2 - Lip/jaw tremors during speech disruptions.
3 - Movement of other body parts prior to or during speech disruptions.
Type in any combination of numbers then press Return or Enter.
e.g. 1,2,3 or 2,3 or 3,2 or 1 ',nl].
legalvals(accessory) = [1,2,3].

whenfound(accessory = 9)= [do(reset accessory)
and ....
and do(reset disrupt) and do(restart)].

The Stutter System had 12 whenfound constructs that had never been used. That accounted for 2787 bytes of memory which constituted 3.4 percent of the total size of the system. Furthermore these dead rules constructs were symptoms of serious misconceptions about the use of M.1.

5.2 The Knowledge Base Completeness Module

Knowledge base completeness checking is the checking of the unknown. It is the problem of trying to find something unknown that is missing from the knowledge base. Checking the completeness of a knowledge base usually proceeds in two different directions.
The first one is direct and includes the knowledge engineer and his expert working together on the knowledge base refining and analyzing each of its items. They check the self completeness of every module, the consistency, effectiveness and efficiency of every knowledge base item. They also review each rule and decide if they have to split it, modify it or delete it from the knowledge base. This direction is not within the scope of this dissertation.

The second direction taken in checking the completeness of a knowledge base is indirect. It includes the knowledge engineer working on the structure and representation of the knowledge base. Knowledge base items are analyzed, compared for redundancy, completeness and the correctness of their usage. The second direction is structured, algorithmic and more exhaustive than the direct one. It also uses heuristics and can be easily automated to produce fast and effective results. The techniques presented in this section are classified under the second direction. They include checking the existence of multiple methods of seeking values for expressions and checking questions, rules and facts for redundancies and usage.

5.2.1 Multiple Methods

An expression has multiple methods for seeking its value(s) if it appears in a question and a fact, or in a question and the conclusion of a rule, or in a fact and the conclusion of a rule. During the process of seeking a value(s) for an expression the knowledge base is sequentially searched for every relevant item. The presence of multiple methods for seeking the values of an expression provide grounds
for potential problems in the knowledge base.

The first problem occurs if the expression is declared to be multivalued. In this case all the constructs that conclude values for the expression will fire. The second problem occurs if the inference engine is unable to conclude values for the expression from the first construct, it will keep firing the relevant constructs until a value is found.

The Knowledge Base Completeness module of Validator checks for multiple methods of obtaining values for expressions. Multiple methods might screen the firing of other constructs (if the expression is single valued). They might also aid the inferencing process by providing values where needed (default values). Their usage is tricky and has to be brought to the attention of the knowledge engineer. In the following example (from the COGITO System) the knowledge engineer had both a rule and a question that would give values for number-of-massbus, but he knew exactly what was going to happen.

```
question(number-of-massbus) = 'How many massbus adapters does the system have?'.
legalvals(number-of-massbus) = integer(1,4).
if number-of-massbus is unknown
  then number-of-massbus = 1.
```

The first time the system seeks a value for the expression number-of-massbus, the question will be asked first, if the user’s response is unknown then the rule will set a default value for the expression. While in the following example the warning of the presence of multiple methods revealed a flaw in another system’s structure.
The expression $checkg(G)$ will be set to the value $yes$ no matter what happens in the above rule. According to the M.1 syntax, the second, third, fourth and fifth premises of the rule are always true. So the first premise is the one that will decide whether the rule succeeds or not. If the rule fails because the value of $G$ is not greater than zero then the rule will not set the expression $checkg(G)$ to the value $yes$. The presence of a fact after the rule will set the expression to that value and the rule is redundant.

Another interesting aspect of the usage of different constructs to assert a value for an expression is when the user provides certainty factors along with his answers to questions. The provided certainty will carry out to the conclusion if the level of certainty of a response is less than 100. In that case the system still seeks other constructs in the knowledge base to get a certain value for the expression. If the expression is single valued the system will keep seeking values until it exhausts all possibilities or reaches a 100 percent certainty value. In such cases, even though the knowledge engineer was aware of the structure of the knowledge base and the order in which his questions, facts and rules appear, he has no control over their
firing in the end user system and it is better for him to avoid such multiple usage or to build functions that prevent an end user from using certainty factors.

5.2.2 Facts Validation

After all the facts have been listed for the user to inspect, Validator checks the validity and usage of each fact. For a fact to be valid it has to appear in at least one premise of a rule or the right side of a whenfound. So each fact is checked for its usage in the knowledge base. Unused facts are presented to the knowledge engineer for further inspection.

Unused facts might also explain other types of errors in the knowledge base. In the following example from the Advice System, Validator informed the knowledge engineer that the expression `legavals(semthe)`, whose object is the string `legavals`, attribute `semthe` and value the string `integer`, had never been used.

question(semthe) = ['Delete this course from which semester?','nl'].
legavals(semthe) = integer.

Such an error was typographical. The knowledge engineer misspelled the M.1 metafact `legalvals` by omitting the fifth character from the metafact. This led the inference engine to read that knowledge base item as a fact.

The the developmental test cases that I used to test validator showed a significantly "large" number of unused facts. The COGITO System had 26 unused facts and the Advice System had 10 unused facts.
5.2.3 Questions Validation

After the question expressions are displayed for inspection, they are compared with the legal values tree for cross referencing. Expressions that do not appear in the legal values trees are displayed as unused questions. The program warns the knowledge engineer of the cases where there is no question for an expression whose legal values are provided.

Questions are also checked for redundancy. Two questions are considered to be redundant if both questions seek values for the same expression. Validator does not check if the two questions are identical. Multiple questions might leave serious effects on the system, they might be contradictory or different. Multiple questions have to be deleted from the knowledge base. If the end user did not know the answer to a question and wanted to respond with unknown, the system would repeat the question as many times as the question is found in the knowledge base. All the questions will be repeated if the expression under investigation was declared to be multivalued. The same thing will happen if the user provided a value with a certainty factor less than 100. The system will keep seeking values for the expression until it exhausts all questions or a certain value is found. In the Stutter System one expression had three questions, in the COGITO System three expressions had two questions each, one expression had four questions, one expression had eight questions and one expression has nine questions. In the following example from the Helper System the knowledge engineer provided two questions (the asknumber
The first metafact \texttt{question(what-class)}, allows the user to enter any string as an answer. While the second metafact \texttt{asknumber(what-class)}, does not take other than a number as an answer. The asknumber statement will fire only if the user's response was the adjective \textit{unknown}, a value with a certainty factor less than a 100, or if the expression \texttt{what-class} was declared to be multivalued. So the metafact \texttt{asknumber(what-class)} is extra and will have confusing side effects. In this case, the question construct is not safe for the user, there is no easy way for him to recover from a typographical error. It would be better if the knowledge engineer had used the asknumber instead of the question construct. It is safer because it does not accept anything but numbers.

Questions are also checked with the rules premises, right sides of the \texttt{whenfound} statements, the goals and initial data statements to guarantee that there are no unused questions. This way it is guaranteed that all the expressions whose values are to be provided by the user already have appropriate questions. If the system has to get a value for an expression from the user, M.1 prints a question message of the form: \textit{What is the value of: expression?} The average end user finds it hard to understand, especially if the knowledge engineer used abbreviations for the expression name and attributes. Then there will be no way for an end user to under-
stand what the system means.

Another serious problem with questions is when their expressions appear in the conclusions of rules, to put things in a simpler way, when the knowledge engineer decides to generate rules to seek values for an expression. If the question was not deleted, the issue of multiple methods for seeking values of an expression will rise once more.

5.3 Summary

The development process of a knowledge base is an iterative process (see the spiral model of chapter 1). Many changes and additions to the knowledge base occur during that process. *Validator* is a tool that checks for potential problems in a knowledge base. Such a tool is very helpful for developing more accurate consistent and complete knowledge based systems. The ideas presented in this chapter are generic, with little tuning the algorithms can be adjusted to fit most of the backward chaining shells.
CHAPTER 6

RESULTS

6.0 Background

In this chapter I present the results of running *Validator* on different knowledge based systems. The different types of mistakes that knowledge engineers make while building a knowledge base can be summarized into the following 9 categories: (1) illegal usage of M.1 reserved words; this also includes different wrong usages of the *not* operator and the metaproposition adjectives such as *unknown* in the premises and conclusions of rules. (2) Unused rules; this includes backward as well as forward rules. (3) Unused facts, (4) unused questions, (5) unused legal values (Unused legal values are counted only if two or more legal values of a given expression were not used.), (6) multiple questions, (7) multiple methods; this include expressions that appear in questions and facts, questions and conclusions of rules or facts and conclusions of rules. (8) Rules that use illegal values; this includes mismatch between utilized and legal values, utilized and concluded values and utilized values and facts concerning expressions. (9) Wrong instantiations†.

The tested knowledge based systems were developed in the department of

† The Advice System was the only system that has such type of error. So, it will not be tabulated.
Systems and Industrial Engineering at the University of Arizona. They are classified under three different categories: (1) Masters projects, (2) potential commercial systems and (3) class projects for a course in knowledge engineering (SIE474). The test results of running Validator on the Masters projects and the commercial systems will be presented in some detail and in tabular form (table 6.1), while the test results of the class projects will be presented in tabular form only (tables 6.2, 6.3 6.4).

Multiple occurrences of the same error were counted once. For example, a rule is declared unused (will never fire) based on its conclusion expression. Rules that conclude the same expression are counted as one unused rule if their conclusion expression is not linked to a goal tree. An illegal value of an expression might appear in more than one rule, but it is counted only once.

6.1 Masters Projects

The systems of the Masters projects were all developed in the System and Industrial Engineering Department at the University of Arizona. They were all supervised by this dissertation director (Prof. A. T. Bahill). The systems were all tested by the knowledge engineer and by Prof. Bahill. They were tested to the best ability of the knowledge engineer and Prof. Bahill and were believed to contain no errors.

Advice (82,000 bytes)

This system was developed to help evaluate students study plans of the System and Industrial Engineering of the University of Arizona. The system started
as a class project, then was continued as a Masters project. The testing results showed that out of 57 facts, 9 were not used and 25 appeared in the conclusions of rules. Out of 27 different questions, 6 appeared in conclusions and 4 in facts. The results also showed 2 bugs that will crash the system, 1 wrong usage of utilized values and 3 dead rules. Further investigations of the dead rules showed that 2 of them were active in a version of the system that uses DECtalk from DIGITAL Corporation. The knowledge engineer commented their appearance as premises of other rules but she did not bother to delete or comment these rules.

COGITO (126,000 bytes)

This system was developed to help in making and installing UNIX 4.2BSD on VAX-11 series computers. The system filters the information available in the reference manuals used for installation instructions and presents advice relevant to the user's computer system. Test results showed 5 unused rules, 25 unused facts, 6 different expression with 9, 8, 4, 2, 2 and 2 repeated questions respectively, and 7 different expressions that have multiple methods for seeking values. Further investigation of the knowledge base showed that 2 of the multiple methods used are legal. The knowledge engineer used rules to set default values for expressions when the user did not know the answer.

Helper (50,000 bytes)

This system was designed to aid students using a DOS and UNIX based personal computers laboratory. Test results showed an illegal use of the unknown
adjective, one unused question, two questions for the same expression and 14 wrong usage of utilized values. Another thing that attracted my attention during the review of the data base is the number of questions. Helper has only 11 questions to ask, which is few compared to other systems of its size.

**Stutter (81,000 bytes)**

This system was developed to help with the diagnosis and prognosis of children who may have begun to stutter. The system started as a class project then was continued as a Masters project. Currently the system is under final development and testing for commercial release. Testing the system revealed an expression that has 3 questions, 12 unused forward chaining rules, 5 unused questions and 8 wrong usages of utilized values. Further investigation of the knowledge base showed that 3 of the utilized values will not affect the rule firing of the system because they appeared as premises in disjunctive rules.

**Chromie (100,000 bytes)**

This system was developed to help novice medical professionals identify chromosomal abnormalities. It started as a class project, then was continued as a Masters project. Currently the system is in the process of being commercially released. The knowledge engineer invested a large amount of time developing and structuring the system. Testing the system revealed one dead rule, further investigation showed that a typographical error caused it.
6.2 Commercial Systems

**SRPRES (120,000 bytes)**

This system was developed as a joint project with Solar and Energy Research Facility (SERF) of the University of Arizona. It is a Salt River Project Residential Expert System (Appendix A). It was built during the early stages of development of *Validator*. The different components and prototypes of SRPRES provided test cases while building *Validator*. SRPRES also provided an environment for understanding the pros and cons of the expert system shell (M.1). The system was delivered one year before finishing this research. In the meantime other components were added to *Validator*, and the system was retested. The final testing showed two unused facts, two unused rules and one mismatch between utilized and legal values that will cause the automatic failure of a rule.

**Fundeye (64,000 bytes)**

This system was developed to help with the diagnoses of retinal disease. It started as a student project then expanded in preparation for commercial release. Testing the system revealed no errors.
The following tables show the name, the size and the number of errors in each of the eight different categories of errors found in the Masters projects and the commercial systems. An empty box means no errors were found in that category.

<table>
<thead>
<tr>
<th>System Name</th>
<th>Size (Kbytes)</th>
<th>M.1 reserved words</th>
<th>Unused rules</th>
<th>Unused facts</th>
<th>Unused questions</th>
<th>Unused illegal values</th>
<th>Multiple questions</th>
<th>Multiple methods</th>
<th>Rules using illegal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advice</td>
<td>82</td>
<td></td>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td>Chromie</td>
<td>100</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COGITO</td>
<td>126</td>
<td></td>
<td>5</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Fundeye</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Helper</td>
<td>50</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>SRPRES</td>
<td>120</td>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Stutter</td>
<td>81</td>
<td></td>
<td>12</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>
6.3 Class Projects

Results of testing class projects will be presented in three separate tables, projects of the same year are grouped into the same table.

Table 6.2. Errors Detected in Students Systems (Fall 1988).

<table>
<thead>
<tr>
<th>System Name</th>
<th>Size (Kbytes)</th>
<th>M.1 reserved words</th>
<th>Unused rules</th>
<th>Unused facts</th>
<th>Unused questions</th>
<th>Unused values</th>
<th>Multiple questions</th>
<th>Multiple methods</th>
<th>Rules using illegal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Citsys</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dell</td>
<td>55</td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diet</td>
<td>15</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drug</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Fever</td>
<td>54</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Napsx</td>
<td>27</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kyl</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Macxprt</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oring</td>
<td>37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pimonic</td>
<td>28</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Soundf</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Volcan</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wire</td>
<td>19</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 6.3. Errors Detected in Students Systems (Fall 1987).

<table>
<thead>
<tr>
<th>System Name</th>
<th>Size (Kbytes)</th>
<th>M.1 reserved words</th>
<th>Unused rules</th>
<th>Unused facts</th>
<th>Unused questions</th>
<th>Unused legal values</th>
<th>Multiple questions</th>
<th>Multiple methods</th>
<th>Rules using illegal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baja</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cook</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hypotens</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Job</td>
<td>23</td>
<td></td>
<td>7</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Labd</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PDP11</td>
<td>42</td>
<td></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Pdoc</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Patho</td>
<td>19</td>
<td></td>
<td>7</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Phone</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Patutor</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statcon</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slick</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Ungrad</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>VW</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
Table 6.4. Errors Detected in Students Systems (Fall 1986).

<table>
<thead>
<tr>
<th>System Name</th>
<th>Size (Kbytes)</th>
<th>M.1 reserved words</th>
<th>Unused rules</th>
<th>Unused facts</th>
<th>Unused questions</th>
<th>Unused illegal values</th>
<th>Multiple questions</th>
<th>Multiple methods</th>
<th>Rules using illegal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>AME</td>
<td>30</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMI</td>
<td>52</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automech</td>
<td>102</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barman</td>
<td>14</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECU</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Editor</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expert</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Friz</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Health1</td>
<td>35</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Health2</td>
<td>62</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modex</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motor</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plant</td>
<td>46</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poison</td>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poker</td>
<td>23</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Qaci</td>
<td>38</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rashdec</td>
<td>23</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Readines</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 6.5. Errors Detected in Selected Students Systems (Fall 1985).

<table>
<thead>
<tr>
<th>System Name</th>
<th>Size (Kbytes)</th>
<th>M.1 reserved words</th>
<th>Unused rules</th>
<th>Unused facts</th>
<th>Unused questions</th>
<th>Unused illegal values</th>
<th>Multiple questions</th>
<th>Multiple methods</th>
<th>Rules using illegal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animal</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Autism</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Autism2</td>
<td>16</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chromic0</td>
<td>13</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>19</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Labdes</td>
<td>15</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Solar</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Stutter0</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

6.4 Summary

Ideally we want to build expert systems that pass exhaustive testing and are acceptable by all kind of end users. This goal is not reachable. We will never be able to guarantee the absolute correctness of a system. It is almost impossible to generate and run test cases that exhaust all rules. So, some rules are going to escape developmental testing. Tables 6.1, 6.2, 6.3, 6.4 and 6.5 show that Validator guarantees that each knowledge base construct is checked for validity and consistency. It also guarantees that each rule will fire whenever enough evidence is gathered to satisfy its premises.
CHAPTER 7

FUTURE RESEARCH AND SUMMARY

7.0 Background

Software Automatic and Interactive V&V is easier to perform in an expert system environment than traditional software environment because the expert system's development life cycle has fewer steps and the logic and control are separate. In this dissertation I presented techniques for static Interactive V&V of rule based expert systems. These techniques included checking the syntax and semantics of a knowledge base for erroneous representations and mismatches to ensure the knowledge base correctness completeness and reliability. In chapter two I surveyed the literature of automatic knowledge acquisition (AKA) tools and AV&V tools. AKA and Interactive as well as Automatic V&V tools form an integrated environment to be used together for building a correct and complete expert system. AKA tools are to be used for early prototyping if possible. They are capable of demonstrating the feasibility of the problem. They also help the expert structure and organize his knowledge and build the syntax to be used by the knowledge engineer during the full scale engineering development of the system, AKA tools are also helpful in building the classification modules of the system.
7.1 Future Research

Future research in Interactive V&V of an expert system should go in two directions. The first one is short term, it includes making Validator more powerful by adding more features and components to it. The second direction of research is a long term one, it includes building an expert system shell that has Interactive V&V features as an option built into it.

7.1.1 Short Term Future Research

Future work on Validator includes adding a module to check the completeness and consistency of rules in a knowledge base. This includes (1) checking for redundant rules (rules that have the same set of conclusion and premises), (2) checking for subsumed rules (rules that have the same conclusions and the set of premises of one is a subset of the set of premises of the others), (3) checking for conflicting rules (rules that have the same set of premises and contradicting conclusions) and (4) checking for circular rules (rules that put the system into an infinite loop), for example, the following constructs were meant to allow the user to change his mind about his last answer, and backup to correct it, however what resulted was circular reasoning; it put the M.1 system into an infinite loop.

\[
\text{\texttt{whencached(X=Y)=[(not(Y=oops)) and temp=X].}}
\]
\[
\text{\texttt{whencached(X=Y)={((Y=oops) and (do(reset X),do(reset temp), temp)).}}
\]

Another module that can also be added to Validator is a rule consistency
checking module. This includes (1) searching each rule for internal redundancies, (2) searching each rule for premises that negate each other and (3) searching each rule for matching values for different expressions, for example if a rule tries to match the values of expression1 and expression2 for equality, the intersection of the set of values (legal plus concluded values) of the two expressions should not be empty. Redundancy has no effect on the correctness of a knowledge base, it just increases its size, but contradicting premises and mismatching values will cause the rules that use them to fail.

7.1.2 Long Term Future Research

Some of the solutions to some of the problems can not be implemented by a stand alone Interactive V&V tool. They are dynamic problems that usually arise during run time, the knowledge engineer has no control over them and they have to be incorporated into a shell. So, long term future research includes building an expert system shell that incorporates Validator features plus:

1- A module to handle dynamic arrangement of premises of a rule to insure that the rule will not be searched if it is guaranteed to fail. M.1 for example, seeks the truth value of premises inside a rule sequentially, it does not check if any of the premises is already guaranteed to be false. In the following example from a Media Advisor system, even if the system already has a value in cache for the expression training-budget which is small for example, the inference engine still seeks the sequential truthness of the first three premises.
if stimulus-situation = environmental and stimulus-duration = persistent and appropriate-response = motor and training-budget = large then media = workshop.

2- A module to search the relevant knowledge base rules based on their conclusions and values if necessary. Firing a rule based on its conclusion expression alone does not usually help the inferencing process. It is okay to fire a rule based on its conclusion expression alone if the expression is a goal. Expressions (other than goals) are supposed to be linked to a goal tree based on their values too, so selecting a rule to fire based on its conclusion expression alone might lead to selecting the wrong rule to fire. This selection will lead to the screening of other rules as well as creating an interface problem with the user by asking irrelevant or contradictory questions. Figure 7.1 shows a typical knowledge base with such type of error. If you have a giraffe in mind the system is going to ask you the question concerning the expression fly even though your answer was hooves to the question concerning extremities.
goal = identity.

if fly = yes
then type = bird.

if coat = hair
then type = mammal.

if has-teeth = yes
and pointed-teeth = no and type = mammal
then subtype = ungulate.

if subtype = ungulate
and long-neck = yes
and long-legs = yes
and marking = dark-spots
then identity = giraffe.

question(fly) = 'Does the animal fly?'
legalvals(fly) = [yes,no].

question(coat) = 'What is the coat of animal?'.
legalvals(coat) = [hair, feathers].

question(has-teeth) = 'Does the animal have teeth?'.
question(pointed-teeth) = 'Does the animal have pointed teeth?'.
question(long-neck) = 'Does the animal have long-neck?'.
question(long-legs) = 'Does the animal have long-legs?'.

Figure 7.1 A Knowledge Base That Asks Inconsistent Questions.

3- A module to optimize the search of the knowledge base while seeking values for expressions. Instead of sequentially searching the knowledge base for constructs that can be used to find values for an expression, the knowledge base will be searched for constructs that make the most progress toward completing a goal search (Quinlan, 1983).
7.2 Summary

Checking the consistency of the syntax and the semantics of the knowledge representation helps insure the correctness of the knowledge base. Checking the knowledge base structure and code insures completeness. Debugging the knowledge base insures the system’s reliability. The ideas presented are general and can be applied to any rule based expert system, but the implementation was specific. The goal of this research as well as the goal of verification and validation of conventional software is to produce reliable, correct and complete end user systems.

Interactive V&V techniques presented in this research help in the different areas of an expert system’s life cycle. These include: (1) early error discovery in the prototypes, (2) fewer errors in the delivered system, (3) no logical errors in the delivered system (in theory), (4) increases the productivity of the people involved in building the system, (5) increases the execution efficiency of the system, (6) reduces the development time of the system, (7) improves the maintainability of the delivered system and (8) improves the testability of the system during the different phases of development.
APPENDIX I

SRPRES ENERGY SAVINGS ADVISOR EXPERT SYSTEM
FOR
RESIDENTIAL UTILITY APPLICATIONS†

† This is an updated version of a paper published in the Proceedings of THE 3rd NATIONAL CONF. ON MICROCOMPUTER APPLICATIONS IN ENERGY Nov. 1988, pp 97-103. Tucson AZ.
INTRODUCTION

This Appendix describes the development and structure of SRPRES (Salt River Project Residential Expert System), a residential energy savings advisor, developed by the University of Arizona under contract for Salt River Project (SRP) and Western Area Power Administration (WAPA). The system was designed to provide expertise in support of SRP residential energy audit and information efforts. The first version of SRPRES was designed to interact directly with the customer or indirectly through an information specialist. SRPRES obtains information from the customer on his or her specific house and energy use. It provides specific and easy to follow energy management recommendations for that customer’s specific needs. Later versions with more depth can be designed for use by trained auditors to enhance their effectiveness and productivity.

BACKGROUND

Nineteen percent of the energy consumed in the United States goes to residential buildings and is distributed as follows: 14 percent for heating and air conditioning, 4 percent for water heating and 1 percent for other appliances, reflecting the importance of creating an energy savings program for the residential sector. It has been demonstrated that homeowners are willing to learn more about their home energy use and how to reduce energy costs by investing a little time and money to conserve and manage their energy use. At the same time, utility companies are
encouraging customers to use their energy efficiently in an effort to better control their loads. They are trying to implement useful residential energy audit programs by distributing booklets, training energy auditors and information specialists and offering incentives to customers. For such programs to succeed and be useful, a great deal of effort, expense and personnel is required to train and provide energy auditors, back-up energy experts and energy information specialists to answer customer questions. Uniformity and consistency of advice is a major requirement of an effective utility energy information effort. The information given to a customer has to be consistent and tailored to his specific needs, thereby increasing its usefulness and the probability of the customer action. In the past when customers called SRP Customer Services, they received different recommendations, depending on the energy auditor or information specialist available at the time. Advice that is inconsistent, generalized or hard for the do-it-yourselfer is much less likely to promote customer action. Providing a high level of consistent, readily accessible expert advise to an audit and information program typically strains available resources and does not represent the best allocation of the professional energy expert’s time and capabilities.

For a successful residential energy management program, there is a real need to provide the customer, directly or through energy auditors and information specialists, interactive access to in-depth, flexible and tailored expertise without tying up valuable human experts’ time. The advent of powerful microcomputers and knowledge-based "expert" systems provide a practical solution to these needs and a
more cost effective approach to such residential energy management programs.

**EXPERT SYSTEMS and KNOWLEDGE ENGINEERING**

Expert systems (a product of artificial intelligence research in the area of knowledge-based systems) are knowledge rich, logic oriented and highly interactive computer programs. They effectively mimic the input/output behavior of a human expert in narrow and specific domains. They enhance productivity by making expertise available when it is scarce. They also use heuristics, judgmental and common sense knowledge (which might be fuzzy or incomplete due to the lack of information) to infer or explain their line of thinking while trying to draw a conclusion.

An expert system usually contains (1) a knowledge-base (created by the knowledge engineer) that captures an expert’s knowledge stored in structures called production rules, (2) an inference engine or shell that interprets and uses the knowledge base to drive or control the problem solving scenario and (3) a user interface. The user interface plays an important role in evaluating the level of performance and acceptability of a system by the end user.

**EXPERT SYSTEMS APPLIED to ENERGY INFORMATION EFFORTS**

When considering the application of an expert system to a problem, four basic points should be considered to determine if the problem is appropriate for the successful application of an expert system.
1. There are identifiable residential energy management experts at the utility who are successful and capable of identifying appropriate energy savings actions for specific customers to take.

2. Energy experts can verbalize their thought process, identifying what facts, rules-of-thumb and experience information they consider when forming their conclusions. It can take a while, however, to get to the heart of what they really consider and to get beyond the human expert's embellishment of their process.

3. An energy expert can typically provide useful, tailored advice over the phone by questioning the customer about their home, appliances, energy use patterns and other relevant items and then analyzing the situation.

4. For residential energy management, there is normally a wide range of possible actions that can be taken. Only a few will be the most appropriate ones for the customer to undertake in a particular case.

Based on these points, it appears reasonable that the area of residential energy management programs is one that expert systems could be expected to be appropriately applied to. The next question is whether or not the utility can gain from application of expert system techniques in this area. One key benefit that would be useful to the utility is an increase in employee productivity. An approach that would boost the performance of information specialists and energy auditors would be a tangible benefit. Figure 1 illustrates possible benefits of expert systems to residential energy management programs. There are few energy experts who are
capable and provide a high level of performance in terms of analyzing residential
ergy problems in the energy management and conservation opportunities that exist
and selecting the most appropriate actions for a customer to take. Energy informa-
tion specialists and auditors typically have far less training and experience than the
energy expert. Their performance in analyzing an energy problem is typically much
lower than that of the energy expert. However, these specialists and auditors can be
provided by the utility in much larger numbers and with far less investment due to
the lesser requirement for in-depth training and experience. Finally, the typical
residential customer will have considerably less experience and knowledge about
energy savings techniques and opportunities than that of the energy information spe-
cialists and auditors. While there is some overlap between each of these groups, the
more experience, training and performance capability required, the more it will cost
the utility in employee costs. If with the use of an expert system, the effective per-
formance of the energy information specialists and auditors can be increased to
approach that of the typical energy expert within a relatively narrow and well
defined domain, considerable savings can accrue to the utility.
Performance of typical expert system

Gain in human performance due to using an expert system

Performance of different humans

Percentage of people

Figure 1
SRPRES KNOWLEDGE BASE

SRPRES focuses on providing recommendations for energy savings actions and expert advice for detached, single family homes. The knowledge-base contains 125 rules that capture the experts’ knowledge (Example 1), 80 facts that identify the basic states and outputs of the system (Example 2) and 585 metafacts that control the inferencing and provide a friendly interface with the user (Example 3). Recommendations are provided based on comparisons between the knowledge-base and the customer’s specific situation.

Example 1. A Typical Rule

if water heater = electric
and on whole family (on vacation)
and not turned off (water heater)
then turn off (water heater).

Example 2. Two Typical Facts

turn off message(electric water heater) = '
While on vacation it is best to turn off an electric water heater. It only takes a couple of hours to reheat the water and you will save the energy that would otherwise be lost when the water heater is not being used for an extended period of time'.
type (water heater) = electric.
Do you turn off the circuit breaker of the back-up electric water heating system of the solar water heater during summer?

Example 3. A Metafact

KNOWLEDGE REPRESENTATION

In this system I divided the house into four categories (Example 4).

These categories represent the major energy savings topic areas that SRP focuses on in their residential energy savings information efforts. Each category was subdivided into its different constituent objects (Example 5). The objects are the major items or appliances that energy savings action can be taken for.

| Air conditioning | Water heating | Air infiltration | Window and doors shading |

Example 4. The Four Categories of a House

| Gas water heater | Electric water heater | Solar water heater |

Example 5. The Objects of the Water Heating Category
Example 6. Attributes of the Gas Water Heater Object

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>has (water heater, blanket)</td>
<td>yes</td>
</tr>
<tr>
<td>has (water heater, blanket)</td>
<td>no</td>
</tr>
</tbody>
</table>

Example 7. Values of the Water Heater Blanket Attribute

For each object, attributes are defined (Example 6) and values are specified (Example 7) or obtained to provide the basic facts and specifications that will be used in the production rules.

The knowledge representation scheme (object-attribute-value) and production rules are easy to understand, code and seem to effectively reflect the way the utility experts represent and diagnose energy problems. Another major advantage of this scheme is the visibility of the knowledge, especially since the designer (the knowledge engineer) of the system is not the maintainer. The designer had to put extra effort to developing a readable, commented and clear system to help the maintainer modify the system in the future. I also wrote my own C programs and interfaced it with the system to save a file.

Insulation of walls or attic was not considered in the knowledge base because the utility energy information program does not normally address them. This is because the utility feels that this is a complex problem with limited pay back and that most homeowners do not know the existing R-values nor have ready access to
the interiors of the walls and roofs.

THE REASONING PROCESS

The reasoning process is goal driven. The system starts with a statement of the problem and tries to achieve a set of goals during the consultation session. For each goal the system searches its knowledge-base for its values. If values are found, the system generates subgoals to be achieved from the premises of rules for which conclusions match the current goal. These subgoals constitute the new goals for which values are to be achieved. This process is repeated until no more subgoals can be generated by applying the production rules and the facts of the system. If the system fails to provide values for any goal, the user is asked for these values. Hence, rules are selected by matching their conclusions to a current goal and fired by evaluating their conditions.

SYSTEM DEVELOPMENT

The final system took one year to develop. The first prototype system, developed within three months, was naive and inexperienced. Most of the knowledge was collected from books. The prototype demonstrated feasibility of the problem, educated the knowledge engineer and served as a basis of discussion with the expert during the early stages of development. Most criticism was focused on the difference or mismatch between the way books give recommendations and the way human util-
ity experts give recommendations. Books recommend actions such as shading the cooling system if it is exposed to direct sunlight. According to the utility experts, this is hazardous and the payoff is small. Another textbook recommendation was to move the house thermostat if it is exposed to an air current or direct sunlight, but the energy experts say this is expensive, with no payoff and customers usually do not follow such recommendations.

During the early stages of the project, the designated expert was cautious in criticizing the books. After consulting with the panel, she felt more comfortable in disagreeing with that knowledge and pointing out books shortages such as generality.

I built the expert system based on the knowledge provided by a primary or designated expert. The designated expert and project team also consulted with a panel of utility experts. During the final stages of development, the panel became more involved in the project. Considerable effort was required to resolve conflicts between overlapping, and sometimes conflicting, knowledge of various experts.

The expert system should model the performance of a single expert. This can either be a real human expert or a hypothetical construct. The knowledge engineer should strive to establish a knowledge-base that models a single expert’s knowledge, corrected or enhanced by others, rather than a committee, thus avoiding a confused, conflicting base of knowledge. When resolving conflicts in facts, approaches or methodologies, the knowledge engineer should try to integrate the
appropriate information to form a self-consistent whole that responds as a single expert.

The user interface is menu driven. A dialogue is carried on with the customer, the system asks simple questions and provides a menu of possible answers (Example 8).

```
<table>
<thead>
<tr>
<th>Do you turn off the circuit breaker of the back-up electric water heating system of the solar water heater during summer?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. yes</td>
</tr>
<tr>
<td>2. no</td>
</tr>
<tr>
<td>3. oops</td>
</tr>
</tbody>
</table>
```

Example 8. A Typical M.1 Question

The *oops* option allows the user to recover from mistakes. It allows him to change the answer to the previous question or go back all the way to the beginning of a module. Such an error recovery option was time consuming and error prone. The knowledge engineer had to provide control structures that allows the end user to backtrack to every construct that has been affected by the wrong answer, change the answer and return him to the same point in the consultation.
In response to a question the user has the following options:
1. answer the question,
2. type unknown if he is not sure about the answer,
3. quit the system,
4. ask for explanations,
5. type oops to change previous answers.

Example 9 Set of possible user’s options
At the end of each session, the user has the option of redoing the whole session with the set of possible recommendations already saved for him into a file for him to take.

SYSTEM EVALUATION

At the end of each development stage, I had a panel of evaluators from WAPA and SRP provide comments on the usability of the system. Based on the input from the panels of experts and evaluators, the project team worked with the designated expert to integrate appropriate changes into the knowledge-base while maintaining self-consistency. The system was also tested during a computer show in a shopping mall. The customers were mall visitors who used the system with little help. They received a printout of a file containing the recommendations from SRPRES. They were generally happy with the system’s performance and felt the information provided would be useful to them. It was also suggested that illustrations of the major items under consideration by the program (Figure 2) be provided in conjunction with the program to clarify questions that might arise. When appropriate these illustrations could be referenced by the program. The system was
also tested for internal consistency of the knowledge-base by Validator.

We suggest pictures like this be provided for air conditioners, water heaters etc.

Figure 2
TECHNOLOGY TRANSFER

The major objective of the project was to investigate the usefulness of applying expert systems to utility's residential energy audit and information program. As an experimental development to evaluate a new technique, the development tool (M.1) was selected for power and flexibility, not for distribution purposes. Due to the limiting license agreement with the developers of M.1, Teknowledge, multiple distribution copies are not allowed of the run-time system of the expert system shell. Once this program is evaluated by SRP and WAPA, it is recommended that a follow-up effort be undertaken to make any useful revisions of the knowledge base and user interface and to transfer them to an expert system shell that would permit wide distribution by SRP and WAPA. This would permit the widest possible technology transfer to other customers and maximize the program's usefulness to SRP and WAPA. Since the knowledge base is utility/location specific, modified or new knowledge bases should be developed tailored to the needs of each SRP and WAPA customers' group (or knowledge base related group of SRP and WAPA customers) utilizing this approach.

CONCLUSION

One underlying goal was to provide SRP and WAPA with some experience with expert systems so they could begin to explore how such techniques might be usefully employed by utilities to enhance productivity and ameliorate some of
their problems. This effort was successful in providing an introduction to expert systems and knowledge engineering through an exercise that involved WAPA and SRP personnel and resulted in a useful example. Many other possible utility applications have been identified for future development including areas such as:

1. An expert advisor for in-home energy auditors.

2. Troubleshooting and equipment diagnostic expert systems.

3. Power system operations advisor for Voltage-VAR control efforts.

The proper application of expert systems and knowledge engineering can quickly go beyond the novelty stage and become a useful tool for utilities to increase productivity and effectiveness, improve reliability and to extend the usefulness of valuable and hard-won knowledge and experience.
REFERENCES


**Machine Intelligence.**


Shortliffe E.H. (1981) ONCOCIN an expert system for Oncology protocol


