Lehigh University
Lehigh Preserve

Theses and Dissertations

1991

A study of problem-solving strategies and their effect on the development of knowledge acquisition tools

Buffi Lynn Clemenko
Lehigh University

Follow this and additional works at: https://preserve.lehigh.edu/etd

Part of the Electrical and Computer Engineering Commons

Recommended Citation
Clemenko, Buffi Lynn, "A study of problem-solving strategies and their effect on the development of knowledge acquisition tools" (1991). *Theses and Dissertations*. 5437.
https://preserve.lehigh.edu/etd/5437

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

A STUDY OF PROBLEM-SOLVING STRATEGIES AND THEIR EFFECT ON THE DEVELOPMENT OF KNOWLEDGE ACQUISITION TOOLS

by Buffi Lynn Clemenko

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1991

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

May 6, 1991

Date

Advisor in Charge

CSEE Department Chairperson

ACKNOWLEDGEMENTS

I would like to thank the people who assisted me in preparing this thesis. I owe special thanks to Dr. Donald Hillman whose work and advice provided extremely helpful with the development of this paper.

CONTENTS

ABSTRACT 1

1. Introduction 2
  1.1 Purpose 2
  1.2 Overview of Thesis 2
  1.3 Background 3
    1.3.1 Key Terms 3
    1.3.2 Overview of Expert Systems 4
    1.3.3 Overview of Expert Knowledge 5

2. Problem Solving Techniques 11
  2.1 Overview 11
  2.2 Cover-and-Differentiate 14
  2.3 Propose-and-Revise 16
  2.4 Acquire-and-Present 19

3. Knowledge Acquisition Tools 23
  3.1 History 23
  3.2 Overview 24
  3.3 SALT 24
    3.3.1 History 25
    3.3.2 Problem-solving Strategy 25
    3.3.3 Acquiring the Knowledge-base 26
    3.3.4 Knowledge Representation 29
    3.3.5 Convergence 30
    3.3.6 Troubleshooting the Knowledge-base 32
    3.3.7 Summary 34
  3.4 MOLE 34
    3.4.1 History 35
    3.4.2 Problem-solving Strategy 35
    3.4.3 Knowledge Representation 36
    3.4.4 Control Knowledge 37
    3.4.5 Acquiring the Knowledge-base 39
    3.4.6 Acquiring Initial Symptoms 40
    3.4.7 Acquiring Covering Knowledge 41
    3.4.8 Acquiring Differentiating Knowledge 42
    3.4.9 Summary 43

4. Thesis Summary 43

REFERENCES 45

Vita 47

LIST OF FIGURES

Figure 1. Example 1.a 5

Figure 2. Example 1.b 6

Figure 3. Example 1.c 7

Figure 4. Three factors which influence human decision-making 10

Figure 5. Overview of interdependencies between knowledge acquisition tools and expert systems 13

Figure 6. Network Representation of a MOLE knowledge base [Eshelman 88] 37

ABSTRACT

Knowledge acquisition is typically the bottleneck of expert system development. Thus, knowledge engineers strive to automate knowledge acquisition with the development of knowledge acquisition tools. Problem-solving strategies are the core of domain-specific knowledge acquisition tools. By reviewing problem-solving strategies we can gain insight into how they are used to guide knowledge acquisition tools during the interrogation of domain experts.

One objective of this thesis is to provide an overview/tutorial of problem-solving strategies for expert systems. A secondary purpose is to specify how problem-solving strategies can be used to enhance the capabilities of knowledge acquisition tools. Three implementations of domain-specific knowledge acquisition tools will be reviewed. Emphasis is placed on applicable domains, knowledge representation, and problem-solving strategies.

Specifically, the problem-solving strategies cover-and-differentiate, propose-and-revise, and acquire-and-present will be discussed. In addition, the knowledge acquisition tools MOLE, SALT, and LAPS are reviewed.

1. Introduction

1.1 Purpose

One objective of this thesis is to provide an overview/tutorial of problem-solving strategies for expert systems. A secondary purpose is to specify how problem-solving strategies can be used to enhance the capabilities of knowledge acquisition tools. Emphasis is placed on applicable domains, and existing implementations of the propose-and-revise and cover-and-differentiate problem-solving strategies.

1.2 Overview of Thesis

The following is a general description of each section of this thesis:

Section 1.4	outlines the relationship between expert knowledge, expert systems, and knowledge acquisition.
Section 2.1	defines problem-solving strategies and how they are used.
Section 2.2	defines the cover-and-differentiate problem-solving strategy.
Section 2.3	defines the propose-and-revise problem-solving strategy.
Section 2.4	defines the acquire-and-present problem-solving strategy.
Section 3.1	gives a brief history of knowledge acquisition tools.
Section 3.2	outlines the importance of knowledge acquisition tools.
Section 3.3	gives a detailed description of SALT, a knowledge acquisition tool which uses the propose-and-revise problem-solving strategy.
Section 3.4	gives a detailed description of MOLE, a knowledge acquisition tool which uses the cover-and-differentiate problem-solving strategy.
Section 4	provides a summary of the thesis.

1.3 Background

Knowledge acquisition is the process of acquiring knowledge from a human expert in a particular field for the development of an expert system. Discussed below are key terms, an overview of expert systems, and an overview of expert knowledge. The overview of expert systems will explain the relationship between knowledge acquisition and the development of an expert system. The overview of expert knowledge will give a detailed description of what knowledge is and how it is acquired.

An overview of expert systems is necessary because the main goal of the automation of knowledge acquisition is to aid in the development of expert systems. An overview of expert knowledge is essential to understanding the complexity of the knowledge acquisition process. The accuracy of an expert system is dependent on the quality of knowledge acquired from the expert.

1.3.1 Key Terms

The following terms are used in this paper and are defined to familiarize the reader with the current technical terms dealing with knowledge acquisition.

Domain Expert:

A person who has expertise in a certain field or domain.

Control Knowledge:

Knowledge which controls the direction the inference engine takes towards determining a solution.

Expert Systems:

Knowledge-based programs which are designed to emulate the problem-solving techniques of a human expert in a specific domain.

Knowledge Acquisition:

The process by which knowledge is acquired from a human domain expert and transformed into structured rules for expert system inferencing.

Knowledge Engineer:

The person responsible for acquiring, structuring and programming expert knowledge.

1.3.2 Overview of Expert Systems

An expert system is a computer program designed to emulate the problem solving techniques of an expert in a specific field. An expert system consists of two essential parts: a knowledge-base, and an inference engine. The knowledge-base is where the knowledge acquired from the domain expert is stored, while the inference engine is responsible for drawing conclusions from that knowledge.

In order for the expert system to formulate conclusions which directly reflect those of the domain expert, all information stored in the expert system's knowledge base must be retrieved directly from the human expert. It is the knowledge engineer's job to retrieve all information needed to fill the knowledge-base from the domain expert. This time-intensive procedure is referred to as the "knowledge acquisition process" and is usually the bottleneck of expert system development. It is for this reason that knowledge engineers strive to automate the

knowledge acquisition process.

1.3.3 Overview of Expert Knowledge

Knowledge is defined as information which has been retrieved, categorized, used, and updated. When a piece of information is retrieved by a human, it is categorized to reflect the relationship between that piece of knowledge and all other pieces of knowledge currently stored in the human's mind. As the human uses that piece of knowledge, new relationships between other pieces of knowledge may be discovered. The knowledge pieces are then updated to reflect the new relationships.

Knowledge is constantly being retrieved, categorized and updated. For example, suppose a child met a black dog for the first time. During the meeting, the dog happened to bite the child. Figure 1 shows the way the child might represent the information retrieved about the dog.

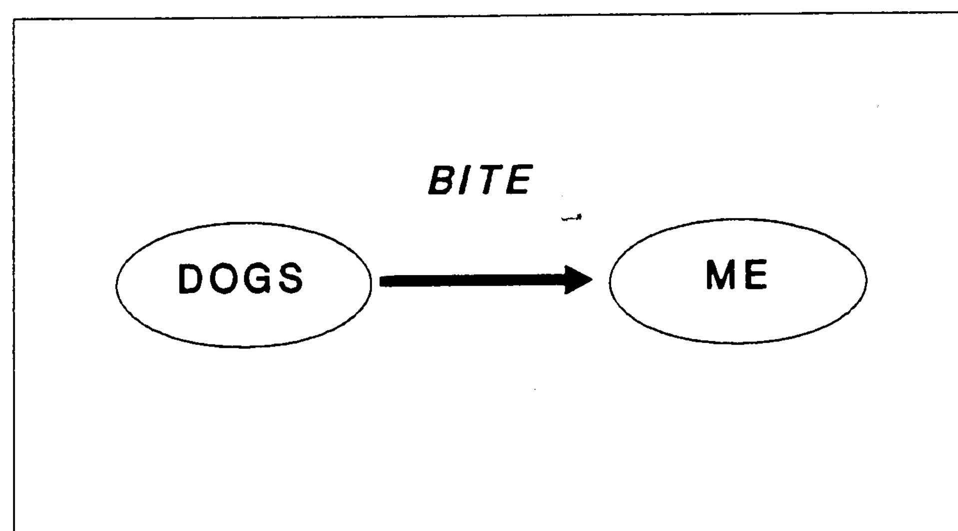

Figure 1. Example 1.a

Figure 1 shows that there is a single piece of knowledge which represents dogs. The relationship between the child and dogs shows that dogs bite.

The next encounter the child has is with a brown dog. This dog does not bite the child, but instead licks the child's face. This is new information which the child must add to his or her internal knowledge base. The new information is added to the knowledge currently stored. The old knowledge must now be updated. The way in which the child would now represent the knowledge of dogs is show in figure 2.

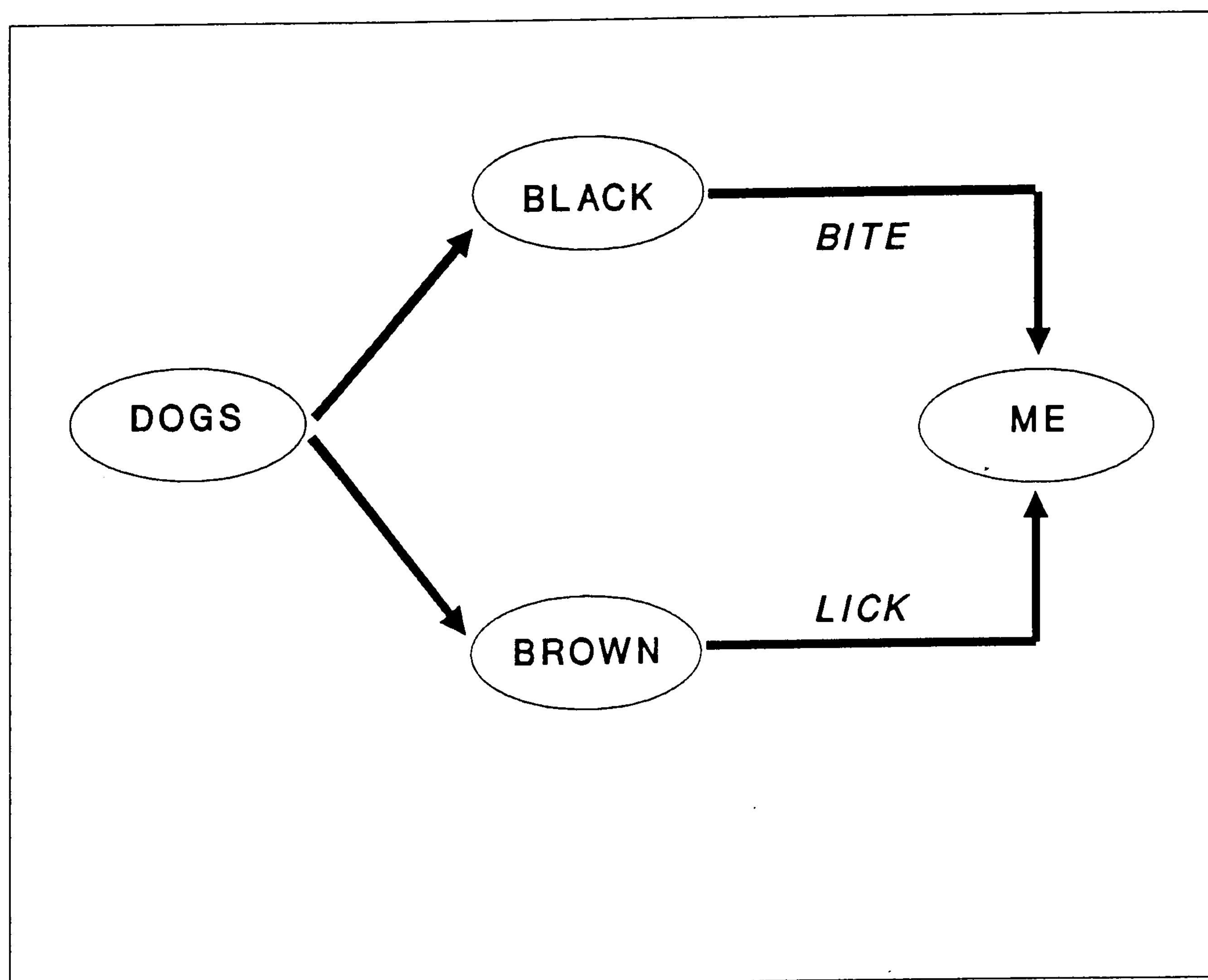

Figure 2. Example 1.b

Figure 2 shows that the piece of knowledge representing dogs is now broken into two separate categories: black dogs and brown dogs. The relationship between the child and dogs gave been revised to represent that black dogs bite and brown dogs lick.

The next encounter the child has is with a black dog. This dog does not bite the child, but again licks the child's face. The child's internal knowledge base is once again reorganized to

represent the newly acquired knowledge. Figure 3 shows how the child's knowledge of dogs might finally be represented.

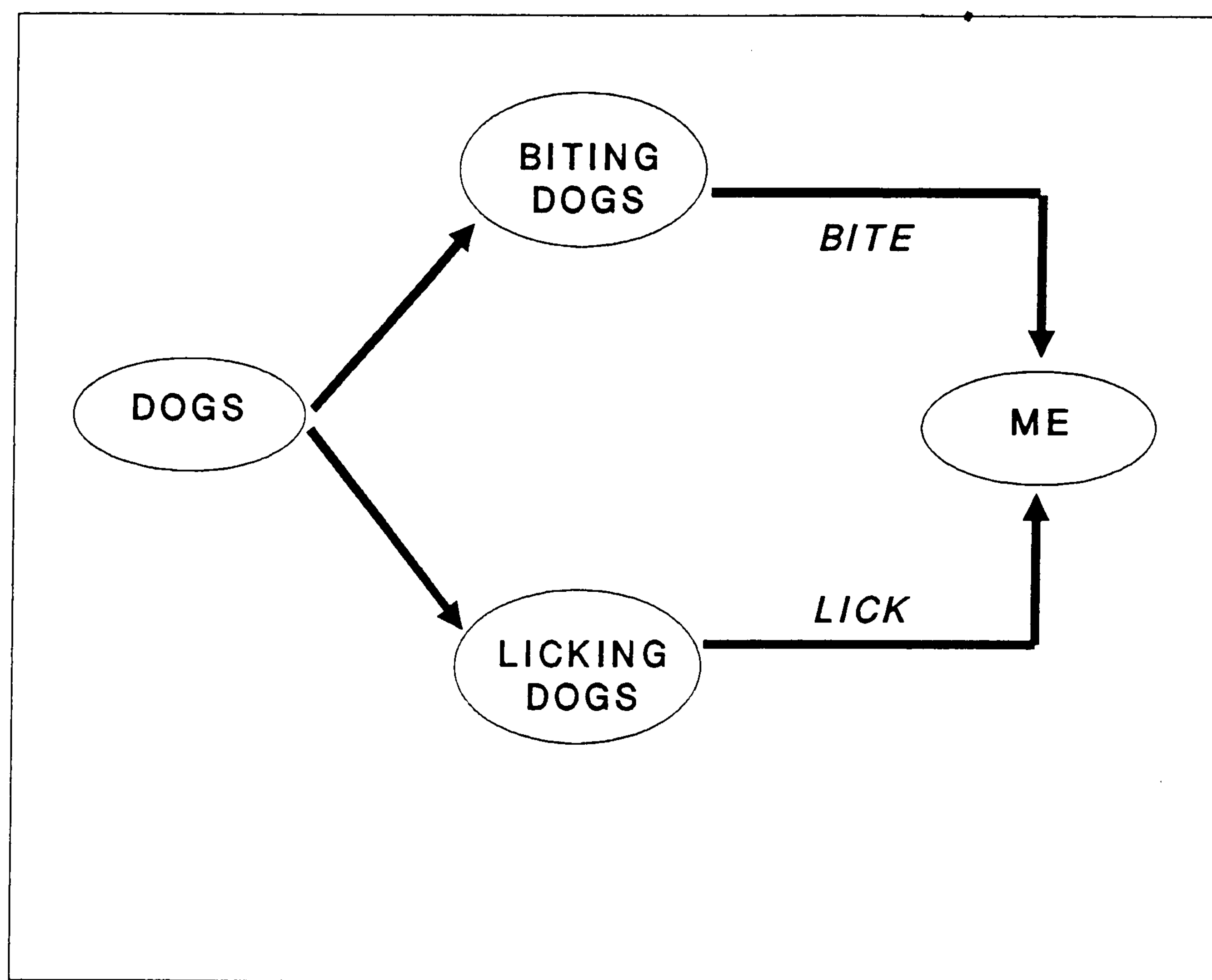

Figure 3. Example 1.c

Figure 3 shows that the piece of knowledge representing dogs is now broken into two new categories: biting dogs and licking dogs. The relationship between the child and dogs has been further revised to represent that biting dogs bite and licking dogs lick.

This was a simple example of how human knowledge is continually being updated as new information is acquired. It was through experience that the child was able to acquire new information about dogs. It was also through experience that the child was able to refine his or her existing knowledge about dogs.

Expert knowledge, or expertise, is defined as the level of experience an expert has in a particular domain. A goal of the knowledge acquisition process is to capture the expertise of the domain expert. Through experience, the mind is able to make interconnections between separate pieces of knowledge currently stored. Experience is the way unrelated facts are transformed into expert knowledge [McGraw 89].

Experience is what gives an expert expertise in a specific domain. That same experience often makes verbalizing his or her thinking process difficult. The more experience a human has performing a specific task, the more that task becomes second nature to him or her. The intermediate steps taken to solve the problem usually become "rules of thumb", or heuristics. These are not easily deciphered by the knowledge engineer because the intermediate steps used to form the conclusion are hidden. When forming conclusions, the domain expert may unconsciously be taking into consideration other related facts. Because the expert is using "rules of thumb", verbalizing his or her own thinking process becomes difficult.

When a domain expert is supplying knowledge about how to perform a task or solve a problem, important information is often unintentionally left out. It is difficult for the human expert to convey all of the outside information which contributes to the decisions the he or she makes. If a driver is asked how to explain the steps involved in driving a car the answer might be similar to:

1) GET INTO CAR
2) START ENGINE
3) DRIVE CAR
4) TURN ENGINE OFF
5) GET OUT OF CAR

Although this set of steps is adequate for driving under normal conditions, there is information missing. If the knowledge engineer would have asked the driver to list the steps taken to drive a car ***in the rain*** the following list might have been given:

1) GET INTO CAR
2) START ENGINE
3) TURN ON WIPERS
4) DRIVE
5) TURN ENGINE OFF
6) GET OUT OF CAR

The fact that the weather has an effect on the decisions made during a driving session was left out the first time. This is because the driver does not consciously ask himself or herself "Is it raining out? If so, then I'd better put the wipers on.", but the weather is outside information which has an effect on the steps the driver will take. To complete the knowledge-base on how to drive a car other information such as time of day, temperature, and driving conditions would need to be included.

Figure 4 shows the factors which influence the decisions humans make when solving a problem or completing a task.

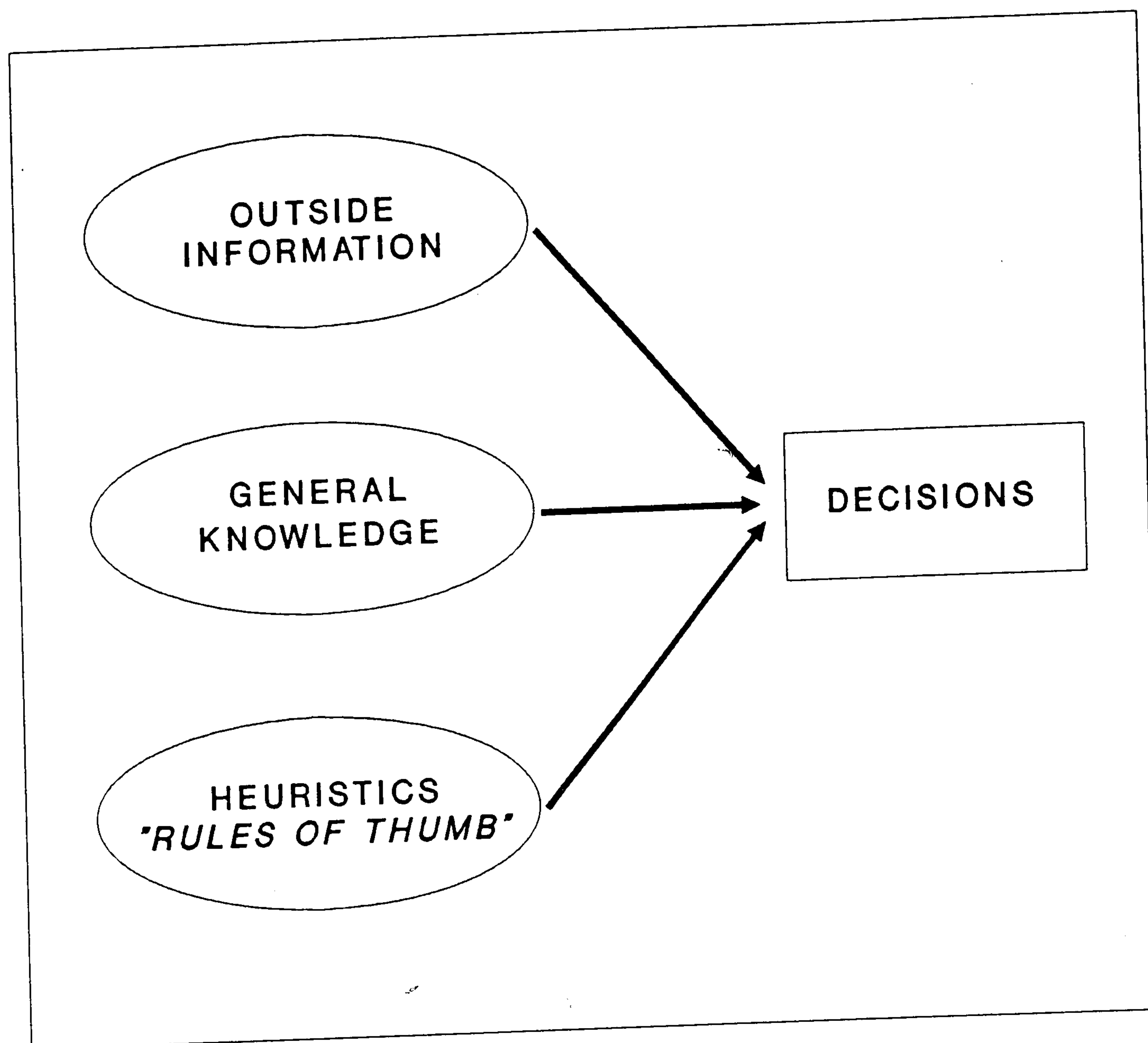

Figure 4. Three factors which influence human decision-making.

By asking the driver the appropriate questions, obtaining all of the outside information, general knowledge, and heuristics which influence driving decisions would be simple for the knowledge engineer. On the other hand, when the domain becomes more complex and foreign to the knowledge engineer, the missing information becomes more difficult to extract from the expert. If the knowledge engineer knew nothing about driving, the first explanation might have seemed to be a suitable one. Because the knowledge engineer usually has limited knowledge of the domain with which he or she is working, detecting when information is missing is not an easy task.

The process of acquiring expert knowledge is difficult and has not been well defined. The ways to acquire knowledge are usually domain-specific and not easily generalized for all domains. The knowledge engineer must conceptualize the structure of the expert's knowledge, and the way in which he or she uses it in order to solve a problem. In addition, conveying each step taken when performing a task or solving a problem is difficult for a human. For these reasons, knowledge acquisition tools try to emulate the problem-solving techniques of an expert. By mimicking the inference process of the human, the knowledge acquisition tool simplifies the process of finding inconsistencies and incompleteness in the knowledge-base. Knowledge acquisition tools which predefine a problem-solving strategy have the advantage of understanding how the knowledge will be used in order to solve a problem.

2. Problem-Solving Techniques

Problem-solving is defined as the identification, selection, and implementation of a sequence of actions that accomplish some task within a specific domain [McDermott 88]. Discussed below is an overview of problem solving techniques followed by a description of three common problem-solving techniques: cover_and_differentiate, propose-and-revise, and acquire-and-present. Tools which use the propose-and-revise, and cover-and-differentiate methods will be discussed later. The acquire-and-present method will not be referred to in later discussions so therefore, only a brief review is given.

2.1 Overview

Problem-solving strategies are designed to mimic the problem solving techniques of a human expert. Many knowledge acquisition tools use problem solving strategies to predefine how the inference engine will use the domain knowledge. The choice of a problem-solving strategy is

domain dependent and affects the organization of the extracted knowledge. A problem-solving strategy must be specific enough to guide the domain expert in defining, analyzing, and testing a knowledge-base [Klinker 88].

When designing a general expert system, a knowledge engineer strives to create a domain-independent inference engine with a domain-specific knowledge base. Separating the inference engine from the knowledge base allows an expert system to be used for many different domains. Limiting an inference engine to using a specific problem-solving strategy limits the domains with which it can be used. On the other hand, using a specific problem-solving strategy increases the inference capabilities for the domains with which it can be used.

A problem-solving method defines the flow of control of the system. It controls the sequence of events needed to form a conclusion, and is designed to emulate the way in which a domain expert solves a problem. At each step of the inference process, the problem-solving method provides a way of identifying possible actions. From that list of actions, the problem solver determines the best possible action. The knowledge used to control the problem solver is stored in the inference engine and is separate from the knowledge base.

Figure 5 presents an overview of the interdependencies between a knowledge acquisition tool and its generated expert systems adapted from Klinker's overview of KNACK [Klinker 88].

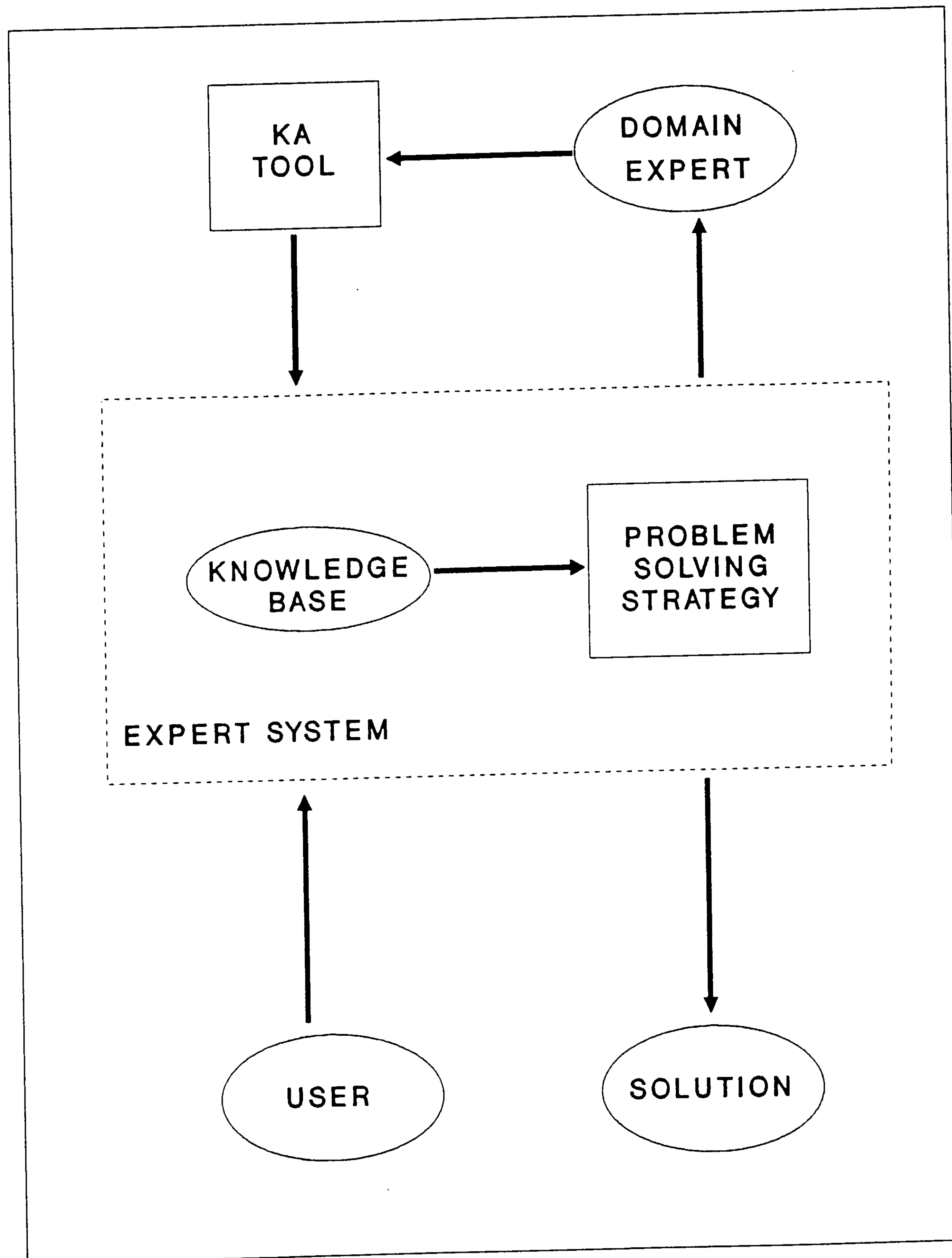

Figure 5. Overview of interdependencies between knowledge acquisition tools and expert systems.

Tools which automate the knowledge acquisition process often use a single problem-solving strategy. Although many exist, the three common problem-solving strategies discussed below include: cover-and-differentiate, propose-and-revise, acquire-and-present.

2.2 Cover-and-Differentiate

The cover-and-differentiate method of problem-solving is defined as the cyclic process of finding a set of initial symptoms, generating a set of explanations for the symptoms, and differentiating between the possible explanations in order to converge on a single solution. The cover-and-differentiate problem-solving strategy is intended to be used primarily for diagnostic tasks. These are tasks which choose a solution (or diagnosis) from a set of possible solutions. Diagnosing diseases would be an example of this type of problem. A knowledge-based system using the propose-and-revise method would select a solution from a predefined set of solutions rather than constructing a unique solution.

An expert solves a problem using the cover-and-differentiate method by first proposing possible solutions that will provide an explanation for the initial symptoms. These solutions are referred to as the covering knowledge. For example, suppose the initial symptom is: The car won't start. The set of covering knowledge might consist of the following:

1) The battery is dead.

2) The car is out of gas.

3) The starter is broken.

The goal of the problem-solver is to converge on a single solution from this set of solutions. Each solution in the set represents a possible candidate for the final solution. The expert will proceed by searching for information that will differentiate the candidate solutions. To differentiate the first candidate from the set of candidates, the following differentiating

knowledge might be obtained:

1) If the battery is dead the lights will not go on.

The next step in this process would be to obtain knowledge which would differentiate the second explanation from the competing explanations in the set. This is an iterative process of obtaining covering knowledge and then differentiating between the parts until all of the covering knowledge has been differentiated. When all of the differentiating knowledge is acquired, the expert is able to choose a final solution by either confirming that solution or rejecting all competing solutions.

The cover-and-differentiate problem-solving method can be classified by the following two rules [Eshelman 87]:

1) For each symptom or abnormal event, propose a set of covering alternatives or explanations.

2) Seek information that will help differentiate these alternatives.

Many problems such as diagnosing diseases, or diagnosing engine problems, can be solved by the process of finding a set of possible solutions and then differentiating between them is a common way for domain experts to solve a problem. Cover-and-differentiate is a common method of solving diagnostic problems.

Knowledge acquisition tools which use the cover-and-differentiate method of problem-solving must focus on this method while acquiring the knowledge from the expert. Extracting knowledge from a domain expert using the cover-and-differentiate method can be cumbersome. Although experts tend to have little problem providing an initial set of symptoms and providing possible explanations for them, domain experts tend to find it more difficult to provide adequate differentiating knowledge. Human experts typically find it difficult to specify exactly

how a piece of information helps to differentiate possible solutions.

MOLE, a knowledge acquisition tool developed by Carnegie Mellon University, generates expert systems that use the cover_and_differentiate problem-solving method. MOLE seeks to elicit the differentiating knowledge from the expert without requiring the expert to have a deep understanding of how it is structured. By separating the covering knowledge from the differentiating knowledge, MOLE is able to disambiguate an under-specified knowledge base and to interactively refine an incomplete knowledge base [Eshelman 87]. MOLE is discussed in detail in section 3.3.

2.3 Propose-and-Revise

The propose-and-revise method of problem-solving is defined as the cyclic process of proposing a design, checking for violations of design constraints, and revising the design according to constraint violations. The propose-and-revise problem-solving strategy is intended to be used primarily for constraint-satisfaction tasks. These are tasks which need a designed solution, including values for all design arguments, which meet all defined constraints. Creating a schedule would be a simple example of this type of problem. A knowledge-based system using the propose-and-revise method would construct a solution or design rather than select an existing one.

A solution generate using the propose-and-revise method is constructed by first proposing values for the design parameters one at a time. Each parameter value is checked to see if it violates any of the predefined constraints on the design. If a constraint is violated, the system will make a correction by changing the design values accordingly. The way in which the system determines an appropriate correction is system dependent. Some systems may choose

the least costly change and apply it to the design. This process of proposing a design and revising it accordingly is continued until all design parameters have values which do not violate design constraints.

The propose-and-revise method is uses the following process to control the problem-solving process [McDermott 88]: .

1). Extend a design by assigning a value to a design parameter.

2). Identify constraint violations on the extension just formed;
   if none, go to step 1.

3). Generate a set of potential corrections for the constraint violation.

4). Select the least costly correction not yet attempted.

5). Tentatively modify the design according to the chosen correction.

6). Identify constraint violations on the modification just formed;
   if any, go to 4.

7). If the design is incomplete, go to 1.

The following example shows how these steps are used to solve a constraint-satisfaction task. The example problem will be to design the perfect family. The following is a list of all design parameters and how each will obtain an initial value.

DESIGN PARAMETERS:	INITIAL VALUES:
Age-of-father	From user
Age-of-mother	= (*Age-of-father* - 1)
Age-of-son	= (*Age-of-father* - 25)
Age-of-daughter	= (*Age-of-brother* - 2)

Each design parameter has at least one design constrain in this example. A design constraint limits the values of the parameters. The design parameters for this example have the following design constraints:

DESIGN PARAMETERS:	CONSTRAINTS:
Age-of-father	> *Age-of-mother*
Age-of-mother	< *Age-of-father*
Age-of-son	< *Age-of-mother*
	= (*Age-of-father* + 25)
Age-of-daughter	= (*Age-of-brother* + 2)
	< *Age-of-mother*
	< (41 - *Age-of-mother*)

For an actual task, all constraints would also include a set of one or more corrections in the event that the constraint is violated. For this example, the only constraint which will need a correction is the third constraint on the parameter *Age-of-daughter*. The correction for this constraint would be to set the *Age-of-mother* = (40 - *Age-of-daughter*). The following set of steps shows how this example problem can be solved using the propose-and-revise problem-solving strategy. Each step will be numbered corresponding to one of the seven propose-and-revise problem-solving steps previously defined.

Step 1. Acquire *Age-of-father* from user. (*Age-of-father* = 35)

Step 2. No constraint violations found.

Step 1. Acquire *Age-of-mother* from equation. (*Age-of-mother* = 34)

Step 2. No constraint violations found.

Step 1. Acquire *Age-of-son* from equation. (*Age-of-son* = 10)

Step 2. No constraint violations found.

Step 1. Acquire *Age-of-daughter* from equation. (*Age-of-daughter* = 8)

Step 2. Constrain violation : (*Age-of-daughter* < 41 - *Age-of-mother*)

Step 3. Set of corrections : (*Age-of-mother* = 40 - *Age-of-daughter*)

Step 4. Select least costly correction.

Step 5. Modify design. (*Age-of-mother* = 32)

Step 6. No constraint violations found.

Step 7. Design complete:

Age-of-father = 35

Age-of-mother = 32

Age-of-son = 10

Age-of-daughter = 8

The design is complete because all design parameters have values and no constraints are violated. By using the propose-and-revise problem-solving strategy we were able to create a design of the perfect family which met all predefined design constraints.

This problem-solving method is used by SALT, a knowledge acquisition tool developed at Carnegie Mellon University. SALT will be described in more detail later.

2.4 Acquire-and-Present

The acquire-and-present problem-solving method is defined as a process of acquiring all necessary information from the expert and then presenting a report of the acquired information. This method used primarily for reporting tasks. The acquire-and-present method is complementary to the problem-solving strategies previously described. While other problem-solving strategies concentrate on the inferencing process, acquire-and-present concentrates

on the extraction of knowledge and the generation of a report from that knowledge. It is most useful for tasks which require a large amount of information to be acquired and presented in a standard form [Klinker 88].

In order to be suitable for the acquire-and-present problem-solving method, a task must have the following characteristics [McDermott 88]:

1) It is possible to document the task with a report.

2) A report for the task will cover a relatively small set of concepts.

A task is most suitable for the acquire-and-present method if it is not just possible, but essential to document the task with a report.

The following are a set of steps which the acquire-and-present problem-solver takes in order to perform a task [McDermott]:

1) Identifies all relevant pieces of information that are appropriate to acquire next.

2) From that set of relevant pieces, one piece of information is chosen to be acquired.

3) A strategy is chosen for acquiring the information.

4) Applies the strategy towards acquiring the information.

5) Integrates the new information with the information previously acquired.

6) Repeats these steps until all relevant information is acquired.

7) Generates a report which documents the task.

KNACK, a knowledge acquisition tool developed by Carnegie Mellon University, is an example of a knowledge acquisition tool which uses the acquire-and-present problem-solving method. KNACK generates an expert system capable of producing structured reports about a specific domain. KNACK structures the report according to the information provided by the domain expert. The domain knowledge acquired from the expert specifies how to generate a report about the specific domain. KNACK acquires the domain knowledge by simply having the domain expert enter the following information [McDermott 88]:

1) A domain model

2) A sample report

3) Sample strategies for acquiring specific information

One of the goals of KNACK was to be simple enough for a domain expert with no programming experience to be able to easily enter his or her domain knowledge.

Some of the tasks KNACK has been used for include: assisting with the creation of a project proposal, assisting with the definition of requirements for software systems, and reporting on designs of electromechanical systems that may be suboptimal form a hardening perspective. To give an indepth description of how the acquire-and-present method is used, an example of a KNACK-built system for creating project proposals follows.

One of the fist steps in starting a new project is the creation of a project proposal. The proposal must contain all relevant information concerning the project in a concise, descriptive manner. The purpose of an expert system developed by KNACK is to help the project leader create the proposal. The expert system will ask the project leader information such as objectives, functionality, and methodology. After obtaining all necessary information from the project leader, the expert system will generate a proposal [McDermott 88].

The knowledge acquisition tool used to create the expert system will acquire all domain-specific information from the domain expert. [McDermott 88] provides an example of task-specific knowledge entered by the domain expert:

> DOMAIN: Assisting with the creation of a project proposal:
>
> *Domain model:* project, objective, task, software (concepts); name, description (concept characteristics for software); KNACK, WRINGER (concept representatives for software).
>
> *Sample report fragment:* The objective of the NAC WRINGER project is to refine KNACK, a knowledge-acquisition tool currently being developed at CMU, so that it can be used to build expert systems that assist with the design of computer networks.
>
> *Sample strategy:* (Question) What are the objectives of the NAC WRINGER project?

This information is what the expert system uses to guide the questioning of the project leader. Because the acquire-and-present problem-solving strategy is complementary to other strategies, it is possible to use a acquire-and-present in conjunction with other methods. The acquire-and-present method can act as a front end to systems which require a large amount of information to be input. It can also be used as a back end for systems which require reports to be generated. A later version of KNACK combines the acquire-and-present method with the propose-and-revise method to broaden the scope of the propose-and-revise systems. Combining the two problem-solving strategies produces expert systems which can handle constructive tasks that need to acquire a large amount of information or require that a document be produced.

When specifying a problem-solving strategy, a knowledge acquisition tool must use the framework provided by the problem-solving strategy to extract knowledge from the domain expert. This is a weakness in that the expert may be required to enter knowledge in a structured format that is unfamiliar to him or her [Klinker 88].

3. Knowledge Acquisition tools

Knowledge acquisition tools are defined as tools which automate the process of extracting knowledge from a domain expert in order to develop an expert system. Discussed below is a history and overview of knowledge acquisition tools followed by descriptions of a few of the tools researched along with some insights obtained from each. Each is an example of a specialized knowledge acquisition tool which uses one of the previously defined problem solving methods.

3.1 History

Developers of expert systems have come to find that the task of knowledge acquisition is usually the most difficult. It is a slow process which can significantly effect the development time of any knowledge-based system. For this reason, much work is being done on the automation of knowledge acquisition. Research is being done world wide on the development of knowledge acquisition tools as a means of decreasing the time involved in developing an expert system.

By 1985, knowledge acquisition had become such a prominent area of research that the American Association of Artificial Intelligence (AAAI) agreed to sponsor the first Knowledge-based Systems for Knowledge Acquisition Workshop (KAW). It was organized in hopes of preventing the duplication of research and providing a means by which all current research could be shared and hopefully integrated. As a result of this first Knowledge Acquisition Workshop, organizations including the Institute of Electrical Engineers, and the Association of Computing Machinery sponsored subsequent workshops [Boose 88a].

Besides preventing the duplication of research, the collaboration of research relevant to the automation of knowledge extraction, allows knowledge engineers to learn from the limitations encountered with previously developed tools. Researching existing tools gives the needed basis for beginning design on a knowledge acquisition tool.

3.2 Overview

Tools developed to automate the knowledge acquisition process can be broken into two distinct categories: general tools, and specialized tools. General tools such as CYC [Lenat 86] and SOAR [Laird 87] make no assumptions about the problem-solving methods. Other tools such as SALT, KNACK, and MORE focus on a single problem-solving strategy. Although specialized tools are limited to specific domains, in those specific domains they are more powerful then general tools.

In conjunction with a tool's problem-solving method, another useful distinction between automated, specialized knowledge acquisition tools is whether they create expert systems that select or that construct a solution [Clancey 84]. MOLE is an example of a knowledge acquisition tool which selects a solution from a predefined set, while SALT constructs a solution which fits a set of constraints [Klinker 88].

3.3 SALT

SALT is an example of a knowledge acquisition tool which uses the propose-and-revise method of problem-solving. Discussed below is SALT's history, problem-solving strategy, acquiring the knowledge base, knowledge representation, convergence, troubleshooting the knowledge-base, and the summary.

3.3.1 History

SALT is a knowledge acquisition tool designed at Carnegie Mellon University. SALT generates a domain-specific knowledge base compiled into rules in OPS5 [Morik 88]. SALT has been used successfully to develop two commercial expert systems. VT, the first expert system developed using SALT, has been used by Westinghouse Elevator Company to custom design elevator systems. One year after the expert system began operation, the number of rules in the knowledge base had tripled. The second expert system developed using SALT was a prototype for a flow_shop scheduler. The schedular routes an order for an escalator or elevator system from the department for engineering, to the manufacturing department, and finally to the department responsible for delivering it [Stout 88].

Using the propose-and-revise problem-solving strategy limits the domains which SALT can be applied. On the other hand, SALT gains power from its understanding of the propose-and-revise method. The expert systems generated by SALT have dealt with domains where the propose-and-revise method of problem-solving is preferred.

3.3.2 Problem-solving Strategy

SALT is used for generating expert systems which use a propose-and-revise problem-solving strategy. Because SALT predefines its problem-solving strategy to be propose-and-revise, it is limited to creating expert systems for synthesis-type domains. Synthesis-type domains involve problems which are solved by constructing solutions rather than choosing from a set of solutions [Garg-Janardan 88].

The propose-and-revise method of problem-solving is defined as the cyclic process of proposing

a design, checking for violations of design constraints, and revising the design according to constraint violations. This process of proposing a design and revising it accordingly is continued until all design parameters have values which do not violate design constraints. The propose-and-revise problem-solving strategy is discussed in detail in section 2.3.

By assuming the propose-and-revise method to be the problem solving strategy used by the expert system, SALT is able to easily structure the knowledge acquired from the domain expert in order to form a working expert system. The following sections will describe how SALT's problem-solving technique is used in representing, analyzing, and applying domain-specific knowledge.

3.3.3 Acquiring the Knowledge-base

The knowledge acquisition techniques used by SALT are highly dependent on the chosen problem-solving strategy. Predefining the problem-solving strategy gives the knowledge acquisition tool an understanding what type of knowledge the problem-solver will need in order to solve a problem. The propose-and-revise method guides the knowledge acquisition process. The three kinds of knowledge SALT's propose-and-revise problem solver needs to acquire from the domain expert take the following form [Marcus 88]:

1. PROPOSE-A-DESIGN-EXTENSION
2. IDENTIFY-A-CONSTRAINT on a part of the design
3. PROPOSE-A-FIX for a constraint violation

The "PROPOSE-A-DESIGN-EXTENSION" is used to define design parameters and propose initial values for those parameters. The "IDENTIFY-A-CONSTRAINT" is used to define constraints placed on the design parameters. These are conditions the design parameters must meet in order for the design to be accepted. The "PROPOSE-A-FIX" is information supplied by the

domain expert which defines a set of possible corrections for constraint violations. This will give possible design corrections in the event that a value for a design parameter violates a design constraint.

SALT interacts with the user via a simple interface. SALT's main interface is the following list of possible choices [Marcus 88]:

1. PROCEDURE	Enter a procedure for a value
2. CONSTRAINT	Enter constraints on a value
3. FIX	Enter remedies for a constraint violation
4. EXIT	Exit interviewer

Enter your command [EXIT] :

If the user entered the choice "PROCEDURE", SALT would display an interface to except specifics about a procedure. A procedure is used to determine a value for a design parameter. A guideline which SALT sets for procedures is that for every design parameter there must be a corresponding procedure. A procedure takes many forms including: asking the user of the expert system, a database look-up, or a mathematical calculation. When entering the procedure, the domain expert is urged to take into account all design constraints which effect the specification of a value.

If the user entered the choice "CONSTRAINT", SALT would display an interface to except specifics about a design constraint. The user is expected to define a constraint for each design parameter and supply a procedure for determining the value of the constraint. The constraint is used to identify limits which have not already been defined in a procedure.

If the user entered the choice "FIX", he or she would be prompted to specify a potential remedy for a specific constraint violation. A fix is used to define revisions of the design parameters in the event that a design constraint is violated. During inferencing, SALT chooses the least costly fix and applies it to the design. SALT allows the user to begin the interview by entering a procedure, constraint, or a fix.

It is not enough for the knowledge acquisition tool to create an expert system. The ability to maintain an expert system by means of adding knowledge incrementally is an essential capability of a knowledge acquisition tool. SALT organizes pieces of the knowledge-base as they are added over time. It keeps track of links between datum and cues the user for appropriate links. SALT also warns the user of missing pieces or inconsistencies in the knowledge base.

Experts enter knowledge into SALT's knowledge base through a user interface. Experts find it easy to list constraints for a solution and can produce values for individual design parameters. It is not easy for the domain expert to have a complete understanding of how all of the pieces of the knowledge fit together. For this reason, SALT allows the user to enter the knowledge piecemeal. SALT keeps track of the structure of the forming knowledge base. SALT will prompt the user when an appropriate link is missing. SALT also keeps track of inconsistencies in the knowledge base. This takes the burden of identifying how all of the steps fit together off of both the knowledge engineer and the domain expert [Marcus 88].

In order to detect inconsistencies or incompleteness in the knowledge-base, SALT must store the extracted knowledge in a structured manner. The following section describes, in detail, SALT's method of knowledge representation.

3.3.4 Knowledge Representation

Knowledge extracted from an expert must have some structured form of representation in order to be properly processed by the expert system. It is virtually impossible to represent the knowledge in the same way in which it is represented in the human mind. Knowledge with in a human mind is said to have an implicit form of representation. The knowledge is stored in a manner in which can cannot be examined by the conscious mind. In contrast, computers represent knowledge in explicit forms. All of the knowledge is accessible and may be reduced to a standard form of binary values. The process of knowledge acquisition is the process of transforming implicit knowledge into an explicit form [Brule 89].

SALT uses a dependency network to represent its pieces of knowledge and how they interact with one another during problem-solving. Each node in the dependency network represents a design parameter. The nodes in the network are linked together with three types of links: contributes, constrains, and suggests-revision-of. For the link from node A to node B, the following definitions apply [Marcus 88]:

Contributes:

> A contributes to B if A is used in a procedure to specify a value for B.

Constrains:

> If A is a constraint and B is a design parameter, then A constrains B if the value of A places some restriction on the value of B.

Suggests-revision-of:

If A is the name of a constraint then A suggests-revision-of B if a violation of A suggests a change to be made to the current value of B.

When new information is entered into the knowledge base, SALT may create a new node to represent that piece of knowledge. Each time a new node is created, the system checks for possible links to or from that node. SALT expects a "contributes-to" link to each node in the dependency network unless it is a constant or an input node. A constant node is defined as a node representing a value which is initialized to some constant value. An input node is defined as a node representing a value which must be imputed by the user.

Representing the imputed knowledge in a standard form allows SALT to troubleshoot the knowledge-base. The following section describes how SALT is able to use the dependency network and the propose-and-revise problem-solving strategy to insure that the imputed knowledge is capable of converging on a single solution.

3.3.5 Convergence

SALT uses the propose-and-revise problem-solving strategy to insure that the knowledge acquired from the expert can converge on a solution. The propose-and-revise problem-solving strategy needs to control the inferencing in the event that a proposed design violates a constraint. Control knowledge guides the expert system in choosing proper actions to propose a fix for violated constraints. While the problem solver is trying to converge on a solution, it is also trying to optimize that solution. When the user supplies the fixes for constraints, information regarding feasibility and preferability is also entered.

When imputing the corrections for constraint violations, it is difficult for the user to recall all contributors to some constraint violation. For this reason, SALT provides the user with a list of design parameters which contribute to the current constraint violation.

All constraint corrections must be less desirable than the original proposed values. The user specifies the preference of individual constraint revisions using integers, one being the most preferred. The integers correspond to a list of reasons why a revision could be less preferred than the original value proposed. This list is supplied by the domain expert. The following list was used for the SALT generated expert system, VT, and provides an example of how a domain expert specifies his or her preferences.

1. Causes no problem
2. Increases maintenance requirements
3. Makes installation difficult
4. Changes minor equipment sizing
5. Violates minor equipment constraint
6. Changes minor contract specifications
7. Requires special part design
8. Changes major equipment sizing
9. Changes the building dimensions
10. Changes major contract specifications
11. Increases maintenance costs
12. Compromises system performance

The information provided for a correction guides the problem-solver in the revision of the proposed value. The problem-solver will begin a revision of the design when the first constraint violation occurs. The most preferred change is chosen first for the revision. If the constraint on that value is still violated, the next preferred change is chosen. This continues until a suitable value is found which satisfies all constraints.

The problem-solver must also consider that it is possible that a solution for one constraint violation may effect other constraint violations. Only if the effects of a fix do not effect other constraints is the revised value considered to be the most preferred. Because fixes may

aggravate other constraint violations, thrashing is a problem which must be addressed. SALT alerts the user of possible thrashing by producing a list of chains of interacting fixes. Each chain in the list represents a constraint whose change make other constraints more likely to be violated.

3.3.6 Troubleshooting the Knowledge-base

Using the dependency network, and the propose-and-revise method of problem-solving, SALT is able to troubleshoot the knowledge-base. It is the task of the problem solver to find a path through the dependency network which satisfies all design constraints and which converges on a single solution. SALT uses completeness checking to insure that the acquired domain knowledge is complete enough to form a working expert system. By using the propose-and-revise method of problem solving, SALT can run sample cases to check the completeness of the knowledge base. SALT also uses the propose-and-revise method to insure that the expert system will be able to converge on a solution.

SALT troubleshoots the imputed knowledge during the knowledge acquisition process by using its understanding of how the propose-and-revise problem solver will use the knowledge. SALT detects cycles in the dependency network and guides the user in breaking them. The problem-solver of a least-commitment expert system would consider all relevant information at each step of the inference process. SALT uses a least-commitment compilation strategy. Each procedure is compiled with data-driven control. A procedure to determine a parameter value will be eligible for use only after all values contributing to that procedure have been specified. If there is a cycle in the dependency network, the requirements for the procedures in that cycle will never be met.

If the problem-solver reaches a cycle, it will get stuck. An example of how SALT handles a cycle in the dependency network is shown by using procedures from the SALT-generated expert system, VT. If the following procedures were entered into VT:

```
HOIST-CABLE-QUANTITY    =
        SUSPENDED-LOAD / HOIST-CABLE-STRENGTH

HOIST-CABLE-WEIGHT      =
        HOIST-CABLE-UNIT-WEIGHT *
        HOIST-CABLE-QUANTITY * HOIST-CABLE-LENGTH

CABLE-WEIGHT            =
        HOIST-CABLE-WEIGHT + COMP-CABLE-WEIGHT

SUSPENDED-LOAD          =
        CABLE-WEIGHT + CAR-WEIGHT
```

a cycle would be formed in the network. The values for HOIST-CABLE-QUANTITY, HOIST-CABLE-WEIGHT, CABLE-WEIGHT, and SUSPENDED-LOAD create a cycle because they are each dependent on one or more of each other. SALT detects such cycles and prompts the user with an appropriate message. The following is an example of a message generated by SALT regarding the cycle previously described [Marcus 88]:

> In the procedures I have been given, there is a loop. The list below shows the values on the loop; each value uses the one below it and the last uses the first:
>
> 1 HOIST-CABLE-QUANTITY
> 2 SUSPENDED-LOAD
> 3 CABLE-WEIGHT
> 4 HOIST-CABLE-WEIGHT
>
> In order to use any procedure, I need some way of getting a first estimate for one of the manes on the list. Which one do you wish to estimate?

This will guide the user in correcting the cycle formed by the imputed knowledge. SALT is able to aid the domain expert in entering correct knowledge without the domain expert needing to structure the knowledge in advance. This is only one of the strengths SALT gained by

predefining the problem-solving strategy. The following section gives a brief summary of the strengths of SALT.

3.3.7 Summary

The predefined problem-solving technique gives SALT the ability to control the incoming knowledge in several different ways. SALT is able to identify relevant domain knowledge, detect potential weaknesses in the expert system and analyze test case coverage. In addition, SALT can detect cycles in the knowledge-base and guide the user in breaking them. Along with proposing a solution to some problem, an expert system developed using SALT can give explanations of conclusions formed. SALT uses knowledge about the propose-and-revise problem-solving strategy to analyze test case coverage.

SALT proved to be successful in using the propose-and-revise problem-solving strategy to guide the knowledge acquisition process. Because of the advantages of predefining a problem-solving strategy, the number of domain-specific knowledge acquisition tools such as SALT is increasing. The next section gives an example of another domain-specific knowledge acquisition tool developed by Carnegie Mellon University.

3.4 MOLE

MOLE is an example of a knowledge acquisition tool which uses the cover_and_differentiate method of problem-solving. Discussed below are MORE's history, problem-solving strategy, knowledge representation, control knowledge, acquiring the knowledge-base, acquiring the initial symptoms, acquiring covering knowledge, acquiring differentiating knowledge, and the summary.

3.4.1 History

MOLE is a knowledge acquisition tool designed at Carnegie Mellon University. It is used for generating expert systems which use a cover-and-differentiate problem-solving strategy. Because MOLE predefines its problem-solving strategy to be cover-and-differentiate, it is limited to creating expert systems for diagnostic-type domains. Diagnostic-type problems are solved by selecting a solution from a set of possible solutions. This is unlike synthesis-type problems which are solved by constructing original solutions.

3.4.2 Problem-solving strategy

Mole predefines its problem-solving strategy to be the cover-and-differentiate method. The goal of a cover-and-differentiate problem-solver is to select a solution (or explanation) from a set of solutions based on an initial set of symptoms.

An expert solves a problem using the cover-and-differentiate problem-solving strategy by first looking at the initial symptoms. Based on the initial symptoms, a set of solutions which can explain the symptoms is generated. This set of solutions is called the covering knowledge. Each solution in the set of covering knowledge is a candidate for the final solution. Once the covering knowledge is found, the expert searches for differentiating knowledge. The differentiating knowledge would be any information which would clarify the differences between the candidates in the covering knowledge. The ultimate goal of the problem-solver is to find enough differentiating knowledge to eliminate all possible solution candidates except one.

MORE separates the covering knowledge from the differentiating knowledge to gain control

over the knowledge and to provide a systematic way of extracting it. The cover-and-differentiate method is described in detail in section 2.2.

Separating the covering knowledge from the differentiating knowledge enables MOLE to standardize the knowledge representation. The following section describes how MOLE uses the cover-and-differentiate method to guide the knowledge representation.

3.4.3 Knowledge Representation

The knowledge base representation is formed by the cover-and-differentiate problem-solving strategy. The knowledge base is represented in the form of a network of nodes representing symptoms and explanations for the symptoms. The each root node in the network represents a final solution to a set of symptoms. Figure 6 shows the following example of a small part of a knowledge-base built by MOLE for the diagnosing of automobile engine problems [Kahn 88]:

The three root nodes: worn crankshaft bearings, worn cylinders, and ignition problems all represent final solutions or explanations. The initial symptoms include: lack of power, excessive engine noise, and engine misfiring. This is a simple example of a MORE knowledge network. In a complete knowledge base, intermediate explanation nodes would be included in the network. Each intermediate explanation would explain the node below it and would likewise be explained by the node above it.

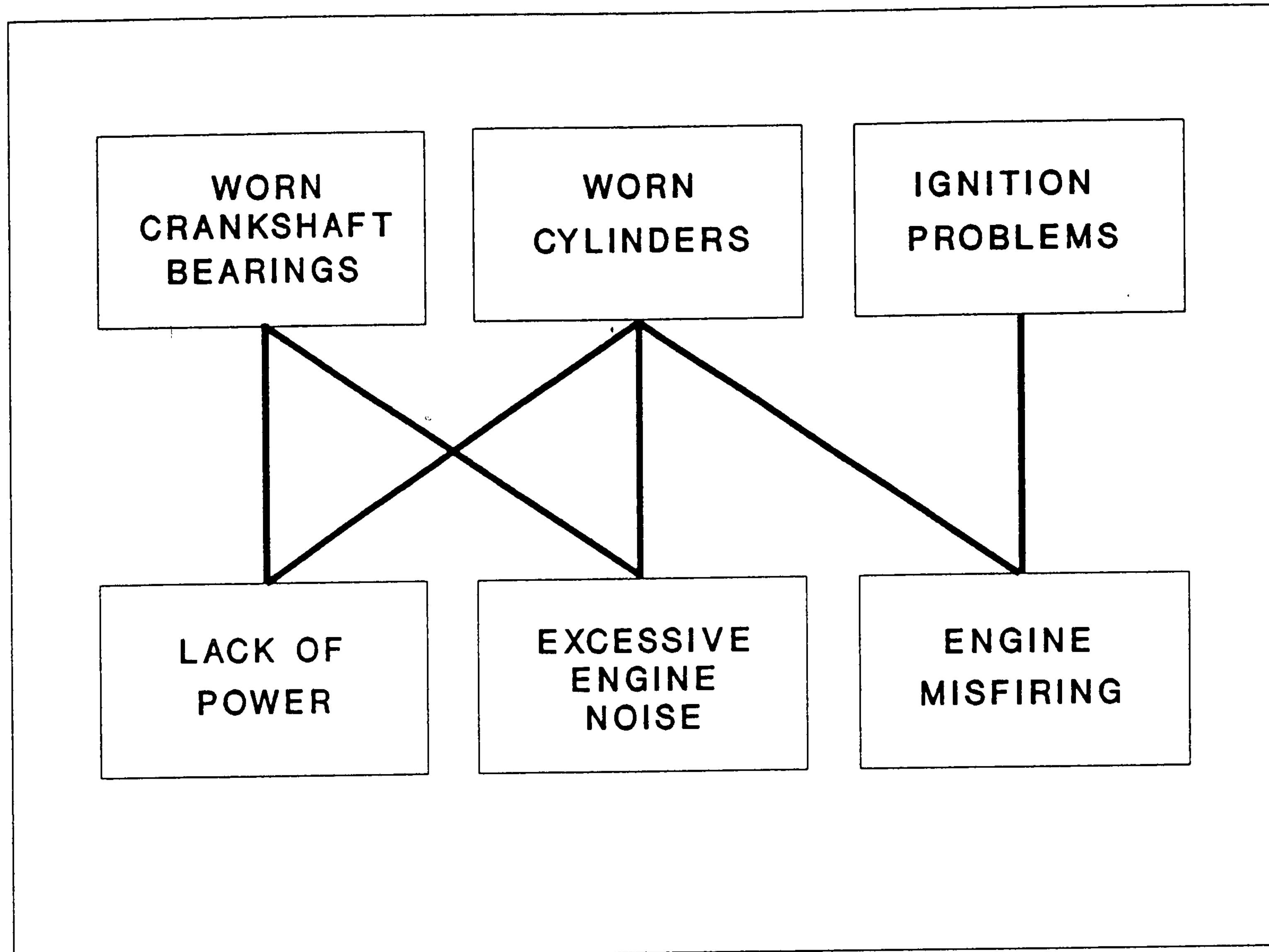

Figure 6. Network Representation of a MOLE Knowledge Base [Eshelman 88].

Because MOLE represents its knowledge in a structured form, it is able to develop standard control knowledge for cover-and-differentiate problem-solving. The following section will describe MOLE's control knowledge and how it manipulates knowledge in the knowledge-base.

3.4.4 Control Knowledge

The control knowledge is defined as the search strategy followed by the expert system's inference engine. MOLE's control knowledge follows the cover-and-differentiate problem-solving strategy. The control knowledge of MOLE is guided by both strong and weak constraints. The strong constraint strictly confines the search strategy, while the weak

constraints simply guide the search strategy.

MOLE's control knowledge must meet the following strong constraint [Kahn 88]:

(1) If an event has at least one potential explanation, the final diagnosis must include at least one of these potential explanations.

This constraint can be describe in terms of exhaustivity. Exhaustivity can be interpreted to mean that every symptom has a cause [Kahn 88]. The strong constraint defines that from each symptom node on the knowledge-base network, there must be a path which ultimately leads to a final explanation. The exhaustivity assumption makes it possible to confirm a hypothesis by ruling out all of the competitors.

MOLE's search strategy must also meet the following weak constraints [Kahn 88]:

(1) For any symptom or state, a single pathway leading to a top-level explanation is preferred.

(2) It is preferable that the various pathways leading from the bottom-level symptoms should converge on as few top-level explanations as possible.

The first weak constraints is to stress simplicity in explanations. The second weak constraint states that each set of symptoms should converge on at most one explanation. The weak constraints guide the search strategy to keep the explanations as simple as possible. At the end of a search, the final subgraph should form a tree with all activated symptoms connected to a single root hypothesis.

MOLE assumes exhaustivity through its strong constraint on the search strategy. MOLE also assumes exclusivity. Exclusivity prevents MOLE from accepting two explanations when one is sufficient. In effect, MOLE is assuming that the alternative explanations for a symptom are mutually exclusive. Once there is evidence of a hypothesis, all competitors are eliminated.

Combining the assumption of exhaustivity with the assumption of exclusivity, MOLE has a strict search strategy which it follows. Because a symptom must be explained exhaustively by some hypothesis, a candidate hypothesis can be confirmed by ruling out its competitors. Furthermore, because only one hypothesis is likely to be true exclusively, a candidate hypothesis can be rejected if there is independent evidence for one or more of its competitors but not for it [Kahn 88].

3.4.5 Acquiring the Knowledge-Base

MOLE acquires the knowledge needs to fill the knowledge base via a user interface. MOLE assumes that the knowledge acquisition tool should be able to take the place of the knowledge engineer. In order for MOLE to be able to completely remove the need for a knowledge engineer to help the domain expert, there are two main problems MOLE seeks to overcome. The two most troublesome feature of the knowledge acquisition process are indeterminateness and incompleteness [Kahn 88]:

- Indeterminateness:
  When specifying associations between events, the expert is likely to be fairly vague about the nature of these associations and events.

- Incompleteness:
  The expert will probably forget to specify certain pieces of knowledge.

Indeterminateness implies that some domain experts are not accustomed to supplying their knowledge in the form which will fit the problem-solving method being used by the knowledge acquisition tool. MOLE tries to be robust enough to handle input from the domain expert which is ambiguous.

Incompleteness implies that it is expected that a domain expert will leave out some pertinent information. For this reason, MOLE allows the knowledge base to be updated incremental. New knowledge may be added to the knowledge base or old knowledge may be refined.

3.4.6 Acquiring the Initial Symptoms

MOLE begins a new knowledge acquisition session by asking the expert to list some of the complaints or symptoms that would tell a potential user there is a problem to be diagnosed. The following is an example of MOLE acquiring knowledge for the expert system that diagnoses problems associated with a coal-burning power plant [Kahn 88]:

```
List possible complaints or symptoms that
might need to be diagnosed:

Complaint:
>> [ NONE ] loss-in-gas
  LOSS-IN-GAS          [ YES NO ]
  Status:              NEW
  Method:              ASK
  Default Value:       NONE
>> Confirm (Yes, No):   [ YES ] <cr>

Complaint:
>> [ DONE ] high-fly-ash-flow
  HIGH-FLY-ASH-FLOW    [ YES NO ]
  Status:              NEW
  Method:              ASK
  Default Value:       NONE
>> Confirm (Yes, No):   [ YES ] <cr>

Complaint:
>> [ NONE ] high-bottom-ash-flow
  HIGH-BOTTOM-ASH-FLOW [ YES NO ]
  Status:              NEW
  Method:              ASK
  Default Value:       NONE
>> Confirm (Yes, No):   [ YES ] <cr>
```

```
Complaint:
>> [ DONE ] dark-ash
  DARK-ASH            [ YES NO ]
  Status:             NEW
  Method:             ASK
  Default Value:      NONE
>> Confirm (Yes, No):  [ YES ] <cr>

Complaint:
>> [ DONE ] <cr>
```

In this example, the expert has entered four complaints that require diagnosis. The fields status, method, and default value are all set with default values which the user has the option of changing. Status field tells whether or not the symptom is new to the system. The method field specifies what method the expert system should use to acquire the value of the symptom. The method field defaults to asking the user. The default value field specifies default value for the symptom. MOLE assumes the default value to be valid unless there is proof otherwise.

3.4.7 Acquiring Covering Knowledge

MOLE tries to acquire covering knowledge from the domain expert after the initial complaints are entered. Using the same example as above, an example of how MOLE precedes to extract the covering knowledge from the expert follows:

```
List possible explanations for
LOSS-IN-GAS:

Possible explanation for LOSS-IN-GAS:
>> [ NONE ] low-heat-transfer
  LOW-HEAT-TRANSFER    [ YES NO ]
  Status:              NEW
  Method:              INFER
  Default Value:       NONE
>> Confirm (Yes, No):  [ YES ] <cr>
```

Possible explanation for LOSS-IN-GAS:
>> [NONE] excess-air high
EXCESS-AIR HIGH [HIGH NORMAL LOW]
Status: NEW
Method: INFER
Default Value: NONE
>> Confirm (Yes, No): [YES] <cr>

Possible explanation for LOSS-IN-GAS:
>> [NONE] <cr>

The default value for method is "infer" which means that the value will be obtained indirectly rather than asking the user. The default value for any event is "YES" but other values such as "HIGH", "NORMAL", or "LOW" may be used. This process of acquiring the covering knowledge is repeated until all of the complaint have potential explanations.

When all complaints have potential explanations, MOLE seeks higher-level explanations. As the knowledge is entered by the domain expert, MOLE builds a network of nodes and links to represent the forming knowledge base. The nodes on the network represent states or events while the links represent explanatory relations between the nodes. This initial network is referred to as the explanation space of the system.

3.4.8 Acquiring Differentiating Knowledge

MOLE begins Acquiring the differentiating knowledge after the covering knowledge has been entered by the domain expert. MOLE seeks to acquire enough knowledge to differentiate each candidate explanation.

MOLE seeks to create as much of the differentiating knowledge as possible before asking the domain expert for help. Some of the explanations do not need differentiating knowledge. Other differentiating knowledge can be inferred from the network. When acquiring

differentiating knowledge from the expert, MOLE tries to exploit the existing knowledge rather than add new knowledge to the knowledge base.

3.4.9 Summary

The predefined problem-solving technique gives MOLE the ability to control the incoming knowledge in several different ways. MOLE is able to identify relevant domain knowledge, detect potential weaknesses in the expert system and analyze test case coverage. Along with proposing a solution to some problem, an expert system developed using MOLE can give explanations of conclusions formed. MOLE uses knowledge about the cover-and-differentiate problem-solving strategy to analyze test case coverage. MOLE proved to be successful in using the cover-and-differentiate problem-solving strategy to guide the knowledge acquisition process.

4. Thesis Summary

Because of the advantages of predefining a problem-solving strategy, the number of domain-specific knowledge acquisition tools such as MOLE and SLAT is increasing. Tools such as LAPS predefine more than one problem-solving strategy. Although predefining problem-solving strategies strengthens the knowledge extraction capabilities of knowledge acquisition tools, knowledge acquisition is still considered to be the bottleneck of expert system development.

Future Knowledge Acquisition Workshops (KAW) will bring the AI community closer to eliminating the problems associated with knowledge acquisition. The KAW was organized in hopes of preventing the duplication of research and providing a means by which all current

research could be shared and hopefully integrated. In addition, the KAW will allow knowledge engineers to learn from the limitations encountered with previously developed tools.

REFERENCES

[Boose 88a]
Boose, J. H., and B. R. Gaines, Knowledge Acquisition Tools for Expert Systems. Knowledge-Based Systems Volume 2. San Diego: Academic Press Inc., 1988.

[Boose 88b]
Boose, J. H., Bradshaw, J. M., and Shema, D. B., Recent Progress in Aquintas: A Knowledge-Acquisition Tool. In Proceeding of the Second European Knowledge-Acquisition Workshop, 2:1-15. Bonn: German Institute for Mathematics and Data Processing, 1988.

[Brule 89]
Brule, James F., and Alexander Blount, Knowledge Acquisition. New York: McGraw-Hill, 1989.

[Clancey 84]
Clancey, W., Classification Problem Solving, In Proceedings of the Fourth National Conference on Artificial Intelligence. Austin, Texas, 1984.

[di Piazza 90]
di Piazza, Joseph S., and Helsabeck, Frederick A., "Laps: Cases to Models to Complete Expert Systems", AI Magazine, (Fall 1990), pp. 80-107.

[Eshelman 87]
Eshelman, L., MOLE: A Knowledge-Acquisition Tool for Cover-and-Differentiate Systems. In Automating Knowledge Acquisition for Expert Systems, ed, S. Marcus, 37-80, Boston: Kluwer Academic, 1987.

[Gaines 88]
Gaines, B., Second-Generation Knowledge-Acquisition Systems. In Proceedings of the Second European Knowledge-Acquisition Workshop, 17:1-14. Bonn: German Institute for Mathematics and Data Processing, 1988.

[Garg-Janardan 88]
Garg-Janardan, Chaya, and Salvendy, Gavriel, A Conceptual Framework for Knowledge Elicitation. In Knowledge Acquisition Tools for Expert Systems Volume 2. SanDiego: Acedemic Press Inc., 1988.

[Giarratano 89]
Giarratano, J., and Gary Riley, Expert Systems: Principles and Programming. Boston: PWS-KENT, 1989.

[Kahn 88]
Kahn, G., MORE: From Observing Knowledge Engineers to Automating Knowledge Acquisition. In Automating Knowledge Acquisition for Expert Systems, ed, S. Marcus, 8-35, Boston: Kluwer Academic, 1988.

[Klinker 88]
Klinker, G., KNACK: Sample-Driven Knowledge Acquisition for Reporting Systems. In Automating Knowledge Acquisition for Expert Systems, ed, S. Marcus, 125-174 , Boston: Kluwer Academic, 1988.

[Laird 87]
Laird, J., A. Newell, and P. Rosenbloom, SOAR: An architecture for general intelligence, Artificial Intelligence 33(1):1-64, 1987.

[Lenat 86]
Lenat, D., M. Prakash, and M. Shepherd, CYC: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. AI Magazine 6(4):65-85, 1986.

[Marcus 88]
Marcus, S., Automating Knowledge Acquisition for Expert Systems. Boston: Kluwer Academic, 1988.

[McDermott 88]
McDermott, J., Preliminary steps Toward a Taxonomy of Problem-Solving Methods. In Automating Knowledge Acquisition for Expert Systems, ed., S. Marcus, Boston: Kluwer Academic, 225-256, 1988.

[McGraw 89]
McGraw, K. L., and Karan Harbison-Briggs, Knowledge Acquisition: Principles and Guidelines. New Jersey: Prentice-Hall, Inc., 1989.

[Morik 88]
Morik, Katharina, Acquiring Domain Models. In Knowledge Acquisition Tools for Expert Systems Volume 2. SanDiego: Acedemic Press Inc., 1988.

[Parsaye 87]
Parsaye, A., Auto-Intelligence User's Manual. Los Angeles: IntelligenceWare, 1987.

[Stout 88]
Stout, J., Caplain, G., Marcus, S., and McDermott, J., Toward Automating Recognition of Differing Problem-solving Demands. In Automating Knowledge Acquisition for Expert Systems, ed., S. Marcus, Boston: Kluwer Academic, 225-256, 1988.

Vita

Buffi Lynn Clemenko, the daughter of Laura Jane Taussig and Howard H. Clemenko, was born on September 3, 1967 in Philadelphia, Pennsylvania. The author received her B.S. in physics from Georgian Court College in 1989 and her M.S. in computer science from Lehigh University in 1991. She is an active member of the ACM. Her hobbies include karate and skiing.