A Computer Troubleshooting Expert System

To Aid Technical Support Representatives

by

Ian C. Cameron

Department of Mathematics and Computer Science

Submitted in partial fulfilment

of the requirements for the degree of

Bachelor of Science (Specialised)

Algoma University College

Sault Ste. Marie, Ontario

April 2005

Abstract

In this modern age computers dot the landscape performing a plethora of functions, and with all of these machines comes the monumental task that is maintenance. Circuit boards short out software becomes corrupted and viruses invade, and dealing with all these problems is the lowly technical support representative. To do this job the computer industry needs more expert representatives than can possibly be found let alone trained. This means that if a system can be created that would make the expertise of an experienced professional could be put at the fingertips of a new technician right out of training both performance and customer satisfaction could be improved immensely.

In this thesis the goal is to address one key question. Would a computer troubleshooting expert system represent the logical next step in troubleshooting aids? To answer this we will explore several topics. The first will be expert systems, the problems that they are best at solving, and how they are usually built. The second will be the Jess rule language which was chosen to build a prototype application. The third topic will be the analysis of the problem faced by the industry, and how it will be met. Finally we will describe the design and implementation of a prototype computer troubleshooting expert system.

Keywords: production rule, production system, expert system, problem domain, knowledge base, inference engine, user interface, expert system shell, experiential knowledge, working memory, rule base, recognise-act cycle, fact, rule, slot, forward chaining, backward chaining, conflict set, fired, Rete algorithm, pattern node, join node, terminal node, activation record, tokens, node sharing, rule engine, interpreted language, rule-based programming, procedural programming, late binding, Shadow fact.

Table of Contents

Abstract		**ii**
Table of Contents		**iii**
Chapter 1	**Introduction**	**1**
1.1	The Ideal Troubleshooting Aid	1
1.2	Summary	3
Chapter 2	**Expert Systems and Production Systems**	**4**
2.1	Expert Systems	4
2.2	Production Systems	7
2.3	The Rete Algorithm	10
2.4	Summary	13
Chapter 3	**The Jess Rule Language**	**15**
3.1	Jess Overview	15
3.2	Jess and Java	18
3.3	The Recognise-Act Cycle in Jess	20
3.4	Summary	22
Chapter 4	**The Computer Troubleshooting Expert System**	**24**
4.1	The Problem	24
4.2	The Current Solutions	27
4.3	Proposed Solution	28
4.4	Summary	31

Chapter 5 Design and Implementation **33**

5.1 Design 33

5.2 Knowledge Base Implementation 36

5.3 GUI Implementation 39

5.4 Summary 41

Chapter 6 Conclusion and Future Work **43**

6.1 Conclusion 43

6.2 Future Development 47

6.3 Summary 50

Bibliography **52**

Appendices **54**

A Questions.java: GUI Class 54

B CTList.java: The Jess Controller 61

C CTKnowledgeBase.clp: The Rule Base Batch File 67

D Nodes.clp: The Knowledge Base Batch File 69

E CTNodeList.java: An Object to Pass the Active List 73

F CTNode.java: An Object for Storing Node Data 75

Chapter 1: Introduction

1.1 The Ideal Troubleshooting Aid

This thesis began with the realization that the computer troubleshooting industry had a fundamental problem. The industry has a great need for expert technicians, but few are available. The current software used to assist technicians assumes that they are already skilled at narrowing down the possible causes of a computer problem, and so they are only useful when a cause to the issue at hand is evident. Into this niche we insert the possibility of an *expert system* to provide the guidance needed by the technicians. Hence the question posed by this thesis is the following: Would a computer troubleshooting expert system represent the next logical step in technical support aids?

To answer this question it is first necessary to first explain what an expert system is and what they can do. Expert systems emulate the thought process of human experts. Expert systems do not handle problems well if the knowledge needed to perform the task is too great, or the knowledge is difficult or impossible to gather and encode for the system. Expert systems are usually created using a *production system*, which allow the knowledge to be encoded in *production rules*. One example of a production system is the Jess rule language, which was used to develop a prototype application for this thesis.

Jess was chosen for this application because it is current, free for academic use, has a text available that is well written by the language's creator, and most

importantly interacted well with a familiar language (namely Java). Jess' interaction with Java allows the programmer to embed Jess code in Java programs, and gives Jess the advantage of access to all Java's APIs. When using Jess with Java the programmer must be careful because a bug in one system can cause strange errors in the other. Jess is very efficient and provides many useful features for improving the efficiency of its operation based on the system that is being built.

The computer troubleshooting expert system presents the technician with a list of guiding questions that, when answered, provide the system with information about what problem sets the issue may belong to, and which ones it does not belong to. It then generates another list of questions that apply to subsets of those identified. When all possible sub-sets have been narrowed down to individual possible causes the system returns a list of possible solutions. This process can be saved midway through by saving the current list of questions and possible solutions.

1.2 Summary

In sum, this thesis will explore expert systems and their functions, the Jess rule language, the problems faced by the computer troubleshooting industry, and the computer troubleshooting expert system. Each of these topics will be covered in their own chapter that explains the information pertinent to this thesis. Finally we

will bring together this information and seek to answer the question posed in the start of this chapter, and explain the future work needed to complete the troubleshooting expert system.

Chapter 2: Expert Systems and Production Systems

In this chapter we will discuss the fundamentals of expert systems design and functionality. This information has been broken down into three main sections that grow more specific and detailed as they go. The first section contains information about the definition of an expert system, the components that make up a typical expert system, and the information needed to determine if a problem is suitable for the development of such a system. The second section talks about the components of a production system and their functions because the expert system created for this thesis was developed using a production system. The third section provides an in depth discussion of the Rete algorithm, which is utilised by most modern production systems.

2.1 Expert Systems

An Expert system is "a model and associated procedure that exhibits, within a specific domain, a degree of expertise in problem solving that is comparable to that of a human expert."[6]. To do so an expert system must have in depth knowledge of the *problem domain* for which it was built. This knowledge is usually collected from interviews with human experts, although it may also come from other research materials such as books or the internet. While this knowledge gives the expert system many advantages over a conventional system it also limits the system. Expert systems are only able to solve problems in a relatively small,

predefined, problem domain. If the problem domain is too large they become extremely inefficient, and are almost impossible to create. An expert system must also be capable of explaining its behavior[1]. In the very least it should be able to list the steps it took to reach its conclusions, and many more elaborate expert systems are designed so that they can provide the reasoning behind those steps. It is often also required that the expert system be capable of dealing with uncertainty [1]. In many instances the expert system is designed to reason in the face of conflicting or incomplete evidence, and it is expected to make conclusions and rate them on their accuracy.

Every expert system has three main components, the *knowledge base*, *inference engine*, and *user interface*[7]. The knowledge base stores the information known about the given problem domain. It is usually made up of rules that define the domain specific knowledge human experts use to make decisions. The inference engine applies what is known about the current situation or problem to the knowledge that is stored in the knowledge base to draw conclusions. In the process conclusions may lead to new information about the current situation, which in turn may lead the inference engine to draw more conclusions which creates a *chain of inference*. The user interface provides interactivity with the user (which may itself be another software system). A user interface may be as simple as a text prompt or as complex as a remote GUI running across a network. In most expert systems the user interface and inference engine are bundled together so that the knowledge base can easily be modified or replaced to change

the domain in which the system has expertise. This combined inference engine and user interface is referred to as an *expert system shell* (as seen in Diagram 2.2.1) [5]. Most expert systems can only be modified to encompass knowledge of a similar problem domain to that of the original. Despite this some general purpose expert system shells do exist, and are commercially available as programming languages and debugging environments for building expert systems.

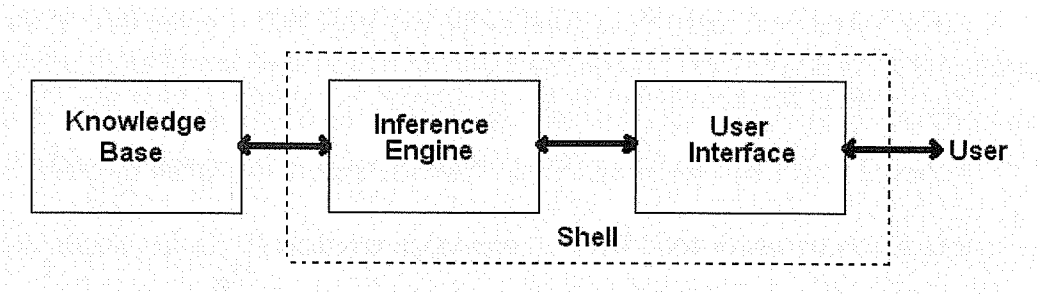

Figure 2.2.1: The Layout of a Typical Expert System

Expert systems have been used successfully to perform many commercial applications. In 1973 MYCIN was created to aid in medical diagnosis and was then used to create EMYCIN (or Empty MYCIN), which was the first general purpose expert system shell. In 1976 PROSPECTOR was created to perform geological analysis, and proved to be extremely successful. The OPS expert system shell was created in 1977, and used by DEC in 1978 to create XCON/R1 for configuring computer systems. These and other systems have made expert systems one of the more profitable technologies to come out of AI research.[5]

There are several questions that must be addressed when contemplating creating an expert system as a solution to a given problem acording to Giarratand and Riley[5]. First, "can the problem be efficiently solved using conventional programming?" If a conventional program can be created that solves the given

problem efficiently then it is unlikely that an expert system could be created that is of comparable performance. Second, "is the domain well bounded?" For an expert system to be created it is necessary to know exactly what it will need to know. Third, "is there a need and a desire for an expert system?" If the problem has already been sufficiently solved then the expert system will never be used. Fourth, "is there at least one human expert who is willing to cooperate?" An expert system will only be as good as the expertise that is given to it; if the expert from whom the knowledge is being derived is unwilling then the information retrieved won't be very good. Fifth, "can the expert(s) explain the knowledge so that it can be understood by the knowledge engineer?" If the person collecting and encoding the knowledge can't understand it, then the system will not properly reflect the knowledge of the human expert(s). Finally, "is the problem-solving knowledge mainly heuristic and uncertain?" If the expert's problem-solving approach is mainly based on experience (*experiential knowledge*) or an organised trial and error process, then an expert system is probably the right approach.

An expert system can be written in any programming language, but the vast majority are created using production systems. For this reason production systems will be discussed in the next section.

2.3 Production Systems

Most current expert systems are built using production systems (e.g. Jess, CLIPS, Ops, etc.). Production systems are made up of three main components, the

working memory, *rule base*, and *recognise-act cycle*[8]. The best analogy for how an expert system that is written using a production system works is that the working memory is like the human short-term memory. Working memory contains all information about the current situation. The rule base represents long-term memory. It stores all the information that is known from past experience. The recognise-act cycle represents the thinking process that draws conclusions based on analysis of the current situation, and how it relates to what is known from past experience.

Working memory is usually made up of an ordered list of *facts*. A fact is a single cohesive set or chunk of data [5]. Interrelated pieces of information are stored in a single fact so that the recognise-act cycle can easily match them with the required conditions for a *rule*. For example if the system needed to know about a given door the user may describe it as being made of wood, with steel hinges, and it has no window, and the system would store all of these attributes in a single fact. Each fact may have one or more *slots*. A slot is a place where a single piece of information is stored. For example the door fact above would probably have four slots including one to state that it is a door we are referring to. If the system only dealt with doors this would be unnecessary.

The rule base of a production system is made up of production rules (hence the name "production systems"). Production rules are of the form "If *Condition,* then *Action*". Production systems can be made so that they are natively either *forward chaining* or *backward chaining*. Forward chaining rules are of the form "If

DesiredDataPresent, then *DrawConclusion*", and so the system performs a search of the knowledge base for possible conclusions that can be made based on the data present in working memory[8]. Forward chaining inherently does a breadth first search following all chains of inference simultaneously[5]. This means that forward chaining is most efficient when there are many short chains of inference (the graph is broad and shallow). Backward chaining rules present themselves as "If *Goal*, then *ProveSubGoals*", and therefore the system performs a search of the knowledge base for data that can prove the goals in working memory. Backward chaining naturally performs a depth first search following all chains of inference one at a time[5]. This means that backward chaining is best used when dealing with a few long chains of inference (the graph is deep and narrow).

The recognise-act cycle performs pattern matching operations between the facts in working memory and the rules in the rule base. When the conditions for a given rule are met they are added to the *conflict set.* When all matching rules are in the conflict set the system must perform some kind of *conflict resolution* strategy. This could be as simple as executing all rules in the order in which they where entered into the system, or as complex as using a heuristic to determine the best rule to execute first. Once a rule has been chosen from the conflict set its action is executed, and the rule is said to have *fired.* When a rule fires it may alter the state of working memory, and so after firing a rule the recognise-act cycle restarts with pattern matching. This process repeats itself until it either runs out of rules in the conflict set, or it runs into a halt command.

Production systems can perform the role of a general-purpose expert systems shell, and in fact can be utilised in the creation of any software application. This is because production systems are equivalent in power to a Turing machine[8]. Despite this, these systems are seldom used for general programming because most applications would be less efficient and harder to implement using a production system than if they were implemented in a language such as C++ or java. To clarify how a production system really works (particularly the one used for this thesis), it is now necessary that we go into detail describing the *Rete algorithm*, which is the most commonly used implementation of the recognise-act cycle.

2.2 The Rete Algorithm

In the recognise-act cycle it is obvious that the efficiency of the system will depend greatly on the algorithm used to match facts to rules. If the system had to check every fact, or combination of facts, to every rule on each iteration of the cycle, then the system would be horribly inefficient. Luckily most of the time when a rule fires it usually changes only a small portion of the contents of working memory. This means that if the system can remember the matches that occurred on the previous iteration then the system only has to perform pattern matching with the facts that have changed when the last rule fired. The most important algorithm that has been found for doing this is the Rete algorithm

created in 1982 by Charles L. Forgy [2]. Variations on this algorithm have been used in several rule engines, including OPS5, ART, CLIPS, and Jess.

Rete (pronounced "ree-tee") is Latin for net, and so it will come as no surprise that it works by building a network of interconnected nodes[4]. Each node represents one or more conditions of a rule and has one or two inputs and any number of outputs. Input nodes are placed at the top of the network and output nodes at the bottom. The input nodes separate facts according to their head (identifies the kind of fact, doors would go in door facts while windows would go in window facts), and in the network progressively finer discriminations and associations between facts are made until finally the facts reach the bottom where nodes representing the rules reside. When a fact or set of facts are taken as input for a node in the network, the node will output those facts if they pass the test within it. If a set of facts filters all the way down the network to the bottom then it has passed all the conditions for one of the rules, and so this rule will be added to the conflict set.

The internal nodes between the inputs and the outputs can be divided into two categories. They are either *pattern nodes* or *join nodes*. Pattern nodes always occur above join nodes, and filter the inputs so that only relevant facts are presented to the join nodes. Pattern nodes have a single input and perform a test on a single fact. Join nodes have two inputs, and these nodes perform tests that involve more than one fact. Join nodes remember any facts that come in on either of their inputs. The "left" input can receive one or more facts as input, and the

"right" input can only receive a single fact. Join nodes always produce two or more facts as output.

The rules that have been defined in the system are represented in the network by *terminal nodes*. Terminal nodes have one input and no output. Upon receiveing input they package the incoming list of facts with the rule that they represent to create an *activation record*. The activation record is then added to the conflict set.

To run the Rete network the system simply has to pass every new fact to the inputs at the top of the network. First the pattern nodes will run every test that can be done with only one fact, effectively filtering out all facts that do not apply to the join nodes. The join nodes then determine what interrelations between facts match those that are required to match the conditions for the rules in the system. These then feed into the terminal nodes, which create activation records that are added to the conflict set.

So we now have a system that performs pattern matching operations only on facts that are added, but can it handle the retraction of facts. This is done by sending *tokens* instead of facts through the network. A token contains the fact which is to be added or removed from the network, and it also contains a command as to what to do with the fact. When a node in the network receives a token it checks if the fact meets its test, and if it does it will perform the prescribed action. If it is adding the fact then it will add the fact to its memory, and if it is retracting then it will remove it from its memory. It will then create a new token to pass as output.

If an incoming token negates one that it has previously output, it will create a new token that performs the change in the rest of the network. If a terminal node receives a retracting token it searches for the associated activation record and removes it from the conflict set.

Over the years each system implementing the Rete algorithm has made successive improvements on the implementation of the network. The simplest of these optimizations is *node sharing*. Using a purist's implementation of the Rete network would mean that there would be one input node for every fact that is required for every rule, even if two rules required the same fact the network would still have two input nodes for that same fact. By sharing nodes that are identical for more than one rule, the algorithm becomes more efficient both in the amount of computations needed to perform pattern matching, and in the amount of memory used to store previously encountered facts.

2.3 Summary

Expert systems are extremely useful in emulating human experts when few are available or assisting human experts under extreme conditions when the performance of human experts becomes unpredictable. These systems are limited by their inability to handle problems outside of the relatively small problem domains for which they can be programmed. This means that they are only suitable for handling a very specific set of problems. Luckily these problems can be identified by answering the six questions previously listed. In short these

questions identify whether another type of system could solve the problem, and if the knowledge needed to create an expert system can be collected and represented in the system.

Expert systems are usually created using production systems. These systems are made up of a working memory, a rule base, and a recognise act cycle. The working memory is made up of an ordered list of facts. Each fact contains a cohesive set of data about the current situation faced by the system. The rule base is made up of a list of production rules, which encode knowledge about how the system should react to a given situation represented in working memory. The recognise-act cycle recognises what rules apply to the current situation, and decides which of those rules' actions to perform next.

The most commonly used method for performing the recognise act cycle of a production system is the Rete algorithm. This algorithm improves efficiency by remembering the pattern matching done on previous iteration of the recognise act cycle so that it only has to react to the immediate changes to working memory. To do this it creates a network of nodes, each of which performs a test on the facts given to it and saves those that match its required patterns. Results that are outputted from the bottom of the network activate specific rules; adding them to the conflict set.

Chapter 3: The Jess Rule Language

In the last chapter we discussed the components of a typical production system and how they work. In this chapter we will be talking about a specific production system, Jess. The expert system developed for this thesis was written in the Jess language, and so it is important to describe the features, benefits, and inner workings of the Jess *rule engine* to gain a better understanding of how the troubleshooting expert system really works.

This chapter has three main sections. The first section is a general outline of the Jess language, the features it provides to the programmer, and the reasons behind choosing it for this expert system. The second section provides deeper insight into the most important feature of Jess, it's interactivity with Java. The third section describes the implementation of the recognise act cycle used by Jess, and how its behaviour can be monitored, adjusted and refined.

3.1 Jess Overview

For the application developed for this thesis it was decided that the most appropriate language was the Jess, or Java Expert Systems Shell, language [3]. Developed in the late 1990's at Sandia National Laboratories [4], Jess is a general purpose programming language written specifically with expert systems in mind. Jess was originally inspired by CLIPS, and the two do have great similarities in the rule languages they support. This has made it easy in the past for CLIPS applications to be ported over to Jess (some speculate that this feature was vital in the initial success of the Jess rule language). Despite these similarities, the two languages are fundamentally different. Jess is a *dynamic*, *Java centric*

language, where CLIPS is not. This makes Jess ideal for web-based applets, where CLIPS would be difficult to use at best.

Jess also has some functions that CLIPS does not, and some Jess functions have the same syntax as those in CLIPS but different semantics. One function that Jess has that CLIPS does not is the ability to treat working memory like a relational database by defining possible queries that can be called by the program at any time. The query returns a java.util.Iterator of all the tokens containing facts that match the query. This allows the programmer to easily step through the returned data.

Jess is an *interpreted* language that involves both *rule-based programming* and *procedural* commands. When using rule-based commands the inference engine supports both forward and backward chaining. In Jess backward chaining is implemented through forward chaining in the background, so the system is really only using forward chaining. When using procedural calls the language is syntactically very similar to LISP. All function calls are encapsulated in parentheses and are in prefix notation, and all commands are function calls.

Functions in Jess use *late bindings* to support the languages dynamic nature[4]. This means that a function call can be made in a function or rule definition before the function itself is defined. When the function is then defined it is properly linked to the call. This is most important when a function definition is changed during execution because in Jess the new definition will affect all calls to the function. Even those functions already called in previous function and rule definitions are redirected to the new function definition.

Many rule-based systems can have hundreds of rules, and so preventing adverse interactions between rules can be a monumental task. To make building such a system easier, the designers of Jess made it possible to partition the rule base into *modules*. When a *run* command is called on the system only the rules in the *focus module* can fire. During execution a rule may call for a change in focus and so change which rules can fire. Modules also define a *name space*, which means that two rules can have the same name without there being a problem as long as they belong to different modules. This use of modules makes Jess ideal for large rule based systems.

There were several reasons that Jess was chosen for this thesis. The first was that Jess was free for academic use. This was an important consideration as economics are always a concern where a university student is concerned. The second consideration was the availability of a text that gave a comprehensive treatment of the language. Jess in Action by Earnest Friedman-Hill was instrumental in learning the language and the essentials for creating an expert system. The third reason for selecting Jess was its strong interaction with Java. Having a strong background in Java made Jess an ideal approach to learning rule based programming. The final reason for choosing Jess was that it is one of the more recent rule engines to be developed. This meant that when it was developed many of the lessons learned from the experience that arose from older rule engines were incorporated into the implementation of the language.

3.2 Jess and Java

Jess has a very strong relationship with Java. In fact Jess is written and executed in Java. This relationship proves useful in that Java code can be executed in Jess, therefore Java objects can be instantiated in Jess. Not only can Jess call Java; the reverse is also true. Jess code can be directly called from Java. This means that Jess can be embedded directly in Java and new Jess commands can be easily written in Java. The close relationship between Jess and Java also means that Jess can access all Java APIs. This ability makes access to networking, graphics, database management, and a range of other useful tools very easy. The close relationship with Java is probably the most important feature of the Jess language; without it Jess would be of little interest.

Jess also has an interesting way of allowing interaction between the working memory and JavaBeans. *Shadow facts* are facts that derive their slot values from those contained in JavaBeans. There are two distinct types of shadow facts to reflect when they obtain their contents. *Static shadow facts* set the slot values whenever the fact is placed in working memory, and they do not change their value even if the values in the associated JavaBeen change. *Dynamic shadow facts* change slot values whenever those in the associated JavaBean change, so their value may change at any time. As the values in a dynamic shadow fact change, they may cause rules to be *activate* or *retract* rule activations. This allows for Jess to respond to data that is in a state of flux.

There is one disadvantage to combining Jess and Java in a program. This is that errors in embedded Jess code can cause very strange errors to present themselves in Java. When debugging the system it can be difficult to locate the error as the symptom may present

itself in one subsystem while it is caused in the other. One example of this is when a missing bracket in a Jess file caused it to fail to load without causing an error, and when the needed information was accessed by Java it threw a null pointer exception. In this case the erroneous data (actually lack of data) had to pass through the rule engine, and two Java objects before presenting the error in a third object. The fact that Jess is interpreted means that a syntax error will only present itself on runtime, and when embedded in Java it will only present itself as an exception that has little to do with the actual nature of the problem. In this example a null pointer exception won't make you immediately think of a possible syntax error.

Some interesting problems can also be encountered because of the fact that Java variables are *strictly typed* while Jess variables are *untyped*. This causes problems when Jess calls an overloaded java method. For instance if a Java method has two different implementations, one for a string parameter and one for a boolean parameter, and Jess calls that method with the value FALSE then it will be impossible to predict which implementation Jess will execute because Jess can translate FALSE into either a string or boolean value. To deal with this it may be necessary to place the value in an explicit rapper class. So if it was the intended that the boolean value be passed it would be necessary to create and pass a java.lang.Boolean object.

Another difference between Jess and Java is that Jess has no array data structure. Instead the basic data structure in Jess is the list. This is no problem in most cases because arrays can be converted to lists and vice versa, but Jess has no way of representing multidimensional arrays. If a multidimensional array is passed to Jess it will dutifully

convert it to a one dimensional list of the arrays contents. Currently the only way to deal with this problem is to handle the multidimensional array in Java and only access individual data elements from Jess. This solution is very easy as it is simply a matter of creating an abstract data type that acts like the multidimensional array and creating an instance in Jess.

3.3 The Recognise-Act Cycle in Jess

Jess uses a modified Rete algorithm for efficient pattern matching. It implements node sharing to increase efficiency, and uses different kinds of nodes in the network to implement *conditional elements* such as not and test as well as special behaviours for backward chaining. Jess also has built in functions for providing information about the Rete network and individual nodes within it. The *watch compilations* command provides feedback from the Rete network. With this command Jess returns information about nodes added to and shared in the network each time a new rule is added to the system. The *view* command can be used to display a graphical representation of the Rete network. The *matches* command takes the name of a rule and gives information about all the given rules join nodes.

The implementation of the Rete algorithm that is used makes Jess extremely efficient. On some systems Jess can outperform CLIPS by a factor of 20 or more on many problems. On an 800MHz Pentium III Jess has been clocked at over 80,000 rules fired per second and almost 600,000 pattern matching operations performed per second using Sun's HotSpot JVM [4]. There are few rule engines faster.

There are several ways that a programmer can alter the way that Jess handles conflict resolution. It has two built in conflict resolution strategies *depth* (the default) and *breadth* that can be explicitly selected using the *set-strategy* command. Depth always executes the most recently activated rule first, and so inherently does a breadth first search. Breadth executes the rules in the order that they are activated, so the oldest activation always fires first. The programmer can also define their own conflict resolution strategy by creating a Java class that implements the jess.Strategy interface. In doing so the programmer can use heuristic methods of determining the order of execution, or any other method they want. Loading this new strategy is as easy as calling *set-strategy* on the class file.

The conflict resolution strategy can also be manipulated by setting the *salience* value of a given rule. The salience gives the priority of each given rule, and defaults to zero. This method is used to give either higher or lower priority to rules that define special cases. If the system requires that a given rule fire immediately, such as responding to an emergency, then the rule should be given a high salience value. If a rule should always be left as a last resort, such as trying all non invasive options before trying surgery on a patient, then the rule should be given a low salience value. This process of setting salience values should always be used only for special cases because if over used it will be detrimental to performance. It is also considered bad style to use a large number of salience values because it negates the non-determinism of the system by making the rule based program perform more like a conventional procedural program.

3.4 Summary

The Jess language provides the programmer with a full-featured production system that is well suited to producing modern expert systems and other rule based programs. By making Jess free for academic use the developers have put this powerful rule engine within reach of students everywhere for development of projects such as this one. The addition of a readily available textbook written by the creator of the language also encourages development using Jess, but what clinches the deal is the easy interaction with java. All these combined make Jess a very attractive production system.

Jess was designed from the very beginning to encourage integration with Java. Jess can call any Java method, and Java can call any Jess function. This unlocks the use of Java's powerful APIs for use by Jess. It also makes it possible to embed Jess in Java programs. Jess can even allow facts in working memory to respond to changes in values held by JavaBeans. The only hitch is that the added complexity of multiple interacting systems can sometimes create some interesting side effects and debugging headaches. This is a small price to pay for all that functionality.

Jess has several features surrounding the recognise act cycle that help the programmer obtain the desired performance. It uses an optimised Rete algorithm to provide efficient pattern matching, and provides functions to allow the programmer to observe the changes to the network as they occur so that they can modify the rules to improve performance. The rule engine provides two built in conflict resolution strategies, breadth and depth, and allows the developer to create their own. Jess also provides the ability to modify salience

values to compensate for special cases that do not fit into the regular conflict resolution scheme. These and other features have made Jess an interesting language to study.

Chapter 4: The Computer Troubleshooting Expert System

In the last two chapters we have covered background information on expert systems, production systems, and the Jess rule language. It has come time that we delve into the heart of this thesis. In the coming chapter we will flesh out the inspiration behind this thesis, the problem that it is trying to solve, and how that problem will be solved.

The chapter is divided into three key sections. The first introduces the problems faced by the technical support industry. The second will provide information about the software that is currently in place. The third will describe how the current inadequacies will be met by the introduction of a computer troubleshooting expert system.

4.1 The Problem

Computer troubleshooting is an integral part of the modern computer industry. To honor warrantees and satisfy their customers needs, hardware and software manufacturers need to be able to determine the root causes of computer problems so that the proper assistance can be given from the company that produced the part or program that is causing the malfunction. With the incredible number of computer systems involved, providing these services with house calls and local service centers is virtually impossible, if not simply unreasonably expensive. This has meant an increasing need for technical support to be done over the phone from call centers across the globe. Large numbers of people are employed at

relatively low pay rates providing this support, and so the problem of finding skilled service representatives proves difficult. Customers who call in are generally frustrated and even sometimes hostile, and this makes providing support demanding and stressful. This causes employee turnover rates to be very high.

When troubleshooting computer systems there are many possible issues that could be encountered, but almost all of them have at some point been encountered before. In fact after only a few months of work a support representative will have encountered most of the problems that they receive during a given work day. In fact after six months on the phones a technician is considered to have encountered every issue that currently exists at least once. The majority of the calls that are received deal with issues that they are likely to encounter at least once a week. If a technician can learn to recognize these common issues then their troubleshooting becomes extremely efficient, but this requires a fair degree of skill and a natural ability to recognize the symptoms of these common problems. Very few of those who work in the technical support industry will have the ability to attain this level of expertise, and with the high turnover rate of employees even fewer will actually reach that level. Those who do become expert technicians rarely stay very long, and are quickly seen as good workers and promoted to positions that involve little in the way of time on the phones. These issues compound to create a situation where a given call center will have a precious few expert troubleshooters on the phones at any one time.

The question is how to make this expertise available to all technicians while on the phone. If we could, then the stress of starting would be reduced because the increase in ability to deal with customer issues would inspire confidence. Any reduction of job stress is almost guaranteed to reduce turnover. It would also increase the rate at which those who have the ability to become experts attain that status, and it would increase the performance of those that would not have otherwise had access to that expert advice. These increases in efficiency would simultaneously increase both employee and customer satisfaction.

The greatest costs in the technical support industry are human resources, training, and the overhead of running each call center. The availability of a computer troubleshooting expert system may decrease these costs in several ways. By decreasing turnover and making the job easier to perform, the training costs could be decreased. The increased efficiency of technicians would also reduce the number of them that would be needed, and therefore decrease the number of centers needed. This diminishing need for employees and centers would decrease both overhead and human resources costs. By making the job easier such a system would also decrease the skills required to do the job, and thus make it easier to find willing employees. Reducing the skills needed by the work force would reduce the starting wage required to attract employees. This would make such a system very commercially viable.

4.2 The Current Solutions

Technical support companies currently use a combination of several systems to aid their support representatives in solving the issues that they encounter. First they have data about known issues with respect to specific pieces of hardware or software. This is stored on websites that are linked from information about the system owned by the caller. This is a good way of providing information to technicians about problems inherent in a given system (known glitches or design flaws). Second they have small troubleshooting checklists that work as a web site that allows the technician to find solutions to certain general problems, but that require them to know at least approximately what the problem is and gives only a list of possible solutions to use in a trial and error process. Thirdly they use a searchable database of solutions to specific problems. This database is really only useful if you know the specific problem (i.e. a specific error message) otherwise the representative is likely to face a huge list of results to their search with no advice as to which ones to try first. If a specific problem is known then the database system is extremely efficient.

The known issues system and database are very useful and efficient in certain situations (in my experience I never did find a use for the checklists as their solutions were too inefficient). In general use none of these systems work well because they require a technician who has knowledge of how the system works so that they can ask relevant questions that narrow down the problem to a few specific solutions. This clearly means that what is needed is a system that asks

guiding questions and narrows the field of possible problems until the number of possible solutions is reasonable enough that the technician can efficiently solve the problem. A technician working with this new system would be able to follow a dependable process to solve most problems encountered on the job. They would first check for known issues for the given system, then they would use the new system to narrow down the problem to a specific list of solutions, which could then be retrieved from the database system.

4.3 Proposed Solution

To complement the current systems in place and provide the expertise that is needed to guide the average technical support representative through an efficient and effective troubleshooting process the logical next step is to develop an expert system that represents the problem solving expertise of human experts in the field. This system would be required to ask questions that would guide the technician through testing the system and narrowing down the possible sources of the problem. It should then provide a list of solutions to the possible causes.

The system should allow the user to save the current state of troubleshooting for later retrieval by another technician. This is so that the troubleshooting process does not have to be repeated if a call is accidentally disconnected. The saved state should be accessible to the user by way of a reference. A reset switch should also be provided that restarts the troubleshooting process. The system should also be easy to expand to accommodate changes in technology as they become available

to the customers. Ease of use and efficiency will be important if the system is to be widely accepted.

This application exhibits nondeterministic characteristics in that if a problem could have more than one possible cause the system must explore all possibilities concurrently. This would be done by asking questions that relate to all possible causes. This kind of behavior would be difficult to perform with conventional programming, but is ideal for an expert system. The problem domain is well bounded, in that the system only has to have knowledge of the most common problems. A human expert is unlikely to know solutions to all computer problems, so it is acceptable for the system to be able to increase efficiency for those issues that make up the majority of the calls. As can be seen above there is a great need for a system of this type to provide assistance where none currently exists. There are human experts from whom the knowledge can be retrieved, namely several colleges and myself who are willing to participate. This knowledge is also particularly easy to translate to a rule based structure. The problem-solving knowledge is mainly a combination of experiential knowledge and an organized trial and error process. This means that this particular application meets all of the requirements for being an ideal problem to be solved by an expert system as listed in section 2.1 of this thesis.

The main subsystem of the computer troubleshooting expert system as concerns this thesis is the inference engine / knowledge base. It is implemented as a basic search program for a *directed acyclic graph* (*DAG*) using the Jess rule language.

Each *internal node* represents a sub-domain of the problem set for which the system has knowledge. The *leaf nodes* represent specific solutions to known computer problems. As the DAG is traversed downward the problem set is broken down into successively narrower problem domains until all the possible solutions are known. The information about each node is stored in a fact that is always asserted into working memory when the system is reset. The inferences are handled by a single rule that handles all interactions with the GUI front end and manages the traversal of the graph.

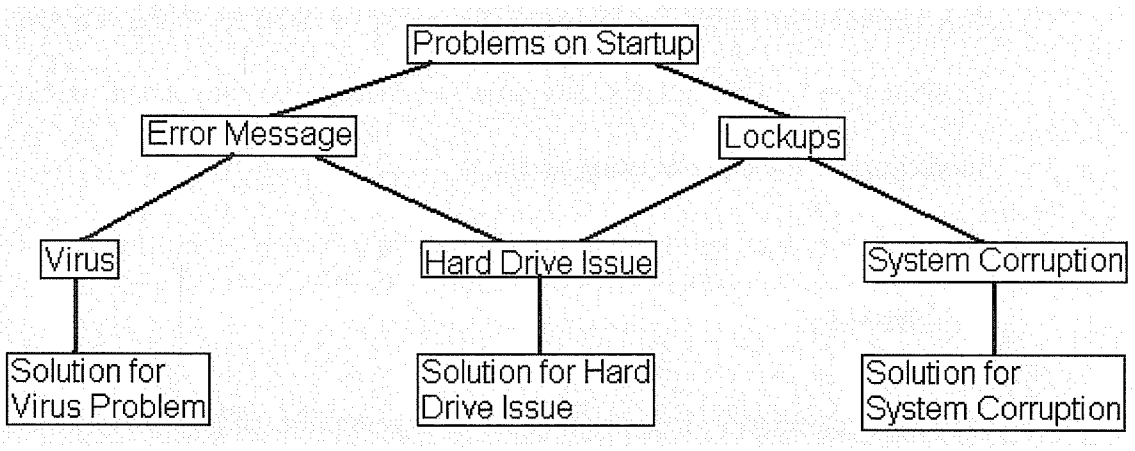

Figure 4.3.1: An example DAG structured knowledge base.

As you can see in Figure 4.3.1 two domains may share a sub-domain. This is because two very different symptoms may have the same root cause. In this case there are two possible problems that can cause a problem on start-up, an error message and a lockup. Also we see that either a virus or a hard drive issue may cause an error message, and each of these have one possible solution. This is similar to lockups, which can be caused by either a hard drive issue, or system corruption (which also has one solution).

Each node on the given DAG has knowledge of what its parent nodes are, but generally a *parent node* does not know what its *children* are. The child activates, is put on the list of those to be tested, when its parent domain is confirmed to be a domain in which a possible solution to the current problem may reside. Because of this, it is possible to add sub domains, or solutions, to a given problem domain without modifying the existing nodes. This organisation does have one drawback: for an existing sub domain to be activated by a new one's confirmation, the old node would have to be modified. For this purpose a second method of activation was included in which the parent holds a list of children to be activated upon confirmation. This allows any new knowledge to be added without any modification of the pre-existing nodes.

2.4 Summary

The technical support industry requires a large number of support representatives who can troubleshoot computer problems effectively and efficiently, but expert technicians are few and far between. This causes a fundamental problem in which new technicians flounder. Having not yet obtained the level of expertise needed to properly perform the job, they face the stress of feeling incompetent. Many support representatives will never become an expert because they will either quit, or do not possess the skills needed to reach that goal.

Currently, several software aids are in place to assist technical support representatives. These tools can be very useful when solving certain types of

problems. Specifically, they solve problems to which the root cause is obvious, and issues that involve known hardware and software design and construction flaws. The problem lies in that there is no aid provided that helps to narrow down the cause of an unknown issue.

This is where a computer troubleshooting expert system would provide the ideal solution. The system would work by asking the user guiding questions to narrow down the possible causes of the problem and provide a list of possible solutions. It should allow a save of the current state of troubleshooting, so that later it could be reloaded. It would also be made easy to expand; adding new problems to the domain specific knowledge. This system would compliment the existing software to provide comprehensive support to technicians on the phones.

Chapter 5: Design and Implementation

In the last chapter we explained the problem for which the computer troubleshooting expert system was produced, why an expert system is the right approach, the functionality required of the system, and a high level view of how they will be achieved. In this chapter we will delve deeper into the internal workings of the computer troubleshooting expert system. To do so it is divided into three sections. The first one explains the interactions between the various elements that make up the system. The second section gives direct code examples to illustrate how the knowledge base/ inference engine guide the system through the troubleshooting process. The third describes how the GUI provides interactivity for the user.

5.1 Design

The Computer Troubleshooter prototype was implemented using Jess embedded in Java. This means that the Jess rule engine is instantiated as an object in a Java program. A Jess controller object was created so that the rule-based system would present an easier interface with the Java system. It presents itself as a simple abstract data type with basic methods for retrieval of the list of questions for the user.

During normal operation the GUI prompts the user with a list of yes or no questions. The responses are stored in an array, which is then passed to the Jess controller. The Jess controller then compares the array of Boolean values with the

confirmation values (answers that confirm the validity of each problem domain) that where given with the questions from the rule engine. For those questions whose answers match the confirmation values, the Jess controller places domain confirmations in working memory. The Jess controller then issues a run command to the rule engine, which fires the rules that where activated by the domain confirmations. As the inference engine fires the rules it adds the domain question and confirmation values to a list, which is then retrieved by the Jess controller. The Jess controller then compiles a new array of questions, which are then returned to the GUI. The GUI then presents this new list of questions to the user. The process then repeats itself until the user resets the system is reset by the user, or the system runs out of questions to ask. This process can be seen in Figure 5.2.1.

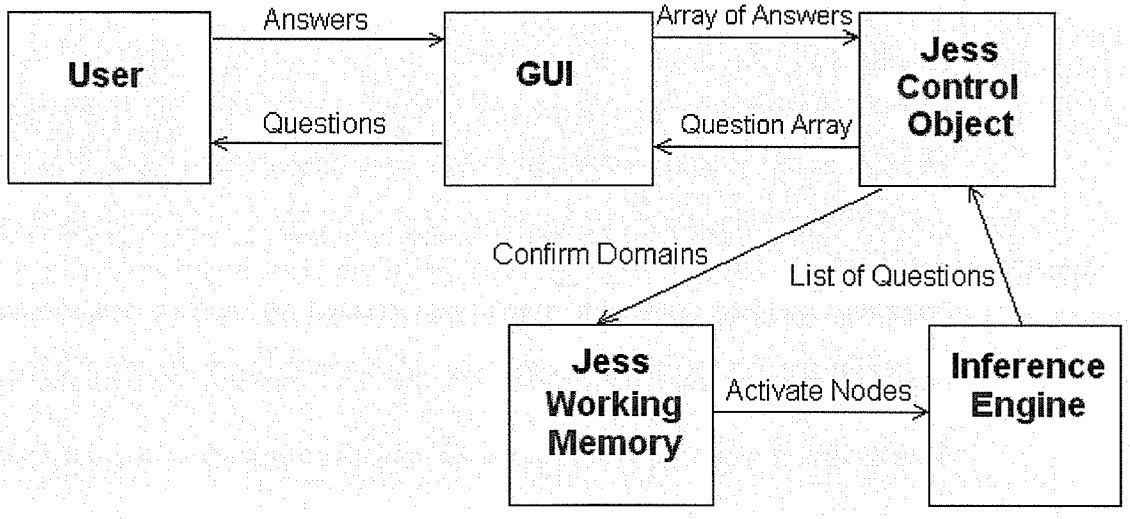

Figure 5.2.1: The information flow in the computer troubleshooter.

The system must also deal with several situations other than the normal narrowing of the problem domains. It also has to handle possible solutions when they are found. When the inference engine encounters solution nodes it adds them to the

list just like the problem domain nodes. When encountered by the Jess controller it puts them in a reserved section of the question array. They are then recognized by the GUI and presented to the user as solutions. The system must also be capable of saving and reloading the current state of execution. By saving the current list of questions and confirming values held by the Jess controller the system can save the current state of the troubleshooting process. Resetting the rule engine to its starting state and restoring these values will resume the troubleshooting where it left off. The system must also be capable of resetting the troubleshooting process at any time. Resetting the troubleshooting system is simply a matter of calling a reset command on the rule engine from the Jess controller.

The system is stored in seven separate files on the host system; five of which are important. These are the GUI class file, the Jess controller class file, rule base batch file, the knowledge base batch file, and the Jess jar file. The GUI class file functions as the driver program that initializes the entire system and provides the user interface. The Jess controller class file loads the rule engine from the Jess jar file and executes the rule base batch file to initialize the rule engine with all rules and all initial facts. The rule base batch file provides the rules that are loaded into the rule engine upon initialization and calls the knowledge base batch file. The knowledge base batch file loads the nodes of the DAG that define the chains of inference. Alterations to the problems that can be handled by the system are possible through modification of the knowledge base batch file.

The last two files used by the system are the class files that define an abstract data type used by the system. When passing the list of questions and solutions from the rule engine to the Jess controller the transition is handled by giving both entities references to the same instance of this abstract data type. This object stores the name, type, text, link, confirming value, and list of triggers held by each node that is currently active. Using this information the Jess controller can pass the questions to the GUI and confirm the nodes that are given proper answers.

5.2 Knowledge Base Implementation

The nodes are stored in facts which are asserted when the system is loaded, and automatically loaded into working memory whenever the system is reset. These nodes are asserted using a single deffacts command. Each of these nodes stores the name of the node, the type (problem domain or solution), the text message for the user, the confirmation value, possibly a web link, a list of the domains to which this node belongs (which if confirmed will cause the node to activate), and a list of triggers (nodes to activate if the node is confirmed). If the node is a problem domain node the text message for the user will be a yes or no question whose answer will confirm or deny that the problem may belong in the nodes domain. If it is a solution node then the text will contain a title describing the solution. The web link will link to information such as tests that may be performed to answer the question given, or an in-depth description of a given solution.

For example the node in Example 5.3.1 defines the domain of all problems that cause the system to be unable to enter windows. As you can see it is a problem domain node so it is of type question, and it has a question in the text slot that will confirm or deny that the problem belongs to the given problem. If the answer given by the user is returned by the GUI as a false it means that the system can't get into windows, and therefore the system will confirm this domain. The link is blank, so this node does not have a web site that explains proper testing to find an answer to the posed question. This node belongs to the initial domain, meaning when the system is reset it will activate this node. For a node that represented a sub domain of this one the domain slot would contain cant_enter_Win, and so the node would be activated when the Jess controller asserted the fact (domain cant_enter_Win). This node does not have any triggers. If it did have triggers then upon confirmation of the node the Jess controller would assert the fact (trigger name_of_triggered_domain), and the triggered domain would be activated.

```
(node  (name cant_enter_Win)
       (type question)
       (text "Can the system get in to Windows?")
       (answer FALSE)
       (link "")
       (domain init))
```

Figure 5.3.1: An example node fact.

Among other things, the rule base batch file defines a single rule that handles all inferences in this system. This rule can be seen in Example 5.3.2. This rule is a good example of the prefix notation that is inherent in the Jess language. When

using the defrule function in Jess the first argument is the name of the rule, the second argument is a string describing the rule, the third argument is a list of conditions that must be met for the rule, the fourth is "=>", and the fifth is a list of actions to perform if the rule fires. This rule will fire under two possible conditions separated by the "or" conditional element. The first possibility is that the rule will fire if a domain fact in working memory matches a domain that is a member of the list of domains defined in one of the node facts. The second possible activation will occur if a trigger fact in working memory matches the name of a node fact. When the rule fires it calls the add method of the active list, which is the Java object to which the Jess controller has access.

```
(defrule ask_question
        "takes submitted questions and executes method for adding to list"
        (or     (and    (domain ?i)
                        (node   (domain $?d&:(member$ ?i $?d))
                                (name ?name)
                                (type ?type)
                                (text ?text)
                                (answer ?answer)
                                (link ?link)
                                (trigger $?trigger)))
                (and    (trigger $?n)
                (node   (name ?name&:(member$ ?name $?n))
                        (type ?type)
                        (text ?text)
                        (answer ?answer)
                        (link ?link)
                        (trigger $?trigger))))
=>
(call ?*active_list* add ?name ?type ?text ?answer ?link ?trigger))
```

Figure 5.3.2: The rule for adding nodes to the active list.

5.3 GUI Implementation

The GUI class starts by initializing a menu bar containing the reset and logout commands. It then instantiates a Jess controller object, which in turn initializes the knowledge base and retrieves the first list of questions for the user. Once this is done it initializes a question window by creating an instance of itself with a second constructor that takes the Jess controller as an argument. This new instance retrieves the list of questions and creates a window to display them to the user. Upon submission of the answers it passes them to the Jess controller, destroys the current question window, and creates a new instance of the GUI class that will in turn display the new list of questions. When the list of questions runs out the new instance of the GUI class will display the list of solutions that has been generated.

What follows is a set of screen shots that shows the computer troubleshooter in use. In Figure 5.3.1 the reset and logout options can be seen along with the initial set of questions. Upon selecting no and submitting we come to Figure 5.3.2. In this shot the user has stated that an error message occurred. Submitting brings us to Figure 5.3.3. Here they have stated that the error occurs after the windows splash screen and so leading to Figure 5.3.4. Here the system has run out of questions, and so is listing all solutions encountered during execution. To restart the process the user simply has to exit the current question window and go to the restart in the "File" menu on the task bar.

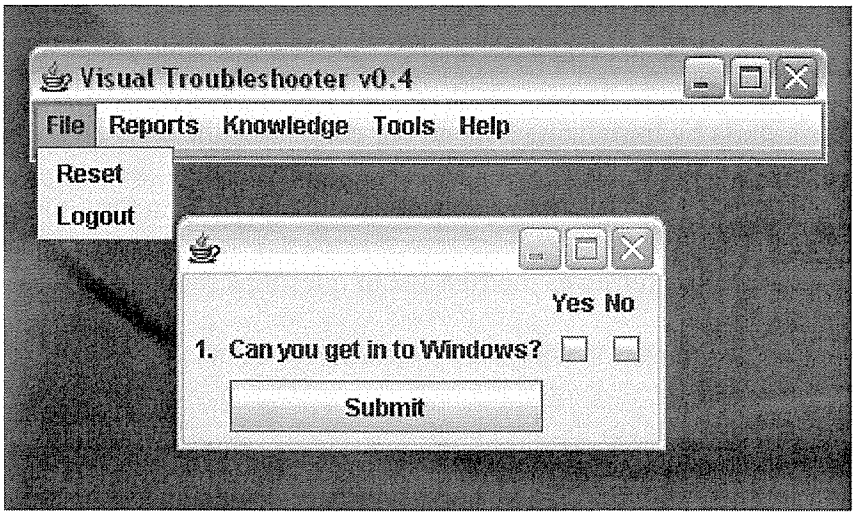

Figure 5.3.1: The initial questions and menu bar

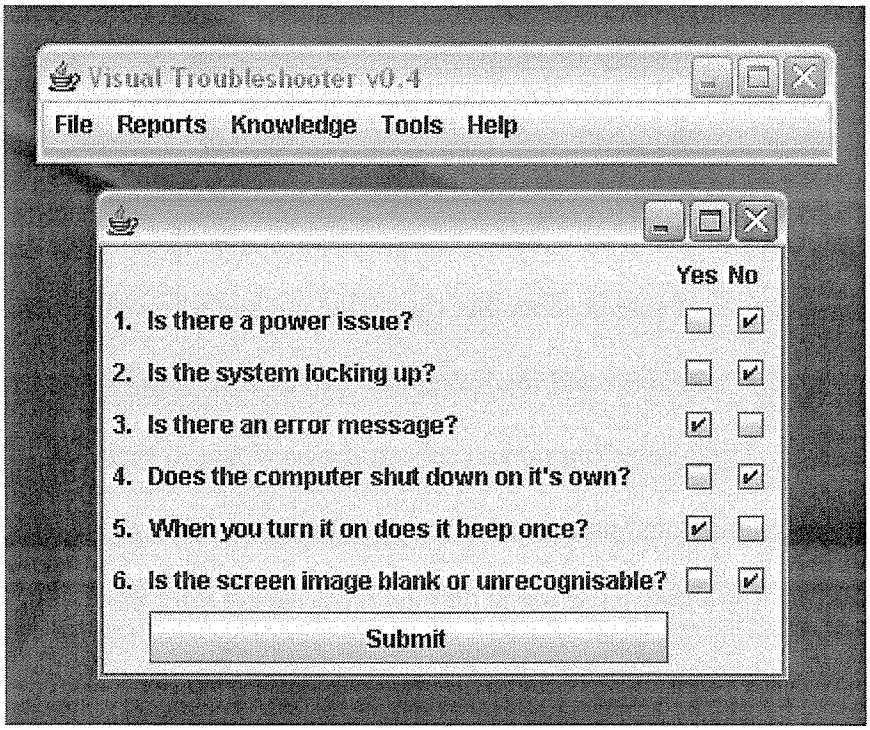

Figure 5.3.2: System is encountering an error message.

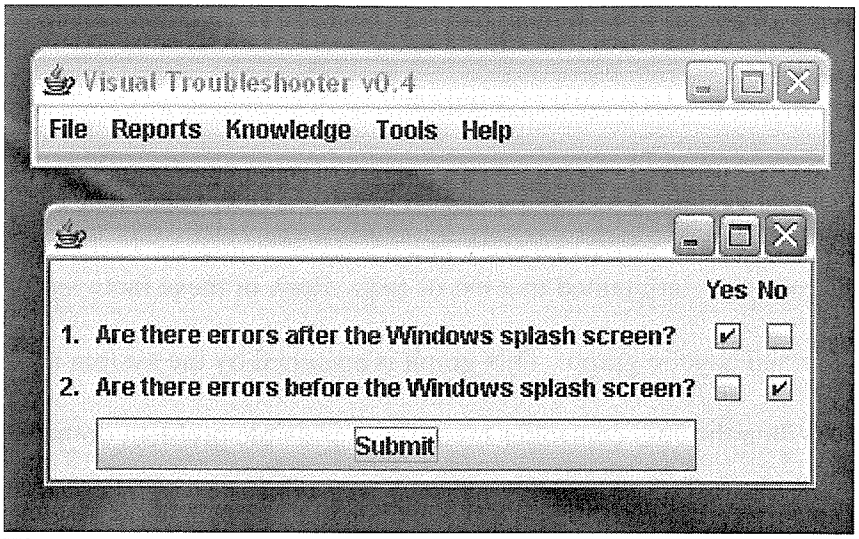

Figure 5.3.3: The error occurs after the windows splash screen.

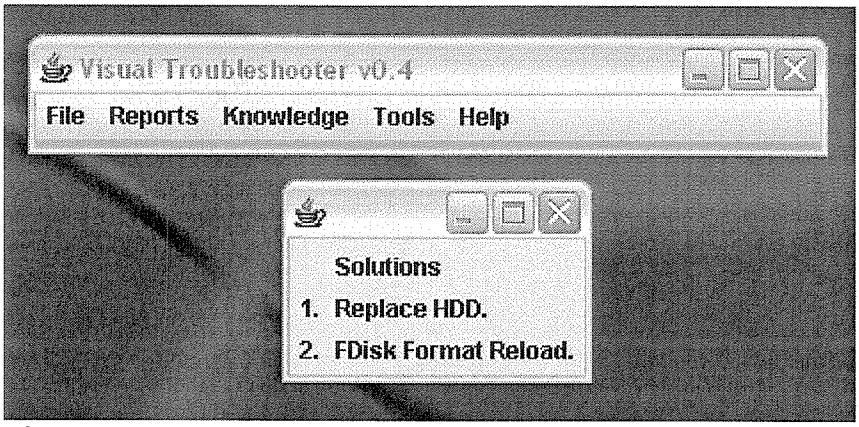

Figure 5.3.4: The system presents a list of possible solutions.

5.4 Summary

The computer troubleshooting expert system created for this thesis presents the user with a list of questions. Upon answering them the system processes the input and decides in which possible problem domains the issue may lie. Once the possible problem domains are identified the system presents another list of questions, each of which pertains to one of the possible problem domains

selected. This process is repeated until the possible problems are small enough that the system can produce a list of solutions.

All the questions and all of the information needed for the system to go through its thought process are contained in a list of facts. Each of these facts represents a node in a directed acyclic graph. This graph is searched by the system using a single rule. When the Jess controller identifies a possible problem domain this rule searches the facts for sub-domains that relate to the domain submitted. When these facts are identified their questions and other needed information is passed to the Jess controller to be presented to the user.

Chapter 6: Conclusions and Future Work

Over the course of this thesis we have covered a variety of topics. First, we discussed expert systems, production systems, and the Rete algorithm. Second, we explored the Jess language, its interaction with Java, and its implementation of the Rete Algorithm. Third, we provided insight into the problems faced by the computer troubleshooting industry, the current software aids used, and the solution proposed by this thesis. Finally we explained how a prototype implementation was designed and implemented. We will now seek to summaries the concepts that have been discussed and how future work can be done to produce the final application.

This chapter has two key sections. The first is a summary of the information provided in the preceding chapters. The second will describe the planned future development of the computer troubleshooting expert system.

6.1 Conclusions

At the beginning the goal of this thesis was to find a better way to provide assistance to technicians in the computer technical support industry. This came from the discovery that the current aids to troubleshooting were woefully inadequate. A niche existed into which a new system could be placed to perform the functions not currently provided. An interest in the techniques that have been developed by artificial intelligence research led me to the possibility that a computer troubleshooting expert systems may just hold the answer. This raised

an interesting question which then became the premise of this thesis. Would a computer troubleshooting expert system represent the next logical step in technical support aids?

Expert systems are systems that emulate the thought process of human experts. They are made up of a knowledge base to store the thought process, an inference engine that applies that thought process to the current situation, and a user interface that provides interaction with the user. Expert systems are only appropriate for solving specific types of problems. Many problems are inappropriate because the knowledge needed is too vast, it is not attainable, or the problem can more efficiently solved by other means. Expert systems are commonly created using production systems, which are made up of a rule-base, working memory, and recognize-act cycle. The rule base is made up of a series of production rules that represent knowledge about the thought process. The working memory is made up of an ordered list of facts known about the current situation. The recognize-act cycle draws conclusions by matching rules to facts. This is usually implemented using the Rete algorithm, which creates a network that identifies facts that match the requirements for rules and remembers past matching operations. This creates a production system which is efficient at storing and applying the knowledge that is necessary for the creation of an expert system.

Jess is the production system used for creating the prototype application for this thesis. In comparison to the other production systems available Jess was found to

be more current, cheaper, provided a well written text (written by the languages creator), and most importantly interacted better with a familiar language (namely java). Jess' interaction with Java allows programs written in each language to call functions written in the other language. This allows the embedding of Jess code in Java programs, and allows Jess to utilize Java APIs. This interactivity is very useful but must be used wisely to avoid creating bugs that are difficult to locate and repair. All this functionality is great, but it would all be for naught without a good recognise act cycle. For this Jess implements a modified Rete algorithm and provides tools for optimizing its performance and that of the conflict resolution strategy. All told Jess has proved to be a useful system in the creation of the computer troubleshooting expert system.

The computer troubleshooting industry has a chronic problem with too few expert technicians available to answer the phones. As a result they have to make do with less qualified staff, which causes customer and employee satisfaction to be poor at best. To aid technicians they provide systems that help with specific issues, but do not provide guidance as to how to use an organized process to narrow down the problem and identify the underlying issue. One possible solution would be to put an expert system in place to fill this niche. This system would provide the user with a list of questions that would successively narrow down the problem until solutions can be found. The system would also allow the user to save the state of troubleshooting to prevent multiple technicians from having to do the same work twice. Together with the other systems already in place this system

will allow a new technician to confidently handle most of the calls they handle without needing help.

The computer troubleshooting expert system provides guiding questions to the user, and upon receiving answers uses them to narrow down the possible problem sets to which the issue faced may belong. It then provides questions that apply to those problem domains identified. This occurs until a list of possible solutions can be produced that accounts for all possible causes. The state of this process can be saved at any time by saving the list of current questions and solutions, and the process can be reset by simply calling a reset on the rule based system. All of the information needed for the system to go through the aforementioned processes is contained in a list of facts each of which represents a node in a directed acyclic graph. To search this graph, a single rule is used that identifies the nodes that should activate when a given domain is identified by the Jess controller. In doing so it adds the nodes information to a list that in turn provides the questions to the user.

All told the information in this thesis has produced a clear answer to the original question. A computer troubleshooting expert system is the ideal next step in technical support aids. This is apparent for the several reasons. Firstly, it would compliment the current systems by providing much needed guidance as to how to approach troubleshooting a system when the problem isn't explicitly known. In other words an expert system would provide all of the functionalities needed by the computer troubleshooting community. Secondly, it fits the description of an

ideal problem to be solved by an expert system. In fact it is exactly this type of problem that expert systems where developed to solve. Thirdly, a final commercial version of this system would be relatively easy to produce based on the lessons learned from development of the current prototype. This future development will be discussed in the coming section.

6.2 Future Development

The prototype created for this thesis was developed to provide insight as to the effort and skills needed to create the final system. As such the implementation provided a very limited functionality. The problem domain handled was restricted to covering only a small subset of computer problems. This was done to simplify the collection of the needed information, and any commercial application would need to accommodate a greatly expanded problem domain. The system was also limited by the amount of research that had been completed at the time of implementation. Much research has been done since the development of this prototype, and the information gained has provided insight into how the model could be improved.

The model used to develop the prototype allows the DAG to extend to great depth with no lower limit. It also allows for solutions to occur at any level of the tree, and it is expected that for any one problem their will only be one solution. This was originally done to prevent the production of a large list of solutions that would be inefficient to try in linear order, a problem that was encountered with

troubleshooting checklist systems. The problem with this method is that the system uses forward chaining, which is distinctly more efficient in a broad shallow graph. By limiting the depth of the graph the system could become more efficient.

The depth of the graph could be limited by requiring that a problems identification process be limited to dividing problems into general domains, sub-domains, problem types, problems, and solutions where each is successively lower on the tree. This would limit the depth of the tree, and it would also guarantee that all solutions would be encountered at the same time, thus negating the need to take pains to distinguish between questioning nodes and solution nodes. It would also be recognized that not all problems will be specific, and so may have multiple solution nodes. As long as there aren't too many solutions per problem the previously mentioned problem could be avoided. A solution node could also have a layer of sub nodes that provide questions that give the user insight as to the likelihood that a specific solution is going to solve the problem, and so direct the user as to which solutions to try first. This would make the system run more efficiently and predictably.

Another alteration that would be advantages is to implement the nodes as rules instead of as facts. This would mostly affect the system performance when the system is reset because the Rete network would have to process all the node patterns on every reset as opposed to just loading them right in the network when it is initialized and never having to reload them. It is also a matter of convention.

Normally static knowledge is always represented as rules, and only information about the current problem is stored in working memory as facts. This is a long held convention.

It would also be necessary to implement an explanation facility. The explanation facility was never implemented in the prototype as this was a relatively minor feature that contributed little to the knowledge of computer troubleshooting expert system construction. An explanation facility would be implemented by submitting all information about activated nodes to a Java object that would store information about the chains of inference that have been followed, and the nodes would have to be modified to store more information about the reasoning that may have led to that node becoming active. When submitted to the explanation object the new information in the nodes would be related to the chains of inference to create explanations for how the system reached its current state. It should also discard data about chains of inference that are negated to save memory space. When solutions are reached the explanation object should only hold information about the chains of inference that led to solutions. This would be the most difficult change to the system that would need to take place to make it ready for commercial use, but it would mostly involve implementing of Java classes.

Finally, as stated before, any commercial system would have to handle a much wider problem set. To do this it would be necessary to collect information about what the most common problems are. After eliminating known system issues and

problems which explicitly present the cause (i.e. error messages) a list could be created that covers all issues that should be handled by the expert system. This list would then be presented to troubleshooting experts to determine the steps that they take to identify which issue is present on a given call. This identification process would then have to be encoded into the rules needed to allow the system to recognize this expanded problem domain.

Much of this process would require the participation of the computer troubleshooting company by which the system would be used. The system would have to be tailored to a single company because a company that makes modems is likely to receive a different set of common problems than a company that sells entire computer systems. Some common problems will inevitably be similar, so the system would not have to be completely reengineered to produce a new system for a different company.

6.3 Summary

Over the course of this thesis we have explored all the technologies and techniques used in the development of a computer troubleshooting expert system. We first explained expert systems and what problems they can solve. Then we moved into the Jess rule language, and the features that benefit this application. This led into discussion of the problem we wish to solve, and how it can be solved. The implementation of the prototype was then discussed, and now we have described the future work necessary to complete the project.

Development of a final application would require certain changes to be made to the current prototype. Firstly the system would need to handle an expanded problem domain. This would require a study to be done of the most common problems faced by technicians. Secondly, a reworking of the current application code would be required to improve efficiency. These changes would mostly be small and easy to implement. Finally the system would need an explanation facility to provide the reasoning behind the systems choice of solutions. This would help the users to understand what is known about the problem.

Bibliography

[1] Bratko, Ivan. "Prolog programming for Artificial Intelligence: Third Edition", Addison Wesley, Harlow Eng., 2001 chapters 15-16.

[2] Forgy, Charles L. "Rete: A fast Algorithm for the Many Pattern / Many Object Pattern Match Problem,", Artificial Intelligence 19, 1982 pages 17-37.

[4] Friedman-Hill, Ernest. "Jess, The Java Expert System Shell Version 5.2", Sandia National Laboratories, Livermore CA, May 2001, http://herzberg.ca.sandia.gov/jess/docs/52/

[3] Friedman-Hill, Ernest. "Jess In Action Rule-Based Systems in Java", Manning, Greenwich CT, 2003

[5] Giarratand, Joseph and Gary Riley. "Expert Systems Principles and Programming: Third Edition", PWS Publishing Company, Boston MA, 1998

[6] Ignizio, James P. "Introduction to Expert Systems: The Development and Implementation of Rule-Based Expert Systems", McGraw-Hill, Inc., New York NY, 1991

[7] Khera, Dheeraj (Raj). "The Impact of Expert Systems", Khera Communications, Inc., 1998, http://www.morebusiness.com/running_your_business/businessbits/v2n8.brc

[8] Luger, George F. "Artificial Intelligence Structures and Strategies for Complex Problem Solving Fifth Edition", Addison Westly, Essex England, 2005

[9] Morris, Jason. "Jess Inventor Opines About Rule Engines and Java", Jupitermedia Corporation, October 2003, http://www.devx.com/Java/Article/17651

[10] Walker, Terri C. and Richard K. Miller, "Expert Systems Handbook : An Assessment of Technology and Applications", Prentice-Hall, Englewood Cliffs NJ, 1990

[11] Whitehouse, Peter, "INFORMATION PROCESSING & TECHNOLOGY EDITION 10.010204", wOnKo THES@NE, Brisbane Australia, 1992-2005, http://www.wonko.info/cybertext/ai/ai3.htm

[12] Wu, Chaur G. "Modeling Rule-Based Systems with EMF", Chaur G. Wu, 2004, http://www.eclipse.org/articles/Article-Rule%20Modeling%20With%20EMF/article.html

Appendix A: Questions.java: GUI Class

Purpose:

Provides the GUI class for a computer troubleshooting expert system.

Dependencies

This class utilises the SpringUtilities class available at:
http://java.sun.com/docs/books/tutorial/uiswing/layout/example-1dot4/SpringUtilities.java

Constructors:

One constructor is provided that initialises the menu bar and another initialises a new list of questions or solutions.

Public Methods:

main() - initialises the system
actionPerformed(ActionEvent) - implements GUI interaction

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;

public class Questions extends JFrame implements ActionListener
{
	private int num;  //index of the last question/answer(indexed from one)
	private static CTList list;  //the jess controler object
	private int[] answers=new int[25];  //the list of answers for the current
                                           //questions
	private JCheckBox[][] b;  //the check boxes for the current questions on
                                  //display
	private JFrame myWindow;  //the current question/answer window

//main method: used to initialise the system
	public static void main(String[] args)
	{
		Questions window = new Questions();
		window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		window.setVisible(true);

	}

//constructor for desplaying the menu bar
	public Questions()
	{

		list= new CTList();
		Questions window = new Questions(list);

		setSize(400, 60);
		setTitle("Visual Troubleshooter v0.4");
		Container contentPane = getContentPane();
		contentPane.setBackground(Color.lightGray);
		contentPane.setLayout(new BorderLayout());

		JMenu fileMenu = new JMenu("File");
		JMenuItem a;
			a = new JMenuItem("Reset");
			a.addActionListener(this);
			fileMenu.add(a);
			a = new JMenuItem("Logout");
			a.addActionListener(this);
			fileMenu.add(a);
```

```
        JMenu reportsMenu = new JMenu("Reports");
                a = new JMenuItem("Reports go Here");
                a.addActionListener(this);
                reportsMenu.add(a);

        JMenu knowledgeMenu = new JMenu("Knowledge");
                a = new JMenuItem("Search Knowledge");
                a.addActionListener(this);
                knowledgeMenu.add(a);

        JMenu toolMenu = new JMenu("Tools");
                a = new JMenuItem("Tools listed here");
                a.addActionListener(this);
                toolMenu.add(a);

        JMenu helpMenu = new JMenu("Help");
                a = new JMenuItem("Show Help Files");
                a.addActionListener(this);
                helpMenu.add(a);
                a = new JMenuItem("About");
                a.addActionListener(this);
                helpMenu.add(a);


        JMenuBar mBar = new JMenuBar();
        mBar.add(fileMenu);
        mBar.add(reportsMenu);
        mBar.add(knowledgeMenu);
        mBar.add(toolMenu);
        mBar.add(helpMenu);
        setJMenuBar(mBar);
    }


//constructor for displaying the question and answer windows
    public Questions(CTList l)
    {
        list=l;
        String[] questions= list.getCurrent();

        num=0;
        for(int i=0;i<15; i++){
            if (questions[i]==null){
                num=i;
                break;
```

```
                }
                else num=15;
            }

            if(num==0){
                for(int i=15;i<25; i++){
                    if (questions[i]==null){
                        num=i;
                        break;
                    }
                    else num=25;
                }
                displaySolutions(questions);
            }
            else{
                displayQuestions(questions);
            }
        }

//desplays a new question window
    private void displayQuestions(String[] questions){

            JPanel p = new JPanel(new SpringLayout());


            JLabel q[] = new JLabel[num];
            b = new JCheckBox[num][2];

            p.add(new JLabel(" "));
            p.add(new JLabel(" "));
            p.add(new JLabel("Yes"));
            p.add(new JLabel("No"));

            for(int i=0; i<num; i++){
                JLabel number=new JLabel((i+1) + ". ");
                p.add(number);

                q[i]= new JLabel(questions[i]);
                p.add(q[i]);

                b[i][0]= new JCheckBox();
                p.add(b[i][0]);

                b[i][1]= new JCheckBox();
                p.add(b[i][1]);
            }
```

```
                JButton subButton = new JButton("Submit");
                subButton.addActionListener(this);
                JLabel blank1 = new JLabel(" ");
                JLabel blank2 = new JLabel(" ");
                JLabel blank3 = new JLabel(" ");


                p.add(blank1);
                p.add(subButton);
                p.add(blank2);
                p.add(blank3);

                SpringUtilities.makeCompactGrid(p,
                        (num + 2), 4, //rows, cols
                        5, 5, //initialX, initialY
                        5, 5);//xPad, yPad


                myWindow = new JFrame();


                p.setOpaque(true);

                myWindow.setContentPane(p);
                myWindow.pack();

                myWindow.setVisible(true);
        }


//displays the solution list
        private void displaySolutions(String[] solutions){
                JPanel p = new JPanel(new SpringLayout());


                JLabel q[] = new JLabel[num-15];

                p.add(new JLabel(" "));
                p.add(new JLabel("Solutions"));

                for(int i=15; i<num; i++){
                        JLabel number=new JLabel((i-14) + ". ");
                        p.add(number);
```

```
                q[i-15]= new JLabel(solutions[i]);
                p.add(q[i-15]);
        }


        SpringUtilities.makeCompactGrid(p,
                (num - 14), 2, //rows, cols
                5, 5, //initialX, initialY
                5, 5);//xPad, yPad


        myWindow = new JFrame();


        p.setOpaque(true);

        myWindow.setContentPane(p);
        myWindow.pack();

        myWindow.setVisible(true);
    }


//actionPerformed
    public void actionPerformed (ActionEvent e){
        Container content = getContentPane();

//Listener for the question windows
        if (e.getActionCommand().equals("Submit"))
        {
            for(int i=0; i<num; i++){
                if(b[i][0].isSelected()){
                    answers[i]=1;
                }
                else if(b[i][1].isSelected()){
                    answers[i]=0;
                }
            }

            String[] junk = list.getNew(answers);
            Questions window = new Questions(list);
            myWindow.dispose();
        }

//Listeners for the menu bar
        else if (e.getActionCommand().equals("About"))
```

```
                {
                        About window = new About();
                }
                else if (e.getActionCommand().equals("Reset"))
                {
                        list.reset();
                        Questions window = new Questions(list);
                }
                else if (e.getActionCommand().equals("Logout"))
                {
                        System.exit(0);
                }
                else if (e.getActionCommand().equals("Show Help Files"))
                {
                        Help window = new Help();
                }

//Default Listener
                else
                {
                        System.out.println("Error in button interface");
                }
        }//end Action Listener


}//end class
```

Appendix B: CTList.java: The Jess Controller

Purpose:

Provides the Jess controller class for a computer troubleshooting expert system.

Constructor:

One constructor is provided that initializes the Jess rule engine and retrieves the first list of questions for the user.

Public Methods:

getCurrent() - takes no input and returns the current list of questions.

getNext(int[]) - takes a list of zeros and ones that represent the answers given by the user and returns the new list of questions.

reset() - resets the Jess rule engine.

```
import java.*;
import jess.*;
class CTList{
        private static CTNodeList nl;
        private static CTNode[] nodeArray;
        private static String[] textArray;
        private static int[] mapping;
        private static boolean[] answers;
        private static int solNum;
        private static Rete engine;

        public CTList(){
                nl=new CTNodeList();
                Value v;
   try{
                engine = new Rete();
                engine.store("LIST", nl);
                v=engine.executeCommand("(batch CTKnowledgeBase.clp)");
                engine.reset();
                engine.run();

                nodeArray=nl.getList();
   }
   catch(Exception e){}

                textArray=new String[25];
                answers= new boolean[15];
                mapping= new int[25];
                for(int i=0;i<25;i++){
                        mapping[i]=-1;
                }

                int j=0;
                solNum=0;
                String type=null;
                for(int i=0; i<25 && j<15 && solNum<10; i++){
                        type=nodeArray[i].getType();

                        if(type==null){
                                break;
                        }
                        else if(type.equals("question")){
                                textArray[j]=nodeArray[i].getText();
                                answers[j]=nodeArray[i].getAnswer();
                                mapping[j]=i;
                                j++;
```

```
                    }
                    else{
                        textArray[15+solNum]=nodeArray[i].getText();
                        mapping[15+solNum]=i;
                        solNum++;
                    }
                }
            }

            public String[] getCurrent(){
                return textArray;
            }

            public String[] getNew(int[] a){
                for(int i=0; i<15; i++){
                    if(mapping[i]==-1){
                        break;
                    }
                    else if( (answers[i] && a[i]==1) ||
                             (!answers[i] && a[i]==0) ){
                        activate(mapping[i]);
                    }
                }
                cleanSolutions(a);
                update();

                return textArray;
            }


            private void activate(int i){
try{
                Value v= engine.executeCommand("(assert (domain " +
                                                nodeArray[i].getName() + "))");
                String[] triggers=nodeArray[i].getTrigger();
                for(int j=0; j<triggers.length; j++){
                    v= engine.executeCommand("(assert (trigger " + triggers[j] + ")");
                }
}
catch(Exception e){}
            }

            private void update(){
try{
                engine.store("LIST", nl);
                Value v=engine.executeCommand("(bind ?*active_list* (fetch LIST))");
```

```
                nl.clear();
                engine.run();
                nodeArray=nl.getList();
}
catch(Exception e){}

                for(int i=0;i<15;i++){
                        textArray[i]=null;
                        mapping[i]=-1;
                }

                int j=0;
                solNum=0;
                String type=null;
                for(int i=0; i<25 && j<15 && solNum<10; i++){
                        type=nodeArray[i].getType();

                        if(type==null){
                                break;
                        }
                        else if(type.equals("question")){
                                textArray[j]=nodeArray[i].getText();
                                answers[j]=nodeArray[i].getAnswer();
                                mapping[j]=i;
                                j++;
                        }
                        else{
                                textArray[15+solNum]=nodeArray[i].getText();
                                mapping[15+solNum]=i;
                                solNum++;
                        }
                }
        }

        private void cleanSolutions(int[] a){
                int k=10;
                for(int i=0;i<k; i++){
                        if(textArray[15+i] != null){
                                break;
                        }
                        else if(a[15+i] != -1){
                                k--;
                                for(int j=0; j<k; j++){
                                        textArray[15+j]=textArray[15+j+1];
                                        a[15+j]=a[15+j+1];
```

```
                                        solNum--;
                                }
                                textArray[15+k]=null;
                                a[15+k]=-1;
                                mapping[15+k]=-1;
                                i--;
                        }
                }
        }
        public void reset(){
try{
                engine.store("LIST", nl);
                Value v=engine.executeCommand("(bind ?*active_list* (fetch LIST))");

                nl.clear();
                engine.reset();
                engine.run();
                nodeArray=nl.getList();
}
catch(Exception e){}

                textArray=new String[25];
                answers= new boolean[15];
                mapping= new int[25];
                for(int i=0;i<25;i++){
                        mapping[i]=-1;
                }

                int j=0;
                solNum=0;
                String type=null;
                for(int i=0; i<25 && j<15 && solNum<10; i++){
                        type=nodeArray[i].getType();

                        if(type==null){
                                break;
                        }
                        else if(type.equals("question")){
                                textArray[j]=nodeArray[i].getText();
                                answers[j]=nodeArray[i].getAnswer();
                                mapping[j]=i;
                                j++;
                        }
                        else{
                                textArray[15+solNum]=nodeArray[i].getText();
                                mapping[15+solNum]=i;
```

```
                    solNum++;
                }
            }
        }
    }
```

Appendix C: CTKnowledgeBase.clp: The Rule Base Batch File

Purpose:

Initializes the Jess Rule engine for the Computer Troubleshooting Expert System.

Initialization:

This batch file is executed when the system is initialized when the system is started to load the rule into the Rete network, and otherwise initialize the rule-based part of the system.

```
(defglobal ?*active_list* =(fetch LIST))

(deftemplate node "describes a subset of the problem domain"
        (slot name)
        (slot type)
        (slot text)
        (slot answer)
        (slot link (default ""))
        (multislot domain)
        (multislot trigger))

(deffacts init "initial problem domain" (domain init))

(batch Nodes.clp)

(defrule ask_question
        "takes submitted questions and executes method for adding to list"
        (or (and (domain ?i)
                (node (domain $?d&:(member$ ?i $?d))
                        (name ?name)
                        (type ?type)
                        (text ?text)
                        (answer ?answer)
                        (link ?link)
                        (trigger $?trigger)))
          (and (trigger $?n)
                 (node (name ?name&:(member$ ?name $?n))
                        (type ?type)
                        (text ?text)
                        (answer ?answer)
                        (link ?link)
                        (trigger $?trigger))))
        =>
        (call ?*active_list* add ?name ?type ?text ?answer ?link ?trigger))
```

Appendix D: Nodes.clp: The Knowledge Base Batch File

Purpose:

Provides The domain specific knowledge for the Computer Troubleshooting Expert System.

Initialization:

This batch file is executed when the system is initialized when the system is started to load the nodes into the working memory.

Modification:

Each node should fill the slots in the following template definition:

```
(deftemplate node "describes a subset of the problem domain"
    (slot name)
    (slot type)
    (slot text)
    (slot answer)
    (slot link (default ""))
    (multislot domain)
    (multislot trigger))
```

```
(deffacts knowledge_base "knowledge base"
;;;;;;;;;;;;;enter nodes between this comment and the next;;;;;;;;;;;;;

        (node   (name cant_enter_win)
                (type question)
                (domain init)
                (text "Can you get in to Windows?")
                (answer FALSE))

        (node   (name video)
                (type question)
                (domain cant_enter_win)
                (text "Is the screen image blank or unrecognisable?")
                (answer TRUE))

        (node   (name no_post)
                (type question)
                (domain cant_enter_win)
                (text "When you turn it on does it beep once?")
                (answer FALSE))

        (node   (name shutdown)
                (type question)
                (domain cant_enter_win)
                (text "Does the computer shut down on it's own?")
                (answer TRUE))

        (node   (name startup_error)
                (type question)
                (domain cant_enter_win)
                (text "Is there an error message?")
                (answer TRUE))

        (node   (name lockup)
                (type question)
                (domain cant_enter_win)
                (text "Is the system locking up?")
                (answer TRUE))

        (node   (name power)
                (type question)
                (domain cant_enter_win)
                (text "Is there a power issue?")
                (answer TRUE))

        (node   (name no_video)
```

```
        (type question)
        (domain video)
        (text "Is the screen black?")
        (answer TRUE))

(node   (name distorted_video)
        (type question)
        (domain video)
        (text "Is the video distorted?")
        (answer TRUE))

(node   (name post_error)
        (type question)
        (domain startup_error)
        (text "Are there errors before the Windows splash screen?")
        (answer TRUE))

(node   (name win_loading_error)
        (type question)
        (domain startup_error)
        (text "Are there errors after the Windows splash screen?")
        (answer TRUE))

(node   (name post_lockup)
        (type question)
        (domain lockup)
        (text "Does the system lockup before the Windows splash screen?")
        (answer TRUE))

(node   (name win_loading_lockup)
        (type question)
        (domain lockup)
        (text "Does the system lockup after the Windows splash screen?")
        (answer TRUE))

(node   (name replace_vid)
        (type solution)
        (domain no_video, distorted_video)
        (text "Replace the video card.")
        (answer TRUE))

(node   (name reset_cmos)
        (type solution)
        (domain post_error)
        (text "Reset the CMOS.")
        (answer TRUE))
```

```
(node  (name ffr)
       (type solution)
       (domain win_loading_error)
       (text "FDisk Format Reload.")
       (answer TRUE))

(node  (name rephdd)
       (type solution)
       (domain win_loading_error)
       (text "Replace HDD.")
       (answer TRUE))

(node  (name chkdsk)
       (type solution)
       (domain win_loading_lockup)
       (text "Run chkdsk /r in the recovery console.")
       (answer TRUE))

;;;;;;;;;;;;;;;;enter nodes between this comment and the previous;;;;;;;;;;;;;;;;
       )
```

Appendix E: CTNodeList.java: An Object for Passing the Active List

Purpose:

Provides a method for transporting all data pertaining to the list of activated nodes from the Jess rule engine to the Jess controller object.

Constructor:

One constructor is provided that initializes the member variables.

Public Methods:

add - adds a node to the current list of those that have been activated.
clear - reinitializes the list
getList - returns the current list of active nodes.

```
public class CTNodeList{
	private static CTNode[] list= new CTNode[25];
	private static int count=0;
	private static String oldName[]=new String[25];

	public CTNodeList(){
		for(int i=0; i<25; i++){
			list[i]=new CTNode();
			oldName[i]=null;
		}
	}
	public void add(String name, String type, String text, boolean answer, String link,
			String[] trigger){

		if(checkNew(name)){
			list[count].setName(name);
			list[count].setType(type);
			list[count].setText(text);
			list[count].setAnswer(answer);
			list[count].setLink(link);
			list[count].setTrigger(trigger);
			oldName[count]=name;
			count++;
		}
	}
	public void clear(){
		for(int i=0; i<25; i++){
			list[i]=new CTNode();
			oldName[i]=null;
		}
		count=0;
	}
	public CTNode[] getList(){
		return list;
	}

	private boolean checkNew(String name){
		boolean b=true;
		for(int i=0; i<count;i++){
			if(oldName[i].equals(name)){
				b=false;
				break;
			}
		}
		return b;
	}
}
```

Appendix F: CTNode.java: An Object for Storing Node Data

Purpose:

Provides a system for encapsulating all the necessary data about an activated node.

Constructor:

One constructor is provided that initializes the member variables.

Public Methods:

Set and get methods are available for the member variables.

```
class CTNode{
        private String name, type, text, link;
        private boolean answer;
        private String[] trigger;

        CTNode(){

                name=null;
                type=null;
                text=null;
                link=null;
        }

        public void setName(String n){
                name=n;
        }
        public String getName(){
                return name;
        }


        public void setType(String t){
                type=t;
        }
        public String getType(){
                return type;
        }


        public void setText(String t){
                text=t;
        }
        public String getText(){
                return text;
        }


        public void setLink(String l){
                link=l;
        }
        public String getLink(){
                return link;
        }
```

```
        public void setAnswer(boolean a){
                answer=a;
        }
        public boolean getAnswer(){
                return answer;
        }


        public void setTrigger(String[] t){
                trigger=t;
        }
        public String[] getTrigger(){
                return trigger;
        }

}
```