An Automated Verification Tool for Expert Systems

by

Alexandra Y. Pau

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Bachelor of Science and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 28, 1996

Author ....
Department of Electrical Engineering and Computer Science
May 3, 1996

Certified by ....
Professor Kenneth Haase
MIT Media Laboratory
Thesis Supervisor

Acc ....
rgenthaler
Chairman, Departmental Committee on Graduate Thesis
Electrical Engineering and Computer Science

An Automated Verification Tool for Expert Systems

by

Alexandra Y. Pau

Submitted to the
Department of Electrical Engineering and Computer Science

May 7, 1996

In partial fulfillment of the Requirements for the Degree of Bachelor of Science in Electrical Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The automated verification tool is developed to help users in building or modifying a rule-based expert system. The design of this tool differs from that of other verification tools in two major components. First, this tool provides a front-end to existing expert system shells. Users interact with their expert system through a user-friendly interface. All rules are entered in simple "if-then" statements rather than an expert system shell language. Verification is performed in meta-language before the rules are translated to any specific expert system language. In addition, the tool provides a visual representation of the state of a knowledge base. The inference relationship among rules of a knowledge base is illustrated by a directed graph. The tool automatically performs verification and syntax checking. Users are notified of errors and conflicts through popup message windows and by highlighting nodes in the graphical analyzer.

Thesis Supervisor: Kenneth Haase
Title: MIT Media Laboratory

ACKNOWLEDGEMENT

I would like to thank my group supervisor at JPL, Ursula Schwuttke, for giving me the opportunity to work on this project and for her guidance through my internship there. I also want to extend my sincere thanks to Bonnie Traylor, my mentor for this project. Thank you for your patience, your suggestions, and your company. Many thanks to Diana Hamilton and the whole AEG gang for making my internship a fun learning experience.

I am very grateful to my thesis advisor, Kenneth Haase, for his input and suggestions. He guided me through my research project, from providing me background resources to proof-read my thesis write-up.

I especially want to thank my parents for their unfailing support and encouragement. You sacrifice so much so that I can pursue my dreams. Thank you for everything.

Many thanks to my friends and CCF for making MIT a memorable experience. Last but not least, a big thanks to Him who made all these possible.

Table of Contents

1 Introduction 10
- 1.1 Expert Systems 10
- 1.2 Expert Systems at JPL 10
- 1.3 Proposed Solution 12

2 Background 14
- 2.1 Previous Work 14
- 2.2 Background on Verification Algorithms 16

3 Approach 18
- 3.1 The Design 18
- 3.2 Graphical User Interface (GUI) 19
- 3.3 Parser 21
- 3.4 Verification 22
- 3.5 Automated Graphical Analyzer 25

4 Experimental Evaluation 32
- 4.1 Guidelines for Evaluation 32
- 4.2 Insurance Plan Advisor 32
- 4.3 FDIR Boresight 37

5 Conclusion 40
- 5.1 Successes 40
- 5.2 Failures 40

6 Future Work 42
- 6.1 Feature Improvements 42
- 6.2 Author's Comments 42

Bibliography 44

List of Figures

Figure 3.1: Flow chart for the design of verification tool for expert systems 18
Figure 3.2: Graphical user interface for the automated verification tool........................20
Figure 3.3: Calculating the intersection positions of the nodes and their connecting edge 30
Figure 4.1: Pop up message window displaying verification result for Insurance Plan Advisor...33
Figure 4.2: Graphical analyzer for the rule base of the Insurance Plan Advisor 34
Figure 4.3: Pop up message window displaying the verification result of the modified Insurance Plan Advisor system. 36
Figure 4.4: Graphical analyzer for the rule base of the modified Insurance Plan Advisor system...37
Figure 4.5: Pop up message window displaying verification result for Boresight. 38
Figure 4.6: Graph analyzer for the rule base of Boresight system................................39

Chapter 1

Introduction

1.1 Expert Systems

Expert systems have been widely used in industry, business, medicine, and government. The applications of expert systems have been increasingly broad, and include certain generic benefits such as providing expertise where it is scarce, speeding up analysis time, and collectively capturing and preserving knowledge from the top experts in a field. Almost all existing expert systems in use are task specific. That is, they are built for one specific narrow domain task. Traditionally, all theoretical research, software tools, and expert system languages are developed for specific applications. One drawback of these systems is that they are built in a specific expert system language and, therefore, cannot readily be used for other tasks even if the goals of the tasks are similar and same methods and algorithms apply. The development cycle of an expert system is usually long and complex, from data acquisition to verification and validation. More and more, engineers are exploring general application expert systems and tools for aiding expert system development, to save time and resources, and to shorten the development cycles.

1.2 Expert Systems at JPL

At the NASA's Jet Propulsion Laboratory, embedded rule-based expert systems are widely used for real-time data monitoring, diagnosis of system anomalies, and preserving expertise of spacecraft analysts and operation specialists. In the new age of "faster, better, cheaper" missions, developing systems quickly and cost efficiently is heavily emphasized. Traditionally, expert systems at JPL are built individually for their specified tasks. A typical expert system takes more than a year to develop. In developing an expert system at JPL, the scientists or engineers define the specifications and objectives of a system. The specifications are then passed on to software engineers who often are not familiar with the domain of knowledge that is to be captured in an expert system. As a result, a large portion of software engineers' time is spent in acquiring and understanding the specifications given to them. This building process is time consuming and cost-inefficient. In addition, all expert systems are built in a limited time frame. Updating expert systems after they have been put to use is inevitable, be it adding new rules or changing existing rules. The system updating process usually involves rehiring the original software engineers who developed the system. However, these software engineers have already moved on to other projects. They often cannot be found or are not available to take on the updating task immediately. An alternative is to hire any available software engineers. But they would have to spend time to learn the background knowledge of the system. Either way, the procedure is impractical and time-consuming.

Methods for reducing the development cycle of expert systems are in high demand. Most of the inefficiency in a system development cycle occurs during the knowledge transfer between the experts and the software engineers. The experts provide the detailed guidelines of system performance, and the software engineers translate these guidelines to an expert system. Eliminating the role of software engineers in expert system development can reduce the development cycle time significantly. Software engineers are not needed if the development task is in the hands of the experts who know the particular domain of knowledge.

The software engineers are necessary in traditional expert system development for two major reasons. First of all, expert systems are developed in expert system shell languages. The commercially available shell languages are designed similarly to other programming languages and are often too complicated for non-programmers. The scientists/engineers using the systems often do not have the time or willingness to learn an expert system language needed to develop or upgrade an expert system. In addition, verification and validation is the most complex aspect of expert system development. Even experienced software developers are concerned with successful verification of the system. It is too complicated for experts from JPL, who are non-programmers, to verify an expert system. Verification and validation are critical to the success of an expert system. Unfortunately, the verification capability provided by most commercial shells is limited to syntax check and type validation, leaving most of verification in the hands of the developers. As a result, software engineers usually are hired to collect the necessary knowledge from the experts and then translate it to a functional expert system. However, the experts providing the knowledge for the system often have other responsibilities, the knowledge transfer from the experts to the software engineers is ad-hoc and expand over a period of weeks and months. A major portion of the system development time is spent in transferring knowledge from the experts to the software engineers, rather than in coding the specifications to the expert system. In addition, misunderstanding and misrepresentation often occur during knowledge transfer between the experts and software engineers due to inadequate communication skills and the software engineers' lack of understanding of the knowledge domain.

1.3 Proposed Solution

We propose an automated tool for building and verifying expert systems characterized by rules adhering to grammar restrictions imposed by the meta-language. The tool plays the role of software engineers in conventional expert system developments. It will allow experts to program an expert system themselves by entering rules through a user-friendly interface in simple IF-THEN rule statements. The tool will automatically perform verification such as checking for misspelling, inconsistency in variable declaration, rule conflicts, and duplicated rule names. More over, a graphical analyzer supplements the above automated verification. It visual displays the inference relationship among the rules in the rule base, with arrows going from one rule to another indicating the inference sequence in the rule base. A user can easily analyze the inference relationship of the rule base by looking at the graphical display, then edit rule and re-verify the rule base until the rule base until the analyzer displays the desired inference pattern. After completing the verification, the rules are then translated into an expert system shell language.

The design of this tool promises a large degree of flexibility. Since all verification is done in a knowledge base internal to the tool called meta-knowledge, the tool is independent of any expert system language. It can be easily adopted for expert systems in any shell language by providing a simple interface that translates the verified rules into an expert system shell language of choice. In addition, with an automated verification module that displays inference relations among the rules in a rule base, users with little or no knowledge of expert system languages and expert system developments can build systems without external help, and with more confidence than with conventional expert system shells, which require complex programming expertise and extensive verification and validation.

Chapter 2

Background

2.1 Previous Work

An expert system that works for an interesting domain cannot be completely analyzed automatically, either because not enough semantic information is provided, or because there are questions that cross the undecidability barrier (Landauer, 1989). In the process of building an expert system, extensive tests and evaluations of the system is conducted throughout all stages of development. The procedure of testing to make sure that the system is operating according to its pre-defined criteria is called verification and validation.

Verification and validation are performed at various stages of development. There is semantic verification, a simple check of spelling and value types; static analysis, checking for redundant clauses and rules; inference analysis, checking for conflict rules, isolated rules, and circularity in inference rules. Verification is the act of comparing the expert system's result for a given problem in the domain with answers provided by experts of the field to verify that the system is producing the right answer. Therefore, verification often requires detailed knowledge of the system being built, and cannot be easily generalized. However, a major portion of verification for rule-based systems can be automatically performed by the expert system building tool because it involves analysis of the syntax, structure, and inference of the rules of first-order logic.

Extensive research has been conducted in verification and validation of rule-based expert systems. Hundreds of papers on verification and validation algorithms and tools have been published in the past. Some algorithms are purely theoretical arguments and were never evaluated in actual applications; some were implemented in expert systems that work for narrow domains only. Few tools for aiding expert system development exist in the growing field of expert system community. There are no applicable tools that meet demands of the highly specialized expert systems in JPL.

The expert systems described below are typical expert systems. We will examine the verification and validation methods used and the systems' performance.

The Rule Checker Program (RCP) was developed at Stanford University in 1982. It was developed as part of the ONCOCIN project and thus is application-specific. It has the ability to check knowledge bases solely on the basis of their syntax. It can also check a knowledge base in the absence of an inference engine. However, it makes certain assumptions about the rule base. For example, it assumes the reasoning of the rule base is monotonic, rules conclude certain value for a given item of data, and all rules are assigned to particular context which tells the inference engine when to fire the rules. RCP takes on an exhaustive approach in verification. For example, the deficiency check of RCP checks to see if every possible permutation of the parameters and their values is covered by some rule in the table. This method of checking becomes intractable for systems with large rule base.

The CHECK program was developed at Lockheed corporation in the mid 1980s as a verification module for their expert system shell language, called the Lockheed Expert System (LES). LES is a rule-based tool containing both forward-chaining rules and backward-chaining rules. Similar to RCP, the verification of CHECK program is based on three cache tables, the "if-if" table, the "then-then" table, and the "then-if" table. The entries of these tables are created by comparing the clauses of each pair of rules. For a system of n rules, it performs n*(n-1)/2 comparisons for each table.

The Expert System Validation Associate (EVA) system was a long term project at Lockheed Corporation to develop a general-purpose verification and validation tool for expert systems. EVA was built in Prolog, but is not tied to any particular rule language or expert system shell language. It provides a set of translation programs which translate the rule languages of several widely used shells to an internal form based on first-order logic.

EVA is one of the few tools that attempts to create a general-purpose verification and validation. It also includes translation programs to convert rules from other shell languages to Prolog. However, it is unclear as to how much of their described features were implemented or how successful the project was (Preece, 1992).

All verification tools developed thus far are intended to aid software engineers in designing expert systems. They are developed in one particular shell language and incompatible with other shell languages. EVA attempts to make the tool more portable by providing translation programs from other shell languages to Prolog. Nevertheless, the development environment is still in an expert system shell language. This research project attempts to build an automated verification tool that is independent of any expert system shell language and provide extensive aid in verification. The development environment will be in simple IF-THEN English statements, independent of any shell language. Thus, users with little programming experience and unfamiliar with verification procedure can program an expert system using this tool. Furthermore, this tool provides capabilities to aid developers analyze inference sequences and the relations among rules in a rule base.

2.2 Background on Verification Algorithms

Many methods of verification and validation for rule-based systems have been developed in recent years. The most popular models used for verification are mathematical approach and linguistic approach. The linguistic approach checks a rule base for correctness by analyzing the syntax and semantics of its use of language. Most of them are variations of exhaustive comparisons in which each clause is checked against all other clauses in the rule base (Preece, 1992). The computation time of this approach grows in proportion to the combinatorial of total number of rules. The verification becomes impractical for large rule base systems. The mathematical approach analyses the structure of a rule base, which reflects the logical processes behind an expert's reasoning. The mathematical approach is more precise and rigorous. The verification algorithm for this research project is based on the mathematical model.

Verification is an evaluation of a rule base correctness in comparison to a specification of its desired behavior. There is a substantial amount of published work in verification of rule-based expert systems. Unfortunately, there is no one established guideline of criteria and definition for verification in the literature. All researchers define and follow their own verification criteria and authors use their own terminologies to express the concept of verification. However, the different terminologies fundamentally come down to five basic principles:

- *Consistency*, making sure that there are no conflicts among rules. All situations should be well defined. For example, two rules consisting of same antecedent clauses have to draw same conclusion statements.

- *Completeness*, checking for universal applicability of the rule base. That is, the rule base should be able to draw conclusion for any situation in its defined domain.

- *Irredundancy*, assuring that everything in the rule base is there for a good reason. No variables, no clauses, and no rules are redundant.

- *Connectivity*, analyzing groups of rules for the entire dynamic process of inference. The interaction among the rules in any situation should be proper.

- *Distribution*, checking for even distribution in usage of rules in the rule base. Any instance should only have inference to a small portion of the rules. Similarly for a given variable, the set of rules accessing that variable should be a small subset of the rule base.

All verification criteria discussed in published papers can be categorized under a subset of the above five principles. A large portion of the mentioned verification criteria can only be performed with additional knowledge of the specific rule-based system. For example, the missing rules criteria can only be tested if the goals of the system is known. However, by analyzing the structure of rules and the inference relationships among the rules, certain types of error checking can be performed solely based on the rules in a rule base, without additional knowledge of the system. These verification criteria that depend only on rules in a rule base is the focus of this project.

Chapter 3

Approach

3.1 The Design

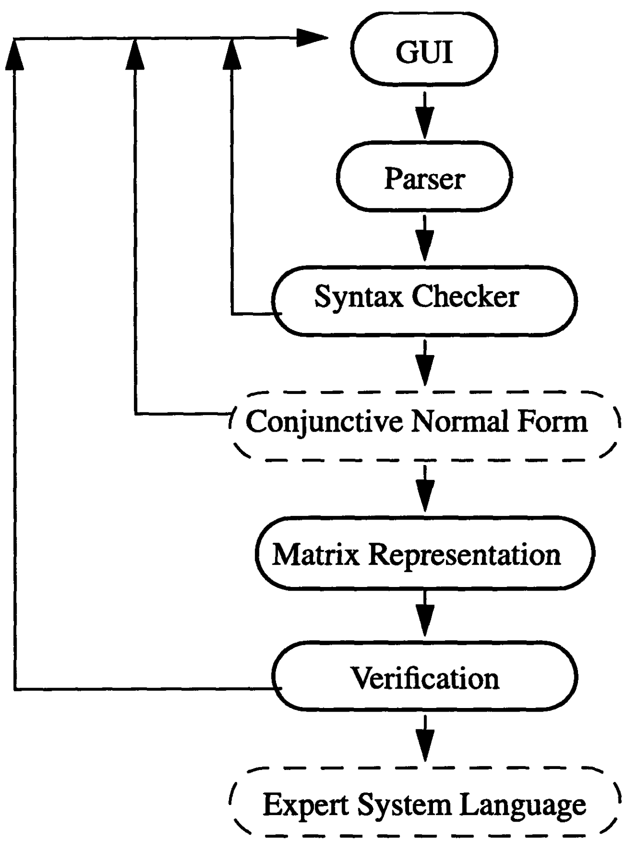


Figure 3.1: Flow chart for the design of the verification tool for expert systems

The primary goal of this project is to develop a protocol application to aid users with no programming background to develop rule-based expert systems on their own, without the help of experience software engineers. Alternatively, this tool will enable JPL engineers and scientists to add or change rules in a rule base by themselves, without wasting time and resources to track down the original system developers. With this emphasis in mind, we developed an automated verification tool for real time rule-based expert systems called SOCRATES.

The design of this application reflects three major differences between this tool and other existing ones. First, the application allows experts with little or no programming experience to build knowledge-based systems on their own, through a user-friendly interface, without the help of software engineers. Second, the verification module is self-contained and independent of any expert system shell language. All of the verification are done in the meta-language, before the rules are translated into a shell language for execution. Performing verification in a self-contained meta-language provides portability across various expert system languages in different platforms. The third and most significant feature is its graphical analyzer. The graphical analyzers are automatically generated by the application to display the relations among the rules of a rule base graphically in a flowchart style for easy comprehension.

In building an expert system using SOCRATES, users insert and edit rules through a user-friendly GUI. Each rule is parsed and then checked for simple syntax errors and duplication (in rule name only). If the rule has no syntax error and the rule name does not exist already, it is entered into the rule base and meta-knowledge. Once the user has finished entering all the rules and press the verification button on the pull-down menu bar, the tool generates several incidence matrices. Upon completing the verification, a graphical analyzer window will appear displaying all rules (represented by nodes) and the relationship among the rules (represented by a line with an arrow indicating the direction of the inference). If additional changes are needed, the user can add more rules through the rule editor on the GUI, and re-process the changes. Once all changes are completed, the user can generate it into an expert system language of his choice. All of the above procedures are performed in real-time.

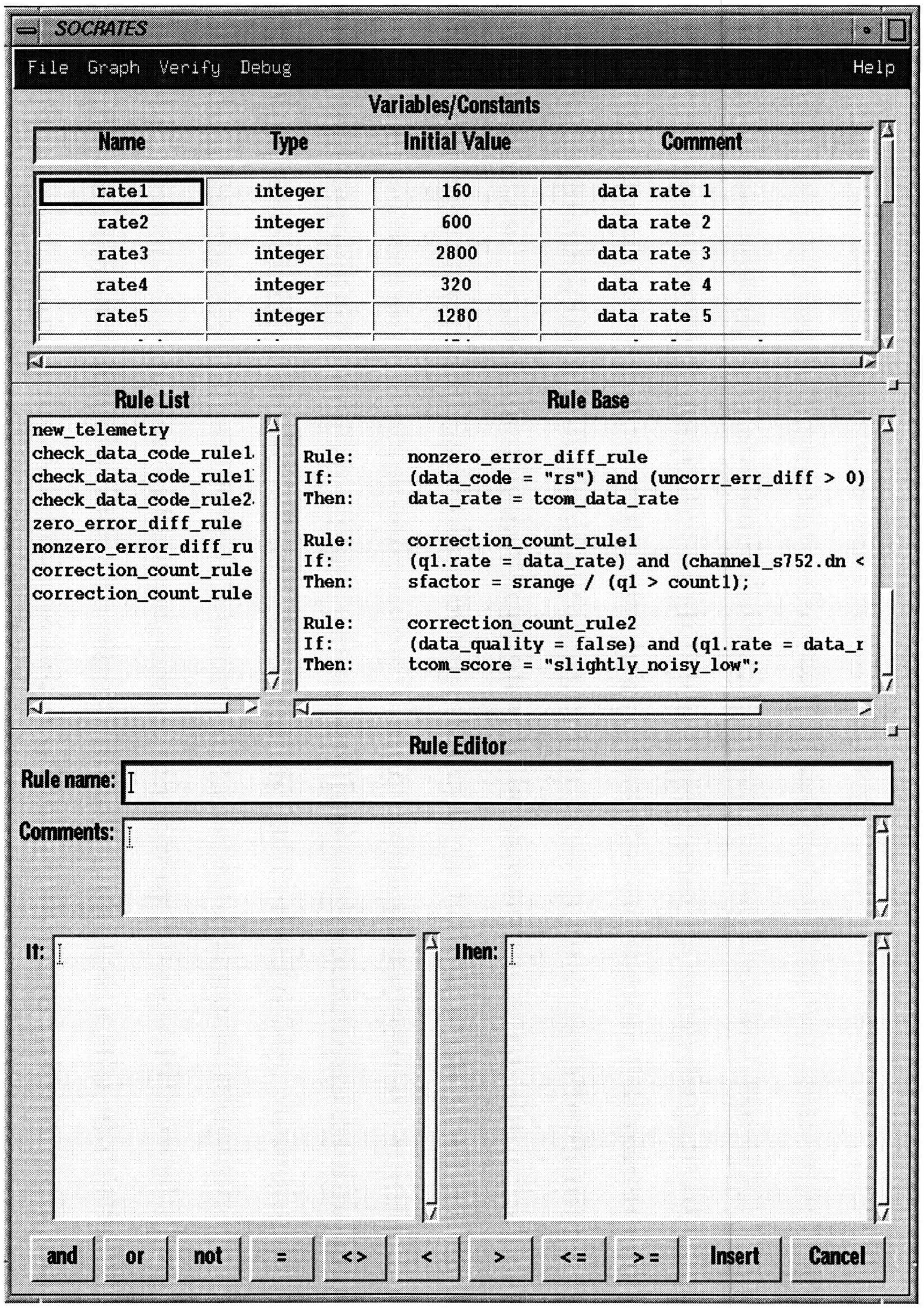
The feature of translating a rule base into an expert system language is not implemented in this project due to time constraint. It is a simple syntax translation that can be added once the desired shell language to convert into is known.

3.2 Graphical User Interface (GUI)

The graphical user-interface consists of a constant/variable window, a rule index column, a rule base window, and a rule editing window (see Figure 3.2). The constant/variable window is for declaration of variables, their type, and their initial values. The rule index column displays the names of all rules in the rule base for easy access. The rule base window displays complete rules inserted in the rule base. The rule-editor window is divided into a rule name field, a comments field, a IF field, and a THEN field. The tool only accepts a rule if it is complete, that is, if it has a rule name, if clauses, and then clauses.

The GUI is divided into three expandable sections: top for constant/variable window, middle for rule index and rule name, and bottom for rule editor window. The user can choose the size of each window by clicking on one of the two divider lines and move it up or down.

Each window and each field within that window has its own editing functions in a popup menu. The constant/variable window has insert row, delete row, copy and paste row, and delete cell. The rule index field has search (to find a particular rule in the rule base) and delete (for removing a rule from the rule base). All fields in the rule-editor window have copy, paste, and delete.



Name	Type	Initial Value	Comment
rate1	integer	160	data rate 1
rate2	integer	600	data rate 2
rate3	integer	2800	data rate 3
rate4	integer	320	data rate 4
rate5	integer	1280	data rate 5

```
new_telemetry
check_data_code_rule1
check_data_code_rule1
check_data_code_rule2
zero_error_diff_rule
nonzero_error_diff_ru
correction_count_rule
correction_count_rule
```

```
Rule:     nonzero_error_diff_rule
If:       (data_code = "rs") and (uncorr_err_diff > 0)
Then:     data_rate = tcom_data_rate

Rule:     correction_count_rule1
If:       (q1.rate = data_rate) and (channel_s752.dn <
Then:     sfactor = srange / (q1 > count1);

Rule:     correction_count_rule2
If:       (data_quality = false) and (q1.rate = data_r
Then:     tcom_score = "slightly_noisy_low";
```

Figure 3.2: Graphical User Interface for the Automated Verification Tool

The constant/variable window is in the format of a table with four columns: *Name*, *Type*, *Initial Value*, and *Comments*. The column *Name* allows input in any combination of characters as long as it is not a number. The input to column *Type* is validated as it is entered to ensure that it is one of the three allowable values: STRING, FLOAT, and INTEGER. The input to column *Initial Value* is validated against the value in its corresponding *Type* column to make sure they are consistent.

The rule name field and the rule base display field adjust automatically to guarantee that the most currently inserted rule is shown in the viewing window.

The rule-editor window performs simple syntax check and duplicate rule name check as a rule is inserted in the rule base. It also ensures that the entered rule is complete, in other words, it has a name, an IF statement, and a THEN statement. If a rule is unacceptable, a message window will popup explaining the reason why the rule is not inserted into the rule base.

3.3 Parser

3.3.1 Domain of the Parser

All inputs are entered in one of the two formats, either as variable declaration through the variable/constant window or as a simple rule through the rule-editor window. The inputs are parsed and stored in data structures after they are entered. Some basic checks, such as duplicated rule names, syntax errors, and illegal variable assignments are performed on a rule when it is parsed. If the rule is error-free, then it is added to the rule-base and to the data structure. A permissible input rule has to be in the form of IF-THEN statement, with a rule name, if clause, then clause, and an optional comments section. The parser also verifies that the input is in the form of standard first-order logic.

3.3.2 Lex and Yacc

The parser is programmed in Lex & Yacc. Lex & Yacc are tools often used for writing compilers and interpreters. Their specialty is looking for patterns in the inputs. It is concise, readable, and thus easy to edit and maintain. When user presses the *insert* button, a C program extracts the input rule from the GUI and passes the string to Yacc. Yacc in turn calls Lex to identify tokens (break the input string down into recognizable patterns). If the parser cannot break it down into recognizable patterns, it returns a syntax error to notify the user. Otherwise, it stores the string in a data structure according to pre-defined patterns either through the variable/constant window or the rule-editor window.

3.4 Verification

3.4.1 The Inference Engine

In a rule-based expert system, the set of known facts at an instant in time is called a *situation*. The inserted initial facts combined with the rules shift the rule base from one situation to another. The action is called rule inference. The possible inference ordering in the rule base is called the *inference engine*.

3.4.2 Verification Criteria

Verification is the static and dynamic analysis of the logic of knowledge to reveal potential errors (Preece, 1992). Since this research project is an expert system building tool, no knowledge is assumed beyond the entering rules. Thus, only three of the five principles can be tested: *redundancy, consistency,* and *connectivity. Redundancy* include isolated rules, duplicated rules, and subsumed rules. Isolated rules are the unusable rules in a rule base that is neither inferenced by any other rule nor inference to any other rule. Two rules are duplicated if they have identical clauses both in hypothesis and in conclusion. Rule A is subsumed by Rule B if the hypothesis of Rule B is a subset of the hypothesis of Rule A, and rules A and B have identical conclusions. *Consistency* implies there are no conflicting rules. Two rules in a rule base are conflicting if they fire under the same situations yet draw different conclusions. *Connectivity* principle tests for circularity and missing rules. A rule base has circularity if it contains some set of rules that an interminable loop could occur when the rules are inferenced. Missing rules only can be detected when the goal states of the expert system is known. Since this is a general application tool, not a specific expert system, the goal states of a particular system are unknown. Therefore, missing rules criteria cannot be verified automatically. However, at the time of building a specific system, a user has knowledge about the system beyond just the rules, such as the goal of the system and the domain of the system. Therefore, missing rule criteria can be evaluated by the user with the aid of the graphical rule analyzer.

3.4.3 Verification Algorithm

Christopher Landauer at The Aerospace Corporation suggested a clever way of modeling a rule base. The basic idea is to use incidence matrices to represent relationships among the rules. For example, a RV incidence matrix represents the rule-variable relationship in the rule base.

RV(i,j) = 1, if variable j is in rule i,

RV(i,j) = 0, otherwise

Similarly, a RC matrix represents the rule-clause relationship of a rule base, and a CV matrix represents the clause-variable relationship of a rule base.

The verification algorithms for this tool is based on Landauer's idea of matrix representation. Four matrices were built: CC matrix, CH matrix, RVc matrix, and RVh matrix. CC matrix indicates the presence of clauses in the conclusion of all rules. It has rules for the row entries and clauses for the column entries. For example, CC(i,j) indicates whether or not clause j is present in the conclusion statements of rule i. Similarly, row i of CC matrix indicates all clauses in the conclusion of rule i. CH matrix indicates the presence of clauses in the hypothesis of rules. It also has rules for the row entries and clauses for the column entries. CH(i,j) indicates whether or not clause j is present in the hypothesis of rule i. RVc matrix and RVh matrix are the rule-variable matrices representing conclusion statements and hypothesis statements respectively. They both have rules for the row entries and variables for the column entries. RVc(i,j) indicates whether or not variable j is present in the conclusion statements of rule i. Similarly, RVh(i,j) indicates whether or not variable j is present in the hypothesis statements of rule i.

Let us define an inference R graph such that

$\text{inferR} = (\text{CC})(\text{CH}^{\text{tr}})$.

InferR is a mxk matrix with rules indexing both rows and columns. A non-zero entry in the inferR matrix implies an inference between two rules in the rule base.

$(\text{CC})(\text{CH}^{\text{tr}})\ (i, j) >= 1$, if rule i inferences rule j.

The algorithm works as follow: entry (i,j) in inferR matrix is the product of row i of CC matrix and column j of CH^{tr} matrix. The non-zero entries in row i of CC matrix represent all clauses in conclusion of rule i. Similarly, the non-zero entries in column j of CH^{tr} matrix, which is same as row j of CH matrix, represent all clauses in hypothesis of rule j. Thus, entry (i,j) in inferR matrix is only non-zero when row i of CC matrix and column j of CH^{tr} matrix have at least one clause in common.

3.4.4 Verification Implementation

Using the inference matrices, the criteria conflict rules, circular chaining, subsumed rules, and isolated rules were tested. Two rules are conflicting if they have identical hypothesis statements (same number of clauses in the hypothesis statements and each pair of clauses are equal) and different conclusion statements. Rule conflicts are checked using CH matrix and CC matrix. Circular chaining occurs when a rule node is a child node of its own children. In terms of graph, circular chain exists if there is a path traveling from a node to itself with no backtracking. Circular chaining is examined using depth first search. As it traverse through a node, it checks to see if the node has been visited. If the node is not visited, mark it visited; if the node is visited, it is a circular chain. Subsumed rule is a subset of redundancy. One rule is subsumed by another if two rules have identical hypothesis statements but the conclusion statements of the first rule is a subset of the conclusion statement of the second rule. An isolated rule may or may not be redundant. It is not redundant if it represents an inference of one level deep. On the other hand, it is redundant if it represents a rule drawing conclusion clauses that are not used anywhere in the rule base and the conclusion clauses are not the goal states. Therefore, an isolated rule could or could not be useful. The tool identifies the isolated rules for the user to decide what to do with it.

3.5 Automated Graphical Analyzer

3.5.1 Visual Representation of Relations Among the Rules

To create a visual representation of the rule base structure, the rules are displayed graphically in directed graph format. Each rule is represented by a node, and inference relationship among the rules is represented by a line with an arrow pointing in the direction of inference. An inference matrix of the rules can be easily converted to a graph of the above mentioned format. By making a vertex for each row and column, and an edge for each nonzero entry in the inference matrix, a visual representation of the rule in the rule base can be created. Verifications such as connectivity and distribution can be spotted at a glance on the visual rule analyzer.

3.5.2 Automated Graph Layout Algorithm

The difficulty of an automated graph representation lies in arranging the layout for easy visual interpretation. Basically, given a particular rule base with known relationships among the rules, how can the vertices be automatically positioned so that there are minimum edge crossings? This is the most crucial problem as far as readability of a graph is concerned. There are extensive literature on the research of graph layout algorithms. A simple algorithm called the Relative Degree Algorithm is implemented in this research project. The algorithm was developed by Marie-Jose Carpano and implemented in a project called GTIVX, a computer aided decision analysis tool which facilitate interactive structural systems analysis (Carpano, 1980). The idea behind this algorithm is simple -- the more vertical the edges are, the fewer the crossings. This algorithm assumes that all nodes are assigned a level number. It arranges the order of nodes on a single level by examining their relationship with the nodes one level higher than them and one level lower than them.

Take the simple case of a two-level cycle: free graph where the flow is unidirectional (top to bottom, for example), if all children nodes on the lower level is right below their parent nodes on the top level, there would be no edge crossings. Hence, for every permutation on node arrangements of level i, the method consists defining a permutation of nodes for level (i+1) such that the vertices are placed as directly beneath their parent nodes on level i as possible. This concept can be expressed concisely in the matrix formalization.

Let

x_i be a node on level 1 of a two-level graph.

y_i be a node on level 2 of the two-level graph.

A_{xy} be a m x k boolean matrix indicating the presence of edges from x_i to y_j.

Then,

$A_{xy}(i, j) = 1$, if there is an edge from x_i to y_j.

$A_{xy}(i,j) = 0$, if there is no edge from x_i to y_j.

Let

A_x be the matrix obtained by dividing each coefficients a_{ij} of A_{xy} by $\Sigma_k a_{ik}$ and transposing it.

A_y be the matrix derived from A_{xy} by dividing each of its coefficients a_{ij} by $\Sigma_k a_{ik}$.

$X = (x_1, x_2, ..., x_k)$ be the vector of the x-coordinate of the level 1 nodes.

$Y = (y_1, y_2, ..., y_k)$ be the vector of the x-coordinate of level 2 nodes.

Then,

$$Y^{(p)} = A_x X^{(p)}$$

and

$$X^{(p+1)} = A_y Y^{(p)}$$

The p^th iteration of the x-coordinates for level 1 nodes and level 2 nodes can be calculated using the above equations.

3.5.3 Adopting the Algorithm for Verification Tool

The *Relative Degree* algorithm mentioned above has three assumptions which does not apply in this Verification Tool.

1. The algorithm assumes that the graph is cycle free.

2. The algorithm assumes an ordered hierarchy graph that flows unidirectionally.

3. The algorithm only applies to a two-level graph.

The input data is filtered to remove all circular paths. Then, each node is assigned a rank number to decide which level of the graph it belongs to. Using depth first search, the first node visited (root node) is in level 0, the next node visited is in level 1, the (i+1)th node visited is in level (i+1). When a node is visited more than once by various paths, the highest level assignment becomes its new rank. The inference matrix and the ranking array are then used to calculate the actual positioning of the nodes in the tcl/tk graphical analyzer window.

The *Relative Degree* algorithm arranges nodes one level at a time for minimum edge crossing. The calculated positions of nodes are used to distinguish the relative position of nodes of the same level. At each level, the nodes are displayed equidistant apart, ordered according to their calculated positions. An additional feature is added which checks for overlapping node replacements. If two nodes have the same calculated position, they will overlap when drawn in the graph analyzer window. When this situation is detected, one of the nodes is moved away so that all nodes have unique positions.

3.5.4 Generating Graphs in Tcl/Tk

Tcl/Tk is a programming language developed by John Ousterhout at the University of California, Berkeley. The syntax is simple and thus easy to use, yet it has very useful graphical facilities. TCL is the basic programming language, and TK is a ToolKit of widgets similar to those of Xlib and Motif. Unlike Xlib or Motif, Tcl/Tk is a self contained language that can be executed on its own. But if users wish to use the widgets with other programming languages such as C or C++, the Tcl/Tk system can be configured to work with other programming languages. The main advantage of Tcl/Tk is its simple syntax. A useful application can be developed quickly and concisely in a short period of time, even if the developer is not fluent in Tcl/Tk.

The Tcl/Tk window has a simple manu bar with four submenus: File, Graph, View, and Help. It can print the graph to a file, graph either rules or clauses and their relations, and has an option to show rule names and clause names next to the nodes. The Tcl/Tk window is divided into two sections: a graph window and a display window. The graph window shows a visual graph of rules in a rule base. The display window displays the content of rules/clauses when a user clicks on a node in the graph window. Both windows have horizontal and vertical scroll bars for viewing purpose.

3.5.5 Calculating the Intersection Position

In Tcl/Tk, a line is drawn by specifying the X and Y coordinates of the two end points. Each node, representing a rule, is a solid circle of radius R. Since each node has a surface area greater than zero, it is necessary to calculate where on the circumference of a node does a connecting line intersect. The intersections can be calculated from the relative positions of the two end nodes.

For example, let *node 1* be located at (x1, y1) with radius R, and *node 2* be located at (x2, y2) with radius R (see Figure 3.3). The distance from center of *node 1* to center of *node 2* is calculated by

$$D = \sqrt{(x1-x2)^2 + (y1-y2)^2}.$$

The coordinates on the circumference of *node 1* to *node 2* that specifies their connecting edge are (x1+dx, y1+dy) and (x2-dx, y2-dy) respectively, where

$$dx = \frac{(x2-x1)}{D} \times R$$

and

$$dy = \frac{(y2-y1)}{D} \times R$$

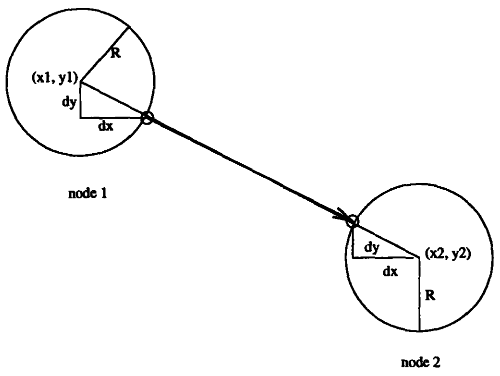


Figure 3.3: Calculating the intersection positions between the nodes and their connecting edge.

Chapter 4

Experimental Evaluation

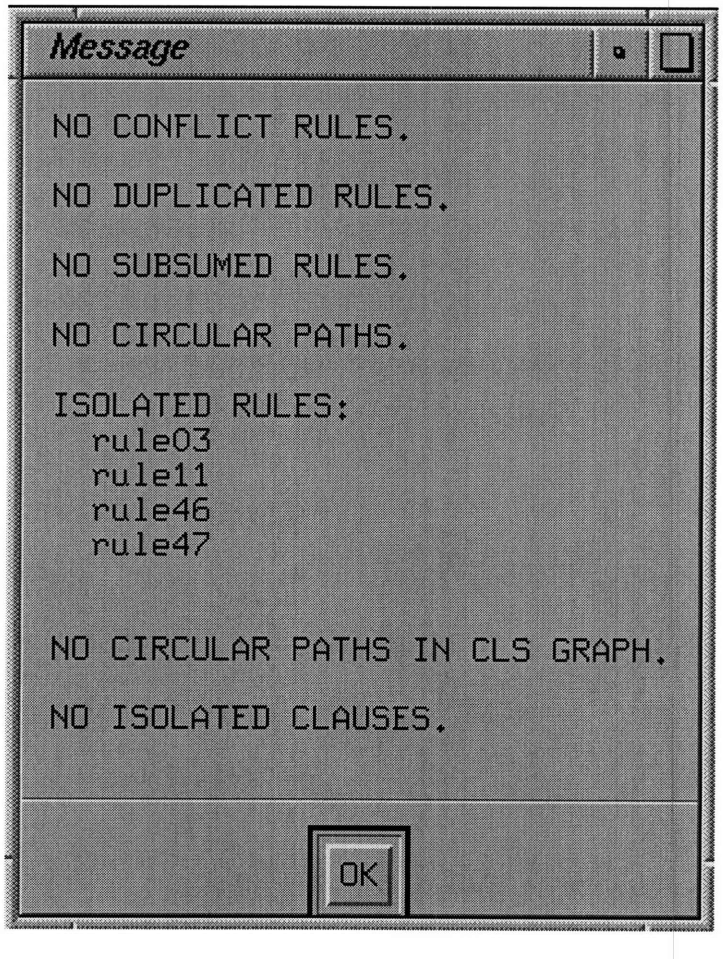
4.1 Guidelines for Evaluation

To evaluate the performance of this tool, it is tested with several rule-based expert systems. Letting a user with no expert system building experience put together a new expert system on the verification tool would be the ideal way to perform a thorough evaluation of the tool. Not can it only evaluate for technical performance, but also for human factor issues like user-friendliness and usefulness of the rule base analyzer. Due to time and resource constraints, the tool was only tested on two existing rule-based systems. The test case systems were already built and put to use. Thus, there are no fatal errors in the system. However, there are several isolated and maybe redundant rules. In addition, some rules were altered just like a user would when updating an expert system. Some new errors were induced as a result. The altered rule bases are again verified by the automated tool. The errors detected by this automated verification tool will be summarize. The errors include duplicated names, duplicated rules with different names, potential circular inference paths in the rule base, and subsumed rules, and conflict rules, as well as errors that the tool missed. In addition, the visual rule analyzer will be evaluated for visual clarity and its usefulness in providing insights into the rule bases.

4.2 Insurance Plan Advisor

The Insurance Plan Advisor is a small rule-based expert system written in Shrubs. It consists of a total of thirty six rules. It decides on a type of investment for the user based on conditions such as whether or not the user has children going to college, the user's stage in life, the user's savings and properties.

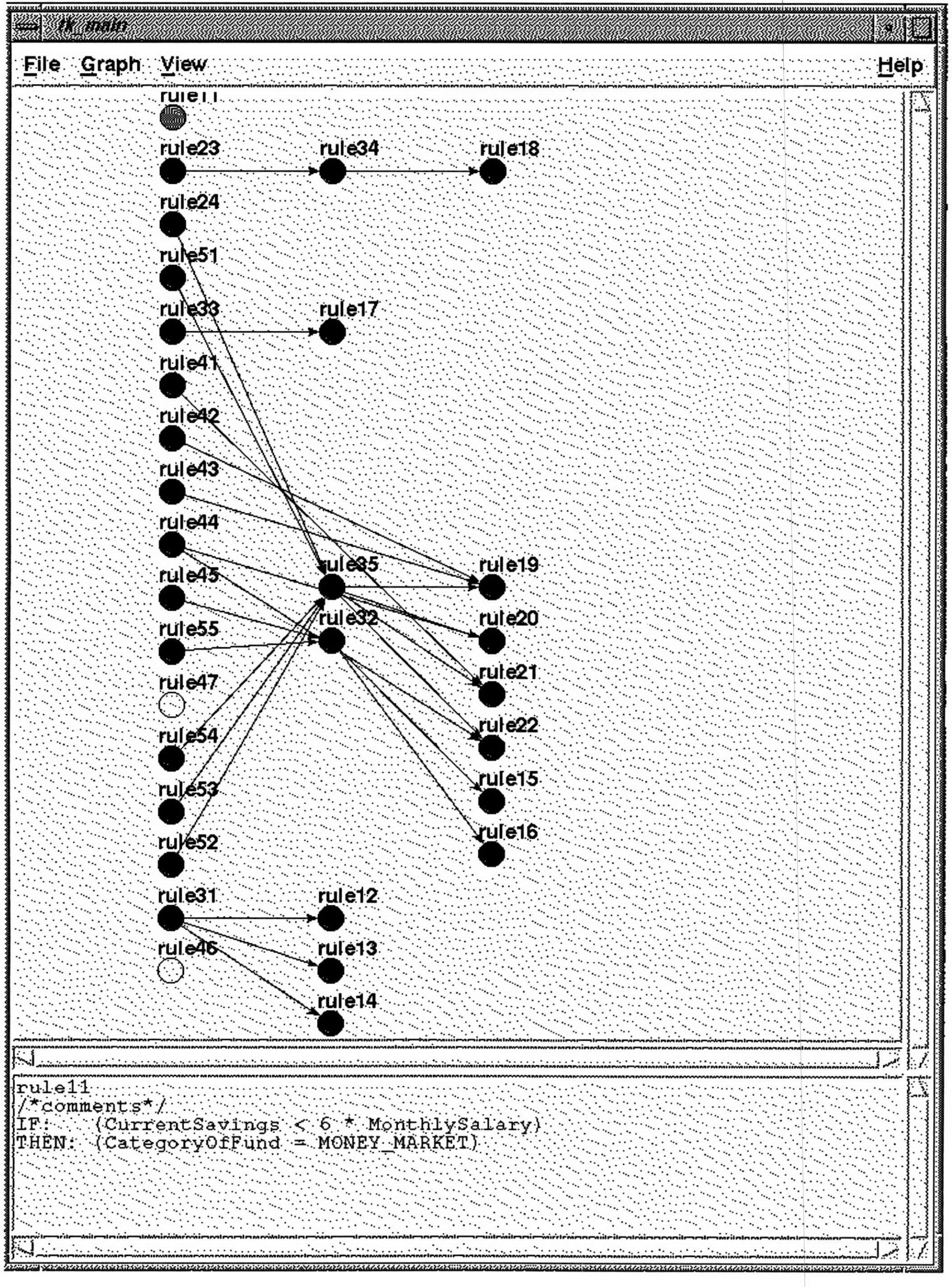
The thirty six rules in the rule base were converted to IF-THEN statement rules, and then tested using the automated verification tool. The verification tool concluded that there are no conflicting rules, no duplicated rules, no subsumed rules, and no circular paths. It detected four isolated rules: rule03, rule11, rule46, and rule47. As for clauses, there are no circular paths in thc clause graph and no isolated rules.



```
NO CONFLICT RULES.

NO DUPLICATED RULES.

NO SUBSUMED RULES.

NO CIRCULAR PATHS.

ISOLATED RULES:
  rule03
  rule11
  rule46
  rule47


NO CIRCULAR PATHS IN CLS GRAPH.

NO ISOLATED CLAUSES.
```

Figure 4.1: Pop up message window displaying verification result for Insurance Plan Advisor.

The graphical analyzer visually displays the inference relationships among the rules. The problem-free rules are represented by blue color nodes. The inefficient or unnecessary rules (such as redundant rules, isolated rules, duplicated rules) are represented by yellow



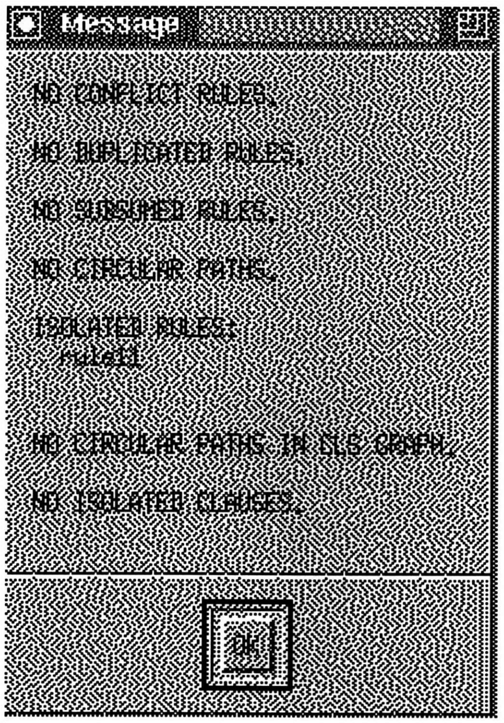
```
rule11
/*comments*/
IF:    (CurrentSavings < 6 * MonthlySalary)
THEN:  (CategoryOfFund = MONEY_MARKET)
```

Figure 4.2: Graphical Analyzer for the rule base of the Insurance Plan Advisor.

colored nodes. The critical rules that result in problems (such as circular chaining and conflict rules) are represented by red colored nodes. The currently selected node is colored in green.

A user can perform a significant amount of analysis just by looking at the graphical analyzer. For example, the user can analyze the Insurance Plan Advisor expert system by looking at Figure 4.1. The goal of this rule-based system is to determine CategoryOfFund to invest in, given some initial facts. The value instantiated to this variable is the only one a user is interested in. Therefore, any rule or clause that does not directly or indirectly lead to assigning value to this variable is redundant. They should be eliminated to keep the size of rule base minimal. Among the isolated rules which are highlighted in yellow, rule03 draws the conclusion (BasicInsuranceCoverage = ADEQUATE), which is a dangling leaf node. Therefore, rule03 can be deleted from the rule base without affecting the behavior of the system. Similarly, rule47 and rule46 draw the dangling conclusion (ChildrenHeaded-ForCollege = FALSE). These two rules are also deleted from the rule base without altering the system behavior. In addition, notice that rule34 is the intermediate node from rule23 to rule18. It can be eliminated by the direct inference of rule18 to rule23.

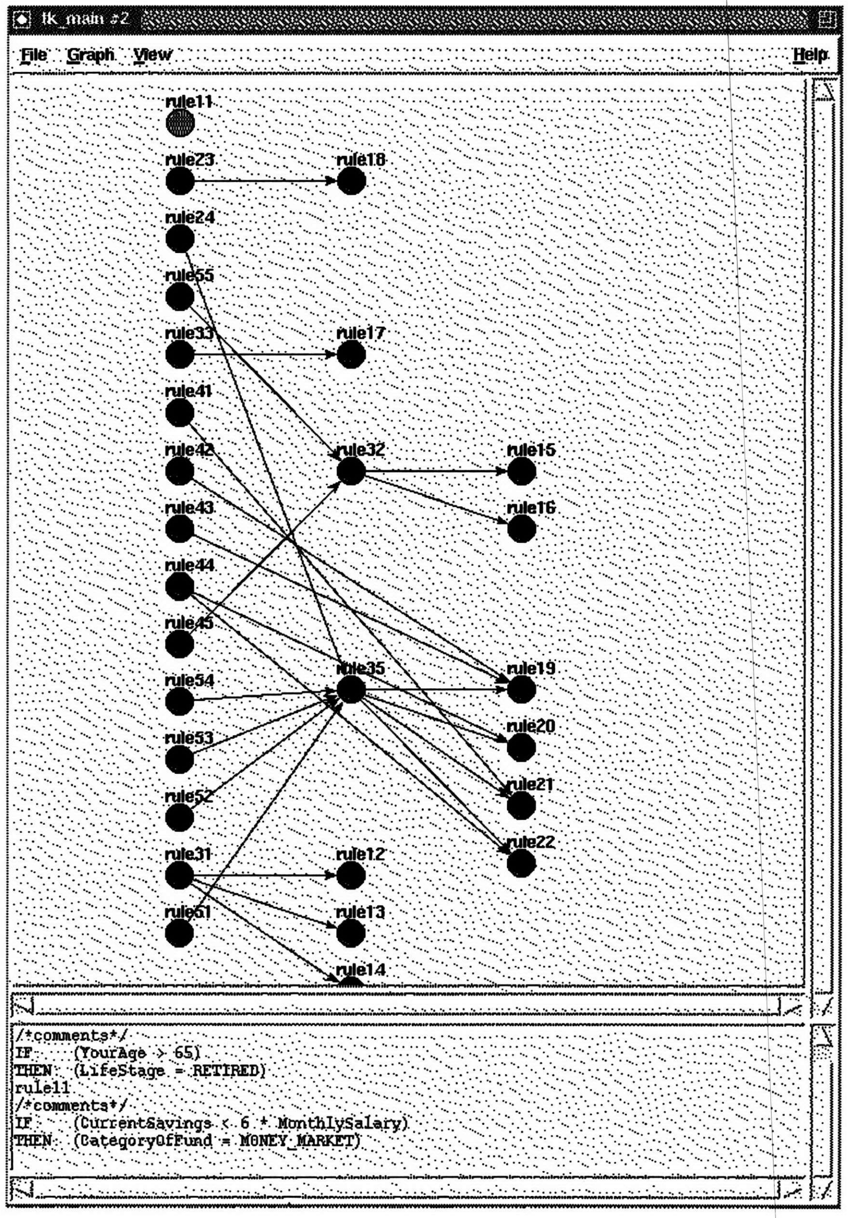
These changes were incorporated to the Insurance Plan Advisor system. Verification was again performed on the modified rule base of Insurance Plan Advisor system. The result shows significant improvements.



```
NO CONFLICT RULES.

NO DUPLICATED RULES.

NO SUBSUMED RULES.

NO CIRCULAR PATHS.

ISOLATED RULES:
  rule11


NO CIRCULAR PATHS IN CLS GRAPH.

NO ISOLATED CLAUSES.
```

Figure 4.3: Pop up message window displaying the verification result of the modified Insurance Plan Advisor system.

The modified rule base contains no error (see Figure 4.3). The only highlighted node on the analyzer graph is rule11, an isolated node. Though rule11 is an isolated node, its conclusion clause is assigning a value to the variable CategoryOfFund. Therefore, it is a relevant rule to the system and should not be modified.

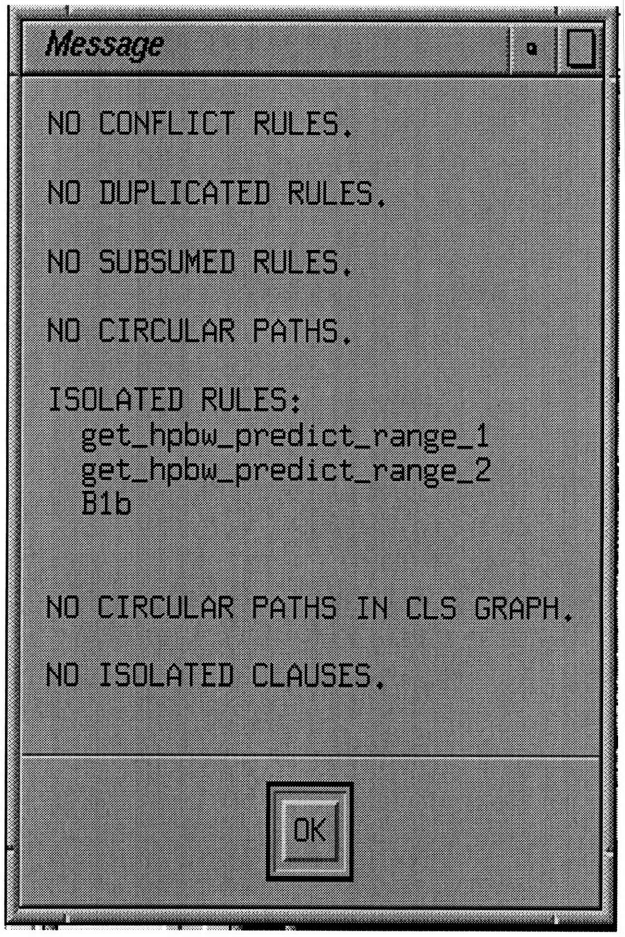


```
/*comments*/
IF    (YourAge > 65)
THEN  (LifeStage = RETIRED)
rule11
/*comments*/
IF    (CurrentSavings < 6 * MonthlySalary)
THEN  (CategoryOfFund = MONEY_MARKET)
```

Figure 4.4: Graphical Analyzer for the rule base of the modified Insurance Plan Advisor system.

4.3 FDIR Boresight

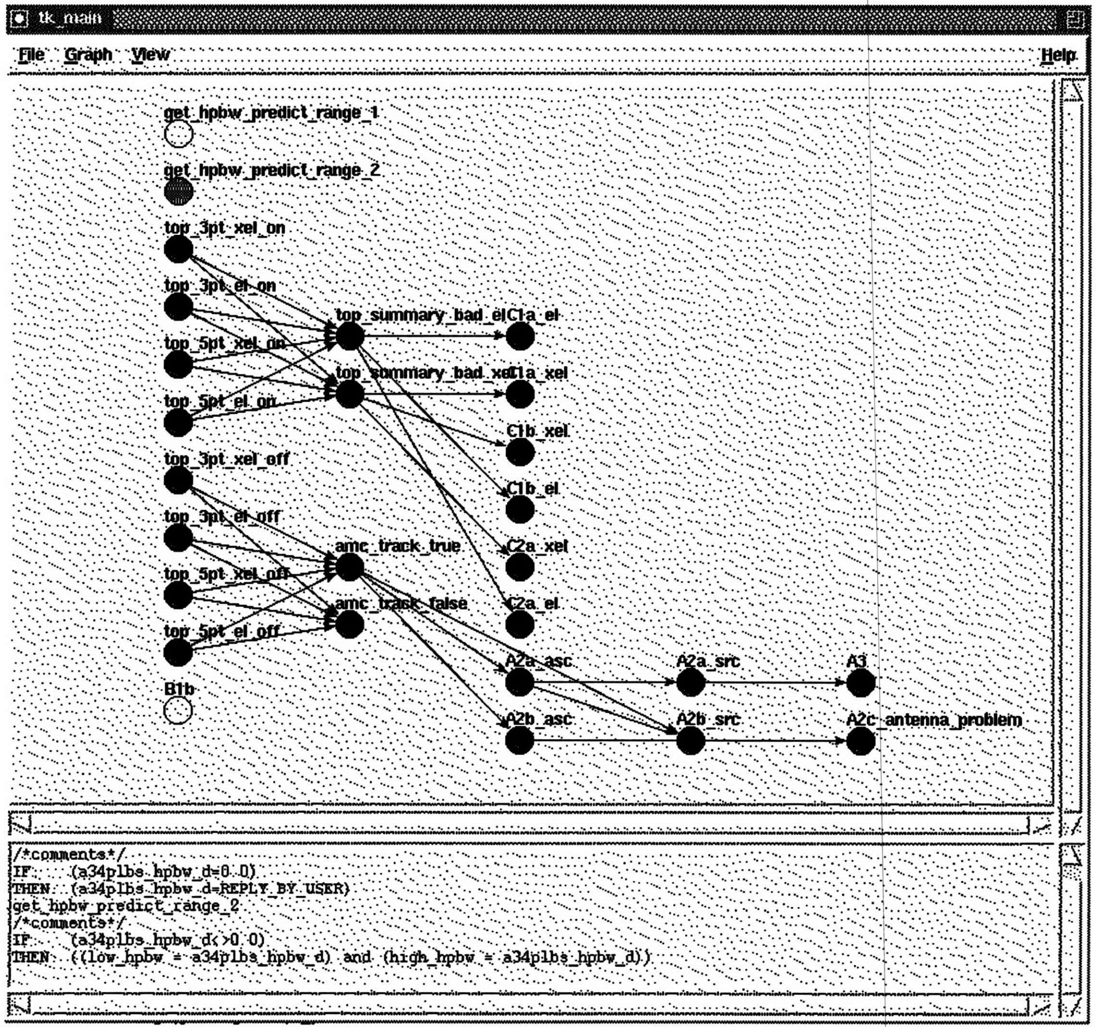
FDIR boresight is a rule-based system for anomaly detection during the boresight process of the Deep Space Network. It was developed at the Jet Propulsion Laboratory in the expert system shell language CLIPS. The rule base of FDIR was translated into simple IF-THEN statement rules so that it is in the syntax of the automated verification tool. Some rules cannot be directly translated and thus need to be modified. For example, for all the rules that prompt user for variables' initial values, a typical initial value was assigned to each. The rules that simply convert input data into desired format, specifically from integer to float, are ignored. The original rule base written in CLIPS has thirty two rules. The equivalent rule base in the automated verification tool has twenty seven rules.



```
NO CONFLICT RULES.

NO DUPLICATED RULES.

NO SUBSUMED RULES.

NO CIRCULAR PATHS.

ISOLATED RULES:
  get_hpbw_predict_range_1
  get_hpbw_predict_range_2
  B1b


NO CIRCULAR PATHS IN CLS GRAPH.

NO ISOLATED CLAUSES.
```

Figure 4.5: Pop up message window displaying verification result for FDIR Boresight.

FDIR Boresight rule base has no conflict rules, no duplicated rules, no subsumed rules, and no circular paths. It merely has three isolated rules: get_hpbw_predict_range_1, get_hpbw_predict_range_2, and B1b.



```
/*comments*/
IF    (a34plbs_hpbw_d=0.0)
THEN  (a34plbs_hpbw_d=REPLY_BY_USER)
get_hpbw_predict_range_2
/*comments*/
IF    (a34plbs_hpbw_d<>0.0)
THEN  ((low_hpbw = a34plbs_hpbw_d) and (high_hpbw = a34plbs_hpbw_d))
```

Figure 4.6: Graph analyzer for the rule base of FDIR Boresight system.

The automatic graph layout algorithm works well for Boresight rule-based system. The inference relationships are clear and edge-crossings are minimal.

Chapter 5

Conclusion

5.1 Successes

The automated verification tool for rule-based expert systems has a lot of potential. The graphical analyzer simplifies the task of verification for expert systems. It is useful for both non-programmers who know little of verification process and software engineers who are experience programmers. The visual analyzer clearly shows the inference interaction among the rules. All problem rules are represented by highlighted nodes. It is extremely helpful for examining the rules. The user can effortlessly trace through the inference paths by following the directed edges through the graph.

Implementing a meta-language front-end to the tool has many benefits. Since the rules are entered in simple IF-THEN statements, users need not learn any expert system shell language, the traditional front-end development environment. Also, verification is done in the meta-knowledge, independent of any expert system shell language. This tool can easily be adopted to work with any shell language, and therefore, is applicable to many rule-based systems developed in various shell languages.

5.2 Failures

The automated graph layout algorithm is inefficient. The analysis graphs for the sample rule-based systems contained a substantial amount of edge-crossing. As the amount of nodes increased, especially in the example of clause graphs where each node represented a clause, the analyzer graph became impractical. When there are large amount of nodes in a graph, the fixed screen size constraints to view only part of an analyzer. For a large rule base, it is possible that two adjacent nodes with a connecting edge linking them together cannot be viewed in the same visible window. If this research is continued in the future, a better algorithm should be implemented instead.

Chapter 6

Future Work

6.1 Feature Improvements

All modules and features in this automated verification tool were developed specifically for this project. No existing customized modules or other commercially available components were used. Due to time constraint, the application neglected several fundamental issues in application development such as ease of use, user-friendliness, and easy to follow instructions, which are critical for real applications. The graphical user interface is an attempt to make the application easy to use, but the operation features are still primitive. Improvements and additional features for the graphical user interface are needed to make the tool more user-friendly. Though a user-friendly graphical interface is not critical to the performance of the verification tool, it is an essential component to a successful package of verification tool.

Much more work is needed in searching for a better automatic graph layout algorithm. The visual analyzer displaying inference relationship among rules in a rule base and highlighting rules of potential problems is the prominent function of this verification tool. The analyzer is useless without a good graph layout algorithm to display the relations among the rules clearly. Thus, a successful automatic graph layout program is critical.

More features can be implemented in the graphical analyzer to aid in analysis of a rule base. For example, one feature could be highlighting the inference path a rule base traces through for a given set of initial facts. This will enable users not only to verify for rule base correctness, checking to see if rules are inferenced correctly, but also to validate the knowledge base, making sure the behavior of the system is what the experts intended.

References

[1] Bellman, K.L. 1990. "The Modelling Issues Inherent in Testing and Evaluating Knowledge Based Systems." *Expert Systems With Applications.* 1:199-215.

[2] Carpano, M.J. 1980. "Automatic Display of Hierarchized Graphs for Computer Aided Decision Analysis." *IEEE Transactions on Systems, Man, and Cybernetics.* 10(11):705-715.

[3] Gansner, E.R., Koutsofios, E., North, S.C., Vo, K.P. "A Technique for Drawing Directed Graphs." IEEE Transactions on Software Engineering. 19(3): 214-230.

[4] Ginsberg, A. 1988. "Knowledge-base Reduction: A New Approach to Checking Knowledge-bases for Inconsistency and Redundancy." *Proceeding 7th National Conference on Artificial Intelligence.* 2:585-589.

[5] Jacob, R.J.K., and Froscher, J.N. 1990. "Software Engineering Methodology for Rule-Based Systems." *IEEE Transactions on Knowledge and Data Engineering.* 2(2):332A.

[6] Landauer, C. 1990. "Correctness Principles for Rule-based Expert Systems." *Expert Systems with Applications.* 1:291-316.

[7] Levine, J.R., Mason, T., Brown, D. Lex and Yacc. O'Reilly & Associates, SEbastopol, ca:1992.

[8] Messinger, E.B., Rowe, L.A. and Henry, R.R. 1991. "A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs." *IEEE Transactions on Systems, Man, and Cybernetics.* 21(1):1-11.

[9] Nguyen, T.A., Parkins, W.A., Laffey, T.J., and Pecora, D. 1985. "Checking An Expert System Knowledge-base for Consistency and Completeness." *Proceeding 9th International Joint Conference on Artificial Intelligence.* 1:375-378.

[10] Ousterhout, John. TCL and the TK Toolkit. Addison-Wesley, 1993.

[11] Preece, A.D., Shinghal, R., and Batarekh, A. 1992. "Principles and Practice in Verifying Rule-based Systems." *The Knowledge Engineering Review.* 7(2):115-141.

[12] Suwa, M., Scott, A.C., and Shortliffe, E.H. 1982. "An Approach to Verifying Completeness and Consistency in a Rule-based Expert System." *AI Magazine.* 3(4):16-21.

[13] Traylor, B., Schwuttke, U., and Quan, A. 1994. "A Tool for Automatic Verification of Real-Time Expert Systems." *AAA-94 Workshop for Validation and Verification of Knowledge-based Systems.* 1:79-83.

[14] Wilson, R.J. 1972. Introduction to Graph Theory. London: Longman.

ACKNOWLEDGEMENT

I would like to thank my group supervisor at JPL, Ursula Schwuttke, for giving me the opportunity to work on this project and for her guidance through my internship there. I also want to extend my sincere thanks to Bonnie Traylor, my mentor for this project. Thank you for your patience, your suggestions, and your company. Many thanks to Diana Hamilton and the whole AEG gang for making my internship a fun learning experience.

I am very grateful to my thesis advisor, Kenneth Haase, for his input and suggestions. He guided me through my research project, from providing me background resources to proof-read my thesis write-up.

I especially want to thank my parents for their unfailing support and encouragement. You sacrifice so much so that I can pursue my dreams. Thank you for everything.

Many thanks to my friends and CCF for making MIT a memorable experience. Last but not least, a big thanks to Him who made all these possible.