

THE UNIVERSITY OF CALGARY

A FAST SEARCHING ALGORITHM FOR EXPERT SYSTEMS IN THE  
POWER AREA

by

Tung Pui Hui

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL ENGINEERING

CALGARY, ALBERTA

MAY 1990

© Tung Pui Hui 1990



National Library  
of Canada

Bibliothèque nationale  
du Canada

Canadian Theses Service    Service des thèses canadiennes

Ottawa, Canada  
K1A 0N4

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

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

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

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

ISBN 0-315-61964-3

Canada

**THE UNIVERSITY OF CALGARY**  
**FACULTY OF GRADUATE STUDIES**

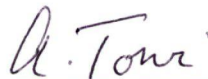
The undersigned certify that they have read, and recommended to the Faculty of Graduate Studies for acceptance, a thesis entitled, "*A Fast Searching Algorithm for Expert Systems in the Power Area*", submitted by Tung Pui Hui in partial fulfillment of the requirements for the degree of Master of Science.



Supervisor - Dr. G. S. Hope  
Department of Electrical Engineering



Dr. O. P. Malik  
Department of Electrical Engineering



Prof. A. Torvi  
Department of Mechanical Engineering

Date: May 14, 1990.

## ABSTRACT

Energy Management System (EMS) is a computerized supervision and control center. It aids operating personnel in controlling and coordinating the operations of different parts of the power system. Expert Systems (ESs) have gradually become a part of EMSs. An important part of an ES is the inference engine, which has a searching algorithm. The searching algorithms currently used in ESs in the power area are slow.

This thesis analyzes the algorithms and outlines criteria for a fast searching algorithm. A fast algorithm is proposed and implemented. A windowing interface is also implemented to test the proposed algorithm.

The algorithm has three processes:

- (1) The dissociation and substitution process breaks down all production rules into simple ones.
- (2) The transformation process transforms the simple rules into a search tree.
- (3) The search process conducts search using facts.

The algorithm can readily be used in power system applications. This is shown by the illustrations involving the alarm-handler/fault-diagnostician ES. Search speed comparisons show that the proposed searching algorithm is much faster than Prolog/V's searching algorithm, which is the most widely used searching algorithm in the power area.

## ACKNOWLEDGEMENTS

The author wishes to express his sincere gratitude to Dr. G. S. Hope for his guidance throughout the course of the project and for his advice and corrections in preparing the manuscript.

Special appreciation goes to Dr. O. P. Malik and other members in the Power Research Group for their constructive criticisms and valuable discussions. Special thanks to Dr. S. Minasiewicz (deceased) who provided assistance and suggestions in developing the algorithm.

Finally, the financial support given by the Department of Electrical Engineering is thankfully acknowledged.

## TABLE OF CONTENTS

TABLE OF CONTENTS .....	v
LIST OF TABLES .....	viii
LIST OF FIGURES .....	ix
LIST OF NOMENCLATURE .....	xi
1. INTRODUCTION .....	1
1.1 ES Applications .....	3
1.2 Object-Oriented Programming And Possible Applications .....	4
1.3 Thesis Objectives .....	5
1.4 Thesis Organization .....	6
2. EMS AND OOP .....	9
2.1 An Introduction To EMS .....	9
2.2 A Proposed EMS .....	10
2.2.1 Features .....	11
2.2.2 Structure .....	12
2.2.3 Tasks .....	14
2.3 Problems Related To Prototyping .....	19
2.4 OOP As A Solution .....	20
2.5 Drawbacks Of OOP .....	22
3. EXPERT SYSTEM SEARCH TECHNIQUES .....	23
3.1 Search Techniques .....	23
3.1.1 Search strategies .....	24

3.1.2 Search methods .....	25
3.1.3 The "best" method or strategy .....	28
3.2 Search Techniques In Power Systems .....	28
3.2.1 Pure forward-chaining .....	29
3.2.2 Backward-chaining depth-first .....	29
3.2.3 Indexed forward-chaining .....	30
3.3 An Ideal Search Technique .....	32
3.3.1 Pros and cons .....	32
3.3.2 Criteria for a suitable technique .....	34
3.4 A Proposed Fast Algorithm .....	36
3.4.1 An important concept: key condition .....	36
3.4.2 The proposed algorithm .....	37
3.5 Merits Of The Proposed Search Algorithm .....	42
3.5.1 Fast search .....	42
3.5.2 Efficiency .....	43
3.5.3 Versatile .....	43
3.6 Disadvantages Of The Proposed Algorithm .....	44
3.6.1 Extra memory space .....	44
3.6.2 Extra time for constructing tree .....	44
4. SMALLTALK AND IMPLEMENTATION OF THE ALGORITHM .....	46
4.1 Smalltalk .....	46
4.1.1 Theoretical foundation .....	46
4.1.2 Features and properties .....	48
4.2 Algorithm Implementation .....	49

4.2.1 Classes created .....	49
4.2.2 Implementation .....	53
4.2.3 Interface .....	56
5. ALGORITHM ILLUSTRATIONS .....	68
5.1 Background .....	68
5.1.1 Single fault examples .....	71
5.1.2 Multiple fault examples .....	79
5.2 Comparison .....	87
5.2.1 Illustration .....	91
6. CONCLUSIONS .....	94
6.1 General Conclusions .....	94
6.2 Future Work .....	96
REFERENCES .....	98
APPENDIX A: The new classes and their implementations .....	103
APPENDIX B: Listing of class protocol .....	118
APPENDIX C: Listing of the rules used in the 5-substation illustrations .....	166
APPENDIX D: Listing of the rules used in speed comparison .....	183



## LIST OF TABLES

Table 5.1 Search times using the proposed algorithm and Prolog/V .....	93
--	----

## LIST OF FIGURES

Fig. 3.1 A simple rule and its corresponding discrimination network .....	31
Fig. 4.1 The Model System and the Referent System .....	47
Fig. 4.2 The System window .....	57
Fig. 4.3 The System window with the 3 choices in the top right subpane .....	58
Fig. 4.4 The Edit window with menus .....	59
Fig. 4.5 An instance of RuleBases is selected in the Edit window .....	61
Fig. 4.6 An instance of SetOfFact is selected in the Edit window .....	62
Fig. 4.7 An instance of StringRules is selected in the Edit window .....	63
Fig. 4.8 An instance of Table is selected in the Edit window .....	64
Fig. 4.9 The Search window with menus .....	65
Fig. 4.10 The Transform window with menus .....	67
Fig. 5.1 The substation schematic .....	69
Fig. 5.2 The substation schematic for the first example .....	72
Fig. 5.3 Example 1 containing alarms for the first example .....	73
Fig. 5.4 Solution paths and explanation for the first example .....	75
Fig. 5.5 The substation schematic for the second example .....	76
Fig. 5.6 Example 2 containing alarms for the second example .....	77
Fig. 5.7 Solution paths and explanation for the second example .....	78
Fig. 5.8 The reprint of the full explanation for the second example .....	80
Fig. 5.9 The substation schematic for the third example .....	81
Fig. 5.10 Example 3 containing alarms for the third example .....	82

Fig. 5.11 Solution paths and explanation for the third example .....	83
Fig. 5.12 The reprint of the full explanation for the third example .....	84
Fig. 5.13 The substation schematic for the fourth example .....	85
Fig. 5.14 Example 4 containing alarms for the fourth example .....	86
Fig. 5.15 Solution paths and explanation for the fourth example .....	88
Fig. 5.16 The reprint of the full result for the fourth example .....	89
Fig. 5.17 The reprint of the full explanation for the fourth example .....	90

## LIST OF NOMENCLATURE

abstraction	a representation of ideas or concepts in the real world.
AI	Artificial Intelligence. An area of research and application aims at teaching computers and machines to imitate human activities.
application domain	the area where the computer program applies.
A*	a search method which visits only the node with the best values which are assigned by two functions.
"B"	a prefix denoting bus-bar. Used only in chapter 5.
backtrack	goes back to the previous node on the search path.
backward-chaining	a search strategy which starts from the goal side.
best-first	a search method which always visits the node with the best value.
bi-directional	a search strategy which starts from both the condition and the goal side.
breadth-first	a search method which always visits the next leftmost sibling node.

"CB"	a prefix denoting circuit breaker. Used in chapter 3 and 5.
child node	a node is a child node of another node if the latter precedes the former in a tree.
class	a collection of objects which all share some commonalities.
class hierarchy	a structure which arranges the classes in the order of generality.
comparison path	the sequence of nodes which have been compared with the facts, and are found true.
condition	a requirement. Conditions constitute the LHS of a rule.
depth-first	a search method which always visits the leftmost child node.
Dictionary	a Smalltalk class. Instances of this class store data as key/value pairs.
dissociation	a process which breaks down a rule into several rules in their simplest form.
EMS	Energy Management System. A computerized power system control centre.

encapsulation	the realization of concepts or ideas in an OOP language.
ES	Expert System. A computer software which imitates human experts' decision making process.
fact	a piece of information describes part of the actual situation.
fault	an apparatus or a section of the power system that malfunctions.
false	a condition is said to be "false" when it does not match a fact.
fire	a rule is said to "fire" when its LHS is true, causing the RHS to become true.
forward-chaining	a search strategy which starts from the condition side.
goal	a conclusion. It constitutes the RHS of a rule.
hashing	uses a function to calculate a value for part or all of a character string, and puts the character string into a table according to the value.
heuristics	a function that chooses one of the several available choices based on some given criteria.

hill-climbing	a search method which always visits the child node with the best value.
indexing	indicates which conditions are related to which rules.
inference engine	a mechanism using knowledge in the KB to infer conclusions regarding actual situations.
inheritance	an object property allowing a class to inherit data and methods from its superclass.
instance	a specific object of a class.
KB	Knowledge Base. A collection of knowledge in a specific area.
key condition	a condition which has the largest sum value and becomes a node in a tree.
"L"	a prefix denoting transmission line. Used only in chapter 5.
leaf node	it has a parent node but no children nodes. A leaf node denotes a goal.
LHS	Left Hand Side. The LHS of a rule is the portion of the rule between the words "if" and "then".

match	a condition is said to "match" a fact when both describe the same thing.
message	an instruction. When passed to an object, a message manipulates the object's data content.
message passing	sending an instruction to an object.
model system	the program execution which imitates the referent system.
method	the definition of a message in a class.
MMI	Man Machine Interface. A program allowing communication between the computer and the user.
MVA	Mega Volt Ampere. A unit used in the measuring of electrical equipment capacity.
node	a tree node. It denotes a key condition.
object	an encapsulation of abstractions.
occurrence number	the number of appearances of a condition among the rules.
OOP	Object Oriented Programming. A new way of computer programming using objects and message passing.



OrderedCollection	a Smalltalk class. Instances of this class store data in an ordered fashion.
parent node	a node is a parent node of another node if the former precedes the latter in a tree.
phenomena	the things that happen in the referent system.
polymorphism	an object property allowing messages to be redefined in different classes.
procedural programming	the traditional way of computer programming using procedural function calls to manipulate data and variables.
protocol	the codes which define the format of a class.
referent system	the part of the real world that is to be imitated.
represent	a condition is said to "represent" several rules if they all contain the condition in their LHSs.
residue rulebase	the remaining of a rulebase which has been partially extracted during the transformation process.
RHS	Right Hand Side. The RHS of a rule is the portion of the rule following the word "then".
root	a root has children nodes but no parent node. Each tree

	has only one root.
RTU	Remote Terminal Unit. A device which collects data information about local power system apparatus, and sends the information to a SCADA system.
rule	a production rule. It stores knowledge in the "if conditions then goal" format.
rulebase	a collection of rules.
RuleBases	a created Smalltalk class. Instances of this class store each rule in two different sets.
"S"	a prefix denoting substation. Used only in chapter 5.
SCADA	Supervisory Control And Data Acquisition. It collects and processes all data information from RTUs and sensors.
search algorithm	the main component of the inference engine. It is responsible for searching the KB for the piece of knowledge applicable to the situation in concern.
search path	the same as comparison path.
Set	a Smalltalk class. Instances of this class store data randomly, and do not allow duplicates.

SetOfFact	a created Smalltalk class. Instances of this class store facts.
sibling nodes	nodes of a tree having the same parent node.
source	the source of a disturbance is the cause of it.
Stack	a created Smalltalk class. Instances of this class store data in the last-in-first-out fashion.
StringRules	a created Smalltalk class. Instances of this class store rules in the character string format.
subclass	a class is a subclass of another class if the former contains only part of the objects of the latter and the objects share commonalities.
subgoal	a condition in a rule and the goal of another.
substitution	a process which substitutes a subgoal in the LHS of a rule with the LHS of another rule whose RHS is the subgoal.
subtree	a tree which is a part of a bigger tree.
sum value	the sum of the occurrence number and the weight. The condition having the largest sum value becomes a key condition.

superclass	a class is a superclass of another class if the latter is a subclass of the former.
symbolic data	data which represents a symbol.
"T"	a prefix denoting transformer. Used only in chapter 5.
Table	a created Smalltalk class. Instances of this class store data as a name/value pair.
transformation	a process which transforms a rulebase into a tree.
tree	a form of knowledge representation facilitating searching. A tree contains a root and at least one leaf node.
Tree	a created Smalltalk class. Instances of this class store trees.
true	a condition is said to be "true" when it matches a fact.
weight	a value to show the relative importance of a condition.

## CHAPTER 1

### INTRODUCTION

Larger and more complex power stations have been built to meet man's endless demand for electrical power. Existing power system facilities have been modified and upgraded in order to be compatible with the newly built facilities. Needless to say, large and complex systems are vulnerable to breakdowns and malfunctions. Power systems are no exception: transformers can overheat for various reasons; transmission lines and their supporting structures can be damaged by storms and other natural disasters; circuit breakers sometimes fail to operate properly; even the protecting circuitry can sometimes malfunction, giving incorrect signals. Any of these problems, if not corrected in time, may propagate through the whole power system, causing systemwide collapse. Although rare, equipment failure induced systemwide collapse does occur. The Hydro-Quebec incident in March of 1989 is such an incident[1].

On March 13, a magnetic storm caused the tripping and shutdown of all Static Var Compensators (SVCs) on the La Grande Network. Without the SVCs, the network became unstable, and all of its 735kV transmission lines tripped subsequently. The tripping of the transmission lines deprived the Hydro-Quebec system of 44% of its total electrical power supply, which led to a systemwide collapse[1].

In addition, intense calculations are necessary to optimize power system

operations and maintain proper power flow. The results of the calculations are often realized through a series of switching.

Resolving the malfunction problems, and performing and realizing the calculations can be effectively carried out via a computerized control centre. An Energy Management System (EMS) is such a computerized control centre. It coordinates and controls the functioning of the different parts of the power system. EMS is an application, in the power area, which utilizes recent computer hardware and software developments.

Ever since its invention, the computer has offered mankind assistance in computational problems and database management. Traditionally, computers have been mainly responsible for processing large quantities of numbers which are difficult and tedious for man to handle. During the 1960's, computers were given another line of duty[2]: Artificial Intelligence (AI). AI is composed of several sub-fields, one of which is Expert System (ES). A lot of research have been conducted regarding both the theoretical and applicational aspects of ES[3].

Along with the birth of computer came the invention of computer languages. They are sets of instructions through which humans tell the computers what to do, when and how to do it. Traditionally, humans create a mathematical model for the application domain regardless of its nature. Then, variables are created to hold data which are passed to subroutines to be processed procedurally. This is known as procedural programming. In 1967, a different kind of programming technique was introduced. SIMULA marked the beginning of Object-Oriented Programming (OOP)[4]. Although first introduced in the mid 60's, OOP did not receive much public attention until the early 80's, when Smalltalk-80 was

made available to the public. The 60's also saw the introduction of the first AI language -- Lisp. It was later joined by Prolog and other AI languages. These computer languages saw little use in non-AI-related areas.

### 1.1 ES Applications

ESs are realized as computer software that assist mankind by processing symbolic knowledge, which usually should not be expressed as numbers. ESs are extremely useful in areas where large quantities of knowledge are involved. A classical example is MYCIN.

MYCIN is an ES capable of diagnosing the type of bacterial infection according to the symptoms, and suggesting therapy[5]. It consists of over 500 rules in its knowledge base. Well-known ESs include the following[5]. PROSPECTOR is an ES used to find ore deposits based on geological information. DENDRAL is capable of determining a chemical compound's molecular structure. DART performs fault diagnosis in computer systems.

ESs are also used in a wide range of areas, such as[5]: agriculture, law, manufacturing, mathematics, meteorology, and military. This shows the extensive use of ES. Needless to say, a power system is also an area where ES can be useful. In fact, ES first appeared in the power area in early 80's[3]. Since then, more and more ESs have been incorporated into power systems.

In general, an ES has a knowledge base, an inference engine, a database, and an interface. The interface allows communications between the user and the ES. The database stores data information which describes the situation of the application domain. The knowledge base contains knowledge which can be used

to solve problems in the application domain. The inference engine is an important component of ES. It contains a search algorithm which is responsible for locating the piece of knowledge applicable to the situation of the application domain. Various search algorithms have been used in ESs for power systems[6-8]. Improvement can be made on these algorithms so that a faster search algorithm can be possible. With a faster search algorithm, an ES can provide results in a shorter period of time, leaving power system operating personnel with more time to react.

## 1.2 Object-Oriented Programming And Possible Applications

It is not surprising that procedural programming languages were developed first because early applications for computers were mainly number crunching. Number crunching is basically a procedural process. For example, in order to find the inverse of a nonsingular matrix,  $A$ , the determinant of  $A$  is first calculated. Then each matrix element's cofactor is determined. The cofactors are subsequently put together to form a second matrix. The product of the second matrix and the determinant of  $A$  gives the inverse of  $A$ . However, it is difficult to set up mathematical models for many real world situations. As computer tasks diversify, new ways of programming which imitate real world situations are invented.

Since computers become more and more involved in our everyday lives, it is natural for computers to simulate different aspects of our world. Our world is not composed of variables or data to be passed between processes. Instead, our world is made up of entities carrying information regarding their own states of existence. This information can be changed only by giving the entities instructions. OOP imitates the above.



Ever since the idea of OOP was introduced in the 60's, there have been some research conducted regarding the definition of object. More specifically, people have been trying to decide which properties of the entities in our world should be preserved in the definition of object. Research has led to several definitions for object[9]. Subsequently, many OOP languages are introduced. Some of them are based on existing computer languages, for example[9,10]:

C related: C++, Objective C.

Lisp related: CLOS (Common Lisp Object System), Flavors.

Prolog related: Concurrent Prolog.

Several application software use the OOP concept. The Graphical Network Interface (GNET)[11] is written in Smalltalk/V, an OOP language. Proteus is an expert system tool incorporating both forward- and backward-chaining searching methods[12]. Pogo combines user interface construction with the OOP concept[12]. ODDESSY (Object-Oriented Database Design System) is a database design system written in Smalltalk-80[12].

The OOP concept is suitable for prototyping software because OOP models the application domain directly, allows reusing existing program codes, and encourages modular programming. Time savings result.

### 1.3 Thesis Objectives

Maintaining proper operations in a power system is a task requiring intensive knowledge in the area and quick response -- something that the operating personnel sometimes failed to provide. The result of this may range from putting an apparatus out of action to a systemwide collapse. The above is especially true

in the alarm-handling/fault-diagnosis area.

The objectives of this thesis are to outline criteria for and to propose a fast searching algorithm for use in the power area. It is also proposed to construct an alarm-handler/fault-diagnostician ES using the proposed algorithm. The ES is part of the EMS proposed by the Power Research Group of the Electrical Engineering Department[13]. The algorithm and the ES are implemented in an OOP language, as the latter is ideal for prototyping large application software such as the proposed EMS. The usefulness of the ES in diagnosing faults (alarms) is demonstrated. Furthermore, the speed of the proposed algorithm is compared to that of the searching algorithm most commonly used in the power area.

Demonstration of the alarm-handler/fault-diagnostician ES is performed using a 5-substation example. Various combinations of alarms are presented to the ES, which then infers the sources of the faults using the knowledge supplied beforehand. The ES also explains how the results are obtained. The speed comparison is performed between the proposed algorithm and Prolog/V's searching algorithm. Both algorithms are written in the same OOP language. Therefore, a fair comparison can be made solely regarding the speed of searching.

#### **1.4 Thesis Organization**

This thesis is organized as follows.

Chapter 2 explains what an EMS is. An EMS proposed for the Power Research Group of the Department of Electrical Engineering is described in detail. The EMS's features, structure, and tasks are presented. It is followed by the discussion of prototyping the EMS software using an OOP language.

Chapter 3 describes the various searching algorithms available. The searching algorithms used in the power area are also identified. Several criteria are then established for the construction of a fast algorithm for the proposed EMS described in chapter 2. An algorithm is subsequently proposed which meets the criteria. The pros and cons of the algorithm are also discussed.

Chapter 4 explains OOP in detail. A theoretical foundation is set for OOP. The Smalltalk language is also introduced. Its features and properties are described. It is followed by the description of the implementation of the proposed search algorithm. A windowing man-machine interface for the algorithm is also presented.

Chapter 5 shows some test results demonstrating the use of an alarm-handler/fault-diagnostician. It is one of many possible applications of the proposed algorithm. The substation schematic used in the tests is briefly described. The results obtained are explained and discussed. There is also an execution speed comparison between the proposed and a popular algorithm. The comparison results are also discussed.

Chapter 6 gives conclusions along with suggested modifications and further research regarding the work done in this thesis.

Appendix A describes how the four Smalltalk properties -- abstraction, encapsulation, inheritance, and polymorphism -- are involved in the formation of the six new classes' protocols.

Appendix B contains a detail list of all the methods defined in the new classes created for the implementation of the proposed searching algorithm, and the classes' immediate superclasses.

Appendix C carries the list of the rules involved in the alarm-handler/fault-diagnostician demonstrations in chapter 5.

Appendix D contains the production rules used in the execution speed comparison in chapter 5.

## CHAPTER 2

### EMS AND OOP

An introduction to EMS is given in the first part of this chapter. Next, the EMS proposed by the Power Research Group of the Department of Electrical Engineering is briefly described. It is followed by the discussion of problems related to prototyping the EMS. Then a new programming technique, OOP, capable of minimizing the problems is presented. Finally, the drawbacks of programming in OOP are discussed.

#### 2.1 An Introduction To EMS

EMS is a coordination and supervisory system. It is built on a Supervisory Control And Data Acquisition (SCADA) system. SCADA receives data and status information on various field equipment via sensors and Remote Terminal Units (RTUs). The information is passed to EMS for processing and interpretation.

There is a vast range of functions performed by different EMSs. Many EMSs have been built, however, there is no standard format. An individual EMS is built according to the builders' individual needs and specifications, and is seldom integrated to another EMS. The only commonalities shared by different EMSs are their general features: 1) a SCADA system; 2) a man-machine interface; and 3) a data processing module, usually involving an expert system of some sort.

The SCADA system, as mentioned, is responsible for collecting data and

status information. Among those collected in a power system are voltage and current magnitudes, frequency deviation, and protective equipment status. As many as 50,000 pieces of such information can be collected[14]. Some information is presented directly to the operator, while other information is run through the data processing module before the result is presented.

The data processing module has two functional submodules: a numeric data processing module, and a symbolic data processing module. The numeric data processing module contains computer programs for load-flow analysis, MVA calculations, time domain simulation, etc. The symbolic data processing module, usually referred to as expert system, is generally divided into two parts: a knowledge base, and an inference engine. The most common form of storing knowledge is production rules[5]. The inference engine contains a mechanism which, based on either the information from the SCADA system or the results from the numeric data processing module, infers a conclusion using knowledge from the knowledge base.

All the above information, results and conclusions are presented to the operator through the Man-Machine Interface (MMI). The MMI also allows communications between the operator and the system. Graphics usually play an important role in the interface. With the help of graphics, the interface gives a clear picture of the power system's operating status.

## **2.2 A Proposed EMS**

The Power Research Group of the Department of Electrical Engineering has proposed an EMS. It is designed to be a long-term project for the group.

This EMS is a computerized supervision and control centre. It centralizes power system operations in order to achieve efficiency. Expert System is used to aid operating personnel in the event of disturbances. The following briefly describes the EMS's selected features, structure, and various tasks[13]. Note that only the software, not the hardware, of the EMS is dealt with in this chapter.

### **2.2.1 Features**

The selected features are: training facilities, user friendliness, testing, machine learning, and expansibility.

#### **(i) Training facilities**

The EMS should provide a built-in tutorial to train operators. This has an obvious advantage over browsing volumes of operator's manual. The tutorial provides the trainees with a quicker and more thorough understanding of the operation of the system. It also increases the trainees' confidence in the system.

#### **(ii) User friendliness**

A comfortable working environment should be provided to the operator. Using a combination of mouse and keyboard input along with windowing structure, the EMS is built for ease of use and understanding. The user also has the freedom to customize the set-up of the window configuration to his/her preference.

#### **(iii) Testing**

The system supports a testing module which can be used to investigate the validity of any conclusions made by either the operator or the system. This detects unsound conclusions and hence prevents would-be disasters. Real-time data from the power system is used in testing.

**(iv) Machine learning**

Some form of learning ability is expected to be incorporated into the software system in the final stage of development so that the system can remember past events. The system can then help the operator resolve present problems by either recalling similar situations, or inducing conclusions from memory.

**(v) Expansibility**

New theory and concepts are bound to evolve during the course of the development of this EMS. Therefore, care must be taken to make sure this EMS can easily adopt new specifications.

**2.2.2 Structure**

The EMS has three components: Database, Logical Search Unit (LSU), and Man-Machine Interface (MMI). The Database, which receives data information from the SCADA system, provides data to the LSU. The LSU processes the data, numerically or symbolically, and reaches a conclusion. The conclusion is presented through the MMI to the user.

**(i) Database**

The SCADA system feeds the database with real-time data information. Each piece of information occupies a destined spot in the database. If changes are made to the power system, the database must be modified accordingly. Modification of the database is done through the MMI.

The database provides a complete, easy to access, and up-to-date set of data information of the power system. The above is essential because the LSU may retrieve any data information from the database at any time.



### **(ii) Logical Search Unit**

The LSU is composed of two modules, which perform two different types of tasks. The Executive module performs mostly routine tasks to achieve a smooth running of the EMS. The Expert System module performs tasks which execute symbolic processing to aid the operator in handling power system disturbances.

Each module maintains its own specialized data structure. A circular buffer contains a short historical record of the power system. This record enables the Executive module to report the performance of the power system as a whole or performance of individual apparatus. The other data structure is a Knowledge Base (KB) used by the Expert System module. The knowledge base provides knowledge upon which the Expert System can draw conclusions. The KB contains three different types of knowledge: data strings, several Knowledge Task Description Files (KTDFs), and Network Topology File (NTF). The data strings contain data and status information of power system apparatus. This information is stored in a character string format which is understood by the Expert System module. The KTDFs are discussed in section 2.2.3. The NTF stores connectivity information of all the apparatus in the power system.

Normally, the expert system tasks are not executed unless they are invoked by either the operator or the power system. On the other hand, the Executive module constantly performs tasks such as updating the different data structures, updating system log, and scheduling the execution of the other tasks.

### **(iii) Man-Machine Interface**

The MMI is the means of communication between the human operator and the system. In order to make the operator feel comfortable working with the inter-

face, and to reduce the operator's work load, the interface should be easy to understand and use. In other words, the interface should be user-friendly. This is achieved by the use of graphics and a combination of mouse and keyboard inputs. The use of windowing structure also enhances user-friendliness by displaying information in a well organized and structured manner.

### **2.2.3 Tasks**

There are three types of tasks in the EMS: Executive module tasks, Expert System module tasks, and MMI tasks. A total of 20 tasks are present among the three types of tasks.

#### **(i) Executive module tasks**

The Executive module performs seven tasks. Most tasks are related to information acquisition and update. Some tasks are run continuously or without the operator's knowledge. The tasks are contained in the Executive Task Description Files (ETDFs). Each ETDF describes the execution procedure of a task. The following briefly describes the seven tasks.

##### **(a) Scheduling**

It is important to schedule an order of execution for the other 19 tasks, because scheduling the task enables the system to run smoothly and efficiently. Requests for execution of tasks are queued. The execution order depends on both the type of task to be executed and the present status of the EMS.

##### **(b) Expert system module**

This task allows the EMS to invoke the Expert System module during disturbances without the operator's knowledge. The Expert System uses the knowl-

edge in the knowledge base to interpret the data from the SCADA system. It then provides the operator a clear view of the power system.

(c) KB data strings update

As mentioned, the Expert System utilizes data in the KB. As new data are collected by the SCADA system, this task is activated and updates the KB data strings.

(d) Historical data update

When new data are collected, this task is also activated. It ensures a short historical record of each power system apparatus is constantly refreshed, so that the apparatus's performance can be displayed upon request.

(e) Historical data acquisition

This task is invoked when the display of historical data of power system apparatus is requested. This request usually comes from the operator through the MMI.

(f) KB statistics recording

The KB statistics recording task keeps a record of the usage of the Knowledge Base. The record may be used by the expert system, along with the Knowledge Base, to induce conclusions.

(g) Flag database input update

When this task is executed, it signals the adding of new data into the database. All tasks related to the database should then be checked to be in compliance with the change of data.

**(ii) Expert system tasks**

There are eight tasks related to the Expert System module of the LSU.

Usually these tasks are not executed. When the power system experiences a disturbance, these tasks are activated either automatically or upon request of the operator. The execution procedures of these tasks are stored in the KTDFs. The following tasks, are described in the KTDFs in the form of production rules, and are part of the KB.

(a) Alarm verification

When the power system experiences a major disturbance, alarms flood in. False alarms may occur. This task uses the data and status information from the power system to verify all alarms.

(b) Alarm summarization

The number of incoming alarms may be overwhelming. This task provides a summarization of related alarms into a short message form to reduce the burden on the operator.

(c) Alarm source determination

Once alarms occur, this task can be invoked to determine the cause (source) of the alarms. Although success is not guaranteed, the Expert System infers conclusion using information available at the KB.

(d) Explanation

The operator can request the expert system to display the reasoning and decision making process used to reach its conclusion. Based on this information, the operator can evaluate the conclusion and decision making process.

(e) Remedial suggestions

As soon as the cause of the disturbance is determined, remedies can be suggested. The relationships between the causes and remedies are stored in the

form of production rules in a KTDF. Upon the operator's request, this task takes the cause, finds and suggests the related remedies to the operator.

(f) Scenario memory update

A scenario includes the alarms occurred, source determined, and remedies suggested. This task records every scenario. These records are later used by the Past Scenario Pattern Matching task.

(g) Past scenario pattern matching

Alarm source determination and remedy suggestion are not the only tasks capable of determining the source of alarms and suggesting remedies. The Past Scenario Pattern Matching task is also able to achieve the above objectives by matching existing alarms with those on the scenario and displaying the determined source and suggested remedies.

(h) Anticipation of related events

Occasionally, events happen in a sequential manner. When this occurs, this task can be activated to predict forthcoming events. This allows operating personnel time to prepare. It also serves as a checking mechanism on the power system apparatus.

**(iii) Man-Machine Interface tasks**

Five tasks are related to the MMI. Most of the tasks are related to data and information I/O. The operator is always involved in these tasks. The following gives a summary of the tasks.

(a) Edit/View

This task allows editing and viewing of four different types of files, which are: database description file, system description files, NTF, and task description

file (includes both KTDFs and ETDFs). The database description file contains instructions regarding the storing of data information in the database and the two data structures in the LSU. The system description files include the Custom Display Configuration File and the Alarm Display Configuration File. The former is related to the display format of the windowing interface. The latter allows the alarms to be displayed in various formats, such as in chronological order and in geographical order. The NTF contains the connectivity information among the apparatus in the power system. The task description files store the execution procedures of the various tasks described in previous sections.

(b) Historical record display

This task causes the scheduling of power system apparatus records to be displayed. The only parameter needed is the name or identification code of the apparatus.

(c) System log

This task is active all the time. It keeps an accurate record of the system's operation. The log is kept for future reference and investigation.

(d) Run/Test

There are two functions involved. The Run function activates the Expert System, which infers conclusions on the input data using knowledge from the knowledge base. The Test function tests the EMS's operation by running test data from a test file, and then comparing the results with those on the test file.

(e) Remedy response

This task prompts for the operator's response regarding remedy suggested by the Expert System. The remedy and response are added to the past scenario

record and the knowledge base statistics record for future assessments.

### 2.3 Problems Related To Prototyping

This EMS is intended to be a long-term project spanning several years. A large part of this project is to prototype the software described in the previous sections. The prototype will subsequently be refined to yield the final software application. There are several problems associated with the prototyping of such a large software system.

First of all, development of the graphics interface with windowing structure intended for the software is, if not difficult, very time consuming. It is supposed to support everything from windows to graphs and charts, and symbols used in displaying power system configurations. The interface must be versatile enough to meet various requirements. Yet it should also be easy to maintain and modify. Also, a lot of time is spent writing source code. The amount of source code to be written for this EMS is obviously very large. This problem is closely related to the capability of the software, i.e. the more functions the software performs, the bigger the software is. The size of the software also depends on the choice of the computer language. Some computer languages produce bulkier source codes than others. Finally, there are the problems related to program bugs. The bulkier the code, the higher the chance bugs exist, and the greater the time required for testing and debugging. It is also easier to create program bugs in some computer languages, especially those with special features -- for example, structures and pointers.

During the course of prototyping, the problematic relation among code size,

bug existence, and debugging will appear again and again. Effort and time to be spent on this aspect, adds to the already huge amount of time and effort to be spent on writing alone. It is therefore important to reduce the magnitude of, if not totally eliminate, the problems.

## 2.4 OOP As A Solution

OOP provides a solution to the first problem because several OOP languages have built-in graphics capabilities. This enables the graphics interface to be built with greater ease. Unfortunately, neither of the remaining two problems can ever be solved. As long as there is software to be written, these problems exist. The most that can be expected is to minimize the problems, i.e. to reduce the time and effort necessary for writing codes, and to minimize the chance for program bugs to exist hence reducing the time necessary for debugging. OOP is also able to minimize these problems through its highly modular nature, and its strong emphasis on code reuse.

Traditional procedural programming languages, on the other hand, do not necessarily have any graphics capabilities. Nor do they possess any properties which can minimize those problems.

How does OOP encourage code reuse? How highly modular is OOP? Before answering these questions, let's briefly look at OOP and two of its important features. OOP is developed on the concept of objects. All objects are grouped into classes and subclasses. Therefore, each class and subclass is an object. The first feature is that each class contains its own data and methods of manipulating the data. Both the methods and data structure of a class are inherited by its sub-



classes. Therefore, methods for manipulating a class's data must reside within that class or its superclass. The other important feature is that the source codes for most of these methods and data structure can be modified at any time. The latter feature facilitates the creation of new classes and methods. If a new method is to be created for a class where a similar method exists, implementing an altered version of the method is all that is required. This creates a new method.

Code reuse is encouraged. By collecting the methods inside a class, OOP alerts programmers to the presence of potentially reusable codes. At the same time, OOP makes it easy for programmers to look for reusable codes.

Since methods and associated data must reside within the same class, it follows that modular programming is enforced. Because of this requirement, a program involving several classes is forced to be divided into modules. Each module is related to only one of the several classes.

Time and effort spent on writing source codes can be greatly reduced by reusing existing code. As demonstrated in the new method creation example, time is spent only on making modifications to existing codes instead of writing the new method from scratch. OOP's modular nature also reduces the effort necessary for writing codes because several small simple modules are much easier to write than a big complex one. The program bug problem is partly solved because of both OOP's features. Code reusing is performed on existing source codes which function properly, hence they are "bug-free". Besides, since only a small area is modified, there is a smaller chance to create bugs. If bugs do occur, they are confined to the modified area. The modular nature of OOP causes programs to be decomposed into small modules, which facilitates program debugging.

## 2.5 Drawbacks Of OOP

There are two drawbacks for OOP, namely, slow execution speed and the need of large memory space [15].

The large memory space requirement stems from the generality of OOP. Large amount of memory space is required to hold the inherited methods and data structure, in addition to those defined in a class. Much of the inherited code is written to provide a wider application range, to increase versatility, although the codes may not be used to their full extent or used at all. An obvious solution is to remove all unnecessary codes. But this may handicap future maintenance and expansion of the software, as those "unnecessary" codes may be needed in the future to fulfill new specifications of the software system.

The slower speed is inevitable because of a large computation overhead. The overhead is related to message passing, where parsing, hashing techniques are performed for input text manipulation.

Despite the shortcomings, OOP is still a better choice for prototyping than procedural programming. It is because the time and effort saved outweighs the drawbacks. Smalltalk/V is the OOP language used for prototyping in this development. A more detailed description of Smalltalk/V is presented in chapter 4.

## CHAPTER 3

### EXPERT SYSTEM SEARCH TECHNIQUES

Many search techniques have been developed[16]. Each has its own advantages and disadvantages. Some feature easy implementation, while others emphasize intelligent decision making. In the first part of this chapter, several commonly used search techniques are described. Not all are suited to power system expert system applications. In fact, most search techniques are not. The shortcomings of the techniques are stated. Then, several criteria for a power system oriented search algorithm are listed. Using these criteria, a fast search algorithm is constructed and presented. A discussion of the algorithm's benefits and drawbacks is then given.

#### 3.1 Search Techniques

Every expert system has a Knowledge Base (KB) and an inference engine. Usually, the KB stores knowledge in the form of production rules and a set of facts. The inference engine contains a mechanism which searches the KB for the right piece of knowledge to solve a problem.

Most search techniques are composed of two parts. Each part is independent of the other. The first part is the search strategy. It controls the general direction of the search. The second part is the search method, which specifies details on how the search is conducted.

Before discussing the search techniques, three keywords must be ex-

plained: goal, condition, and subgoal. A goal is the conclusion. For example, in the following two rules:

If a and b then c

If c and d then e

e is a goal. It is expressed as the RHS of a production rule. Condition is the requirement that determines if a conclusion is true or not. In the above rules, a, b and d are conditions. They are expressed on the LHS of a production rule. If a goal is true, then part or all of its conditions (depending on how the conditions are arranged) must be true, and vice versa. A set of facts is necessary to determine if a basic condition is true. A basic condition is true if it matches any fact in the set. A subgoal is both a condition and a goal. For example, c is a subgoal in the above rules. It is a condition in one rule, and a goal of another rule. Sometimes a condition is satisfied as a subgoal when its dependent conditions are true.

If a condition cannot be matched to a fact or satisfied as a subgoal, it is taken as false.

### **3.1.1 Search strategies**

The search strategy controls the direction of the search, i.e., whether to start from the goal or from the condition. There are three search strategies based on the two directions of search: forward-chaining, backward-chaining, and bi-directional.

#### **(i) Forward-chaining**

In forward-chaining, the search starts from the conditions. If the conditions are not true, the goal of the rule is not considered. If the conditions of a rule

are true, then the rule's goal is also true. When the goal is verified as true, the rule is referred to as "fired". The goal becomes a fact and is added into the set of facts. Because of the newly added fact, all the rules and conditions are checked again. If another rule fires, its goal becomes another fact, and the rules are again checked. This cycle keeps going until no further rule can be fired. Since this strategy focuses on the condition (data), it is also known as "data-driven".

### **(ii) Backward-chaining**

In backward-chaining, the goal is first hypothesized, i.e. assumed to be true. Then effort is spent to prove its conditions. If such effort fails, then that goal is false. Some conditions are subgoals, i.e., they are conditions of some rules, and goals of some other rules. In such cases, the subgoals' conditions are also investigated. This strategy is also known as "goal-driven" because of its emphasis on the goal.

### **(iii) Bi-directional**

This strategy combines both forward- and backward-chaining. On one side, conditions are investigated to see which goal is true. On the other side, hypothesis are made on goals and their conditions are checked. Both processes occur at the same time. They stop when they meet in the middle with a subgoal proven true by both processes. They also stop when they do not meet each other and all possible path have been exhausted.

## **3.1.2 Search methods**

During searching, a question always appears: Should the search proceed to the sibling nodes or the children nodes? In order to provide an answer, heuristics

have to be brought in. Heuristics may also resolve the problem of search explosion. This problem arises if every node is visited in a search. If a large tree is involved, such a process requires an immensely large memory and an extremely long search time. Simple heuristics, such as those used in the breadth-first and depth-first search (described later in this section), may not be capable of solving the problem. More complicated heuristics may suggest search paths which avoid unnecessary node visiting and thus reduce search time. In the simplest term, heuristic is a function that chooses one of the several available choices based on some given criteria. Different criteria result in different heuristic functions, and hence different search methods. Only five common search methods are discussed here. They are[16]: breadth-first, depth-first, best-first, hill-climbing, and A\*.

#### **(i) Breadth-first**

The heuristics used here is simple: always go to the next sibling node if possible. In this method, all nodes at one level are investigated before a node at the next level. This method selects the shortest path possible. However, it also wastes memory storage space and computation time, because many of the nodes visited are not in a solution path.

#### **(ii) Depth-first**

"Always go to the first child node" is the heuristic used in this method. At any level of a search tree, the order of the nodes always goes from left to right. Hence, the leftmost child node is always the first to be visited. The search can go as deep as possible without finding a goal. If a goal is not found in the lowest level, the search backtracks to the previous level and checks the next leftmost node. Any search path that does not end with a goal is removed from memory. There-

fore, the depth-first method promises a small memory space. However, in this method, many of the nodes visited are also not in the solution path. Moreover, it does not guarantee a shortest solution path.

### **(iii) Best-first**

This search uses a numeric function as its heuristics. The function assigns a value to each node. Depending on the implementation, the search selects the node with either the highest or the lowest value from the children and sibling nodes. Therefore, this search selects a path based on a heuristic function. The success of this search method depends on the accuracy of the function. If the function is accurate most of the time, then this search method finds the solution path quickly and easily. If the function is inaccurate, the solution path is not easily found and causes the search to jump around and waste time.

### **(iv) Hill-climbing**

This is basically a depth-first search with the order of nodes selected by a numeric function. Similar to the best-first search, hill-climbing's heuristic function also assigns values to the nodes. But unlike the best-first search, hill-climbing search focuses on children nodes only. The method selects the most suitable child node. The benefit of small memory space is inherited from the depth-first search. The use of a numeric heuristic function minimizes unnecessary node-visiting. This method does not guarantee a shortest solution path because of the nature of depth-first search. Again, the effectiveness of this method depends upon the selection of the numeric function.

### **(v) A\***

The A\* method is basically the best-first method with two numeric func-

tions assigning two different values to each node. One of the values relates the node to the starting node -- the root. The other value relates the node to the goal node. The two values do not have to have the same units. The search routine is basically the same as that of the best-first search. It selects a path in response to the most suitable values. This method has the same advantages and disadvantages as the best-first search.

#### **(vi) Other search methods**

Several other search methods exist. Most of them are basically different combinations of the breadth-first, depth-first search methods, and the two numeric heuristic functions. They are not described in detail because their basic features have just been described and because they are not commonly used.

### **3.1.3 The "best" method or strategy**

Each search method and strategy has its benefits and drawbacks, the effect of which can be significantly influenced by the appropriateness of the heuristic function being used. Each method and strategy is suitable for a certain kind of problem. A method that works well with one problem may be a disaster for another problem, and vice versa. Therefore, a "best" method or strategy does not exist. The only fair statement is that all methods and strategies have equal potential.

It is also difficult to determine the "best" method or strategy for a particular problem. Whether or not the benefits outweigh the drawbacks is usually a subjective opinion.

## **3.2 Search Techniques In Power Systems**



Expert systems are used in power systems for various purposes: fault diagnosis, alarm reduction, restoration, to name a few. Most of the search techniques used in these expert systems have been described in the previous section. The following gives a brief summary of the techniques used in power system expert systems.

### **3.2.1 Pure forward-chaining**

Expert system using pure forward-chaining is rare. However, it exists[6,17]. The search mechanism has been described before, and will not be repeated here.

### **3.2.2 Backward-chaining depth-first**

Most of the expert systems in the power system area use a backward-chaining search strategy[7,18-29], especially Prolog[7,20-29]. Prolog's search technique can be classified as a backward-chaining, depth-first technique. During the search, Prolog verifies each rule by disproving its negated conditions. The search starts with a query from the user. The query is then matched with each rule's goal, starting from the first rule. When a match is found, the rule's conditions are negated and put into a list. Effort is then spent to disprove the (negated) conditions. If a (negated) condition is disproved, it is deleted from the list. When the list is empty, the goal is true and the rule fires. If the rule cannot fire, matching is then performed between the query and the remaining rules. The above continues until either a rule fires or none of the rules can fire.

The disproving of the conditions is performed by matching the conditions with the facts and the rules' goals. If a condition matches a goal, the rule's condi-

tions are negated and put into a list, and the process described in the previous paragraph is performed. The process repeats recursively if a condition in the list matches a goal. If such recurrence occurs, the conditions in the newly created list are investigated before the next condition in the old list is investigated.

### 3.2.3 Indexed forward-chaining

A couple of expert systems use the Rete pattern matching algorithm[8,30-35]. The Rete algorithm can be classified as an indexed forward-chaining search technique. In a simple indexed forward-chaining search technique[36], each condition carries a list of the rules which contain the condition. Matching is then performed between the conditions and facts. A match between a condition and a fact is registered in every rule containing that condition. All rules are subsequently checked to see if any of them fires. If one fires, its goal becomes a fact and the matching is performed. All rules are again checked. The above is repeated until no more rules fire.

In the Rete algorithm[36], both the facts and conditions are grouped into classes. Each condition and fact is represented by a class, attributes, and values. For example, the fact of circuit breaker CB1 is open is represented by a class (circuit breaker), two attributes (name, and status), and two values (CB1, and open). In conditions, values can be variables.

Each rule is compiled into a discrimination network[35,36]. Figure 3.1 shows a simplified rule and its discrimination network. The network is composed of several branches, which represent the conditions in the rule. Each branch consists of nodes which represent the class and attributes with values in each condi-

A simple rule:

If A:uw=13 and B:dx=4 and C:mni=960 then goal.

Note: The format of the conditions for the above rule is: class:attribute=value.  
This format is used only in this example, not in the actual Rete algorithm.

The discrimination network:

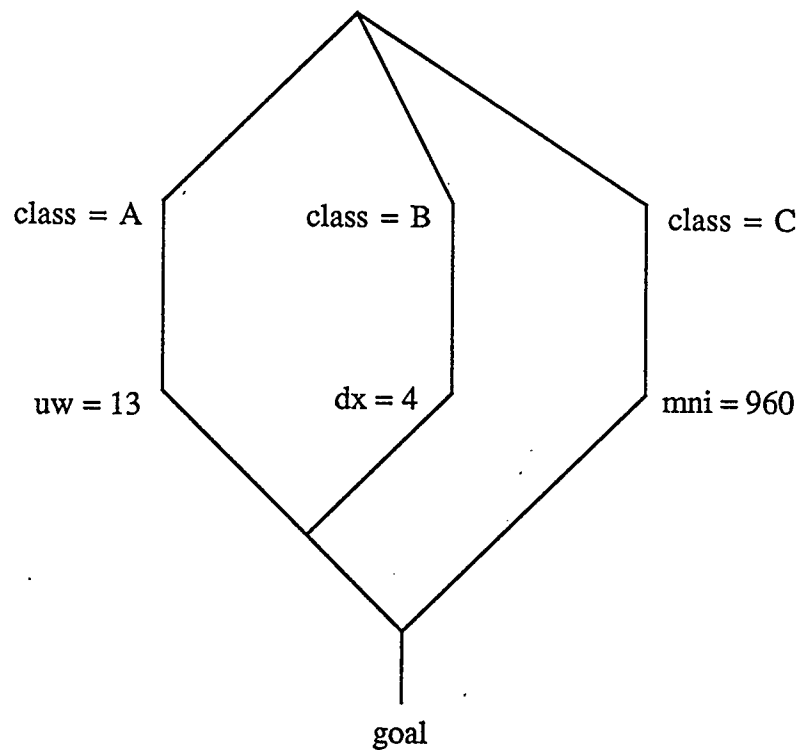


Figure 3.1. A simple rule and its corresponding discrimination network.

tion. Since there can be common conditions among the rules, the networks of all the rules are jointed together through the common branches.

The search begins with matching the conditions with the facts -- classes with classes, attributes with attributes, and values with values. The results are stored in memory. Whenever there are changes to the facts, matching is performed on these changed facts. If all the conditions of a rule match the facts, the rule fires.

### **3.3 An Ideal Search Technique**

The search techniques mentioned above are not flawless. As mentioned, there is no perfect search technique. Each technique has its limits. The same can be said for the different search techniques in the various expert or knowledge-based systems used in the power area. In the following, the pros and cons of the search techniques are discussed. Based on the discussion, several criteria for a power system oriented search technique are defined.

#### **3.3.1 Pros and cons**

Each of the above techniques used is discussed here in terms of its merits and disadvantages.

##### **(i) Pure forward-chaining**

The only benefits of this technique are that it is conceptually simple and easy to implement. Since a goal proven true becomes a fact which can be used to fire other rules, the search algorithm is implemented inside a loop. Inside the loop only two functions are performed: matching the facts with the conditions, and adding goals to the facts. When no more rules can be fired, the loop is exited.

This algorithm wastes time in several ways. Firstly, every rule is investigated, whether or not it fires. If a rule does not fire, the time spent on investigating that rule is wasted. Secondly, this phenomena repeats as many times as the number of iterations. And lastly, the last iteration only serves the purpose of exiting the loop.

### **(ii) Backward-chaining depth-first**

This technique is used by the Prolog language as its search technique. Besides the benefit that this technique is already part of the language, it is also faster than pure forward chaining.

This technique has its drawbacks. Most significant is the time wasted in the rule searching. This is due primarily to the fact that rules containing the conditions matching the facts must be found. A blind search is necessary.

Also, there is the drawback of late binding. During the search, there are actually two processes performed at the same time: constructing the search tree, and finding the solution path. The search first constructs a branch of the tree, then checks if that branch is part of the solution path. If not, it constructs another branch, and checks again. Strictly speaking, only the second process can be called searching. The first process is a burden of the second one. Moreover, a tree is constructed everytime a search is conducted. It goes without saying that the same branch of tree may be built over and over again. This wastes time.

### **(iii) Indexed forward-chaining**

Forward-chaining methods bear, theoretically, more resemblance to human diagnosis process than backward-chaining. When a human expert diagnoses a problem, he usually starts with the facts and works towards a conclusion. For ex-

ample, when a doctor diagnoses a patient, the doctor looks at the symptoms, and gradually works toward a conclusion of which illness the patient has. Seldom does a doctor start with a hypothesis (an illness), and then tries to match that illness' symptoms with the patient's.

An indexed rulebase saves time. It makes the searching of rules much easier and faster than non-indexed rulebase. This speeds up the investigation.

The simple indexed forward-chaining technique has two drawbacks. Firstly, rules with more than one condition are multiply indexed. This wastes time because a rule indexed in several places is investigated again and again even though it failed to fire during the first investigation. Secondly, late binding consumes time, as discussed in previous section.

The Rete algorithm has a pre-constructed network, hence it avoids the drawbacks of late binding. However, it still suffers from a drawback which wastes time. A false condition does not automatically exclude the rules which contain the false condition from the set of rules that may fire. The former rules are still investigated even though they cannot be fired because of the false condition.

### **3.3.2 Criteria for a suitable technique**

A set of criteria is proposed for a power system oriented search technique. This set of criteria results from analysing the pros and cons of other techniques. The set contains only four criteria, which is enough to create a suitable search technique. The criteria are: forward-chaining, depth-first, restrictive intensive indexing, and early binding.

Since forward-chaining resembles the human diagnosis processes, its ad-

vantage over backward-chaining search lies on the basic verification process. In both the forward- and backward-chaining approach, the search must verify the conditions before firing a rule. Hence it is more direct to check data and decide if the rule should fire (forward-chaining) than to pick a rule, check the data and then decide if the rule should fire (backward-chaining). An expert uses the former approach to diagnose a power system problem.

The above forward-chaining strategy is not possible without restrictive intensive indexing of the rulebase. It is essential that the rulebase is intensively indexed so that each condition indicates the next condition that should be checked or the next goal encountered. This is obviously better than simple indexing that only indicates rules related to a particular condition. However, intensive indexing must be restrictive to avoid multiple indexing of a rule. That is, there should be only one path leading to the firing of each rule, so a rule that failed to fire is not investigated again.

It is obvious that the A\*, hill-climbing, and best-first search methods are unsuitable because there are no numeric values involved in the search. The breadth-first search method is also unsuitable because it does not make the best use of the result of node verification. On the other hand, the depth-first method does. Hence, the depth-first search method is the choice.

Early binding can save time. With early binding, the tree is constructed in advance and only once. Then during the search, investigating the nodes is sufficient. The tree is constructed when the expert system is not performing a search. Therefore, only the verification process is performed during the search. After the tree has been constructed, it is stored. Hence, it need not be constructed again.

The two features save considerable time together.

### **3.4 A Proposed Fast Algorithm**

Based on the above criteria, a search algorithm is created. In addition to the mentioned criteria, this algorithm also includes an important concept: key condition. This feature avoids multiple indexing. It also allows smart partitioning of the rules.

#### **3.4.1 An important concept: key condition**

The concept was introduced by Dr. Shi-Jie Cheng, who was a member of the Power Research Group. In an unpublished computer program, Dr. Cheng used a set of handpicked key conditions to represent all the rules in the rulebase. Backward-chaining searches are then conducted using the key conditions. Unfortunately, Dr. Cheng left the group shortly afterwards, and was unable to pursue the idea. The concept was further developed and implemented by the author, with help from members of the group.

This concept avoids multiple indexing of the rules in the rulebase. In this concept, a set of conditions is selected to represent all the rules in the rulebase. This is made possible by the fact that many conditions are common to several rules. Such a condition becomes a key condition, and it is said to represent the rules which have the key condition as a condition. A set of such key conditions are selected in such a way that no two key conditions represent the same rule. This effectively avoids multiple indexing.

A special process also allows intelligent selection of key conditions. Key conditions are selected to maximize the number of rules that can be ignored when



one key condition is found to be false.

Key conditions are applied repeatedly to divide the rules. Therefore, key conditions at the top level lead to several sets of key conditions on the next level. Each key condition on this level then leads to another set of key conditions on the next level down. This carries on until a key condition represents only one rule.

This feature is very beneficial for the search process because the set of key conditions to be investigated is small, and rules are ignored when their key condition are proven false. If all the first level key conditions are proven false, the search terminates.

### 3.4.2 The proposed algorithm

The proposed algorithm can be divided into three parts: dissociation and substitution of rules, transformation, and search. Each one of them is discussed in detail.

#### (i) Dissociation and substitution of rules

The LHSs of the rules are expected to be in a complex format, containing any combinations of the following: subgoals, brackets, "and", "or", and the "not" logic operators. The rules go through two processes: substitution, and dissociation. The former substitutes all subgoals. The latter breaks down the complex format of the LHSs of the rules into a simple one: one that contains only the "and" logic operator. The following explains the two processes in greater detail.

Subgoals are substituted into rules. For example, in the following two rules:

If a and b then c (1)

If c and d then e (2)

goal c of equation (1) -- equation (1) defines a subgoal -- is substituted into equation (2), and the following is obtained:

If a and b then c (1)

If a and b and d then e (3)

If c is a condition or a goal of another rule, further substitution occurs. For example, for the following rules:

If a and b then c (1)

If f and g then c (4)

If c and d then e (2)

If c and h then i (5)

the subgoals are substituted into the other rules and the following is obtained:

If a and b then c (1)

If f and g then c (4)

If a and b and d then e (3)

If f and g and d then e (6)

If a and b and h then i (7)

If f and g and h then i (8)

Notice that this causes the number of rules in the database to increase. It should be pointed out that sometimes (1) and (4) can occur as one combined rule of the form

If (a and b) or (f and g) then c.

Rules of this kind are dissociated, which is the second process performed on the rules. The dissociation process breaks the rules written with any combina-

tion of the three operators and brackets down into rules written in terms of "and" operator only. To illustrate this point, suppose the following rule exists:

If !(a or b and c) then d

where ! denotes the not operator. The rule is subsequently dissociated into two rules:

If !a and !b then d

If !a and !c then d

Both dissociation and substitution of rules are needed because the transformation process assumes that rules are written with the "and" operator only.

#### **(ii) Transformation process**

The rules in the rulebase are transformed into a search tree. The tree's leaf nodes represent the goals, while other nodes represent the conditions of the rules. The transformation of a condition into a node requires a sum value. The sum value is the sum of a weight and an occurrence number.

Each condition carries a weight. The weight indicates the relative importance of the conditions. Each weight has a default value of 0, and can be arbitrarily changed to any value. Each condition's total number of appearances among the rules is calculated, which yields the occurrence number. The occurrence number is then added to the weight of the condition to give the sum value. The condition with the largest sum value becomes the key condition.

There are actually two uses of the weight of each condition: select, and destinate. If two or more conditions have the same largest occurrence number, then their weights can be used to select the key condition from the conditions. On

the other hand, with a large weight, a condition with a small occurrence number can suppress all others and becomes a key condition. Thus, a condition can be destined as a key condition by giving it a large weight. The user can assign the weight as he sees appropriate. The weight can also be assigned or changed by the computer software based on statistics and historical records. These records show whether or not a condition occurs more often than the others and always leads to the firing of a rule.

The following steps describe the formation of a search tree.

(1) A key condition is selected which becomes a first level node of the search tree.

(2) The rules represented by the key condition are extracted from the rulebase and placed into a temporary rulebase. The rulebase, part of which has been extracted, becomes a residue rulebase.

(3) At the same time, the key condition is removed from the extracted rules.

(4) Another key condition is selected from the rules in the temporary rulebase. This key condition becomes a child node of the previous node.

(5) The rules represented by the latest key condition are extracted and put into another temporary rulebase. Again, the rulebase from which the rules have been extracted is called a residue rulebase.

(6) Steps (3) to (5) are performed repeatedly until there is only one rule in the latest temporary rulebase AND the rule has no condition left. The goal of the rule becomes a leaf node connected to the last node formed. The rule is then removed from the temporary rulebase which is subsequently deleted. At this point,

the leftmost branch of the search tree is constructed. (It should be noted that a branch is "constructed" only if it terminates with a leaf node.)

The other branches and nodes of the tree are constructed using the following procedures.

(7) Steps (4) to (6) are performed on the latest residue rulebase to construct the next leftmost branch extending from the node associated with the residue rulebase. When all the rules are extracted from the residue rulebase, the residue rulebase is deleted.

(8) Step (7) is performed repeatedly on the most recent residue rulebase to construct another branch.

(9) The transformation process is complete when all the residue rulebases are deleted and all the branches of the search tree are constructed.

As a result of the way the search tree is constructed, it looks unbalanced, with more nodes and branches to the left of the tree. As a matter of fact, the leftmost nodes, at any level, always represent the same or more number of rules than any of their sibling nodes. Therefore, the leftmost nodes tend to have more prosperous branches than other nodes.

Once a tree is built for an application domain, the tree needs not be rebuilt unless the application domain's characteristics change.

### **(iii) Search**

The search is conducted using a set of facts. A fact is a condition that is true. Therefore, the set of facts contains a set of conditions which are true. The search compares the nodes of the tree with the facts in the set. Comparison starts from the highest level, and goes from left to right. If a node matches a fact

in the set, that node's children nodes are immediately compared with the facts. The comparison proceeds deeper and deeper, until a leaf node is reached. The successful comparison path -- from root to leaf node -- is then stored in memory, and the search backtracks to find other solutions. The search stops when the whole tree has been searched and all solutions found.

If a node fails to match any facts, the sibling node to the right is examined next. If the rightmost node fails, the search backs up to the node's parent node, and proceeds to the node to the right. If the parent node is also a rightmost node, the search backs up one more level. The procedure continues until the first level is reached which terminates the search.

If the leftmost node at any level does not match the facts, then the largest branch from that level can be ignored. This feature effectively and quickly directs the focus of the search onto a small set of rules. Hence, the feature enables the search to fire the correct rule in a short period of time.

### **3.5 Merits Of The Proposed Search Algorithm**

The proposed search algorithm has several merits. First of all, the proposed algorithm performs fast search. Also, it is efficient. Finally, the algorithm is versatile. The following explains the merits in detail.

#### **3.5.1 Fast search**

The algorithm performs fast search because it adopts the early binding feature. This feature separates the tree searching process from the tree construction process. As a result, only the former is executed during search. Therefore, searching is fast.

A fast searching algorithm is essential in Power System (PS) related expert systems because when PS experiences disturbances, its operating personnel have to make decisions within short periods of time. A fast searching algorithm can provide the operating personnel with the necessary information in a short time.

### **3.5.2 Efficiency**

The search algorithm is efficient because it uses forward-chaining and contains both the early binding and key condition features. The forward-chaining strategy eliminates the time-wasting step of investigating the rules one by one before finding a rule that fires. The early binding feature minimizes the number of times the search tree is constructed. And the key condition feature reduces the number of tree-node visiting by allowing a node's descendant nodes to be ignored if the node does not match the facts.

An efficient search algorithm is desirable not only because it makes the best use out of the information available, but it also results in faster search process. As explained in the previous section, fast searching is necessary in PS related expert systems.

### **3.5.3 Versatile**

The algorithm is versatile because it allows the rules to be written in complex formats described in section 3.4.2. As a matter of fact, these are the basic production rule formats. Therefore, it is important that the algorithm allows such formats to exist, understands them, and is able to process them. Lacking the above abilities causes great inconvenience to the users who input and maintain

the production rules.

### **3.6 Disadvantages Of The Proposed Algorithm**

Two things can be regarded as the drawbacks of the proposed algorithm. This algorithm needs extra memory storage space for the storing of the tree. It also needs considerable time for the initial construction of the tree.

#### **3.6.1 Extra memory space**

After the tree is constructed, it must be saved in memory. This is unavoidable for early binding, as the tree must be available for a search. This is the opposite to late binding, where the constructed (partial) tree is temporary and is deleted once the search terminates. Also, in contrast to algorithms such as Prolog which constructs only part of the tree, this algorithm constructs the whole tree, therefore it requires more memory space than its counterpart in Prolog.

The size of the memory required to store a tree depends on the tree's size, which depends on the size of the knowledge base. Since the number of rules in the knowledge base varies among different applications, there is no typical memory size required.

The requirement for large memory space is a drawback, but it is not significant because current computers have larger memories.

#### **3.6.2 Extra time for constructing tree**

It is true that constructing the tree for this algorithm requires more time than constructing trees for some other algorithms. This is due to the fact that the whole tree is constructed. However, this feature can actually save time because



the tree is constructed only once. The tree construction process is independent of the real search process; whereas in other algorithms, tree construction is part of the search process. In other algorithms, such as Prolog, a partial tree is constructed everytime a search is initiated. Very often the same partial tree is built over and over again. As a result, the more frequently a search is conducted using this algorithm, the more time is saved.

The disadvantages discussed in this and the previous sections are undesirable. However, the merits offered by the algorithm outweigh the disadvantages.

## CHAPTER 4

### SMALLTALK AND IMPLEMENTATION OF THE ALGORITHM

The algorithm is implemented on Smalltalk. A discussion of the Smalltalk language is given in the first portion of this chapter. This is followed by the detailed description of the implementation.

#### 4.1 Smalltalk

Smalltalk is an Object Oriented Programming (OOP) language. It is different from traditional procedural programming language such as FORTRAN and C. In this section, a theoretical foundation of OOP is presented. It is followed by the descriptions of object properties and Smalltalk features.

##### 4.1.1 Theoretical foundation

OOP can be best described by a definition. However, there are many definitions for OOP. Among them is the one written by O. L. Madsen and B. Moller-Pedersen[37], which seems quite adequate. Their definition is,

"A program execution is regarded as a *physical* model, simulating the behaviour of either a real or imaginary part of the world."

This means there is a referent system belonging to our world, as shown in figure 4.1, and a model system which is the program execution written in an OOP language. The referent system is the part of the world to be simulated. The

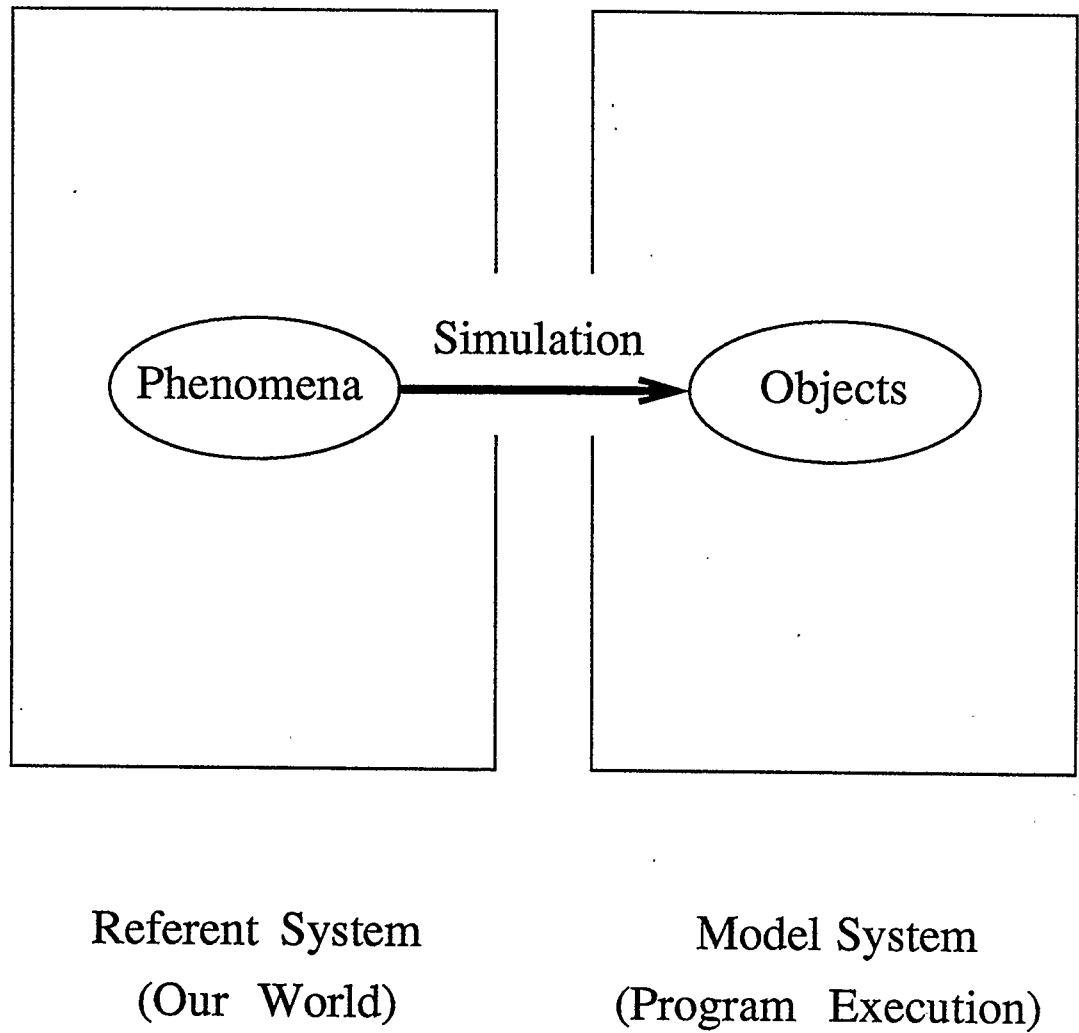


Figure 4.1. The Model System and the Referent System.

things that happen in the referent system are called phenomena. In the model system, objects are created. The phenomena is simulated by passing messages to the objects. In other words, modelling of the referent system is achieved using objects and message passing rather than data, variables and procedural calls, as in procedural languages.

#### **4.1.2 Features and properties**

In Smalltalk, objects are characterized by four properties, which are[38]: abstraction, encapsulation, inheritance, and polymorphism. Smalltalk has several distinct features. Some of them are related to the four properties. The features are[38]: message, method, instance, class, and class hierarchy. Both the properties and features contribute to the simulation of phenomena of the referent system.

In the referent system, each entity carries information regarding its own state. For example, the power generator g1 can be in operational state, producing 1 MegaWatt (MW) of power, or in inoperational state and produces 0 MW. The changing of state is accomplished by sending the generator messages (shut-down, for example).

Abstraction is a representation of ideas or concepts in the real world. "The generator is in \_\_ state", and "producing \_\_ MW" are both abstractions.

Objects are encapsulations of abstractions. Objects with similar data structure are grouped into subclasses and classes. Therefore, each class and subclass carries the data structure common to all objects of that class. An instance is a specific object of a class. For example, the class Generator contains the data variables, state and output. And g1 is an instance of the class, with the data, op-

erational and 1, stored in the variables. These data can be manipulated by sending messages to the instance. So, sending the message "shut-down" to g1 changes the data variable content to inoperational and 0. The messages are defined within the class, and are called methods.

The available classes constitute the class hierarchy. The class Object is the superclass of all classes. Each class then has its subclasses. The inheritance property allows a class's data structure and methods to be inherited by its subclasses. Therefore, a method can manipulate not only the data in the class where the method is implemented, but also in the class's subclasses.

Messages can be redefined in any of the classes or subclasses. This is the polymorphism property. Therefore, sending the same "shut-down" message to an instance of another class changes some other variables. This resembles phenomena in the referent system, where the same "shut-down" instruction can be sent to other entities and produces different results.

## **4.2 Algorithm Implementation**

Implementation of the algorithm involves several classes in the class hierarchy. New subclasses are created under the classes. Both the classes involved and the new subclasses created are described. It is followed by the descriptions of the implementation of the algorithm. Finally, an interface for running the algorithm is presented.

### **4.2.1 Classes created**

Six subclasses are created during the implementation of the algorithm. The following briefly describes the six subclasses. A more detailed description of

the subclasses can be found in Appendix A. A listing of the subclasses', and their superclasses', class protocol is given in Appendix B.

#### **(i) StringRules**

Its superclass is Dictionary. Data is stored in the key/value pair format in class Dictionary. The keys are unique, whereas the values can be the same. Both key and value are objects. Therefore, they can be anything: an integer, a character string, or a set. The pair can be retrieved by searching for either the key or the value.

The StringRules subclass is created to hold the production rules in their original form. Each goal, a character string, is a key, and each rule's entire LHS -- in a long character string -- constitutes the value. Each instance of StringRules can be given a name which is stored in the variable, name.

#### **(ii) RuleBases**

This is also a subclass of Dictionary. The RuleBases subclass is designed to hold the substituted and dissociated production rules. In this subclass, the key is a set containing a goal, and the value is another set containing each and every condition of a rule. Therefore, rules with identical goal can coexist because every set is unique, regardless of its content. The RuleBases has two variables: name, and stringRule. The former stores the name of the instance of RuleBases, and the latter stores the name of the instance of StringRules from which the instance of RuleBases is obtained.

#### **(iii) Table**

Table is also a subclass of Dictionary. This subclass facilitates the creation of weight tables which store the conditions' weights. The condition/weight

pair constitutes the key/value pair. The keys are unique. The values take integers only. Table has three variables: name, stringRule, and ruleBase. They store the name of the table, and the names of related instances of StringRules and RuleBases, respectively. Several weight tables can be created from the same instance of RuleBases to assign different weight patterns to the conditions.

#### **(iv) Tree**

Tree's superclass is OrderedCollection. As the name suggests, OrderedCollection stores objects in an ordered fashion. The objects are arranged according to the order the objects entered the collection. The collection can be searched by either the object or the position index.

A subtree has a node, also known as the root, and branches coming out of the node. Connecting these branches with other subtrees' roots ultimately forms a tree. In this implementation, search trees are constructed using instances of subclass Tree. Each search tree contains many instances. Each instance is actually a subtree, and can be viewed as a series of empty slots. It has a variable, root, to store the subtree's root's name, and the empty slots functionally resemble branches. Putting an instance into another instance's empty slot resembles connecting a root to a branch. Therefore, a search tree is composed of instances within instances of subclass Tree. The order of the instances in another instance represents the left-to-right order of the nodes in the search tree. Besides root, the Tree subclass has four other variables: stringRule, ruleBase, table, and name. The first three store the name of related instances of StringRules, RuleBases and Table. The last variable stores the name given to the search tree.

#### **(v) SetOfFact**

Its superclass is Set. Set collects objects in an unordered manner. Duplicated objects are discarded. The objects are hashed to facilitate the searching of objects in instances of Set.

SetOfFacts is created to store facts. Its only variable, name, stores the name given to the instance.

#### **(vi)Stack**

This is also a subclass of OrderedCollection. Successful search paths are stored in instances of Stack. Therefore, the objects stored in instances of Stack can be nodes of the search tree. The nodes on the instances of Stack always bear a simple relationship: every node is a child node of the node below it on the stack. Stack has six variables: ruleBase, tree, setOfFact, stringRule, table, and name. The first five store the names of related instances of RuleBases, Tree, SetOfFact, StringRules, and Table, respectively. The sixth one stores the name of the instance of Stack. As indicated later, objects stored in an instance of Stack can be other instances of Stack.

The following gives an example which shows the usage of the six subclasses. Suppose the following production rule is involved:

If a and b then G1

It is stored in an instance of StringRules. 'G1' and 'a and b' are the key and value, respectively. After the substitution and dissociation process, the rule is stored in an instance of RuleBases. 'G1' is put into a set and becomes the key. Another set which contains 'a' and 'b' becomes the value. An instance of Table is created. It has two key/value pairs: 'a'/0 and 'b'/0. The transformation process creates an instance of Tree. Suppose two facts exist: 'a' and 'b'. They are put in-



to an instance of SetOfFact. After the search process, the successful search path is stored on an instance of Stack. Its content is, from top to bottom: 'G1', 'b', and 'a'.

#### **4.2.2 Implementation**

Implementation is divided into three steps. First the substitution and dissociation of rules is implemented. This is followed by the transformation of rules. The search process is last. In order to facilitate the description of the implementation, the following convention is used. Instances of each of the above six subclasses are represented by the names of the subclasses in lower case letters with trailing "s" truncated. For example, rulebase represents an instance of subclass RuleBases. Note that both the upper case "R" and "B" have been changed to lower case, and the trailing "s" is omitted. The latter provision allows both the singular and plural forms, i.e. rulebase and rulebases, to exist.

##### **(i) Substitution and dissociation**

The dissociation process takes the value of a key/value pair in a stringrule, and separates the string at each space. The result is an array of objects where each object is a character string. The objects can be classified into three different types: conditions, logic operators, and conditions with either the left or right parenthesis. A "breakdown" message is sent to the array. In that method, a set is created and holds the first object of the array. If the second object is the "and" operator, then the third object is added into the set. However, if the second object is the "or" operator, then the third object is added into a new set instead. Every even numbered object in the array is checked for operator type, and the odd num-

bered objects followed are added into the set(s) accordingly. The message then returns the set(s) created. If the left parenthesis is encountered, then the objects within the pair of parenthesis are sent the same "breakdown" message which returns the set(s) created. These set(s) are then combined with other set(s) accordingly. The goal of the rule then pairs with each set to become a key/value pair and is added into a rulebase.

Sometimes, an original rule is split into several rules, each being represented in the rulebase by two sets. The first set contains the goal and is the key. The second set contains each individual condition and is the value. The goal is true only if all conditions in the value are true. The rules then undergo substitution. First, the dissociated rules are checked for subgoals. If a subgoal exists, then every condition of the rule whose goal is a subgoal is duplicated and added into each and every set of conditions containing the subgoal. The subgoal is subsequently removed from the set(s) of conditions. If more than one rule has the same subgoal as the goal, then the rules whose sets of conditions contain the subgoal are duplicated. The substitution takes place subsequently. The resulting rulebase is used for the transformation process.

## **(ii) Transformation**

A weight table, an instance of subclass Table, is created first. It contains every condition which appears in the rulebase. The conditions are the keys. The conditions' weights are the values. All weights are set to 0 initially, and can be changed later. The rulebase is duplicated. The following steps are performed recursively on the duplicate.

- (1) An instance of subclass Tree is created.

(2) The sum values of the conditions are calculated. A key condition is selected out of the rulebase.

(3) Another tree is created, whose root variable carries the name of the key condition. This tree is added into the previous tree. A node of the search tree is thus created.

(4) The rules represented by the key condition are moved into a temporary rulebase, and the key condition is removed from the rules. The original rulebase is now called a residue rulebase.

(5) Steps (2) to (4) are repeated recursively with the temporary rulebase. With each repetition, a new temporary rulebase, a new residue rulebase and a new tree are created. The repetition ends when only one rule exists in the temporary rulebase and the rule's conditions are all removed.

(6) The goal is stored into the root variable of a new tree, and the temporary rulebase is deleted.

(7) The new tree is added into its previous tree.

(8) Steps (2) to (7) are performed repeatedly on the most recent residue rulebase.

The transformation process terminates when the duplicated rulebase is empty. At this point, the tree construction is completed.

### **(iii)Search**

The search process requires a setoffact, which contains facts. It also needs a stack, which stores the solution path. When the search visits a node, the node is pushed into a stack. If the node does not match the facts, it is popped out of the stack, and its sibling node is visited and pushed into the stack. If the node

matches a fact in the setoffact, the search visits the node's children node. If the search reaches a goal, then the existing stack is duplicated and pushed into a master stack, which is also an instance of Stack. The search then visits other nodes and pushes other solution paths into the master stack. When the search ends, the master stack contains all the solution paths.

### **4.2.3 Interface**

An interface between the expert system and the user is constructed using the class Pane. The interface includes four windows, which are: System window, Edit window, Search window, and Transform window.

#### **(i) System window**

When the expert system is invoked, the System window appears, as shown in figure 4.2. On the top left pane, the choice "fast algorithm" appears. If this is selected, three choices appear on the top right pane: edit, search and transform, as shown in figure 4.3. They invoke the Edit, Search and Transform windows respectively. The bottom pane is reserved for message display.

#### **(ii) Edit window**

When the edit choice in the System window is selected, the Edit window appears, as shown in figure 4.4. This window allows the contents of rulebases, setoffacts, stringrules, and tables to be modified. The window is divided into upper and lower panes. The upper pane is subdivided into left, center, and right subpanes. On the left subpane, four choices are available: RuleBases, SetOffFacts, StringRules, and Tables. Selecting any one of them displays that subclass' instances in the middle subpane. Selecting an instance displays its content in the

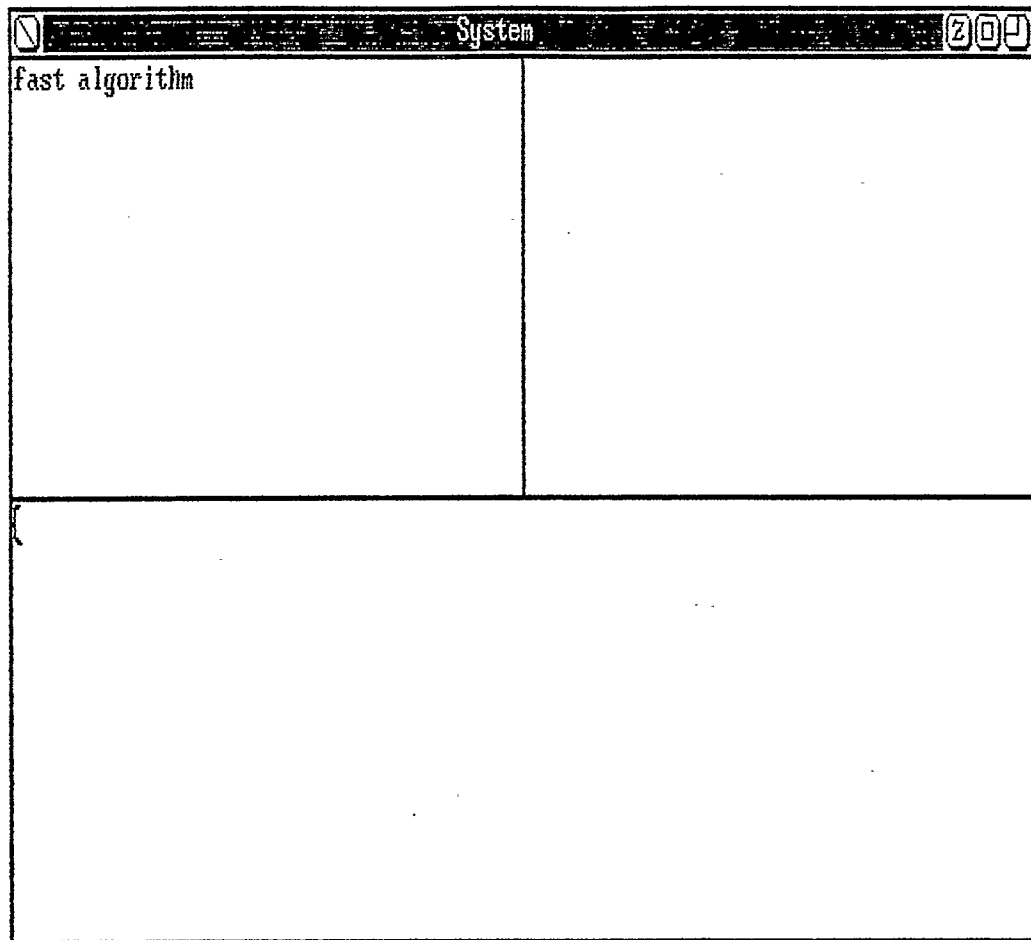


Figure 4.2. The System window.

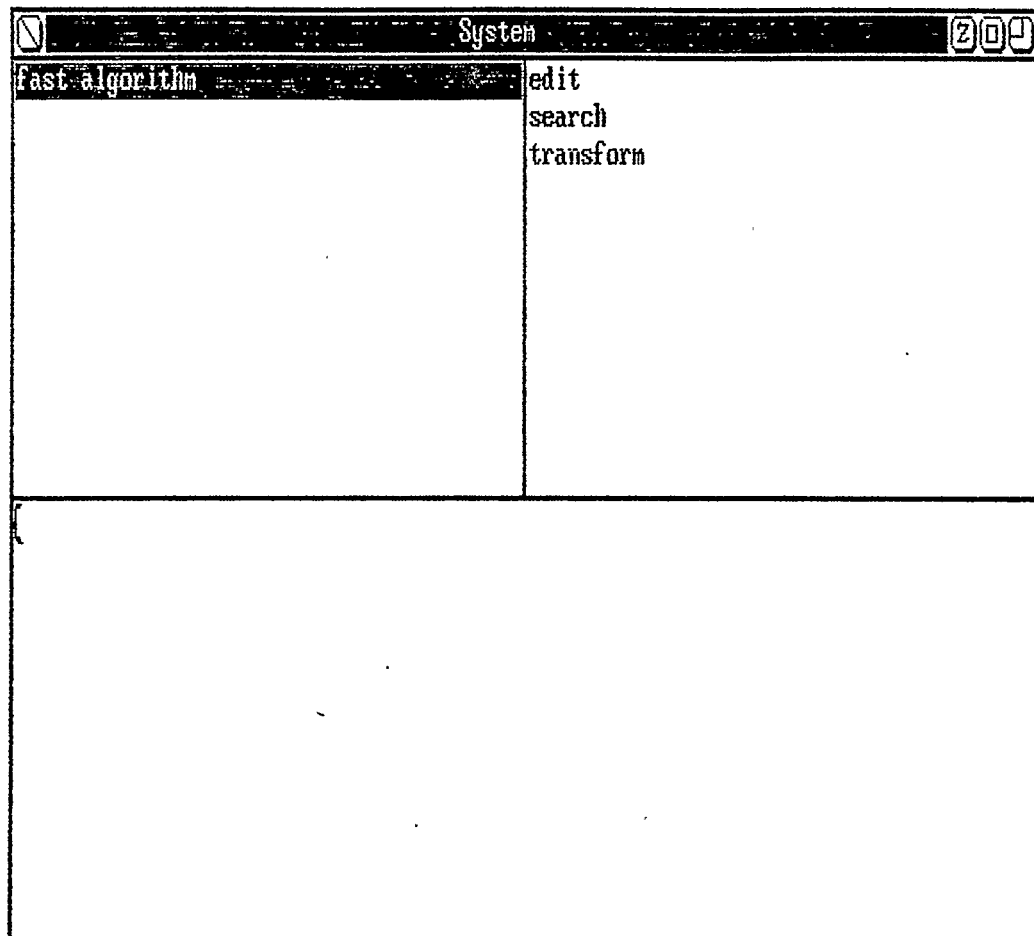


Figure 4.3. The System window with the three choices in the top right subpane.

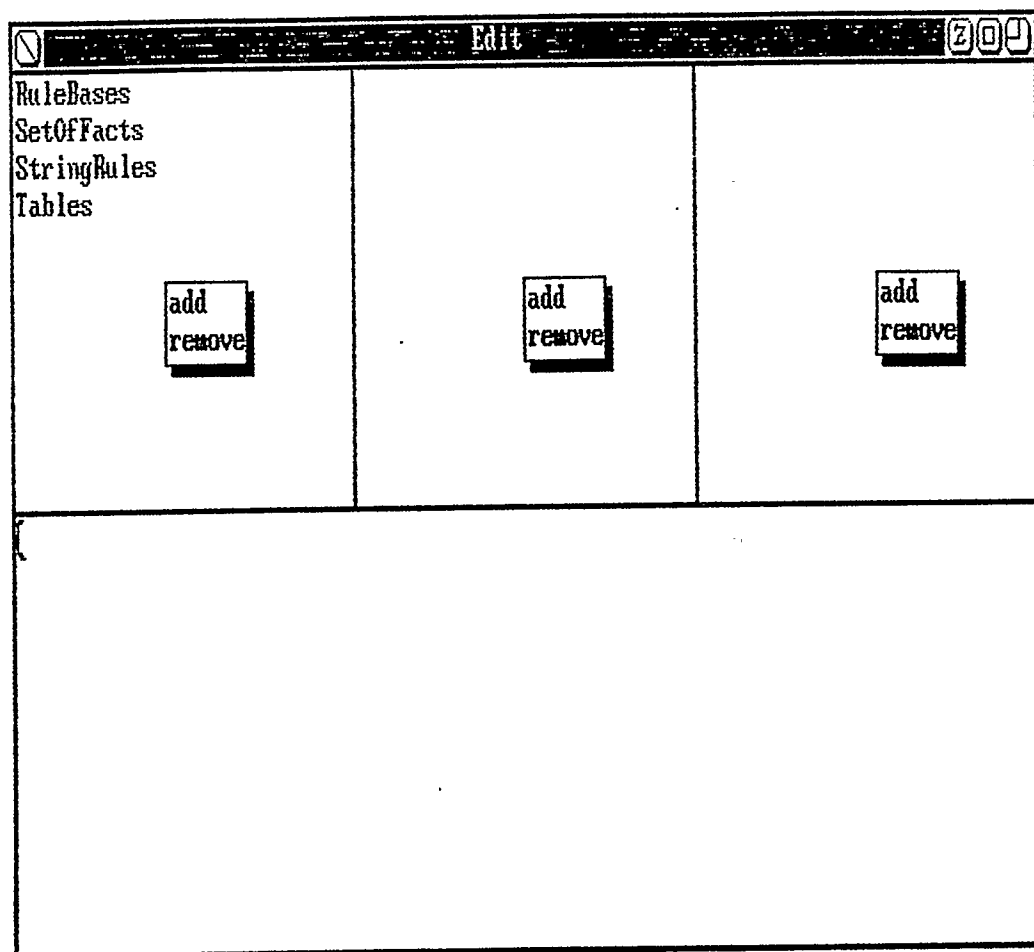


Figure 4.4. The Edit window with menus.

right subpane. If a table or stringrule is selected, only the keys are displayed. If a rulebase is selected, the key/value pairs appear. Otherwise, the facts are displayed. The above are shown in figures 4.5 to 4.8. A menu allowing two choices, add and remove, is available in each subpane. If add is selected, the user is prompted for the object to be added. The remove choice deletes the object selected.

If an object is selected in the right subpane, it or its associated object(s) appears in the lower pane, where they can be modified.

### **(iii) Search window**

The Search window allows searches to be conducted. This window resembles the Edit window except that a middle pane is inserted. The names of the available trees, setoffacts and stacks are displayed in the left, centre and right subpanes respectively, as shown in figure 4.9. In the left subpane, there is a menu which provides only one choice: search. If this is selected, the user is prompted for the names of the setoffact and stack to be used. The selected tree is subsequently searched, using the setoffact supplied by the user. The resulting master stack is stored under the name given by the user. The search result is also displayed in the middle pane. Explanation is provided in the lower pane. A tree's content cannot be displayed. But a setoffact's or a stack's content can be displayed in the middle pane by selecting the instance. A stack can also be removed. This is achieved by selecting the instance and choosing the only choice on the menu, remove, in the right subpane. The middle pane also has a single-choice-menu. The choice allows the expert system to explain the search result stored on the stacks. The explanation is displayed in the lower pane.

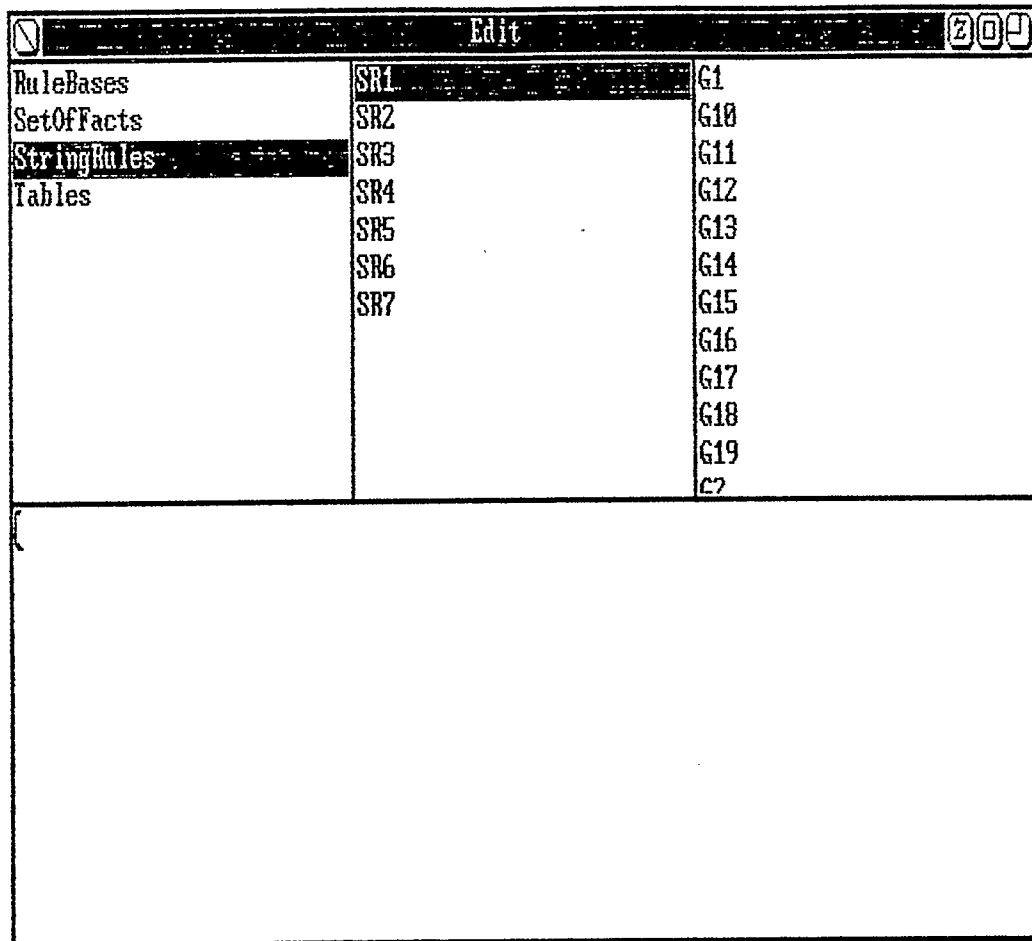


Edit		
RuleBases	RB1	G1 / g5 f5 e5 d5 b5 a5
SetOfFacts	RB2	G10 / b2 g2 f2 e2
StringRules	RB2	G11 / b2 j2 h2 f2 e2
Tables	RB3	G12 / b2 k2 h2 f2 e2
	RB4	G13 / b2 n2 m2 h2 f2 e2
	RB5	G14 / b3 e3 c3
	RB6	G15 / c3 a3
	RB7	G16 / c3
		G17 / a4 d4 b4
		G18 / f4 e4 d4
		G19 / e5 c5 b5 a5
		G2 / d1 a1 d4

Figure 4:5. An instance of RuleBases is selected in the Edit window.

Edit		
RuleBases	SOf1	a1
SetOfFacts	SOf2	b1
StringRules	SOf3	b2
Tables	SOf4	e2
	SOf5	f1
	SOf6	f2
	SOf7	g1
		h2
		m2
		n2

Figure 4.6. An instance of SetOfFact is selected in the Edit window.



Edit		
RuleBases	SR1	G1
SetOfFacts	SR2	G10
StringRules	SR3	G11
Tables	SR4	G12
	SR5	G13
	SR6	G14
	SR7	G15
		G16
		G17
		G18
		G19
		G2

Figure 4.7. An instance of StringRules is selected in the Edit window.

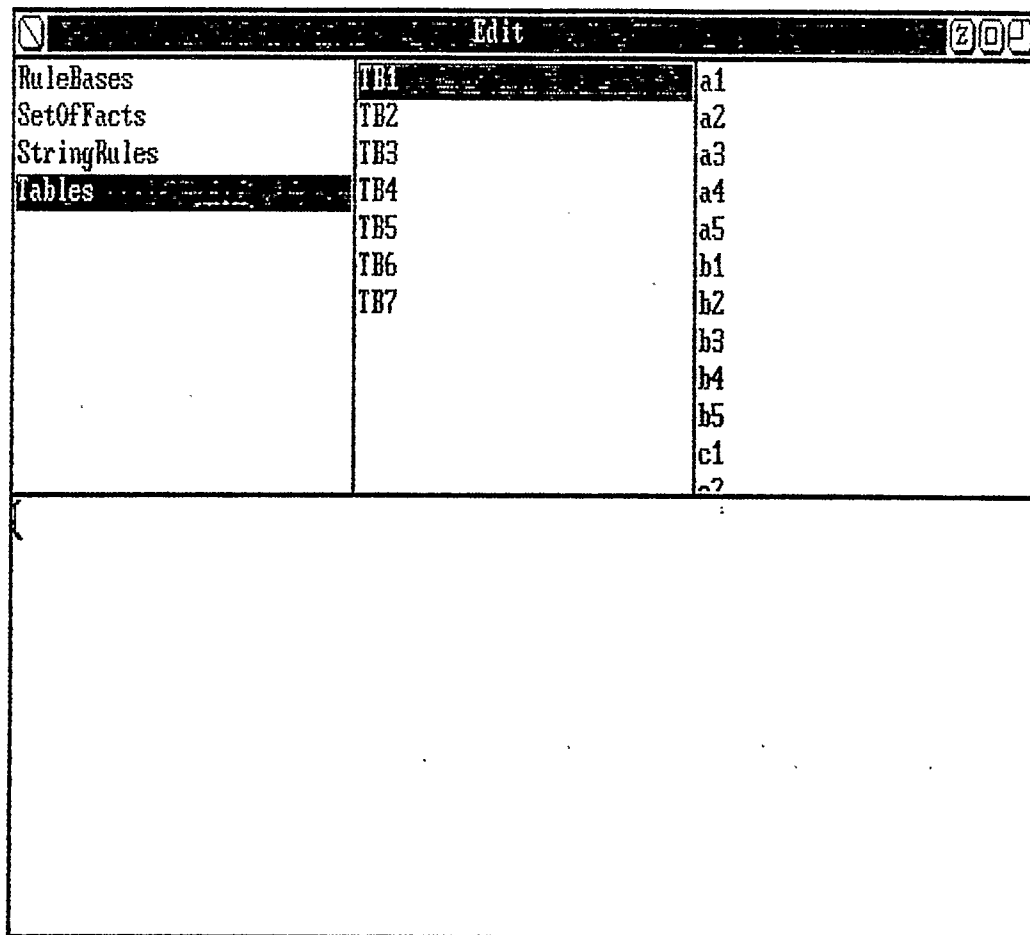


Figure 4.8. An instance of Table is selected in the Edit window.

Search		
TR1	SOF1	PS1
TR2	SOF2	PS2
TR3	SOF3	PS2
TR4	SOF4	PS3
TR5	SOF5	PS4
TR6	SOF6	PS5
TR7	SOF7	PS6
		PS7
<div>search</div>		
<div>remove</div>		
<div>explain</div>		

Figure 4.9. The Search window with menus.

#### (iv) Transformation window

The dissociation and substitution, and transformation processes can be activated only in the Transform window. This window also resembles the Edit window, except that the upper pane is subdivided into four subpanes, as shown in figure 4.10. Names of the available stringrules, rulebases, tables, and trees are displayed in the four subpanes. Each subpane has a menu. The stringrules' menu only allows the choice of dissociation, which includes substitution. The result is put into the rulebase specified by the user. The rulebases' menu provides four choices: remove, create table, relationship, and transformation. The second choice, create table, creates a new weight table whose name is given by the user. When the fourth choice, transformation, is chosen, the selected rulebase is transformed. The resulting search tree is put into the tree specified by the user. Before the transformation takes place, the user is prompted for the weight table to be used. The remaining two choices also appear in the tables' and trees' menu. The remove choice removes the instance selected. The relationship choice shows the names of other subclasses' instances related to the selected instance. Contents of the instances can be displayed -- except trees -- in the lower pane.

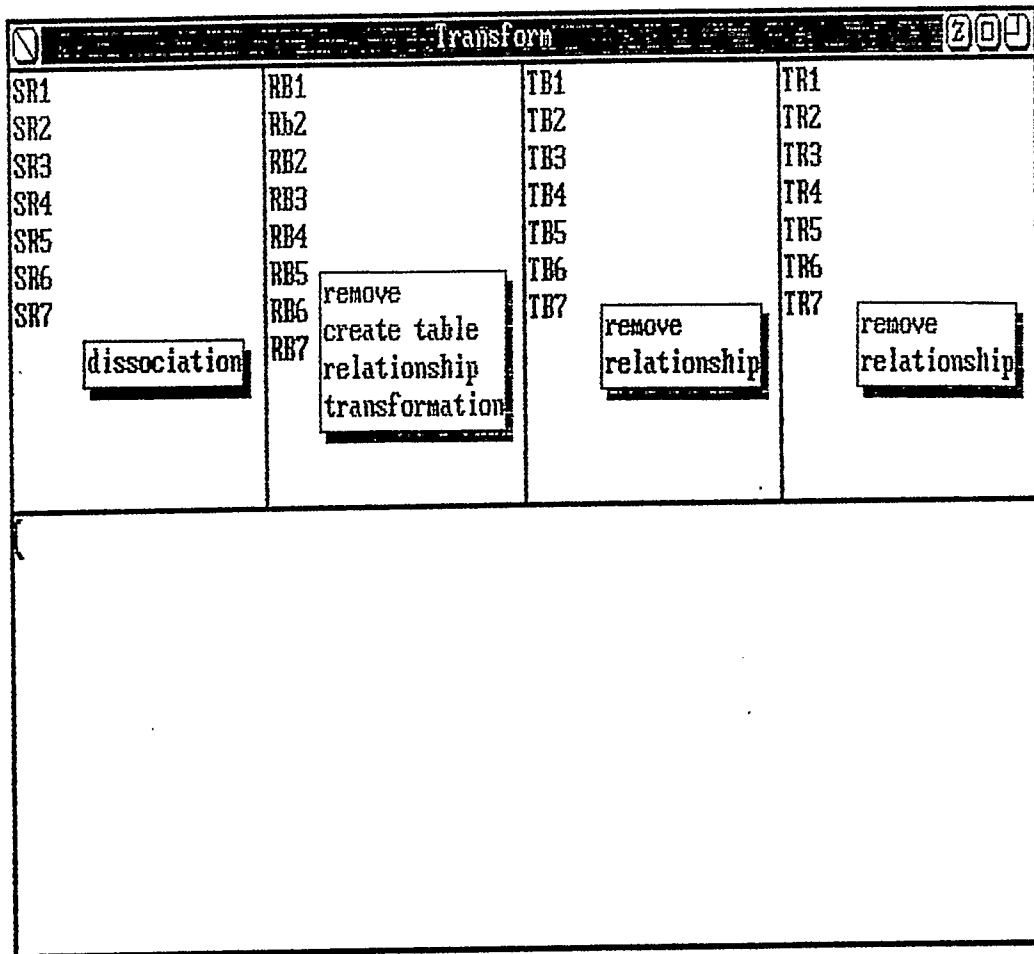


Figure 4.10. The Transform window with menus.

## CHAPTER 5

### ALGORITHM ILLUSTRATIONS

This chapter contains five illustrations. The first four illustrations demonstrate the algorithm's ability to handle various combinations of alarms in power substations. The fifth one compares the execution speed between this algorithm and the search algorithm used in Prolog.

#### 5.1 Background

Electricity is transmitted at very high voltage to minimize power losses. Part of the electricity is stepped-down to lower voltages at substations to feed local consumptions. Therefore, bus-bars and transformers form the skeletons for most substations. Normally, double bus-bar systems are employed in substations to prevent bus-bar malfunctioning from disrupting the electricity flow. Substations are interconnected by transmission lines. These apparatus are protected by circuit breakers. When an apparatus develops a fault, its associated circuit breakers trip to avoid damages.

Tests of the algorithm are performed on a 5-substation model. A schematic for the model is shown in figure 5.1. There is a pair of 240kV bus-bars in every substation. In addition, each of substations S1, S2, S3, and S5 also has a pair of 138kV bus-bars. Two transformers serve each of the above four substations. Each transformer is connected to a 240kV and a 138kV bus-bar. Transmission lines are also connected to the bus-bars, transmitting electricity both into and out



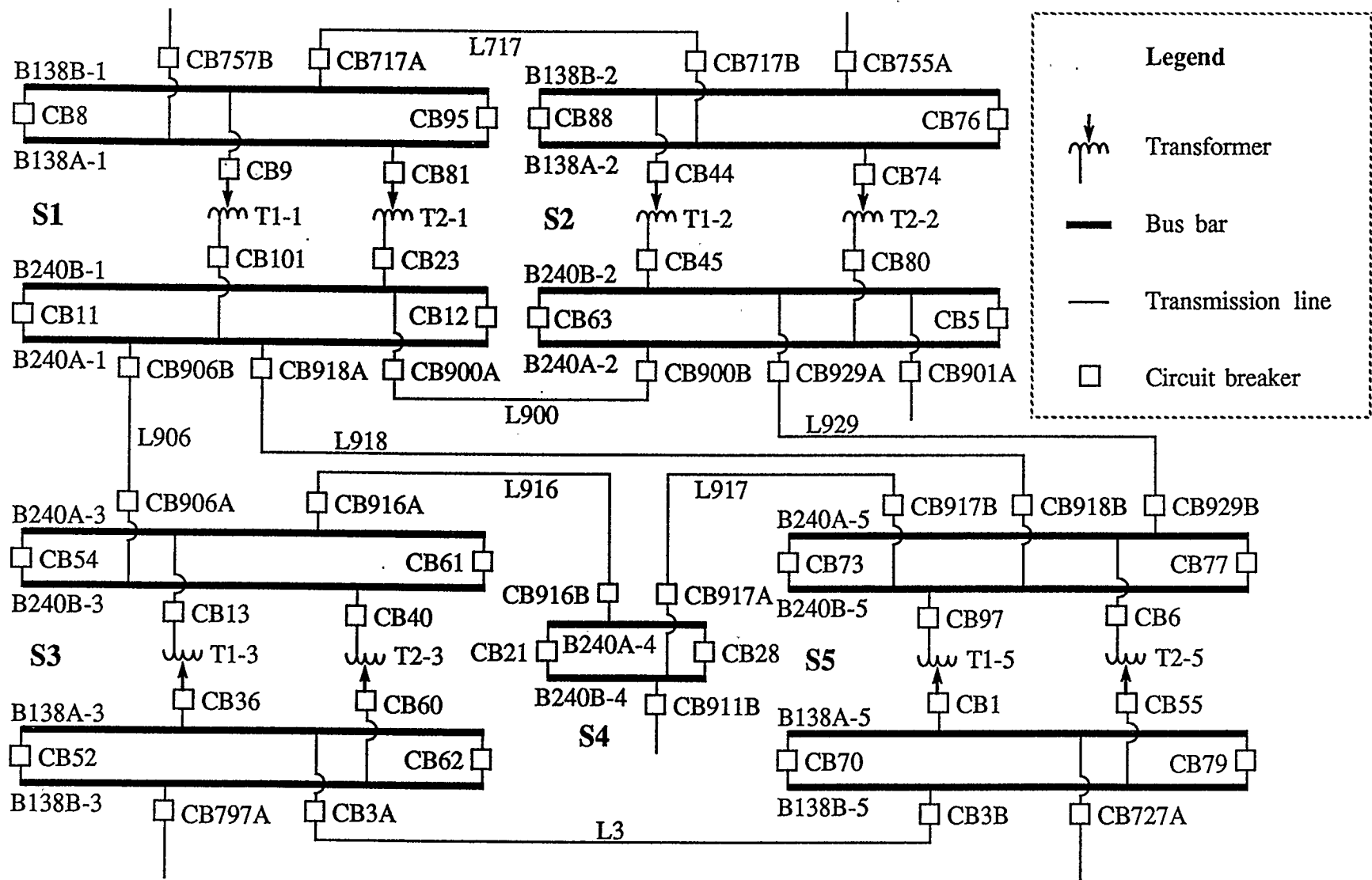


Figure 5.1. The substation schematic.

of the substations. Each of the bus-bars, transformers, and transmission lines is protected by several circuit breakers. Each device is given a name. Bus-bars' names start with the letter "B". Transformers' names start with the letter "T". Transmission lines' names start with the letter "L". And circuit breakers' names start with letters "CB".

Several assumptions are made when writing the production rules to handle alarms. Firstly, only alarms indicating that circuit breakers have tripped are considered. Secondly, when an apparatus develops a fault -- for example, a transmission line short-circuits -- all circuit breakers connected to it trip. And lastly, at most only one circuit breaker fails to trip when a fault occurs.

With the above assumptions, a total of 118 rules are created. The rules conceive the knowledge for most faults that can occur in the substations. Eighteen of the 118 rules deal with the 18 bus-bars in the five substations. Eight rules are related to the eight transformers in substations S1, S2, S3, and S5. Another eight rules are devoted to the eight transmission lines interconnecting the five substations. These 34 rules assume that all circuit breakers function normally, and that all circuit breakers connected to an apparatus trip when it develops a fault. For example, the following rule states that when a bus-bar is faulty, the four circuit breakers connected to it trip:

If CB8\_tripped and CB95\_tripped and CB757B\_tripped and  
CB81\_tripped then B138A-1\_fault

However, in reality, circuit breakers may fail to trip. Therefore, the remaining 84 rules are dedicated to situations where a circuit breaker does not trip when a fault occurs. For example, the following three rules describe the situation that a fault

occurs and a circuit breaker fails to trip:

If CB8\_tripped and CB95\_tripped and CB757B\_tripped and  
CB23\_tripped and !CB81\_tripped then possible\_B138A-1\_fault

If CB8\_tripped and CB95\_tripped and CB757B\_tripped and  
CB23\_tripped and !CB81\_tripped then CB81\_failstotrip

If CB8\_tripped and CB95\_tripped and CB757B\_tripped and  
CB23\_tripped and !CB81\_tripped then possible\_T2-1\_fault

Where "!" denotes the "not" logic operator. Note that in the above situation, it is logical to suggest that the fault occurs at either the bus-bar or the transformer because either situation, when combined with the failed-to-trip CB81, results in the tripping of the four circuit breakers. A list of the 118 rules can be found in Appendix C.

The 118 rules are subsequently dissociated into 184 simpler rules which do not contain the "or" logic operator among the conditions. The latter rules are then transformed into a search tree.

### 5.1.1 Single fault examples

The first example deals with the simple situation that a transmission line short-circuits. As shown in figure 5.2, transmission line L3 develops a fault. Subsequently, circuit breakers CB3A and CB3B trip. Two alarms indicating the tripping of circuit breakers are received. The alarms become facts and are stored into Example 1, an instance of SetOfFact. As shown in figure 5.3, Example 1 contains two facts: CB3A\_tripped and CB3B\_tripped. A search is conducted. As expected, the search concludes that transmission line L3 is faulty. The resulting solution

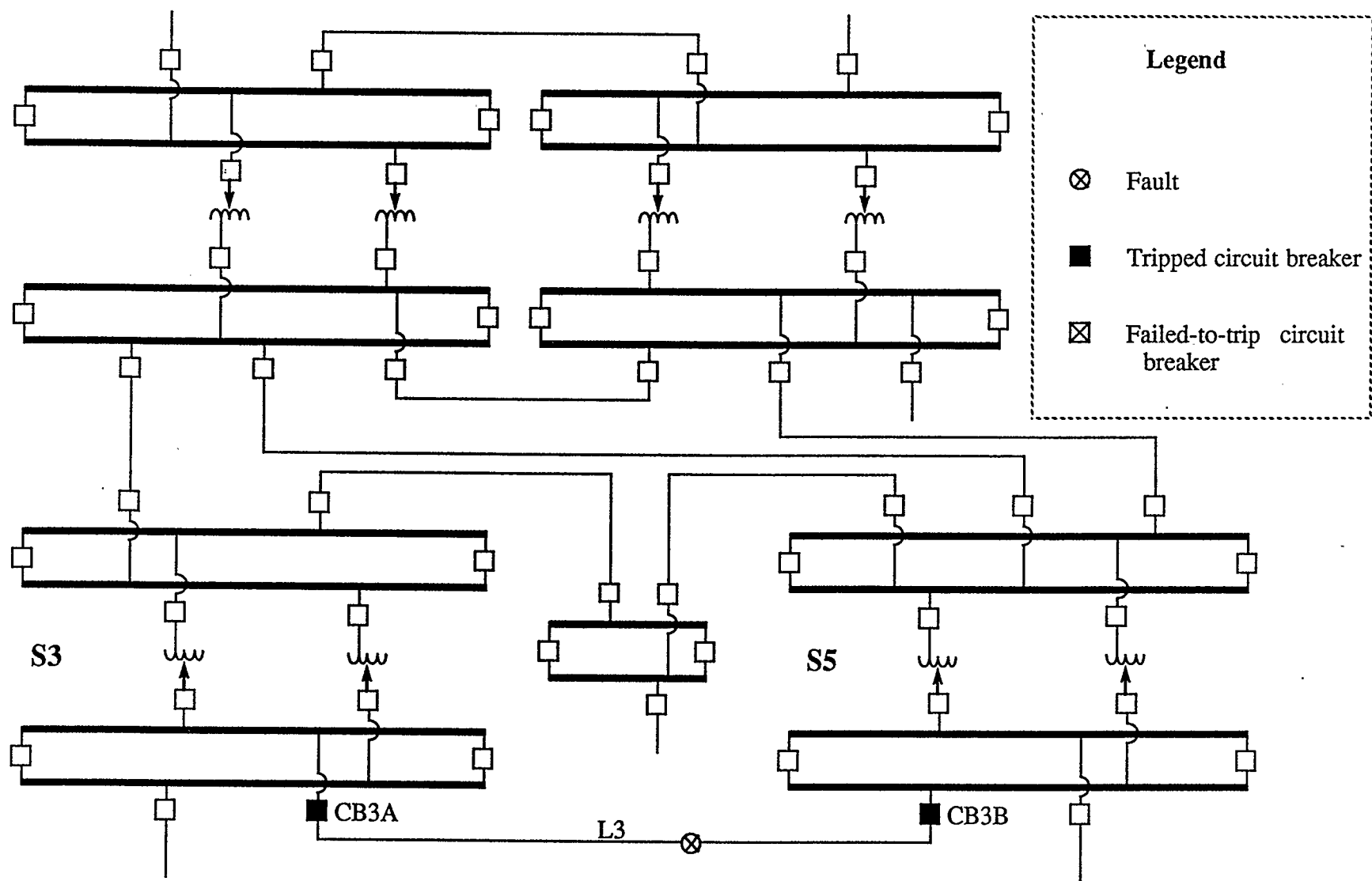


Figure 5.2. The substation schematic for the first example.

Edit		
RuleBases	Example 1	CB3A_tripped
SetOfFacts		CB3B_tripped
StringRules		
Tables		

Figure 5.3. Example 1 containing alarms for the first example.

path and explanation is shown in figure 5.4. In figure 5.4, the top left, top center, and top right subpanes show the available instances of Tree, SetOfFact, and Stack respectively. The solution path is stored on a stack. Content of the stack is displayed in the middle pane. It shows the sequence of tree nodes visited which leads to the solution:

CB3B\_tripped->CB3A\_tripped->L3\_fault

The explanation, shown in the bottom pane, indicates the rules involved:

If CB3A\_tripped and CB3B\_tripped then L3\_fault

and the facts used:

CB3B\_tripped, CB3A\_tripped.

In the second example, a fault occurs at bus-bar B138A-2, causing circuit breakers CB88, CB76, CB717B, and CB74 to trip. However, CB74 fails to trip, forcing CB80 to trip. The above can be seen in figure 5.5. The four alarms are received and stored into Example 2, another instance of SetOfFact. As seen in figure 5.6, four facts are present in Example 2: CB717B\_tripped, CB76\_tripped, CB80\_tripped, and CB88\_tripped. A search is subsequently performed. The results are shown in figure 5.7. The solution path is put into a stack, Solution 2. It shows that a circuit breaker fails to trip:

CB74\_failstotrip

and the fault occurs at either the bus-bar or the transformer:

possible\_B138A-2\_fault

possible\_T2-2\_fault

Again, the rules and facts involved are shown in the explanation in the bottom pane. Because of the limited space in the bottom pane, the full explanation is re-

Search		
Illustration	Example 1	Solution 1
CB3B_tripped->CB3A_tripped->L3_fault		
If CB3A_tripped and CB3B_tripped then L3_fault. And the following facts are true: CB3B_tripped CB3A_tripped.		

Figure 5.4. Solution paths and explanation for the first example.

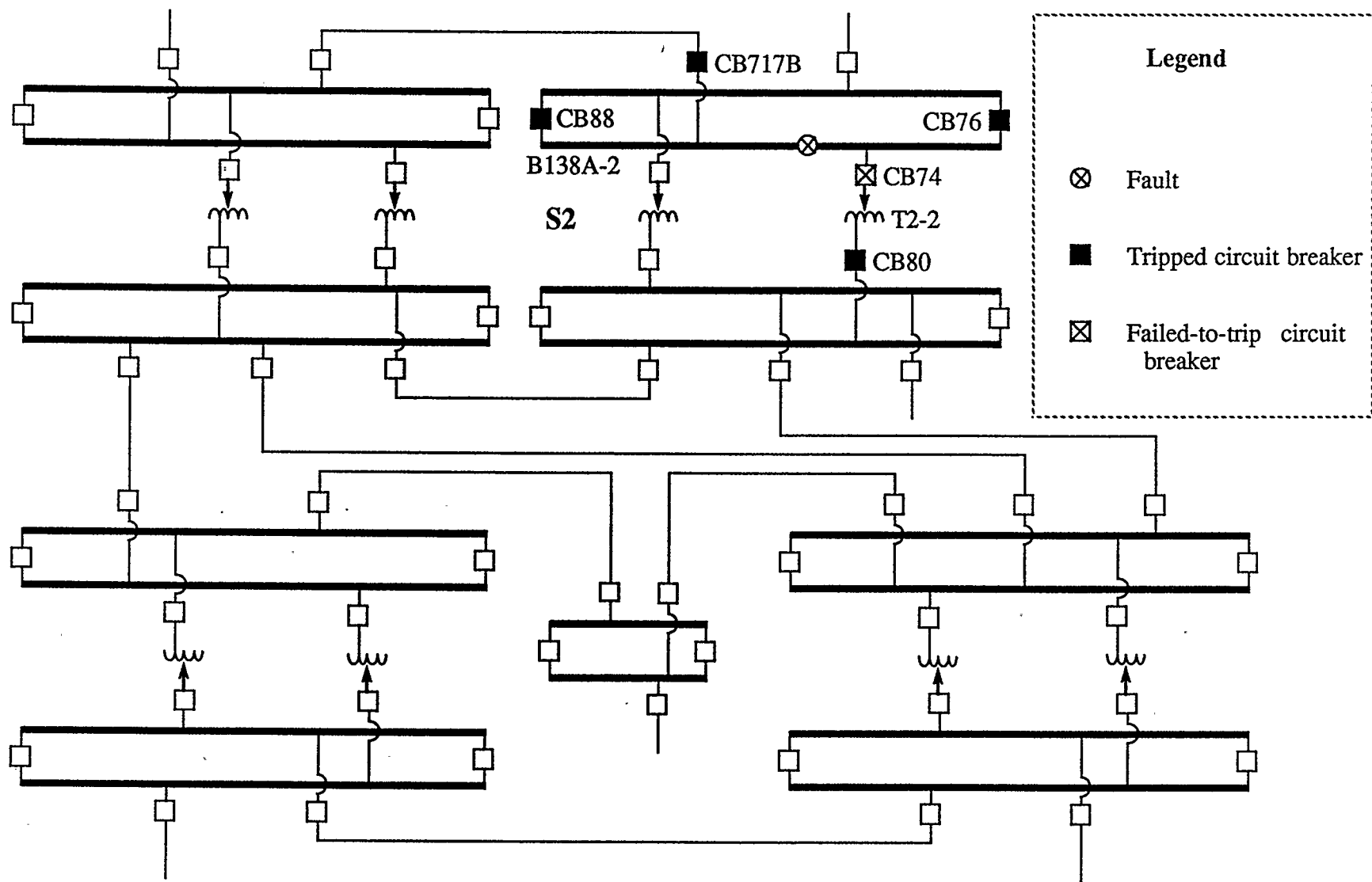


Figure 5.5. The substation schematic for the second example.



Edit		
RuleBases	Example 2	CB717B tripped
SetOfFacts		CB76 tripped
StringRules		CB88 tripped
Tables		CB88 tripped

Figure 5.6. Example 2 containing alarms for the second example.

Illustration	Example 2	Solution 2
		<p>CB76_tripped-&gt;CB88_tripped-&gt;CB717B_tripped-&gt;!CB74_tripped-&gt;CB88_tripped  -&gt;possible T2-2 fault</p> <p>CB76_tripped-&gt;CB88_tripped-&gt;CB717B_tripped-&gt;!CB74_tripped-&gt;CB88_tripped  -&gt;CB74_failstotrip</p> <p>CB76_tripped-&gt;CB88_tripped-&gt;CB717B_tripped-&gt;!CB74_tripped-&gt;CB88_tripped  -&gt;possible B138A-2_fault</p>
		<p>If CB88 tripped and CB76 tripped and CB717B tripped and CB88 tripped and  !CB74 tripped or (CB63 tripped and CB5 tripped and CB900B tripped and  CB74 tripped and !CB88 tripped) then possible T2-2 fault.</p> <p>And the following facts are true: CB76_tripped CB88_tripped CB717B_tripped  !CB74_tripped CB88_tripped.</p> <p>If CB88 tripped and CB76 tripped and CB717B tripped and CB88 tripped and  !CB74 tripped then CB74_failstotrip</p>

Figure 5.7. Solution paths and explanation for the second example.

printed in figure 5.8.

### 5.1.2 Multiple faults examples

The following two examples demonstrate the algorithm's ability to handle multiple faults. In the first example, faults occur in three substations. As figure 5.9 shows, faults occur on bus-bar B240B-1 in substation S1, transformer T2-3 in substation S3, and bus-bar B138A-5 in substation S5. Circuit breakers CB11, CB12, CB23, CB900A, CB40, CB60, CB70, CB79, CB1, and CB727A tripped. These alarms are received and stored in an instance of SetOfFact, Example 3, as shown in figure 5.10. A search provides the results in figure 5.11. The stack, Solution 3, contains three solution paths. As expected, they indicate that the sources of the alarms are B240B-1, T2-3, and B138A-5:

B240B-1\_fault

B138A-5\_fault

T2-3\_fault

Explanations in the bottom pane give the related rules and facts. Again, because of limited space, the full explanation is reprinted in figure 5.12.

The second example has two faults and two failed-to-trip circuit breakers. As figure 5.13 shows, a fault occurs on bus-bar B240B-3 in substation S3. Another fault occurs on transmission line L917. Circuit breakers CB906A and CB917A fail to trip. As a result, eight circuit breakers tripped: CB54, CB61, CB40, CB906B, CB917B, CB28, CB21, and CB911B. The alarms are stored in Example 4, as shown in figure 5.14. The search quickly identifies CB906A and CB917A to be the circuit breakers which failed to trip:

If CB88\_tripped and CB76\_tripped and CB717B\_tripped and  
 CB80\_tripped and !CB74\_tripped or (CB63\_tripped and  
 CB5\_tripped and CB900B\_tripped and CB74\_tripped and  
 !CB80\_tripped) then possible\_T2-2\_fault.  
 And the following facts are true: CB76\_tripped CB88\_tripped  
 CB717B\_tripped !CB74\_tripped CB80\_tripped.

If CB88\_tripped and CB76\_tripped and CB717B\_tripped and  
 CB80\_tripped and !CB74\_tripped then CB74\_failstotrip.  
 And the following facts are true: CB76\_tripped CB88\_tripped  
 CB717B\_tripped !CB74\_tripped CB80\_tripped.

If CB76\_tripped and CB717B\_tripped and CB755A\_tripped and  
 CB44\_tripped and CB74\_tripped and !CB88\_tripped or  
 (CB88\_tripped and CB717B\_tripped and CB755A\_tripped and  
 CB44\_tripped and CB74\_tripped and !CB76\_tripped) or  
 (CB88\_tripped and CB76\_tripped and CB717B\_tripped and  
 CB80\_tripped and !CB74\_tripped) or (CB88\_tripped and  
 CB76\_tripped and CB717A\_tripped and CB74\_tripped and  
 !CB717B\_tripped) then possible\_B138A-2\_fault.  
 And the following facts are true: CB76\_tripped CB88\_tripped  
 CB717B\_tripped !CB74\_tripped CB80\_tripped.

Figure 5.8. The reprint of the full explanation for the second example.

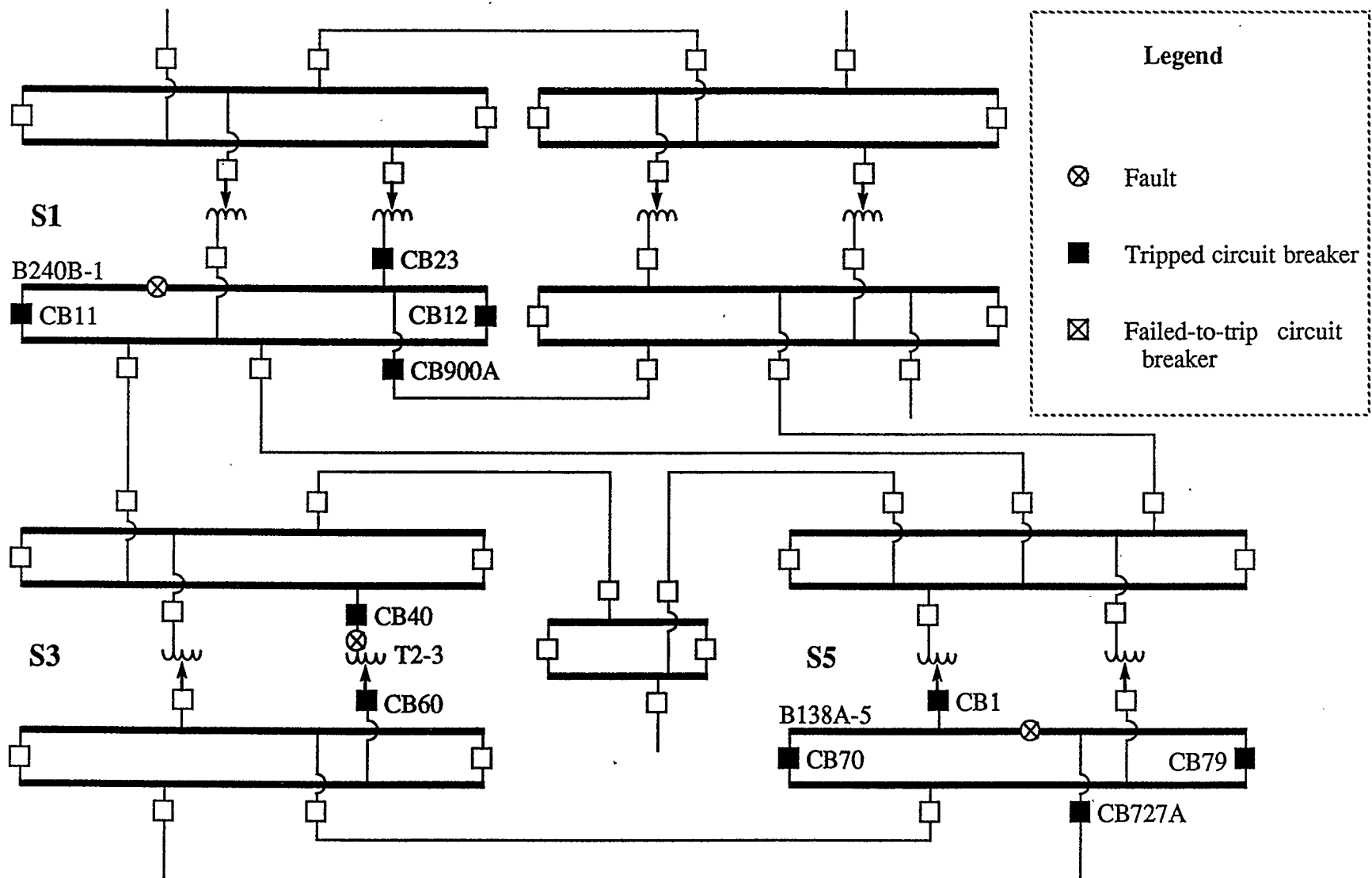


Figure 5.9. The substation schematic for the third example.

Edit		
RuleBases	Example 3	CB11 tripped
SetOfFacts		CB12 tripped
StringRules		CB1 tripped
Tables		CB23 tripped
		CB40 tripped
		CB60 tripped
		CB70 tripped
		CB727A tripped
		CB79 tripped
		CB900A tripped

Figure 5.10. Example 3 containing alarms for the third example.

Illustration	Example 3	Solution 3
CB12 tripped->CB11 tripped->CB900A tripped->CB23 tripped->B240B-1 fault CB79 tripped->CB70 tripped->CB1 tripped->CB727A tripped->B138A-5 fault CB60 tripped->CB40 tripped->T2-3 fault		
If CB11 tripped and CB12 tripped and CB23 tripped and CB900A tripped then B240B-1 fault. And the following facts are true: CB12 tripped CB11 tripped CB900A tripped CB23 tripped.  If CB70 tripped and CB79 tripped and CB1 tripped and CB727A tripped then B138A-5 fault. And the following facts are true: CB79 tripped CB70 tripped CB1 tripped		

Figure 5.11. Solution paths and explanation for the third example.

If CB11\_tripped and CB12\_tripped and CB23\_tripped and  
CB900A\_tripped then B240B-1\_fault.  
And the following facts are true: CB11\_tripped CB12\_tripped  
CB900A\_tripped CB23\_tripped.

If CB40\_tripped and CB60\_tripped then T2-3\_fault.  
And the following facts are true: CB60\_tripped CB40\_tripped.

If CB70\_tripped and CB79\_tripped and CB1\_tripped and  
CB727A\_tripped then B138A-5\_fault.  
And the following facts are true: CB1\_tripped CB79\_tripped  
CB727A\_tripped CB70\_tripped.

Figure 5.12. The reprint of the full explanation for the third example.



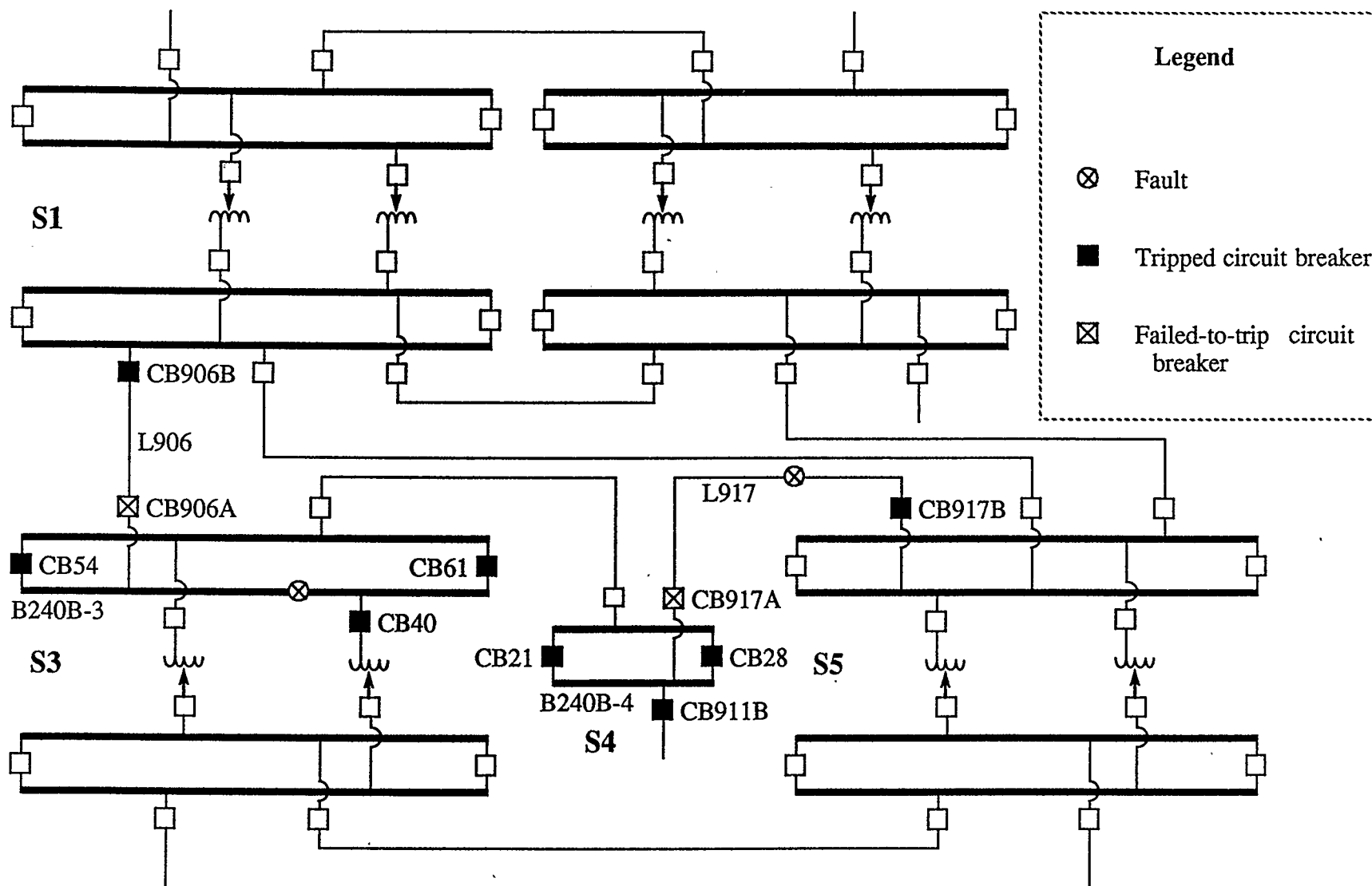


Figure 5.13. The substation schematic for the fourth example.

Edit		
RuleBases	Example 4	CB21 tripped
SetOfFacts		CB28 tripped
StringRules		CB40 tripped
Tables		CB54 tripped
		CB61 tripped
		CB906B tripped
		CB911B tripped
		CB917B tripped

Figure 5.14. Example 4 containing alarms for the fourth example.

CB906A\_failstotrip

CB917A\_failstotrip

and that the faults may have occurred at bus-bar B240B-3 or B240B-4, or transmission line L906 or L917:

possible\_B240B-3\_fault

possible\_B240B-4\_fault

possible\_L906\_fault

possible\_L917\_fault

The above results are stored in Solution 4. Its content is displayed in the middle pane in figure 5.15. The bottom pane in the figure contains the explanations of the results. Both the middle and bottom panes are too small to display the full results and explanations. Therefore, they are reprinted in figures 5.16 and 5.17, respectively.

## 5.2 Comparison

A Smalltalk version of Prolog, called Prolog/V, is available in the Smalltalk/V programming environment. It combines the characteristics of both Smalltalk/V and Prolog. Prolog/V is written in Smalltalk. Thus Smalltalk features such as class hierarchy, polymorphism, and inheritance can be found in Prolog/V[39]. The Prolog/V predicates are actually Smalltalk messages. They are all implemented in the class Prolog. New predicates can be added into the class. As a result, there is a substantial difference between Prolog/V and standard Prolog. However, they use the same searching algorithm. Prolog/V presents its solutions in a different way than standard Prolog. While the latter shows the solutions one

Illustration	Example 4	Solution 4
<p>CB54 tripped-&gt;CB61 tripped-&gt;!CB906A tripped-&gt;CB906B tripped-&gt;CB40 tripped -&gt;possible B240B-3 fault</p> <p>CB54 tripped-&gt;CB61 tripped-&gt;!CB906A tripped-&gt;CB906B tripped-&gt;CB40 tripped -&gt;possible L906 fault</p> <p>CB54 tripped-&gt;CB61 tripped-&gt;!CB906A tripped-&gt;CB906B tripped-&gt;CB40 tripped -&gt;CB906A failstotrip</p> <p>CB28 tripped-&gt;CB21 tripped-&gt;CB911B tripped-&gt;CB917B tripped-&gt;!CB917A tripped -&gt;possible I917 fault</p>		
<p>If CB61 tripped and CB906A tripped and CB916A tripped and CB13 tripped and CB40 tripped and !CB54 tripped or (CB54 tripped and CB906A tripped and CB916A tripped and CB13 tripped and CB40 tripped and !CB61 tripped) or (CB54 tripped and CB61 tripped and CB906B tripped and CB40 tripped and !CB906A tripped) or (CB54 tripped and CB61 tripped and CB60 tripped and CB906A tripped and !CB40 tripped) then possible B240B-3 fault.</p> <p>And the following facts are true: CB54 tripped CB61 tripped !CB906A tripped CB906B tripped CB40 tripped</p>		

Figure 5.15. Solution paths and explanation for the fourth example.

```

CB54_tripped->CB61_tripped->!CB906A_tripped->CB906B_tripped
->CB40_tripped->possible_B240B-3_fault
CB54_tripped->CB61_tripped->!CB906A_tripped->CB906B_tripped
->CB40_tripped->possible_L906_fault
CB54_tripped->CB61_tripped->!CB906A_tripped->CB906B_tripped
->CB40_tripped->CB906A_failstotrip
CB28_tripped->CB21_tripped->CB911B_tripped->CB917B_tripped
->!CB917A_tripped->possible_L917_fault
CB28_tripped->CB21_tripped->CB911B_tripped->CB917B_tripped
->!CB917A_tripped->possible_B240B-4_fault
CB28_tripped->CB21_tripped->CB911B_tripped->CB917B_tripped
->!CB917A_tripped->CB917A_failstotrip

```

Figure 5.16. The reprint of the full result for the fourth example.

If CB61\_tripped and CB906A\_tripped and CB916A\_tripped and CB13\_tripped and CB40\_tripped and !CB54\_tripped or (CB54\_tripped and CB906A\_tripped and CB916A\_tripped and CB13\_tripped and CB40\_tripped and !CB61\_tripped) or (CB54\_tripped and CB61\_tripped and CB906B\_tripped and CB40\_tripped and !CB906A\_tripped) or (CB54\_tripped and CB61\_tripped and CB60\_tripped and CB906A\_tripped and !CB40\_tripped) then possible\_B240B-3\_fault.  
 And the following facts are true: CB54\_tripped CB61\_tripped !CB906A\_tripped CB906B\_tripped CB40\_tripped.

If CB11\_tripped and CB12\_tripped and CB906A\_tripped and CB918A\_tripped and CB101\_tripped and !CB906B\_tripped or (CB54\_tripped and CB61\_tripped and CB906B\_tripped and CB40\_tripped and !CB906A\_tripped) then possible\_L906\_fault.  
 And the following facts are true: CB54\_tripped CB61\_tripped !CB906A\_tripped CB906B\_tripped CB40\_tripped.

If CB54\_tripped and CB61\_tripped and CB906B\_tripped and CB40\_tripped and !CB906A\_tripped then CB906A\_failstotrip.  
 And the following facts are true: CB54\_tripped CB61\_tripped !CB906A\_tripped CB906B\_tripped CB40\_tripped.

If CB21\_tripped and CB28\_tripped and CB917B\_tripped and CB911B\_tripped and !CB917A\_tripped or (CB73\_tripped and CB77\_tripped and CB97\_tripped and CB917A\_tripped and CB918B\_tripped and !CB917B\_tripped) then possible\_L917\_fault.  
 And the following facts are true: CB28\_tripped CB21\_tripped CB911B\_tripped CB917B\_tripped !CB917A\_tripped.

If CB28\_tripped and CB916B\_tripped and CB917A\_tripped and CB911B\_tripped and !CB21\_tripped or (CB21\_tripped and CB916B\_tripped and CB917A\_tripped and CB911B\_tripped and !CB28\_tripped) or (CB21\_tripped and CB28\_tripped and CB917B\_tripped and CB911B\_tripped and !CB917A\_tripped) then possible\_B240B-4\_fault.  
 And the following facts are true: CB28\_tripped CB21\_tripped CB911B\_tripped CB917B\_tripped !CB917A\_tripped.

If CB21\_tripped and CB28\_tripped and CB917B\_tripped and CB911B\_tripped and !CB917A\_tripped then CB917A\_failstotrip.  
 And the following facts are true: CB28\_tripped CB21\_tripped CB911B\_tripped CB917B\_tripped !CB917A\_tripped.

Figure 5.17. The reprint of the full explanation for the fourth example.

at a time, the former presents all solutions at the same time.

Comparison is made on the searching speed between the proposed algorithm and Prolog/V's search algorithm. This comparison is fair as both algorithms are written in Smalltalk, and both give complete solutions.

### 5.2.1 Illustration

The comparison procedure is as follows.

- (1) A set of production rules is created. For simplicity, the conditions and goals of the rules consist of at most three alphabets.
- (2) The rules are added into the appropriate rulebases for the two algorithms.
- (3) A search tree is constructed for the proposed algorithm.
- (4) Five sets of facts are created and stored.
- (5) Five searches are subsequently conducted with each algorithm using the set of rules. Each search uses one of the five sets of facts. The five searches fall into three categories, namely, Single, Multiple, and Recursive. The Single category focuses on situations where only one rule is involved, such as the first example in section 5.1.1. The Multiple category focuses on situations where several rules are involved. However, these rules bear no relationship among each other. The Recursive category focuses on situations where recurrence occurs, i.e., the proving of a rule requires the proving of another rule which requires the proving of another rule ..., and so on. Each of the last two categories includes two searches, which are: control, and worst-case. In the control search, the same sets of facts are used in each comparison. This shows the effect of the number of rules on the

search speed. In the worst-case search, the longest time required to reach a solution in the category is selected.

(6) Times needed to find the solutions are compared.

In this illustration, five comparisons are performed. In other words, steps (1) to (6) are repeated five times. In each comparison, more rules are involved. In the first comparison, only three rules are created. In the second one, seven more rules are added. Ten additional rules are created for each of the remaining three comparisons. The time needed in each search is put into table 5.1. The rules used in the comparisons are listed in Appendix D.

As table 5.1 shows, the Prolog search algorithm takes more time to reach a solution as more rules are involved than the proposed algorithm. In most cases, the number of rules has little effect on the proposed algorithm's search time. However, the Prolog search algorithm's search time increases significantly as the number of rules increases. The most dramatic results can be seen in the last row of the table. In that row, Prolog's search time increases more than 250 times from the first to the fifth comparison, whereas its counterpart has increased only by a factor of 13.2. In short, the proposed algorithm is faster than Prolog's search algorithm.



Time in ms			Comparison (Number of rules)				
			First (3)	Second (10)	Third (20)	Fourth (30)	Fifth (40)
Category	Single		110 (~0)	330 (50)	990 (50)	2140 (50)	3240 (60)
	Multiple	control	110 (50)	500 (50~60)	1320 (50~60)	2750 (50~60)	4010 (60)
		worst	110 (50)	500 (50~60)	1490 (60)	3570 (60)	5000 (60)
	Recursive	control	110 (50)	390 (50~60)	1100 (50~60)	2310 (50~60)	3460 (50~60)
		worst	110 (50)	500 (50~60)	3840 (220)	12410 (550)	28340 (660)

Table 5.1. Search times using the proposed algorithm (in brackets) and Prolog/V.

## CHAPTER 6

### CONCLUSIONS

As the demand for electrical power increases, larger and more complex power systems are designed and built. Existing power systems are also expanded and upgraded. Accompanying this trend towards larger power systems is the need for Energy Management Systems (EMSs) capable of not only managing the existing power systems, but also accommodating future expansions and modifications. As power systems become more complex, it becomes more difficult for operating personnel to maintain proper control of a power system when disturbances occur. ES has been introduced as a solution. An EMS incorporating an ES has been proposed by the Power Research Group of the Department of Electrical Engineering.

#### 6.1 General Conclusions

Every ES has an inference engine. It is a mechanism which infers conclusion(s) on the existing facts using knowledge. This knowledge always exists in extremely large quantities (otherwise, the human brain would be able to handle the knowledge and there would be no need for ES). The inference engine is basically a searching mechanism which locates the applicable knowledge in the pool of available knowledge. The search mechanisms used in ESs in the power area are slow.

In this thesis, *a fast searching algorithm is developed*, which has several

advantages over other algorithms used in the power area. The advantages are as follows:

(1) *The search algorithm separates the tree construction and search processes.* The separation results in improved response as the search process becomes independent of the construction process.

(2) *The separation also eliminates repeated construction* of the same partial search trees.

(3) *The key condition feature* adopted by the algorithm enables the search tree to be constructed in a way that *facilitates rapid elimination of nonapplicable production rules* during search. Hence, the search process is both efficient and fast.

The proposed EMS is large. Prototyping such a large software using procedural language consumes a disproportionately large amount of time. Instead, an Object-Oriented Programming (OOP) language is proposed. OOP has several advantages over procedural programming. First of all, OOP encourages *reuse of program codes*, thus reducing time and effort required to develop and maintain application software. Secondly, OOP *enforces modular programming*. The enforcement is achieved through the basic definition of objects -- that objects encapsulate both data and methods manipulating these data. Modular programming simplifies the task of writing application software by simply dividing the software into modules. In this case, each module is a class. Lastly, OOP models the application. OOP *simulates the relations among the various entities in the application*. Therefore, it is easier to fine-tune the software to the user's requirement.

Since the proposed algorithm will become part of the EMS software, the al-

gorithm is implemented in an OOP language -- Smalltalk. A windowing interface is also constructed to allow the algorithm to be run and tested. *Comparisons* between the fast searching algorithm and Prolog/V's searching algorithm, which is the most commonly used search algorithm in the power area, *show* that the former *reaches a solution in less time* than the latter. The time difference magnifies significantly as the number of rules involved increases. The fast searching algorithm also shows enormous *reduction in time needed for a recursive search*, especially when the recurrence is intense. Illustrations also demonstrate the algorithm's ability to *serve as an alarm-handler/fault-diagnostician*. Besides stating the search results, the alarm-handler/fault-diagnostician also provides explanations of the results.

## 6.2 Future Work

Several improvements and follow-up works are suggested in the following.

Firstly, a more efficient parser should be constructed. The parsing algorithm currently used in the dissociation process is written for the sole purpose of parsing. Efficiency of the parsing algorithm has not been considered. Optimizing the parsing algorithm would save time, and result in a faster dissociation process.

Secondly, the range of logic operators accepted by the algorithm should be expanded. In its present form, the algorithm accepts only "and", "or", and "not" logic operators. By accepting a wider range of operators, the algorithm becomes more versatile and hence user-friendly.

Thirdly, the algorithm and the windowing interface should be incorporated into the proposed EMS. Modifications should be made so that the search algo-

rithm should be activated both manually by the operator and automatically by the EMS. Changes should also be made to automatize the dissociation and transformation processes.

Fourthly, the knowledge base should be expanded to accommodate all the knowledge necessary to perform the proposed EMS's expert system tasks outlined in chapter 2.

Fifthly, a conflict resolving mechanism should be added to the proposed algorithm. It resolves possible conflicts within the knowledge supplied by the expert, or in the solutions suggested by the search algorithm.

And finally, a subclass should be created for search trees of power substations. Since substations have similar characteristics, their search trees should be similar. Therefore, the trees should be grouped to form a subclass.

## REFERENCE

- [1] D. Soulier, "The Hydro-Quebec System Blackout of March 31, 1989", IEEE Power Engineering Review, Vol. 9, No. 10, Oct. 1989, pp. 17-18.
- [2] H. Schildt, "Artificial Intelligence Using C", Osborne McGraw-Hill, Berkeley, California, USA, 1987.
- [3] Z. Z. Zhang, G. S. Hope, O. P. Malik, "Expert Systems in Electric Power Systems -- A Bibliographical Survey", IEEE/PES, 1989, Winter Meeting, Paper No.: 89 WM 212-2.
- [4] D. Thomas, "What's in an Object?", BYTE magazine, March 1989, pp. 231-240.
- [5] J. Hsu, J. Kusnan, "The Fifth Generation: The Future of Computer Technology", Windcrest Books, USA, 1989, pp. 162-167.
- [6] G. De Montravel, "A Real Time Expert System for Alarm Handling in the Future EDF Regional Control Centres", CIGRE SC39, Sept. 1986.
- [7] R. Fujiwara, Y. Kohno, T. Sakaguchi, H. Suzuki, "An Intelligent Load Flow Engine for Power System Planning", IEEE Transactions on Power Systems, Vol. PWRS-1, No. 3, Aug. 1986, pp. 302-307.
- [8] E. Cardozo, T. Perry, S. N. Talukdar, "The Operator's Assistant -- An Intelligent, Expandable Program for Power System Trouble Analysis", IEEE Transactions on Power Systems, Vol. PWRS-1, No. 3, Aug. 1986, pp. 182-187.
- [9] P. Wegner, "Learning the Language", BYTE magazine, March 1989, pp.

245-253.

- [10] C. V. Ramamoorthy, P. C. Sheu, "Object-Oriented Systems", IEEE Expert, Fall 1988, pp. 9-15.
- [11] R. Wheatley, B. J. Frey, G. S. Hope, O. P. Malik, "Object Based Expert Systems in Integrated Power System Analysis", Proceedings of Canadian Conference on Electrical and Computer Engineering, Montreal, Quebec, Canada, Sept. 1989, pp. 1059-1062.
- [12] W. Kim, F. H. Lochovsky, "Object-Oriented Concepts, Database, and Applications", ACM Press, New York, New York, 1989, pp. 127-197.
- [13] R. Wheatley, T. Hui, "Proposal for Integrated Energy Management System Development Toolkit", Department of Electrical Engineering, May 1989.
- [14] B. F. Wollenberg, "Feasibility Study for an Energy Management System Intelligent Alarm Processor", Proceedings 1985 Power Industry Computer Applications Conference, San Francisco, California, pp. 249-254.
- [15] B. J. Cox, Object-Oriented Programming -- An Evolutionary Approach, Addison-Wesley Publishing Company, USA, 1986.
- [16] N. C. Rowe, "Artificial Intelligence Through Prolog", Prentice Hall, Englewood Cliffs, New Jersey, USA, 1988.
- [17] Y. Akimoto, D. B. Klapper, W. W. Price, H. Tanaka, K. A. Wirgau, J. Yoshizawa, "Transient Stability Expert System", IEEE Transactions on Power Systems, Vol. 4, No. 1, Feb. 1989, pp. 312-320.
- [18] T. Sakaguchi, "Development of a Knowledge Based System for Power System Restoration", IEEE Transactions on Power Apparatus and Systems, Vol. PAS-102, No. 2, Feb. 1983.

- [19] E. N. Dialynas, A. V. Machias, C. A. Protopapas, "An Expert System Approach to Designing and Testing Substation Grounding Grids", IEEE Transactions on Power Delivery, Vol. 4, No. 1, Jan. 1989, pp. 234-240.
- [20] C. Fukui, J. Kawakami, "An Expert System for Fault Section Estimation Using Information from Protective Relays and Circuit Breakers", IEEE Transactions on Power Delivery, Vol. PWRD-1, No.4, Oct. 1986.
- [21] B. D. Russell, K. Watson, "Power Substation Automation Using a Knowledge Based System -- Justification and Preliminary Field Experiments", IEEE Transactions on Power Delivery, Vol. PWRD-2, No. 4, Oct. 1987, pp. 1090-1097.
- [22] S. J. Lee, C. C. Liu, S. S. Venkata, "An Expert System Operational Aid for Restoration and Loss Reduction of Distribution Systems", IEEE Transactions on Power Systems, Vol. 3, No. 2, May 1988, pp. 619-626.
- [23] K. Nara, S. Yamashiro, Y. Yanaura, "Application of an Expert System to Decisions on Countermeasures Against Snow Accretion on Transmission Lines", IEEE Transactions on Power Systems, Vol. 3, No. 3, Aug. 1988, pp. 1052-1058.
- [24] F. Gubina, M. Mihelcic, A. Ogorelec, "An Approach to Power Network Fault Location Diagnosis", Proceedings of Symposium on Expert Systems Application to Power Systems, Stockholm-Helsinki, Aug. 1988.
- [25] A. J. Germond, D. Niebur, L. Palmieri, M. Stalder, "A Rule Based System for Substation Monitoring: The Switching Operation", Proceedings of Symposium on Expert Systems Application to Power Systems, Stockholm-Helsinki, Aug. 1988.



- [26] H. E. Dijk, G. P. T. Roeiofs, A. W. van der Weegen, "An Expert System for Power System Restoration", Proceedings of Symposium on Expert Systems Application to Power Systems, Stockholm-Helsinki, Aug. 1988.
- [27] S. J. Cheng, O. P. Malik, G. S. Hope, "An Expert System for Voltage and Reactive Power Control of a Power System", IEEE Transactions on Power Systems, Vol. 3, No. 4, Nov. 1988, pp. 1449-1455.
- [28] A. A. Girgis, M. B. Johns, "A Hybrid Expert System for Faulted Section Identification, Fault Type Classification and Selection of Fault Location Algorithms", IEEE Transactions on Power Delivery, Vol. 4, No. 2, Apr. 1989, pp. 978-985.
- [29] J. L. Chen, Y. Y. Hsu, "An Expert System for Load Allocation in Distribution Expansion Planning", IEEE Transactions on Power Delivery, Vol. 4, No. 3, Jul. 1989, pp. 1910-1918.
- [30] C. C. Liu, K. Tomsovic, "An Expert System Assisting Decision -- Making of Reactive Power/Voltage Control", IEEE Transactions on Power Systems, Vol. PWRS-1, No. 3, Aug. 1986, pp. 195-201.
- [31] P. Ackerman, C. C. Liu, S. Pope, K. Tomsovic, "An Expert System As a Dispatchers' Aid for the Isolation of Line Section Faults", IEEE Transactions on Power Delivery, Vol. PWRD-2, No. 3, Jul. 1987, pp. 736-743.
- [32] E. Cardozo, S. N. Talukdar, "A Distributed Expert System for Fault Diagnosis", IEEE Transactions on Power Systems, Vol. 3, No. 2, May 1988, pp. 641-646.
- [33] S. F. Borys, R. T. Goeltz, K. M. Hemmelman, S. L. Purucker, R. D. Rasmussen, B. E. Tonn, "Communication Alarm Processor Expert System",

Proceedings of Symposium on Expert Systems Application to Power Systems, Stockholm-Helsinki, Aug. 1988.

- [34] D. Lubkeman, T. Taylor, "Applications of Knowledge-based Programming to Power Engineering Problems", IEEE Transactions on Power Systems, Vol. 4, No. 1, Feb. 1989, pp. 345-352.
- [35] C. C. Liu, K. Tomsovic, S. Zhang, "Efficiency of Expert System As On Line Operating Aids", Proceedings of the 9th Power Systems computation Conference, Cascais, Portugal, Sept. 1987, pp. 695-701.
- [36] C. L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence, Vol. 19, 1982.
- [37] O. L. Madsen, B. Moller-Pedersen, "What object-oriented programming may be -- and what it does not have to be", proceedings of ECOOP '88 European Conference on Object-Oriented Programming, Oslo, Norway, Aug. 1988.
- [38] L. J. Pinson, R. S. Wiener, "An Introduction to Object-Oriented Programming and Smalltalk", Addison-Wesley Publishing Company, USA, 1988.
- [39] "Prolog.doc", a documentary file in the Smalltalk/V software, Digitalk Inc., CA, USA.

## APPENDIX A

### The new classes and their implementations

Implementation of the new classes is discussed in the following sections. Each of the six new classes is described in terms of the four Smalltalk properties: abstraction, encapsulation, inheritance, and polymorphism. The data protocol and part of the function protocol, i.e. data and messages, are presented and explained. A complete list of the messages can be found in Appendix B. The convention used in section 4.2.2 is adopted in this appendix.

#### New class -- RuleBases

**Abstraction --** A rulebase is a collection of production rules, each of which has two parts: a goal, and the conditions. In each rule, all the conditions are operands of only one logic operator -- "and". Quite often, the goals also serve as indexes through which the rules are extracted from and inserted into the rulebase. Each rulebase is a result of the dissociation of the set of stringrules entered by users. The rulebases may then be transformed into search trees. Each rulebase also has a name and carries information about the related set of stringrules.

**Encapsulation --** The above abstraction is encapsulated into the following protocol. The goal and conditions of each rule are put into two different sets. One set contains an object which is the goal, while

the other set contains objects which are the conditions.

Data -- All data are private. There is no shared and global shared data.

name -- An object identifies the name of the rulebase.

stringrule -- An object identifies the instance of StringRules from which the rulebase is created.

Messages -- The following is a partial list of the messages for the class. A detail listing is given in Appendix B.

transformWith: aTable -- Answers a search tree constructed from the receiver and the argument. The argument, aTable, is an instance of class Table.

nameSR: aString -- Stores the name of the instance of StringRules from which the receiver is created. The name, aString, is an instance of class String.

srName -- Answers the name of the instance of StringRules from which the receiver is created.

findKey: aSet value: aSet -- Answers the key in the receiver whose key/value pair contains the same objects, respectively, as the argument set. Both arguments are instances of class Set.

Inheritance -- The new class is implemented as a subclass of class Dictionary, because the latter allows data to be manipulated through an index -- the key. Class Dictionary's data and messages are inherited. Some of the messages are polymorphically redefined.

Data -- The following are the inherited data.

contents -- An array containing the objects stored in the instance of class Dictionary.

elementCount -- The number of objects stored in the instance of class Dictionary.

Messages -- All except three messages are inherited and used as is. A list of the messages is given in Appendix B under class Dictionary. The three redefined messages -- values, add:, at: put: -- are mentioned later in this section.

Polymorphism -- Since both the keys and values in an instance of RuleBases are sets of objects while the keys and values in an instance of Dictionary can be any objects, so the two messages which insert key/value pairs are polymorphically redefined. Besides, the message returning a bag containing all the values is also redefined.

add: anAssociation -- Add an association, which is a key/value pair, into the receiver. Both the key and value must be instances of class Set. If not, an error message is displayed.

at: aKey put: aSet -- Add the key/value -- aKey/aSet -- pair into the receiver. If aKey already exists in the receiver, its associated value is replaced by aSet. Both aKey and aSet must be instances of class Set. If not, an error message is displayed.

values -- Answers a set containing the contents in each value set of all the key/value pairs.

### New class -- SetOfFact

Abstraction -- A setoffact is basically a set containing facts. There is no difference between a setoffact and an ordinary set, except that the former has a name. The name is a character string which can be used to distinguish the different setoffacts.

Encapsulation -- The above abstraction is encapsulated into the following protocol.

Data -- The data is private. There is no shared and global shared data.

name -- An object identifies the name of the instance of SetOfFact.

Messages -- Only two messages have been created for this class. The following is a description of the two messages. A detail listing is given in Appendix B.

nameSelf: aString -- Gives the receiver a name. The name, aString, is an instance of class String.

selfName -- Answers the name of the receiver.

Inheritance -- The new class is implemented as a subclass of class Set. All data and message protocol are inherited. There is no polymorphically redefined message. The following is the data and part of the messages inherited. A complete list of messages is given in Appendix B.

Data -- The following are the inherited data protocol.

contents -- An array containing the objects stored in the instance of class Set.

elementCount -- The total number of objects stored in the instance of class Set.

Messages -- The following are some of the inherited functional protocol.

sameAs: aSet -- Answers true if the receiver and the argument contain identical objects, else answers false.

subsetOf: aSet -- Answers true if the receiver is a subset of the argument, else answers false.

union: aSet -- Answers a set of objects which is the union of the receiver and the argument.

Polymorphism -- There is no polymorphically redefined message.

### **New class -- Stack**

Abstraction -- A stack is a collection of objects in an ordered manner. The objects are manipulated in the last-in-first-out sequence. In other words, new objects are always put on the top of the stack, and only the top object can be removed from the stack. Since the stacks are created primarily to store solution paths of searches, each stack should carry information regarding the search itself, such as: the set storing the facts, the search tree involved, the origin of the tree, and so on. Each stack also has a name.

Encapsulation -- The above abstraction is encapsulated into the following protocol.

- Data -- All data are private. There is no shared and global shared data.
- name -- An object identifies the name of the stack.
  - tree -- An object identifies the search tree involved.
  - setoffact -- An object identifies the set storing the facts.
  - rulebase -- An object identifies the rulebase from which the search tree is constructed.
  - stringrule -- An object identifies the instance of StringRules from which the rulebase is created.
  - table -- An object identifies the weight table related in the construction of the search tree.
- Messages -- The following are explanations of several messages for the class. A detail listing can be found in Appendix B.
- emptyStack -- Answers true if the receiver is an empty stack, else answers false.
  - push: anObject -- Places an object on top of the stack.
  - pop -- Answers the object removed from the top of the stack.  
If the stack is empty, answers nil.
  - nameTr: aString -- Stores the name of the search tree related to the receiver. The name, aString, is an instance of class String.
  - trName -- Answers the name of the search tree related to the receiver.
- Inheritance -- Since a stack collects objects in an ordered fashion, so the new class is implemented as a subclass of class OrderedCollection.



All the data and messages of class `OrderedCollection` are inherited by subclass `Stack`. However, because of the limited choice of stack operations, many of the inherited messages are useless. The following list the inherited data and some messages.

A complete list of messages can be found in Appendix B.

Data -- The following are the inherited data protocol.

`contents` -- An array containing the objects in the collection.  
`endPosition` -- The position index of the last object in the collection.  
`startPosition` -- The position index of the first object in the collection.

Messages -- The following are the inherited functional protocol.

`size` -- Answers the number of objects in the receiver.  
`includes: anObject` -- Checks if the receiver contains the argument. Answers a Boolean value.  
`do: aBlock` -- For every object in the receiver collection, performs aBlock.  
`copyFrom: beginning to: end` -- Answers a new collection which contains only the objects between beginning and end of the receiver.

Polymorphism -- Many inherited messages become useless and have to be redefined due to the fact that stacks can only be popped and pushed. The following outlines messages which are polymorphically redefined.

all add messages -- All nine messages which intend to

	add an object into the collection are redefined to display error messages when used.
all remove messages --	All four messages for removing objects from the collection are redefined to display error messages when used.
all retrieve messages --	The after:, at:, before: messages which retrieve objects anywhere in the collection are also redefined to display error messages when used.
replace and concatenate messages --	Both the replaceFrom: to: and , messages are redefined to display error messages when used.

### **New class -- StringRules**

Abstraction --	A stringrule is a collection of production rules written in a different format. Each rule is divided into two parts: the goal in the form of a character string, and all the conditions and logic operators in another character string. In other words, each rule is represented by two character strings. This is the original form of representation of a production rule entered by users. The goals also serve as indexes through which the manipulation of rules is performed. Dissociation may then be performed on each string.
----------------	---

grule to create a rulebase. Each stringrule also carries a name.

Encapsulation -- The above abstraction is encapsulated into the following protocol.

Data -- The data is private. There is no shared and global shared data.

name -- An object identifies the name of the instance of StringRules.

Messages -- The following is a partial list of the messages for the class. A detail listing can be found in Appendix B.

dissociate -- Answers an instance of RuleBases which is created from the dissociation of the receiver.

nameSelf: aString -- Gives the receiver a name. The name, aString, is an instance of class String.

selfName -- Answers the name of the receiver.

Inheritance -- Since manipulation of the rules is performed through referring to the goals, so the new class is implemented as a subclass of Dictionary. Except for two messages, all the data and messages protocol of class Dictionary are inherited. The followings are the descriptions of the inherited data and messages.

Data -- The following are the inherited data.

contents -- An array containing the objects stored in the instance of class Dictionary.

elementCount -- The number of objects stored in the instance of class Dictionary.

Messages -- All except two messages are inherited. A list of the messages can be found in Appendix B under class Dictionary. The two re-

defined messages -- add:, at: put: -- are mentioned later in this section.

**Polymorphism --** Since both the keys and values in an instance of StringRules are character strings, so the two messages which add key/value pairs into a stringrule must be redefined to meet the requirement.

**add: anAssociation --** Add an association -- a key/value pair -- into the receiver. Both the key and value must be instances of class String. Otherwise, an error message is displayed.

**at: aKey put: aString --** Add the key/value -- aKey/aString -- pair into the receiver. If aKey already exists, its associated value is replaced by aString. Both the key and value must be instances of class String. If not, an error message is displayed.

### **New class -- Tables**

**Abstraction --** A table is simply a collection of data pairs. Each data pair contains a name and a numerical value. The name in the data pair is the means through which the data pair can be modified. Each table should carry information about its related rulebase and the instance of StringRules. Each table also has a name.

**Encapsulation --** The above abstraction is encapsulated into the following protocol.

- Data -- All data are private. There is no shared and global shared data.
- name -- An object identifies the name of the table.
- rulebase -- An object identifies the related rulebase.
- stringrule -- An object identifies the instance of StringRules from which the rulebase is created.
- Messages -- The following is a partial list of the messages for the class. A detail listing can be found in Appendix B.
- newWeightAt: aString to: aNumber -- Assigns a new numerical value to the data pair whose name matches the first argument. The name, aString, is an instance of class String, and the new value, aNumber, is an instance of class Number.
- maximizeWeightAt: aString to: aNumber -- Assigns a maximum value to the data pair whose name matches the first argument. The maximum value is selected between the original value in the data pair and the second argument.
- nameRB: aString -- Stores the name of the rulebase related to the receiver.

The name, aString, is an instance of class String.

rbName --

Answers the name of the related rulebase.

Inheritance -- The new class is implemented as a subclass of class Dictionary. The data and most of the messages of class Dictionary are inherited and used as is. Only two of the inherited messages are polymorphically redefined, which are discussed later in this section. The following are the inherited protocol.

Data -- The following are the inherited data.

contents -- An array containing the objects stored in the instance of class Dictionary.

elementCount -- The number of objects stored in the instance of class Dictionary.

Messages -- All except two messages are inherited and used as is. A list of the messages can be found in Appendix B under class Dictionary. The two redefined messages -- add:, at: put: -- are mentioned later in this section.

Polymorphism -- Since the values in all key/value pairs must be numbers, so the two messages which add key/value pairs must be polymorphically redefined.

add: anAssociation -- Add a key/value pair into the receiver. The value must be an instance of class Number. Otherwise, an error message is displayed.

at: aKey put: aNumber -- Add the aKey/aNumber pair into the receiver. If aKey already exists, its associated value is replaced by aNumber. The second argument, aNumber, must be an instance of class Number. If not, an error message is displayed.

### **New class -- Tree**

Abstraction -- In a common perception of a tree, there are three kinds of nodes. The root is the origin of the tree and has only children nodes. It has no parent or sibling nodes. A leaf node has parent and may be sibling nodes, but no children nodes. A tree node lies between the root and the leaf nodes, and has both parent and children nodes. It may also has sibling nodes.

A second kind of perception of a tree, which is as popular as the previous perception and is adopted in this abstraction, is that a tree is itself a subtree, and every subtree is composed of other subtrees. Each subtree has a root and subtree(s) connected to the root.

A special kind of subtree exists which has only root and no subtrees. This is equivalent to the leaf node. The subtrees in each subtree maintain an order of precedence. The order exists during the lifetime of the tree, and is not modified.

Each tree has a name and should also carry information regarding its construction, i.e., the stringRule, ruleBase, and table re-

lated to the construction. A tree may subsequently be involved in searching.

Encapsulation -- The subtree perception is encapsulated into the following protocol.

Data -- All data are private. There is no shared and global shared data.

name -- An object identifies the name of the tree.

root -- An object identifies the root of a subtree.

rulebase -- An object identifies the rulebase from which the tree is constructed.

stringrule -- An object identifies the instance of StringRules from which the rulebase is created

table -- An object identifies the weight table related in the construction of the tree.

Messages -- The following are explanations of several messages for the class. A detail listing can be found in Appendix A.

searchWith: aSetOfFact -- Answers a stack storing the solution paths which are results from the searching of the receiver using the argument. The argument is an instance of class SetOfFact.

nameTb: aString -- Stores the name of the related table. The name, aString, is an instance of class String.

tbName -- Answers the name of the related weight table.

Inheritance -- Because of the order of precedence of subtrees inside a subtree, the new class is implemented as a subclass of class Or-



deredCollection. All the data and messages of class OrderedCollection are inherited by subclass Tree. Partial description of the inherited data and messages can be found in the section describing the new class Stack. A complete list of messages can be found in Appendix B.

Polymorphism -- There is no polymorphically redefined message.

## APPENDIX B

### Listing of class protocol

The following sections list the class data and methods. The listing includes both the new classes -- RuleBases, SetOfFact, Stack, StringRules, Table, and Tree -- and their immediate super classes -- Dictionary, Set, and OrderedCollection. Discussions on how the four Smalltalk properties -- Abstraction, Encapsulation, Inheritance, and Polymorphism -- are preserved in the new classes are placed in Appendix A.

#### Super class -- Dictionary

```
Set subclass: #Dictionary
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

#### Dictionary methods

```
add: anAssociation
  "Answer anAssociation. Add anAssociation to the
  receiver."
  | index element |
  index := self findKeyIndex: anAssociation key.
  (element := contents at: index) == nil
    ifTrue: [
      elementCount := elementCount + 1.
      contents at: index put: anAssociation]
    ifFalse: [element value: anAssociation value].
  self adjustSize.
  ^anAssociation
```

```
associationAt: aKey
```

```

    "Answer the Association whose key equals aKey
    from the receiver. If not found,
    report an error."
  ^self
    associationAt: aKey
    ifAbsent: [self errorAbsentKey]

associationAt: aKey ifAbsent: aBlock
    "Answer the Association whose key equals aKey
    from the receiver. If not found,
    evaluate aBlock (with no arguments)."
  | answer |
  ^(answer := self lookUpKey: aKey) == nil
    ifTrue: [aBlock value]
    ifFalse: [answer]

associationsDo: aBlock
    "Answer the receiver. For each key/value pair
    in the receiver, evaluate aBlock with that
    pair as the argument."
  super do: aBlock

at: aKey
    "Answer the value of the key/value pair whose key
    equals aKey from the receiver. If
    not found, report an error."
  | answer |
  ^(answer := self lookUpKey: aKey) == nil
    ifTrue: [self errorAbsentKey]
    ifFalse: [answer value]

at: aKey ifAbsent: aBlock
    "Answer the value of the key/value pair whose key
    equals aKey from the receiver. If not found,
    evaluate aBlock (with no arguments)."
  | answer |
  ^(answer := self lookUpKey: aKey) == nil
    ifTrue: [aBlock value]
    ifFalse: [answer value]

at: aKey put: anObject
    "Answer anObject. If the receiver contains
    the key/value pair whose key equals aKey,

```

```

        replace the value of the pair with anObject.
        Else add the aKey/anObject pair."
    self add: (Association key: aKey value: anObject).
    ^anObject

```

```

deepCopy
    "Answer a copy of the receiver with shallow
    copies of each element."
    | answer |
    answer := self species new.
    self associationsDo: [:element |
        answer add: element copy].
    ^answer

```

```

do: aBlock
    "Answer the receiver. For each value
    in the receiver, evaluate aBlock with
    that value as the argument."
    super do: [ :association |
        aBlock value: association value]

```

```

duplicate
    "Answer a copy of the receiver with
    duplicate copies of each element.
    Receiver -- a Dictionary.
    Return -- a copy of the receiver."
    | answer |
    answer := self species new.
    self associationsDo: [:element |
        answer add: element deepCopy].
    ^answer

```

```

errorAbsentKey
    "Private - Report an error to the effect
    that the desired key was not found in the
    receiver."
    self error: 'Key is missing'

```

```

findKeyIndex: aKey
    "Private - Answer the index position of the
    key/value pair in the receiver whose key equals
    aKey or the index of the first empty position
    where such an pair would be stored."

```

```

| index answer anAssociation |
index := contents size.
answer := (aKey hash) \\ index + 1.
[((anAssociation := contents at: answer) isNil)
 or: [aKey = anAssociation key]]
  whileFalse: [
    (answer := answer + 1) > index
    ifTrue: [answer := 1]]. "wrap-around"
^answer

grow
  "Private - Answer the receiver doubled in
  size to accomodate more key/value
  pairs."
  | aDictionary |
  aDictionary := self class new: contents size * 2.
  self associationsDo: [ :anAssociation |
    aDictionary add: anAssociation].
  contents := aDictionary contents

includes: anObject
  "Answer true if the receiver contains the
  key/value pair whose value equals anObject,
  else answer false."
  self do: [ :element |
    element = anObject
    ifTrue: [^true]].
  ^false

includesKey: aKey
  "Answer true if the receiver contains aKey,
  else answer false."
  ^(self lookUpKey: aKey) notNil

inspect
  "Open a dictionary inspector window
  on the receiver."
  DictionaryInspector new openOn: self

keyAtValue: anObject
  "Answer the key in the receiver whose paired value
  equals anObject. If not found, answer nil."
  ^self keyAtValue: anObject ifAbsent: [nil]

```

```

keyAtValue: anObject ifAbsent: aBlock
    "Answer the key in the receiver whose paired value
    equals anObject. If not found, evaluate aBlock
    (with no arguments)."
```

```

    self associationsDo: [ :anAssociation |
        anAssociation value = anObject
            ifTrue: [^anAssociation key]].
    ^aBlock value
```

```

keys
    "Answer a Set containing all the keys
    in the receiver."
    | answer |
    answer := Set new: self size * 2.
    self associationsDo: [ :assoc |
        answer add: assoc key].
    ^answer
```

```

keysDo: aBlock
    "Answer the receiver. For each key
    in the receiver, evaluate aBlock with
    the key as the argument."
    self associationsDo: [ :anAssociation |
        aBlock value: anAssociation key]
```

```

lookUpKey: aKey
    "Private - Answer the association
    in the receiver whose key equals
    aKey or nil if it doesn't exist."
    | index limit anAssociation |
    limit := contents size.
    index := (aKey hash) \\ limit + 1.
    [((anAssociation := contents at: index) == nil)
    or: [aKey = anAssociation key]]
    whileFalse: [
        (index := index + 1) > limit
            ifTrue: [index := 1]. "wrap-around"
    ]
    ^anAssociation
```

```

occurrencesOf: anObject
    "Answer the number of key/value pairs in the receiver,
    whose values are equal to anObject."
    | answer |
```

```

answer := 0.
self do: [ :element |
    element = anObject
        ifTrue: [answer := answer + 1]].
^answer

rehashFrom: anInteger
    "Private - Rehash the keys of the receiver
    from the index position anInteger
    to the last index position."
| index size searchIndex anAssociation |
size := contents size.
index := anInteger.
[(index := index + 1) > size
    ifTrue: [index := 1]. "wrap-around"
(anAssociation := contents at: index) isNil]
whileFalse: [ "test next assoc for relocation"
    searchIndex := self findKeyIndex: anAssociation key.
    (contents at: searchIndex) isNil
        ifTrue: [ "found assoc to move"
            contents
                at: searchIndex
                put: anAssociation.
            contents
                at: index
                put: nil]]

remove: anObject ifAbsent: aBlock
    "Remove the key/value pair whose value is anObject
    from the receiver dictionary. This method
    reports an error since the values are
    not unique in a dictionary, the keys are."
^self invalidMessage

removeAssociation: anAssociation
    "Answer the receiver after anAssociation has been
    removed from it. If anAssociation is not in the
    receiver, report an error."
    self removeKey: anAssociation key

removeKey: aKey
    "Answer the receiver with the key/value pair whose
    key equals aKey removed. If such a pair is not found,
```

```

        report an error."
    self removeKey: aKey ifAbsent: [self errorAbsentKey].

removeKey: aKey ifAbsent: aBlock
    "Answer aKey. Remove the key/value pair whose key
    equals aKey from the receiver. If such a pair
    is not found, evaluate aBlock (with no arguments)."
    | index |
    index := self findKeyIndex: aKey.
    (contents at: index) == nil
        ifTrue: [^aBlock value].
    contents at: index put: nil.
    elementCount := elementCount - 1.
    self rehashFrom: index.
    ^aKey

select: aBlock
    "For each key/value pair in the receiver, evaluate
    aBlock with the value part of the pair as the argument.
    Answer a new object containing those key/value pairs
    for which aBlock evaluates to true."
    | answer |
    answer := self species new.
    self associationsDo: [ :each |
        (aBlock value: each value)
            ifTrue: [answer add: each]].
    ^answer

shallowCopy
    "Answer a copy of the receiver which shares
    the receiver elements."
    | answer |
    answer := self species new.
    self associationsDo: [:element |
        answer add: element].
    ^answer

storeOn: aStream
    "Append the ASCII representation of the
    receiver to aStream from which the
    receiver can be reinstantiated."
    | firstTime |
    firstTime := true.

```



```

aStream
  nextPutAll: '(';
  nextPutAll: self class name;
  nextPutAll: ' new)'.
self associationsDo: [ :assoc |
  firstTime
    ifFalse: [aStream nextPut: $;].
  aStream
    cr;
    nextPutAll: 'add: ('.
  assoc storeOn: aStream.
  aStream nextPut: $).
  firstTime := false].
firstTime
  ifFalse: [aStream nextPutAll: ';yourself'].
aStream nextPut: $)

```

```

values
  "Answer a Bag containing all the values of the
  key/value pairs in the receiver."
  | answer |
  answer := Bag new.
  self associationsDo: [ :assoc |
    answer add: assoc value].
  ^answer

```

### Super class -- OrderedCollection

```

IndexedCollection subclass: #OrderedCollection
instanceVariableNames:
  'startPosition endPosition contents '
classVariableNames: ''
poolDictionaries: ''

```

### OrderedCollection class methods

```

new
  "Answer an instance of OrderedCollection
  capable of holding 12 elements initially."
  ^self new: 12

```

```

new: anInteger
  "Answer an initialized instance of OrderedCollection

```

capable of holding anInteger number of elements."  
 ^((super new) initPositions: anInteger

OrderedCollection methods

, aCollection

"Answer an OrderedCollection containing all  
 the elements of the receiver followed by  
 all the elements of aCollection."

^self copy  
 addAll: aCollection;  
 yourself

add: anObject

"Answer anObject. Add anObject after the  
 last element of the receiver collection."

endPosition = contents size  
 ifTrue: [self putSpaceAtEnd].  
 endPosition := endPosition + 1.  
 contents at: endPosition put: anObject.  
 ^anObject

add: newObject after: oldObject

"Answer newObject. Insert newObject immediately after  
 the element oldObject in the receiver collection. If  
 oldObject is not in the collection, report an error."

| index |  
 index := 1.  
 [index <= self size]  
 whileTrue: [  
 oldObject = (self at: index)  
 ifTrue: [^self add: newObject afterIndex: index].  
 index := index + 1].  
 ^self errorAbsentElement

add: anObject afterIndex: anInteger

"Answer anObject. Insert anObject at index position  
 anInteger + 1 in the receiver collection. If anInteger  
 is out of the collection bounds, report an error."

self putSpaceAfter: anInteger.  
 ^self at: anInteger + 1 put: anObject

add: newObject before: oldObject

"Answer newObject. Insert newObject immediately before the element oldObject in the receiver collection. If oldObject is not in the collection, report an error."

```
| index |
index := 1.
[index <= self size]
  whileTrue: [
    oldObject = (self at: index)
    ifTrue: [^self add: newObject beforeIndex: index].
    index := index + 1].
^self errorAbsentElement
```

add: anObject beforeIndex: anInteger

"Answer anObject. Insert anObject at index position anInteger - 1 in the receiver collection. If anInteger is out of the collection bounds, report an error."

```
self putSpaceAfter: anInteger - 1.
^self at: anInteger put: anObject
```

addAllFirst: aCollection

"Answer aCollection. Add all the elements contained in aCollection to the receiver before its first element."

```
| index |
index := aCollection size.
[index <= 0]
  whileFalse: [
    self addFirst: (aCollection at: index).
    index := index - 1].
^aCollection
```

addAllLast: aCollection

"Answer aCollection. Add all the elements contained in aCollection to the receiver after its last element."

```
| index size |
size := aCollection size.
index := 1.
[index <= size]
  whileTrue: [
    self addLast: (aCollection at: index).
    index := index + 1].
^aCollection
```

```

addFirst: anObject
    "Answer anObject. Add anObject before
    the first element of the receiver."
    startPosition = 1
    ifTrue: [self putSpaceAtStart].
    startPosition := startPosition - 1.
    contents at: startPosition put: anObject.
    ^anObject

addLast: anObject
    "Answer anObject. Add anObject after
    the last element of the receiver."
    endPosition = contents size
    ifTrue: [self putSpaceAtEnd].
    endPosition := endPosition + 1.
    contents at: endPosition put: anObject.
    ^anObject

after: anObject
    "Answer the element that immediately follows
    anObject in the receiver collection. If anObject
    is not an element of the receiver, report an error."
    ^self
    after: anObject
    ifNone: [^self errorAbsentElement]

after: anObject ifNone: aBlock
    "Answer the element that immediately follows
    anObject in the receiver collection. If anObject
    is not an element of the receiver, aBlock is
    evaluated (with no arguments)."
    | index |
    index := startPosition.
    [index < endPosition]
    whileTrue: [
        anObject = (contents at: index)
        ifTrue: [^contents at: index + 1].
        index := index + 1].
    ^aBlock value

at: anInteger
    "Answer the element of the receiver at index
    position anInteger. If anInteger is an invalid

```

```

    index for the receiver collection, report an error."
| index |
index := anInteger + startPosition - 1.
(startPosition <= index and: [index <= endPosition])
  ifFalse: [
    ^self errorInBounds: anInteger].
^contents at: index

```

```

at: anInteger put: anObject
    "Answer anObject. Replace the element of the
    receiver at index position anInteger with the
    anObject. If anInteger is an invalid index
    for the receiver collection, report an error."
| index |
index := anInteger + startPosition - 1.
(startPosition <= index and: [index <= endPosition])
  ifFalse: [
    ^self errorInBounds: anInteger].
contents at: index put: anObject.
^anObject

```

```

before: anObject
    "Answer the element that immediately precedes
    anObject in the receiver collection. If anObject
    is not an element of the receiver, report an error."
^self
  before: anObject
  ifNone: [^self errorAbsentElement]

```

```

before: anObject ifNone: aBlock
    "Answer the element that immediately precedes
    anObject in the receiver collection. If anObject
    is not an element of the receiver, aBlock is
    evaluated (with no arguments)."
| index |
index := startPosition + 1.
[index <= endPosition]
  whileTrue: [
    anObject == (contents at: index)
      ifTrue: [^contents at: index - 1].
    index := index + 1].
^aBlock value

```

```

copyFrom: beginning to: end
    "Answer an OrderedCollection containing the
      elements of the receiver from index position
      beginning through index position end."
    | answer |
    (answer := self species new: self size)
      startPosition: 1
      endPosition: end - beginning + 1.
    ^answer
    replaceFrom: 1
      to: end - beginning + 1
      with: self
      startingAt: beginning

distributeFrom: start to: stop with: anArray
    "Distribute the contents in anArray evenly among
      the receiver between start and stop.
      Receiver -- an OrderedCollection.
      1st argument -- the beginning.
      2nd argument -- the end.
      3rd argument -- an array.
      Return -- number in the receiver accepted content
        in anArray."
    |period index|
    index := start.
    period := (stop - start + 1) // anArray size.
    anArray do: [:eachSet |
        period timesRepeat: [
            (self at: index) addAll: eachSet.
            index := index + 1]].
    ^index - start

do: aBlock
    "Answer the receiver. For each element in the receiver,
      evaluate aBlock with that element as the argument."
    | index |
    index := startPosition - 1.
    [(index := index + 1) <= endPosition]
      whileTrue: [aBlock value: (contents at: index)]

duplicate
    "Answer an OrderedCollection containing the
      elements of the receiver."

```

```

    Receiver -- an OrderedCollection.
    Return -- a copy of the receiver."
|answer|
answer := self species new: self size.
self do: [:element |
    answer add: element].
^answer

errorAbsentElement
    "Private - Produce a walkback to the effect
    that the desired object was not in the collection."
    ^self error:
        'attempt to access absent element'

grow
    "Private - Answer the receiver expanded in
    size to accomodate more elements."
    self growTo: contents size + self growSize

growTo: anInteger
    "Private - Answer the receiver expanded
    to accomodate anInteger number of elements."
    | aCollection |
    aCollection := Array new: anInteger.
    aCollection
        replaceFrom: startPosition
        to: endPosition
        with: contents
        startingAt: startPosition.
    contents := aCollection

includes: anObject
    "Answer true if the receiver contains an element
    equal to anObject, else answer false."
    | index |
    index := startPosition - 1.
    [(index := index + 1) > endPosition]
        whileFalse: [
            anObject = (contents at: index)
            ifTrue: [^true]].
    ^false

initPositions: anInteger

```

```

    "Private - Answer the receiver after initializing
    it to be an empty OrderedCollection with
    anInteger number of slots."
    startPosition := 1.
    endPosition := 0.
    contents := Array new: anInteger

multiplyFrom: start to: stop with: anArray
    "Multiply the contents in anArray with the receiver
    between start and stop.
    Receiver -- an OrderedCollection.
    1st argument -- the beginning.
    2nd argument -- the end.
    3rd argument -- an array.
    Return -- the multiplied receiver."
    lendl
    end := self selfMultiplyFrom: start to: stop times: (anArray size - 1).
    ^self distributeFrom: start to: end with: anArray

putSpaceAfter: anInteger
    "Private - Answer the receiver with room for an
    element immediately after index position anInteger."
    | index |
    endPosition = contents size
    ifTrue: [self putSpaceAtEnd].
    anInteger = 0
    ifTrue: [
        startPosition = 1
        ifTrue: [self putSpaceAtStart].
        startPosition := startPosition - 1.
        ^self].
    endPosition := endPosition + 1.
    index := self size - 1.
    [index > anInteger]
    whileTrue: [
        self
        at: index + 1
        put: (self at: index).
        index := index - 1]

putSpaceAtEnd
    "Private - Answer the receiver with room for more
    elements following the last element of the collection."

```



```

| size index start |
startPosition = 1
ifTrue: [^self grow]
ifFalse: [
    size := self size.
    start := startPosition // 2.
    index := 0.
    [index < size]
    whileTrue: [
        contents
        at: start + index
        put: (contents at: startPosition + index).
        index := index + 1].
    startPosition := start.
    endPosition := startPosition + size - 1].
index := contents size.
[index > endPosition]
whileTrue: [
    contents at: index put: nil.
    index := index - 1]

```

putSpaceAtStart

"Private - Answer the receiver with room for more  
elements before the first element of the collection."

```

| size index end |
endPosition = contents size
ifTrue: [self grow].
size := self size.
end := contents size + endPosition + 1 // 2.
index := 0.
[index < size]
whileTrue: [
    contents
    at: end - index
    put: (contents at: endPosition - index).
    index := index + 1].
startPosition := end - size + 1.
endPosition := end.
index := 1.
[index < startPosition]
whileTrue: [
    contents at: index put: nil.
    index := index + 1]

```

```

remove: anObject ifAbsent: aBlock
    "Answer anObject. Remove the element anObject from
    the receiver collection. If anObject is not an
    element of the receiver, aBlock is evaluated
    (with no arguments)."
    | index |
    index := startPosition.
    [index <= endPosition]
        whileTrue: [
            anObject = (contents at: index)
                ifTrue: [
                    self removeIndex: index.
                    ^anObject].
            index := index + 1].
    ^aBlock value

removeFirst
    "Remove and answer the first element of the receiver.
    If the collection is empty, report an error."
    | answer |
    startPosition > endPosition
        ifTrue: [^self errorAbsentElement].
    answer := contents at: startPosition.
    contents at: startPosition put: nil.
    startPosition := startPosition + 1.
    ^answer

removeIndex: anInteger
    "Answer the receiver. Remove the element of the receiver
    at index position anInteger. If anInteger is an invalid
    index for the receiver, report an error."
    | index |
    (anInteger between: startPosition and: endPosition)
        ifFalse: [^self errorAbsentElement].
    index := anInteger.
    [index < endPosition]
        whileTrue: [
            contents
                at: index
                put: (contents at: index + 1).
            index := index + 1].
    contents at: endPosition put: nil.
    endPosition := endPosition - 1

```

```

removeLast
    "Remove and answer the last element of the receiver.
    If the collection is empty, report an error."
    | answer |
    startPosition > endPosition
        ifTrue: [^self errorAbsentElement].
    answer := contents at: endPosition.
    contents at: endPosition put: nil.
    endPosition := endPosition - 1.
    ^answer

replaceFrom: start to: stop with: aCollection
    "Answer a new OrderedCollection containing the
    receiver whose elements at index position start
    through stop have been replaced by the elements
    of aCollection."
    | finalSize size index |
    size := aCollection size.
    finalSize := self size + size -
        (stop - start + 1).
    finalSize > contents size
        ifTrue: [self growTo: finalSize + (finalSize // 3 + 10)].
    self
        startPosition: start
        endPosition: contents size.
    self
        replaceFrom: start + size
        to: finalSize
        with: self
        startingAt: stop + 1.
    self
        replaceFrom: start
        to: start + size - 1
        with: aCollection
        startingAt: 1.
    self
        startPosition: start
        endPosition: start + finalSize - 1.
    index := endPosition + 1.
    [index <= contents basicSize]
        whileTrue: [
            contents at: index put: nil.
            index := index + 1].

```

^self

```
selfMultiplyFrom: start to: stop times: anInteger
    "Expand the receiver between start and stop
    anInteger times.
    Receiver -- an OrderedCollection.
    1st argument -- the beginning.
    2nd argument -- the end.
    3rd argument -- number of times of expansion.
    Return -- the end position of the receiver."
    length
    index := stop + 1.
    length := stop - start + 1.
    (length * anInteger) timesRepeat: [
        self add: (self at: (index - length)) shallowCopy.
        index := index + 1].
    ^index - 1
```

```
size
    "Answer the number of elements contained by
    the receiver collection."
    ^endPosition - (startPosition - 1)
```

```
startPosition: start endPosition: end
    "Private - Answer the receiver. Set the position
    of the first and last elements of the receiver,
    to the arguments start and stop respectively."
    startPosition := start.
    endPosition := end
```

### **New class -- RuleBases**

```
Dictionary subclass: #RuleBases
    instanceVariableNames:
        'name stringRule '
    classVariableNames: ''
    poolDictionaries: ''
```

### **RuleBases methods**

```
add: anAssociation
    "Add anAssociation to the receiver. Both the key
    and value must be instances of Set. If not, an
```

error will be displayed.  
 Receiver -- a RuleBases.  
 Argument -- a key/value pair.  
 Return -- the key/value pair."

```
(anAssociation key class = Set)
  ifTrue: [(anAssociation value class = Set)
    ifTrue: [^super add: anAssociation]
    ifFalse: [^self errorValueNotSet]]
  ifFalse: [^self errorKeyNotSet]
```

at: aKey put: aSet  
 "Add the aKey/aSet pair into the receiver. If aKey already exists in the receiver, its associated value will be replaced by aSet. Both aKey and aSet must be instances of class Set. If not, an error message will be displayed.  
 Receiver -- a RuleBases.  
 1st argument -- the key.  
 2nd argument -- the value.  
 Return -- the value."

```
(aKey class = Set)
  ifTrue: [(aSet class = Set)
    ifTrue: [^super at: aKey put: aSet]
    ifFalse: [^self errorValueNotSet]]
  ifFalse: [^self errorKeyNotSet]
```

attachLeafTo: tree  
 "Finds the leaf node from the receiver and attach it to the argument one at a time.  
 Receiver -- a RuleBases.  
 Argument -- a Tree.  
 Return -- no return."

```
!node subtree!
node := self findLeaf.
[node = nil]
  whileFalse: [
    subtree := Tree new.
    subtree nameNode: node.
    tree add: subtree.
    node := self findLeaf]
```

condIsGoal  
 "Find the goal that is a subgoal."

```

    Receiver -- a RuleBases.
    Return -- the subgoal in a set or nil."
|conds goals|
conds := self values.
goals := self keys.
goals do: [:goal |
    (goal subSetOf: conds)
    ifTrue: [^goal]].
^nil

createWeightTable
    "Create a weight table.
    Receiver -- a RuleBases.
    Return -- the new table."
|table|
table := Table new.
(self values) do: [:cond |
    table newWeightAt: cond to: 0].
^table

duplicateAssocAt: aKey times: anInteger
    "Duplicate an association at aKey for anInteger
    times.
    Receiver -- a RuleBases.
    1st argument -- the key whose value is to be
        duplicated.
    2nd argument -- number of duplications.
    Return -- the modified self."
|index array|
array := Array new: anInteger.
index := 1.
anInteger timesRepeat: [
    array at: index put: ((aKey contents) asSet).
    self at: (array at: index) put: ((self at: aKey) asArray asSet).
    index := index + 1].
^self

errorKeyNotSet
    "Private - Report an error to the effect that the
        key in the key/value pair is not an
        instance of class Set."
    self error: 'Key is not a Set'

```

errorValueNotSet

"Private - Report an error to the effect that the  
value in the key/value pair is not an  
instance of class Set."

self error: 'Value is not a Set'

extractAt: aKey

"Answer a string containing the key/value pair at  
aKey.

Receiver -- a RuleBases.

Argument -- the key.

Return -- the string."

|string|

string := aKey asArray at: 1.

string := (string, ' /') asString.

((self at: aKey) isNil)

ifFalse: [(self at: aKey) do: [:each |

string := (string, ' ', each) asString]].

^string

findKey: aKey value: aValue

"Answer the key which is the same as aKey and whose  
value is the same as aValue.

Receiver -- a RuleBases.

1st argument -- the key in a set to be compared.

2nd argument -- the value in a set to be compared.

Return -- the matched key."

|value|

self keysDo: [:key |

(key sameAs: aKey)

ifTrue: [value := self at: key.

((value = aValue) or: [value sameAs: aValue])

ifTrue: [^key]].

^nil

findLeaf

"Finds the leaf from the receiver one at a time.

Receiver -- a RuleBases.

Return -- a leaf node or nil."

(self keys) do: [:key |

((self at: key) size = 0)

ifTrue: [self removeKey: key.

key do: [:goal | ^goal]]].

^nil

findsKey: aKey

"Answers the key in the receiver containing aKey.

If the key is not found, answer nil.

Receiver -- a RuleBases.

Argument -- content of a key.

Return -- the found key or nil."

self keysDo: [:key |

(key includes: aKey)

ifTrue: [^key]].

^nil

formNode: aString with: aTable

"Answer a subtree created with aTable. The subtree's root is aString.

Receiver -- a RuleBases.

1st argument -- name of the subtree's root.

2nd argument -- a weight table.

Return -- a subtree."

!trans keycond keyrules tree!

tree := Tree new.

keyrules := RuleBases new.

tree.nameNode: aString.

self attachLeafTo: tree.

[(self keys) size > 0]

whileTrue: [

trans := self transpose.

keycond := trans maxKeyWith: aTable.

(trans at: keycond) do: [:goal |

keyrules add: (self associationAt: goal).

self removeKey: goal.

(keyrules at: goal) remove: (keycond deset) ifAbsent: ['not found']].

tree add: (keyrules formNode: (keycond deset) with: aTable)].

^tree

goalHasCond: aCond

"Answer the goal whose rule contains aCond.

Receiver -- a RuleBases.

Argument -- a condition.

Return -- the goal or nil."

self do: [:conds |

(aCond subSetOf: conds)



```

    ifTrue: [^(self keyAtValue: conds)].
    ^nil

```

```

maxKeyWith: aTable

```

```

    "Find the key whose set has the maximum number of
    values.

```

```

    Receiver -- a RuleBases.

```

```

    Argument -- a weight table.

```

```

    Return -- the key."

```

```

    |max maxkey sum|

```

```

    max := 0.

```

```

    sum := 0.

```

```

    self keysDo: [:key|

```

```

        sum := (self at: key) size * 10 + (aTable at: (key deset)).

```

```

        (sum > max)

```

```

            ifTrue: [max := sum.

```

```

                maxkey := key]].

```

```

    ^maxkey

```

```

nameSelf: aString

```

```

    "Name the receiver aString.

```

```

    Receiver -- a RuleBases.

```

```

    Argument -- name given to the receiver.

```

```

    Return -- no return."

```

```

    name := aString

```

```

nameSR: aString

```

```

    "Stores the name (aString) of the StringRules
    from which the receiver is created.

```

```

    Receiver -- a RuleBases.

```

```

    Argument -- name of the related StringRules.

```

```

    Return -- no return."

```

```

    stringRule := aString

```

```

returnKey: aKey

```

```

    "Answers the key in the receiver containing aKey.

```

```

    If the key does not exist, it is added to the
    receiver.

```

```

    Receiver -- a RuleBases.

```

```

    Argument -- content of a key.

```

```

    Return -- the key."

```

```

    |key|

```

```

    key := (self findsKey: aKey).

```

```

(key isNil)
  ifTrue: [key := Set new.
    key add: aKey.
    self at: key put: Set new].
^key

```

```

rulesWithGoal: goal
  "Answer a set of goals that are the same as goal.
  Receiver -- a RuleBases.
  Argument -- the goal to be matched.
  Return -- the set."
  |set|
  set := Set new.
  self keysDo: [:key |
    (goal sameAs: key)
    ifTrue: [set add: key]].
^set

```

```

selfName
  "Answer the receiver's name.
  Receiver -- a RuleBases.
  Return -- the receiver's name."
^name

```

```

srName
  "Answer the name of the StringRules
  from which the receiver is created.
  Receiver -- a RuleBases.
  Return -- name of the StringRules."
^stringRule

```

```

subAllSubgoal
  "Substitute all subgoals by their conditions.
  Receiver -- a RuleBases.
  Return -- the modified self."
  |subgoal goal|
  subgoal := self condIsGoal.
  [subgoal isNil]
  whileFalse: [
    goal := self goalHasCond: subgoal.
    self subSubgoal: subgoal at: goal.
    subgoal := self condIsGoal].
^self

```

```

subSubgoal: subGoal at: aGoal
    "Substitute subgoal in a rule with the subgoal's
    conditions.
    Receiver -- a RuleBases.
    1st argument -- the subgoal in a set.
    2nd argument -- the rule containing the subgoal.
    Return -- the modified self."
    |temp set index|
    temp := RuleBases new.
    set := self rulesWithGoal: subGoal.
    (self at: aGoal) remove: (subGoal deset) ifAbsent: ['not found'].
    temp at: aGoal put: (self at: aGoal).
    self removeKey: aGoal.
    temp duplicateAssocAt: aGoal times: (set size - 1).
    index := 1.
    temp do: [:conds |
        conds addAll: (self at: (set asArray at: index)).
        index := index + 1].
    temp associationsDo: [:pair |
        self add: pair].
    ^self

transformWith: aTable
    "Answer the search tree created with aTable.
    Receiver -- a RuleBases.
    Argument -- a weight table.
    Return -- a search tree."
    |trans keycond keyrules tree|
    tree := Tree new.
    tree nameNode: 'root'.
    keyrules := RuleBases new.
    self attachLeafTo: tree.
    [(self keys) size > 0]
    whileTrue: [
        trans := self transpose.
        keycond := trans maxKeyWith: aTable.
        (trans at: keycond) do: [:goal |
            keyrules add: (self associationAt: goal).
            self removeKey: goal.
            (keyrules at: goal) remove: (keycond deset) ifAbsent: ['not found']].
        tree add: (keyrules formNode: (keycond deset) with: aTable)].
    ^tree

```

```

transpose
    "Transposes the receiver into a new RuleBases. The new RuleBase
    uses the values of the receiver as the keys, and the keys of the
    receiver as the values.
    Receiver -- a RuleBases.
    Return  -- a RuleBases."
    |keyset transposed aKey|
    transposed := RuleBases new.
    keyset := self keys.
    (self values) do: [:value |
        keyset do: [:key |
            ((self at: key) includes: value)
            ifTrue: [aKey := (transposed returnKey: value).
                (transposed at: aKey) add: key]]].
    ^transposed

```

```

values
    "Returns a Set containing all the values in the sets of the
    key/set pairs in the receiver.
    Receiver -- a RuleBases.
    Return  -- a Set."
    |answer|
    answer := Set new.
    self do: [:value |
        value do: [:element | answer add: element]].
    ^answer

```

### Super class -- Set

```

Collection subclass: #Set
    instanceVariableNames:
        'elementCount contents '
    classVariableNames: ''
    poolDictionaries: ''

```

### Set class methods

```

new
    "Answer a new Set."
    ^self new: 4

new: anInteger
    "Answer a new Set with an initial

```

capacity of anInteger elements."  
 ^super new initialize: ( 1 max: anInteger)

Set methods

add: anObject  
 "Answer anObject. Add anObject to the receiver  
 if the receiver does not already contain it."  
 | index |  
 anObject isNil  
   ifTrue: [^anObject].  
 self adjustSize.  
 (contents at:  
   (index := self findElementIndex: anObject)) isNil  
   ifTrue: [  
     elementCount := elementCount + 1.  
     ^contents at: index put: anObject].  
 ^anObject

adjustSize  
 "Private - Answer the receiver. If the receiver set is  
 getting full, expand it to accomodate more objects."  
 (elementCount \* 10) >= (contents size - 2 \* 9)  
   ifTrue: [^self grow]

at: anInteger  
 "Access the element at index position anInteger  
 in the receiver. This method reports  
 an error since sets cannot be indexed."  
 ^self errorNotIndexable

at: anInteger put: anObject  
 "Replace the element at index position anInteger  
 in the receiver with anObject. This method  
 reports an error since sets are not indexable."  
 ^self errorNotIndexable

contents  
 "Private - Answer an Array containing  
 contents of the receiver."  
 ^contents

deset

```

    "Answers the only element in the receiver. If it
    has more than one element, report an error.
    Receiver -- a Set.
    Return -- the only object in the receiver or an error message."
    (self size = 1)
    ifTrue: [self do: [:element |
        ^element]]
    ifFalse: [^self errorMultipleElement]

do: aBlock
    "Answer the receiver. For each element in the receiver,
    evaluate aBlock with that element as the argument."
    | index element |
    index := contents size.
    [index > 0]
    whileTrue: [
        (element := contents at: index) == nil
        ifFalse: [aBlock value: element].
        index := index - 1]

errorMultipleElement
    "Private - Report an error to the effect that the
    set contains more than one element."
    self error: 'Set has several elements'

findElementIndex: anObject
    "Private - Answer the index position of anObject in the
    receiver or the first empty element position."
    | index indexedObject lastIndex |
    lastIndex := contents size.
    index := (anObject hash) \ lastIndex + 1.
    [(indexedObject := contents at: index) = anObject]
    whileFalse: [
        (indexedObject == nil)
        ifTrue: [^index].
        (index := index + 1) > lastIndex
        ifTrue: [ "index wraparound"
            index := 1]].
    ^index

grow
    "Private - Answer the receiver expanded
    to accomodate more elements."

```

```

| aSet |
aSet := self species new: contents size * 4 // 3 + 10.
self do: [ :element | aSet add: element].
contents := aSet contents

includes: anObject
    "Answer true if the receiver includes anObject
    as one of its elements, else answer false."
    ^((contents at:
        (self findElementIndex: anObject)) == nil) not

initialize: anInteger
    "Private - Initialize the instance variable
    elementCount to zero, contents to an Array
    of size anInteger."
    elementCount := 0.
    contents := Array new: anInteger

notIncludes: anObject
    "Answer false if the receiver includes anObject
    as one of its elements, else answer true.
    Receiver -- a Set.
    Argument -- an object.
    Return -- a Boolean value."
    ^((contents at:
        (self findElementIndex: anObject)) == nil)

occurrencesOf: anObject
    "Answer 1 if the receiver includes anObject as
    one of its elements, else answer zero."
    (self includes: anObject)
        ifTrue: [^1].
    ^0

rehashFrom: anInteger
    "Private - Rehash the receiver from the index
    position anInteger to the last index position."
    | deleteIndex lastIndex searchIndex testObject |
    lastIndex := contents size.
    deleteIndex := anInteger.
    [(deleteIndex := deleteIndex + 1) > lastIndex
        ifTrue: [ "index wraparound"
            deleteIndex := 1].

```

```

(testObject := contents at: deleteIndex) == nil]
whileFalse: [ "test next object for relocation"
  searchIndex := self findElementIndex: testObject.
  (contents at: searchIndex) == nil
  ifTrue: [ "found object to move"
    contents at: searchIndex
      put: testObject.
    contents at: deleteIndex
      put: nil]]

```

```

remove: anObject ifAbsent: aBlock
  "Answer anObject. Remove the element anObject from
  the receiver collection. If anObject is not an
  element of the receiver, aBlock is evaluated
  (with no arguments)."
  | index |
  index := self findElementIndex: anObject.
  (contents at: index) == nil
    ifTrue: [^aBlock value].
  contents at: index put: nil.
  elementCount := elementCount - 1.
  self rehashFrom: index.
  ^anObject

```

```

sameAs: originalset
  "See if the receiver is the same as the argument.
  Receiver -- a set.
  Argument -- a set.
  Returns -- Boolean value."
  (self size = originalset size)
    ifFalse: [^false].
  self do: [:member |
    (originalset includes: member)
      ifFalse: [^false]].
  ^true

```

```

size
  "Answer the number of elements contained
  in the receiver."
  ^elementCount

```

```

subSetOf: originalSet
  "See if the receiver is a subset of the argument.

```



```

    Receiver -- a set.
    Argument -- a set.
    Returns -- Boolean value."
self do: [:member |
    (originalSet includes: member)
    ifFalse: [^false]].
^true

```

```

union: aSet
    "Returns a union of the receiver and argument.
    Receiver -- a Set.
    Argument -- a Set.
    Return -- a Set."
|answer|
answer := self asSet.
answer addAll: aSet.
^answer

```

### **New class -- SetOfFact**

```

Set subclass: #SetOfFact
instanceVariableNames:
    'name '
classVariableNames: ''
poolDictionaries: ''

```

#### **SetOfFact methods**

```

nameSelf: aString
    "Name the receiver aString.
    Receiver -- a SetOfFact.
    Argument -- name given to the receiver.
    Return -- no return."
    name := aString

```

```

selfName .
    "Answer the receiver's name.
    Receiver -- a SetOfFact.
    Return -- the receiver's name."
    ^name

```

### **New class -- Stack**

```

OrderedCollection subclass: #Stack
instanceVariableNames:
  'stringRule ruleBase table tree setOfFact name '
classVariableNames: ''
poolDictionaries: ''

```

Stack methods

```

, aCollection
  "Answer a Stack containing all the elements of the
  receiver followed by all the elements of
  aCollection. This method reports an error because
  contents of a Stack can only be pushed or popped."
  ^self errorConcateNotAllowed

```

```

add: anObject
  "Add anObject after the last element of the
  receiver. This method reports an error because
  contents of a Stack can only be pushed or popped."
  ^self errorAddNotAllowed

```

```

add: newObject after: oldObject
  "Insert newObject immediately after the element
  oldObject in the receiver collection. This method
  reports an error because contents of a Stack can
  only be pushed or popped."
  ^self errorAddNotAllowed

```

```

add: anObject afterIndex: anInteger
  "Insert anObject at index position anInteger + 1
  in the receiver collection. This method reports
  an error because contents of a Stack can only be
  pushed or popped."
  ^self errorAddNotAllowed

```

```

add: newObject before: oldObject
  "Insert newObject immediately before the element
  oldObject in the receiver collection. This method
  reports an error because contents of a Stack can
  only be pushed or popped."
  ^self errorAddNotAllowed

```

```

add: anObject beforeIndex: anInteger

```

"Insert anObject at index position anInteger - 1 in the receiver collection. This method reports an error because contents of a Stack can only be pushed or popped."

^self errorAddNotAllow

addAllFirst: aCollection

"Add all the elements contained in aCollection to the receiver before its first element. This method reports an error because contents of a Stack can only be pushed or popped."

^self errorAddNotAllow

addAllLast: aCollection

"Add all the elements contained in aCollection to the receiver after its last element. This method reports an error because contents of a Stack can only be pushed or popped."

^self errorAddNotAllow

addFirst: anObject

"Add anObject before the first element of the receiver. This method reports an error because contents of a Stack can only be pushed or popped."

^self errorAddNotAllow

addLast: anObject

"Add anObject after the last element of the receiver. This method reports an error because contents of a Stack can only be pushed or popped."

^self errorAddNotAllow

after: anObject

"Answer the element that immediately follows anObject in the receiver collection. This method reports an error because contents of a Stack can only be retrieved using the pop message."

^self errorAddNotAllow

after: anObject ifNone: aBlock

"Answer the element that immediately follows anObject in the receiver collection. If anObject is not an element of the receiver, aBlock is evaluated (with no arguments). This method

reports an error because contents of a Stack can only be retrieved using the pop message."  
`^self errorRetrieveNotAllow`

`at: anInteger`  
 "Answer the element of the receiver at index position `anInteger`. This method reports an error because contents of a Stack can only be retrieved using the pop message."  
`^self errorRetrieveNotAllow`

`at: anInteger put: anObject`  
 "Replace the element of the receiver at index position `anInteger` with the `anObject`. This method reports an error because contents of a Stack can only be pushed or popped."  
`^self errorAddNotAllow`

`before: anObject`  
 "Answer the element that immediately precedes `anObject` in the receiver collection. This method reports an error because contents of a Stack can only be retrieved using the pop message."  
`^self errorRetrieveNotAllow`

`before: anObject ifNone: aBlock`  
 "Answer the element that immediately precedes `anObject` in the receiver collection. If `anObject` is not an element of the receiver, `aBlock` is evaluated (with no arguments). This method reports an error because contents of a Stack can only be retrieved using the pop message."  
`^self errorRetrieveNotAllow`

`duplicate`  
 "Answer a Stack containing the elements of the receiver.  
 Receiver -- a Stack.  
 Return -- a copy of the receiver."  
`|answer|`  
`answer := self species new: self size.`  
`self do: [:element |`  
   `answer push: element].`

```

^answer

emptyStack
    "Checks if a stack is empty.
    Receiver -- a Stack.
    Return -- Boolean value."
    ^endPosition = 0

errorAddNotAllowed
    "Private - Report an error to the effect that pop
    and push are the only stack operations
    allowed."
    self error: 'Add objects into stacks not allowed'

errorConcateNotAllowed
    "Private - Report an error to the effect that the
    concatenation of two stacks is not
    allowed."
    self error: 'Concatenation of stacks not allowed'

errorRemoveNotAllowed
    "Private - Report an error to the effect that pop
    and push are the only stack operations
    allowed."
    self error: 'Remove objects from stacks not allowed'

errorReplaceNotAllowed
    "Private - Report an error to the effect that pop
    and push are the only stack operations
    allowed."
    self error: 'Replace objects in stacks not allowed'

errorRetrieveNotAllowed
    "Private - Report an error to the effect that pop
    and push are the only stack operations
    allowed."
    self error: 'Retrieve objects from stacks not allowed'

nameRB: aString
    "Stores the name (aString) of the RuleBases
    from which the receiver is created.
    Receiver -- a Stack.
    Argument -- name of the related RuleBases.

```

Return -- no return."  
 ruleBase := aString

nameSelf: aString  
 "Name the receiver aString.  
 Receiver -- a Stack.  
 Argument -- name given to the receiver.  
 Return -- no return."  
 name := aString

nameSOF: aString  
 "Stores the name (aString) of the SetOfFact  
 from which the receiver is created.  
 Receiver -- a Stack.  
 Argument -- name of the related SetOfFact.  
 Return -- no return."  
 setOfFact := aString

nameSR: aString  
 "Stores the name (aString) of the StringRules  
 from which the receiver is created.  
 Receiver -- a Stack.  
 Argument -- name of the related StringRules.  
 Return -- no return."  
 stringRule := aString

nameTb: aString  
 "Stores the name (aString) of the Table  
 from which the receiver is created.  
 Receiver -- a Stack.  
 Argument -- name of the related Table.  
 Return -- no return."  
 table := aString

nameTr: aString  
 "Stores the name (aString) of the Tree  
 from which the receiver is created.  
 Receiver -- a Stack.  
 Argument -- name of the related Tree.  
 Return -- no return."  
 tree := aString

pop

```

    "Retrive the top of the stack.
    Receiver -- a Stack.
    Return -- nil or top of stack."
    (self emptyStack)
    ifTrue: [^nil].
    ^super removeLast

```

```

push: anObject
    "Put an entry onto the stack.
    Receiver -- a Stack.
    Argument -- an object.
    Return -- size of the Stack."
    super add: anObject.
    ^endPosition

```

```

rbName
    "Answer the name of the RuleBases
    from which the receiver is created.
    Receiver -- a Stack.
    Return -- name of the RuleBases."
    ^ruleBase

```

```

remove: anObject ifAbsent: aBlock
    "Remove the element anObject from the receiver
    collection. If anObject is not an element of the
    receiver, aBlock is evaluated (with no arguments).
    This method reports an error because contents of
    a Stack can only be pushed or popped."
    ^self errorRemoveNotAllowed

```

```

removeFirst
    "Remove and answer the first element of the
    receiver. This method reports an error because
    contents of a Stack can only be pushed or popped."
    ^self errorRemoveNotAllowed

```

```

removeIndex: anInteger
    "Remove the element of the receiver at index
    position anInteger. This method reports an error
    because contents of a Stack can only be pushed or
    popped."
    ^self errorRemoveNotAllowed

```

removeLast

"Remove and answer the last element of the receiver.  
This method reports an error because contents of  
a Stack can only be pushed or popped."

^self errorRemoveNotAllow

replaceFrom: start to: stop with: aCollection

"Answer a new OrderedCollection containing the  
receiver whose elements at index position start  
through stop have been replaced by the elements  
of aCollection. This method reports an error  
because contents of a Stack can only be modified  
using the pushed and popped messages."

^self errorReplaceNotAllow

selfName

"Answer the receiver's name.  
Receiver -- a Stack.  
Return -- the receiver's name."

^name

sofName

"Answer the name of the SetOfFact  
from which the receiver is created.  
Receiver -- a Stack.  
Return -- name of the SetOfFact."

^setOfFact

srName

"Answer the name of the StringRules  
from which the receiver is created.  
Receiver -- a Stack.  
Return -- name of the StringRules."

^stringRule

tbName

"Answer the name of the Table  
from which the receiver is created.  
Receiver -- a Stack.  
Return -- name of the Table."

^table

trName



```

    "Answer the name of the Tree
    from which the receiver is created.
    Receiver -- a Stack.
    Return -- name of the Tree."
    ^tree

```

### New class -- StringRules

```

Dictionary subclass: #StringRules
instanceVariableNames:
    'name '
classVariableNames: ''
poolDictionaries: ''

```

### StringRules methods

```

add: anAssociation
    "Add anAssociation to the receiver. Both the key
    and value must be instances of class String. If
    not, an error will be displayed.
    Receiver -- a StringRules.
    Argument -- a key/value pair.
    Return -- the key/value pair."
    (anAssociation key class = String)
    ifTrue: [(anAssociation value class = String)
        ifTrue: [^super add: anAssociation]
        ifFalse: [^self errorValueNotString]]
    ifFalse: [^self errorKeyNotString]

at: aKey put: aString
    "Add the aKey/aSet pair into the receiver. If aKey
    already exists in the receiver, its associated
    value will be replaced by aString. Both aKey and
    aString must be instances of class String. If not,
    an error message will be displayed.
    Receiver -- a StringRules.
    1st argument -- the key.
    2nd argument -- the value.
    Return -- the value."
    (aKey class = String)
    ifTrue: [(aString class = String)
        ifTrue: [^super at: aKey put: aString]
        ifFalse: [^self errorValueNotString]]

```

```
iffalse: [^self errorKeyNotString]
```

```
dissociate
```

```
"Dissociates the rules in the receiver. Answer
a RuleBases containing the dissociated rules.
Receiver -- a StringRules.
Return -- the dissociated rules."
```

```
index returnArray setArray aRulebase!
aRulebase := RuleBases new.
self keysDo: [:key |
    returnArray := (self at: key) breakDown.
    setArray := Array new: returnArray size.
    index := 1.
    returnArray size timesRepeat: [
        setArray at: index put: Set new.
        (setArray at: index) add: key.
        aRulebase at: (setArray at: index) put: (returnArray at: index).
        index := index + 1]].
^aRulebase
```

```
errorKeyNotString
```

```
"Private - Report an error to the effect that the
key in the key/value pair is not an
instance of class String."
self error: 'Key is not a String'
```

```
errorValueNotString
```

```
"Private - Report an error to the effect that the
value in the key/value pair is not an
instance of class String."
self error: 'Value is not a String'
```

```
nameSelf: aString
```

```
"Name the receiver aString.
Receiver -- a StringRules.
Argument -- name given to the receiver.
Return -- no return."
name := aString
```

```
selfName
```

```
"Answer the receiver's name.
Receiver -- a StringRules.
Return -- the receiver's name."
```

^name

## New class -- Table

```
Dictionary subclass: #Table
instanceVariableNames:
    'ruleBase stringRule name '
classVariableNames: ''
poolDictionaries: ''
```

### Table methods

```
add: anAssociation
    "Add anAssociation to the receiver. The value in
    anAssociation must be an instance of class Number.
    If not, an error will be displayed.
    Receiver -- a Table.
    Argument -- a key/value pair.
    Return -- the key/value pair."
    (anAssociation value isKindOf: Number)
        ifTrue: [^super add: anAssociation]
        ifFalse: [^self errorValueNotNumber]
```

```
at: aKey put: aNumber
    "Add the aKey/aNumber pair into the receiver. If
    aKey already exists in the receiver, its associated
    value will be replaced by aNumber. The second
    argument must be an instance of class Number. If
    not, an error message will be displayed.
    Receiver -- a Table.
    1st argument -- the key.
    2nd argument -- the value.
    Return -- the value."
    (aNumber isKindOf: Number)
        ifTrue: [^super at: aKey put: aNumber]
        ifFalse: [^self errorValueNotNumber]
```

### errorValueNotNumber

```
"Private - Report an error to the effect that the
value in the key/value pair is not an
instance of class Number."
self error: 'Value is not a Number'
```

```

maxiGoalWeightAt: aKey to: aWeight
    "Change the weight of aKey in the RuleBases to the
    maximum of aWeight and original weight.
    Receiver -- a Table.
    1st argument -- the key.
    2nd argument -- suggested weight.
    Return -- no return."

```

```

|rb|
rb := RuleBDict at: (self rbName).
rb keysDo: [:key |
    (key includes: aKey)
    ifTrue: [
        (rb at: key) do: [:cond |
            self maximizeWeightAt: cond to: aWeight]]]

```

```

maximizeWeightAt: aKey to: aWeight
    "Change the weight of aKey in self to the maximum
    of aWeight and original weight.
    Receiver -- a Table.
    1st argument -- the key.
    2nd argument -- suggested weight.
    Return -- the maximum weight."
    ^self at: aKey put: (aWeight max: (self at: aKey))

```

```

nameRB: aString
    "Stores the name (aString) of the RuleBases
    from which the receiver is created.
    Receiver -- a Table.
    Argument -- name of the related RuleBases.
    Return -- no return."
    ruleBase := aString

```

```

nameSelf: aString
    "Name the receiver aString.
    Receiver -- a Table.
    Argument -- name given to the receiver.
    Return -- no return."
    name := aString

```

```

nameSR: aString
    "Stores the name (aString) of the StringRules
    from which the receiver is created.
    Receiver -- a Table.

```

```

        Argument -- name of the related StringRules.
        Return -- no return."
    stringRule := aString

newWeightAt: aKey to: aWeight
    "Change the weight of aKey in self to aWeight.
    Receiver -- a Table.
    1st argument -- the key.
    2nd argument -- new weight.
    Return -- the new weight."
    ^self at: aKey put: aWeight

rbName
    "Answer the name of the RuleBases
    from which the receiver is created.
    Receiver -- a Table.
    Return -- name of the RuleBases."
    ^ruleBase

selfName
    "Answer the receiver's name.
    Receiver -- a Table.
    Return -- the receiver's name."
    ^name

srName
    "Answer the name of the StringRules
    from which the receiver is created.
    Receiver -- a Table.
    Return -- name of the StringRules."
    ^stringRule

```

### **New class -- Tree**

```

OrderedCollection subclass: #Tree
    instanceVariableNames:
        'root stringRule ruleBase table name '
    classVariableNames: ''
    poolDictionaries: ''

```

### **Tree class methods**

```

new

```

```

    "Answer a new tree.
    Receiver -- a Tree.
    Return -- a new Tree."
    ^super new initialize

```

### Tree methods

```

attach: anObject
    "Append anObject to the parent node self.
    Receiver -- a Tree.
    Argument -- an object.
    Return -- the position index of anObject."
    self add: anObject.
    ^endPosition

```

```

initialize
    "Private - Initialize the instance variable root
    to nil.
    Receiver -- a Tree.
    Return -- no return."
    root := nil

```

```

isLeaf
    "Checks if the node is a leaf node.
    Receiver -- a Tree.
    Return -- Boolean value."
    (endPosition = 0)
    ifTrue: [^true].
    ^false

```

```

nameNode: aString
    "Name the tree-node aString.
    Receiver -- a Tree.
    Argument -- the name.
    Return -- no return."
    root := aString

```

```

nameRB: aString
    "Stores the name (aString) of the RuleBases from
    which the receiver is created.
    Receiver -- a Tree.
    Argument -- name of the related RuleBases.
    Return -- no return."

```

```

ruleBase := aString

nameSelf: aString
    "Name the receiver aString.
    Receiver -- a Tree.
    Argument -- name given to the receiver.
    Return -- no return."
    name := aString

nameSR: aString
    "Stores the name (aString) of the StringRules from
    which the receiver is created.
    Receiver -- a Tree.
    Argument -- name of the related StringRules.
    Return -- no return."
    stringRule := aString

nameTb: aString
    "Stores the name (aString) of the Table from which
    the receiver is created.
    Receiver -- a Tree.
    Argument -- name of the related Table.
    Return -- no return."
    table := aString

nodeName
    "Answer the name of the node.
    Receiver -- a Tree.
    Return -- name of the node."
    ^root

rbName
    "Answer the name of the RuleBases from which the
    receiver is created.
    Receiver -- a Tree.
    Return -- name of the RuleBases."
    ^ruleBase

searchWith: aSOF
    "Search the receiver and answer a Stack containing
    all the results.
    Receiver -- a Tree.
    Argument -- the set storing facts.

```

```

    Return -- stack containing all the solution paths."
    lindex aStack name pathStack
    pathStack := Stack new.
    aStack := Stack new.
    index := 1.
    self do: [:node |
        name := node nodeName.
        (node isLeaf)
        ifFalse: [
            (((name notStartWithNot) &
              (aSOF includes: name)) |
              ((name startWithNot) &
              (aSOF notIncludes: (name asStringFrom: 2 to: name size))))
            ifTrue: [
                aStack push: name.
                ((self at: index) searchWith: aSOF with: aStack) do: [:each |
                    pathStack push: each].
                aStack pop]]
            ifTrue: [aStack push: name.
                pathStack push: (aStack duplicate).
                aStack pop].
            index := index + 1].
    ^pathStack

searchWith: aSOF with: aStack
    "Search the receiver and answer a Stack containing
    all the results.
    Receiver -- a Tree.
    1st argument -- the set storing facts.
    2nd argument -- stack storing the path currently
    being searched.
    Return -- stack containing all the solution paths."
    lindex name pathStack
    pathStack := Stack new.
    index := 1.
    self do: [:node |
        name := node nodeName.
        (node isLeaf)
        ifFalse: [
            (((name notStartWithNot) &
              (aSOF includes: name)) |
              ((name startWithNot) &
              (aSOF notIncludes: (name asStringFrom: 2 to: name size))))

```



```

        ifTrue: [
            aStack push: name.
            ((self at: index) searchWith: aSOF with: aStack) do: [:each |
                pathStack push: each].
            aStack pop]]
        ifTrue: [aStack push: name.
            pathStack push: (aStack duplicate).
            aStack pop].
        index := index + 1].
    ^pathStack

selfName
    "Answer the receiver's name.
    Receiver -- a Tree.
    Return -- the receiver's name."
    ^name

srName
    "Answer the name of the StringRules from which the
    receiver is created.
    Receiver -- a Tree.
    Return -- name of the StringRules."
    ^stringRule

tbName
    "Answer the name of the Table from which the
    receiver is created.
    Receiver -- a Tree.
    Return -- name of the Table."
    ^table

```

## APPENDIX C

### Listing of the rules used in the 5-substation illustrations

The following are the 118 rules used in the four illustrations involving the 5-substation model.

If CB8\_tripped and CB95\_tripped and CB757B\_tripped and CB81\_tripped then  
B138A-1\_fault

If CB8\_tripped and CB95\_tripped and CB9\_tripped and CB717A\_tripped then  
B138B-1\_fault

If CB88\_tripped and CB76\_tripped and CB717B\_tripped and CB74\_tripped then  
B138A-2\_fault

If CB88\_tripped and CB76\_tripped and CB44\_tripped and CB755A\_tripped then  
B138B-2\_fault

If CB52\_tripped and CB62\_tripped and CB36\_tripped and CB3A\_tripped then  
B138A-3\_fault

If CB52\_tripped and CB62\_tripped and CB797A\_tripped and CB60\_tripped then  
B138B-3\_fault

If CB70\_tripped and CB79\_tripped and CB1\_tripped and CB727A\_tripped then

B138A-5\_fault

If CB70\_tripped and CB79\_tripped and CB3B\_tripped and CB55\_tripped then  
B138B-5\_fault

If CB11\_tripped and CB12\_tripped and CB906B\_tripped and CB101\_tripped and  
CB918A\_tripped then B240A-1\_fault

If CB11\_tripped and CB12\_tripped and CB23\_tripped and CB900A\_tripped then  
B240B-1\_fault

If CB63\_tripped and CB5\_tripped and CB900B\_tripped and CB80\_tripped then  
B240A-2\_fault

If CB63\_tripped and CB5\_tripped and CB45\_tripped and CB929A\_tripped and  
CB901A\_tripped then B240B-2\_fault

If CB54\_tripped and CB61\_tripped and CB13\_tripped and CB916A\_tripped then  
B240A-3\_fault

If CB54\_tripped and CB61\_tripped and CB906A\_tripped and CB40\_tripped then  
B240B-3\_fault

If CB21\_tripped and CB28\_tripped and CB916B\_tripped then B240A-4\_fault

If CB21\_tripped and CB28\_tripped and CB917A\_tripped and CB911B\_tripped  
then B240B-4\_fault

If CB73\_tripped and CB77\_tripped and CB6\_tripped and CB929B\_tripped then  
B240A-5\_fault

If CB73\_tripped and CB77\_tripped and CB917B\_tripped and CB97\_tripped and  
CB918B\_tripped then B240B-5\_fault

If CB9\_tripped and CB101\_tripped then T1-1\_fault

If CB81\_tripped and CB23\_tripped then T2-1\_fault

If CB44\_tripped and CB45\_tripped then T1-2\_fault

If CB74\_tripped and CB80\_tripped then T2-2\_fault

If CB13\_tripped and CB36\_tripped then T1-3\_fault

If CB40\_tripped and CB60\_tripped then T2-3\_fault

If CB97\_tripped and CB1\_tripped then T1-5\_fault

If CB6\_tripped and CB55\_tripped then T2-5\_fault

If CB3A\_tripped and CB3B\_tripped then L3\_fault

If CB717A\_tripped and CB717B\_tripped then L717\_fault

If CB900A\_tripped and CB900B\_tripped then L900\_fault

If CB906A\_tripped and CB906B\_tripped then L906\_fault

If CB916A\_tripped and CB916B\_tripped then L916\_fault

If CB917A\_tripped and CB917B\_tripped then L917\_fault

If CB918A\_tripped and CB918B\_tripped then L918\_fault

If CB929A\_tripped and CB929B\_tripped then L929\_fault

If CB757B\_tripped and CB717A\_tripped and CB9\_tripped and CB81\_tripped and CB95\_tripped and !CB8\_tripped or (CB757B\_tripped and CB717A\_tripped and CB9\_tripped and CB81\_tripped and CB8\_tripped and !CB95\_tripped) or (CB8\_tripped and CB95\_tripped and CB757B\_tripped and CB23\_tripped and !CB81\_tripped) then possible\_B138A-1\_fault

If CB757B\_tripped and CB717A\_tripped and CB9\_tripped and CB81\_tripped and CB95\_tripped and !CB8\_tripped or (CB757B\_tripped and CB717A\_tripped and CB9\_tripped and CB81\_tripped and CB8\_tripped and !CB95\_tripped) or (CB8\_tripped and CB95\_tripped and CB717A\_tripped and CB101\_tripped and !CB9\_tripped) or (CB8\_tripped and CB95\_tripped and CB9\_tripped and CB717B\_tripped and !CB717A\_tripped) then possible\_B138B-1\_fault

If CB76\_tripped and CB717B\_tripped and CB755A\_tripped and CB44\_tripped and CB74\_tripped and !CB88\_tripped or (CB88\_tripped and CB717B\_tripped and CB755A\_tripped and CB44\_tripped and CB74\_tripped and !CB76\_tripped) or (CB88\_tripped and CB76\_tripped and CB717B\_tripped and CB80\_tripped and !CB74\_tripped) or (CB88\_tripped and CB76\_tripped and CB717A\_tripped and

CB74\_tripped and !CB717B\_tripped) then possible\_B138A-2\_fault

If CB76\_tripped and CB717B\_tripped and CB755A\_tripped and CB44\_tripped and CB74\_tripped and !CB88\_tripped or (CB88\_tripped and CB717B\_tripped and CB755A\_tripped and CB44\_tripped and CB74\_tripped and !CB76\_tripped) or (CB88\_tripped and CB76\_tripped and CB755A\_tripped and CB45\_tripped and !CB44\_tripped) then possible\_B138B-2\_fault

If CB62\_tripped and CB36\_tripped and CB60\_tripped and CB797A\_tripped and CB3A\_tripped and !CB52\_tripped or (CB52\_tripped and CB36\_tripped and CB60\_tripped and CB797A\_tripped and CB3A\_tripped and !CB62\_tripped) or (CB52\_tripped and CB62\_tripped and CB13\_tripped and CB3A\_tripped and !CB36\_tripped) or (CB52\_tripped and CB62\_tripped and CB36\_tripped and CB3B\_tripped and !CB3A\_tripped) then possible\_B138A-3\_fault

If CB62\_tripped and CB36\_tripped and CB60\_tripped and CB797A\_tripped and CB3A\_tripped and !CB52\_tripped or (CB52\_tripped and CB36\_tripped and CB60\_tripped and CB797A\_tripped and CB3A\_tripped and !CB62\_tripped) or (CB52\_tripped and CB62\_tripped and CB40\_tripped and CB797A\_tripped and !CB60\_tripped) then possible\_B138B-3\_fault

If CB79\_tripped and CB1\_tripped and CB55\_tripped and CB3B\_tripped and CB727A\_tripped and !CB70\_tripped or (CB70\_tripped and CB1\_tripped and CB55\_tripped and CB3B\_tripped and CB727A\_tripped and !CB79\_tripped) or (CB70\_tripped and CB79\_tripped and CB97\_tripped and CB727A\_tripped and

!CB1\_tripped) then possible\_B138A-5\_fault

If CB79\_tripped and CB1\_tripped and CB55\_tripped and CB3B\_tripped and CB727A\_tripped and !CB70\_tripped or (CB70\_tripped and CB1\_tripped and CB55\_tripped and CB3B\_tripped and CB727A\_tripped and !CB79\_tripped) or (CB70\_tripped and CB79\_tripped and CB3A\_tripped and CB55\_tripped and !CB3B\_tripped) or (CB70\_tripped and CB79\_tripped and CB3B\_tripped and CB6\_tripped and !CB55\_tripped) then possible\_B138B-5\_fault

If CB12\_tripped and CB101\_tripped and CB23\_tripped and CB906B\_tripped and CB918A\_tripped and CB900A\_tripped and !CB11\_tripped or (CB11\_tripped and CB101\_tripped and CB23\_tripped and CB906B\_tripped and CB918A\_tripped and CB900A\_tripped and !CB12\_tripped) or (CB11\_tripped and CB12\_tripped and CB906A\_tripped and CB918A\_tripped and CB101\_tripped and !CB906B\_tripped) or (CB11\_tripped and CB12\_tripped and CB906B\_tripped and CB918B\_tripped and CB101\_tripped and !CB918A\_tripped) or (CB11\_tripped and CB12\_tripped and CB906B\_tripped and CB918A\_tripped and CB9\_tripped and !CB101\_tripped) then possible\_B240A-1\_fault

If CB12\_tripped and CB101\_tripped and CB23\_tripped and CB906B\_tripped and CB918A\_tripped and CB900A\_tripped and !CB11\_tripped or (CB11\_tripped and CB101\_tripped and CB23\_tripped and CB906B\_tripped and CB918A\_tripped and CB900A\_tripped and !CB12\_tripped) or (CB11\_tripped and CB12\_tripped and CB900B\_tripped and CB23\_tripped and !CB900A\_tripped) or (CB11\_tripped and CB12\_tripped and CB900A\_tripped and CB81\_tripped and

!CB23\_tripped) then possible\_B240B-1\_fault

If CB5\_tripped and CB45\_tripped and CB80\_tripped and CB900B\_tripped and CB929A\_tripped and CB901A\_tripped and !CB63\_tripped or (CB63\_tripped and CB45\_tripped and CB80\_tripped and CB900B\_tripped and CB929A\_tripped and CB901A\_tripped and !CB5\_tripped) or (CB63\_tripped and CB5\_tripped and CB900A\_tripped and CB80\_tripped and !CB900B\_tripped) or (CB63\_tripped and CB5\_tripped and CB900B\_tripped and CB74\_tripped and !CB80\_tripped) then possible\_B240A-2\_fault

If CB5\_tripped and CB45\_tripped and CB80\_tripped and CB900B\_tripped and CB929A\_tripped and CB901A\_tripped and !CB63\_tripped or (CB63\_tripped and CB45\_tripped and CB80\_tripped and CB900B\_tripped and CB929A\_tripped and CB901A\_tripped and !CB5\_tripped) or (CB63\_tripped and CB5\_tripped and CB44\_tripped and CB929A\_tripped and CB901A\_tripped and !CB45\_tripped) or (CB63\_tripped and CB5\_tripped and CB929B\_tripped and CB45\_tripped and CB901A\_tripped and !CB929A\_tripped) then possible\_B240B-2\_fault

If CB61\_tripped and CB906A\_tripped and CB916A\_tripped and CB13\_tripped and CB40\_tripped and !CB54\_tripped or (CB54\_tripped and CB906A\_tripped and CB916A\_tripped and CB13\_tripped and CB40\_tripped and !CB61\_tripped) or (CB54\_tripped and CB61\_tripped and CB36\_tripped and CB916A\_tripped and !CB13\_tripped) or (CB54\_tripped and CB61\_tripped and CB916B\_tripped and CB13\_tripped and !CB916A\_tripped) then possible\_B240A-3\_fault



If CB61\_tripped and CB906A\_tripped and CB916A\_tripped and CB13\_tripped and CB40\_tripped and !CB54\_tripped or (CB54\_tripped and CB906A\_tripped and CB916A\_tripped and CB13\_tripped and CB40\_tripped and !CB61\_tripped) or (CB54\_tripped and CB61\_tripped and CB906B\_tripped and CB40\_tripped and !CB906A\_tripped) or (CB54\_tripped and CB61\_tripped and CB60\_tripped and CB906A\_tripped and !CB40\_tripped) then possible\_B240B-3\_fault

If CB28\_tripped and CB916B\_tripped and CB917A\_tripped and CB911B\_tripped and !CB21\_tripped or (CB21\_tripped and CB916B\_tripped and CB917A\_tripped and CB911B\_tripped and !CB28\_tripped) or (CB21\_tripped and CB28\_tripped and CB916A\_tripped and !CB916B\_tripped) then possible\_B240A-4\_fault

If CB28\_tripped and CB916B\_tripped and CB917A\_tripped and CB911B\_tripped and !CB21\_tripped or (CB21\_tripped and CB916B\_tripped and CB917A\_tripped and CB911B\_tripped and !CB28\_tripped) or (CB21\_tripped and CB28\_tripped and CB917B\_tripped and CB911B\_tripped and !CB917A\_tripped) then possible\_B240B-4\_fault

If CB73\_tripped and CB917B\_tripped and CB918B\_tripped and CB929B\_tripped and CB97\_tripped and CB6\_tripped and !CB77\_tripped or (CB77\_tripped and CB917B\_tripped and CB918B\_tripped and CB929B\_tripped and CB97\_tripped and CB6\_tripped and !CB73\_tripped) or (CB73\_tripped and CB77\_tripped and CB55\_tripped and CB929B\_tripped and !CB6\_tripped) or (CB73\_tripped and CB77\_tripped and CB929A\_tripped and CB6\_tripped and !CB929B\_tripped) then possible\_B240A-5\_fault

If CB73\_tripped and CB917B\_tripped and CB918B\_tripped and CB929B\_tripped and CB97\_tripped and CB6\_tripped and !CB77\_tripped or (CB77\_tripped and CB917B\_tripped and CB918B\_tripped and CB929B\_tripped and CB97\_tripped and CB6\_tripped and !CB73\_tripped) or (CB73\_tripped and CB77\_tripped and CB1\_tripped and CB917B\_tripped and CB918B\_tripped and !CB97\_tripped) or (CB73\_tripped and CB77\_tripped and CB97\_tripped and CB917A\_tripped and CB918B\_tripped and !CB917B\_tripped) or (CB73\_tripped and CB77\_tripped and CB97\_tripped and CB917B\_tripped and CB918A\_tripped and !CB918B\_tripped) then possible\_B240B-5\_fault

If CB8\_tripped and CB95\_tripped and CB717A\_tripped and CB101\_tripped and !CB9\_tripped or (CB11\_tripped and CB12\_tripped and CB906B\_tripped and CB918A\_tripped and CB9\_tripped and !CB101\_tripped) then possible\_T1-1\_fault

If CB8\_tripped and CB95\_tripped and CB757B\_tripped and CB23\_tripped and !CB81\_tripped or (CB11\_tripped and CB12\_tripped and CB900A\_tripped and CB81\_tripped and !CB23\_tripped) then possible\_T2-1\_fault

If CB88\_tripped and CB76\_tripped and CB755A\_tripped and CB45\_tripped and !CB44\_tripped or (CB63\_tripped and CB5\_tripped and CB44\_tripped and CB929A\_tripped and CB901A\_tripped and !CB45\_tripped) then possible\_T1-2\_fault

If CB88\_tripped and CB76\_tripped and CB717B\_tripped and CB80\_tripped and

!CB74\_tripped or (CB63\_tripped and CB5\_tripped and CB900B\_tripped and CB74\_tripped and !CB80\_tripped) then possible\_T2-2\_fault

If CB52\_tripped and CB62\_tripped and CB13\_tripped and CB3A\_tripped and !CB36\_tripped or (CB54\_tripped and CB61\_tripped and CB36\_tripped and CB916A\_tripped and !CB13\_tripped) then possible\_T1-3\_fault

If CB52\_tripped and CB62\_tripped and CB40\_tripped and CB797A\_tripped and !CB60\_tripped or (CB54\_tripped and CB61\_tripped and CB60\_tripped and CB906A\_tripped and !CB40\_tripped) then possible\_T2-3\_fault

If CB70\_tripped and CB79\_tripped and CB97\_tripped and CB727A\_tripped and !CB1\_tripped or (CB73\_tripped and CB77\_tripped and CB1\_tripped and CB917B\_tripped and CB918B\_tripped and !CB97\_tripped) then possible\_T1-5\_fault

If CB70\_tripped and CB79\_tripped and CB3B\_tripped and CB6\_tripped and !CB55\_tripped or (CB73\_tripped and CB77\_tripped and CB55\_tripped and CB929B\_tripped and !CB6\_tripped) then possible\_T2-5\_fault

If CB52\_tripped and CB62\_tripped and CB36\_tripped and CB3B\_tripped and !CB3A\_tripped or (CB70\_tripped and CB79\_tripped and CB3A\_tripped and CB55\_tripped and !CB3B\_tripped) then possible\_L3\_fault

If CB8\_tripped and CB95\_tripped and CB9\_tripped and CB717B\_tripped and !CB717A\_tripped or (CB88\_tripped and CB76\_tripped and CB717A\_tripped and

CB74\_tripped and !CB717B\_tripped) then possible\_L717\_fault

If CB11\_tripped and CB12\_tripped and CB900B\_tripped and CB23\_tripped and  
!CB900A\_tripped or (CB63\_tripped and CB5\_tripped and CB900A\_tripped and  
CB80\_tripped and !CB900B\_tripped) then possible\_L900\_fault

If CB11\_tripped and CB12\_tripped and CB906A\_tripped and CB918A\_tripped and  
CB101\_tripped and !CB906B\_tripped or (CB54\_tripped and CB61\_tripped and  
CB906B\_tripped and CB40\_tripped and !CB906A\_tripped) then possi-  
ble\_L906\_fault

If CB54\_tripped and CB61\_tripped and CB916B\_tripped and CB13\_tripped and  
!CB916A\_tripped or (CB21\_tripped and CB28\_tripped and CB916A\_tripped and  
!CB916B\_tripped) then possible\_L916\_fault

If CB21\_tripped and CB28\_tripped and CB917B\_tripped and CB911B\_tripped and  
!CB917A\_tripped or (CB73\_tripped and CB77\_tripped and CB97\_tripped and  
CB917A\_tripped and CB918B\_tripped and !CB917B\_tripped) then possi-  
ble\_L917\_fault

If CB11\_tripped and CB12\_tripped and CB906B\_tripped and CB918B\_tripped and  
CB101\_tripped and !CB918A\_tripped or (CB73\_tripped and CB77\_tripped and  
CB97\_tripped and CB917B\_tripped and CB918A\_tripped and !CB918B\_tripped)  
then possible\_L918\_fault

If CB63\_tripped and CB5\_tripped and CB929B\_tripped and CB45\_tripped and

CB901A\_tripped and !CB929A\_tripped or (CB73\_tripped and CB77\_tripped and CB929A\_tripped and CB6\_tripped and !CB929B\_tripped) then possible\_L929\_fault

If CB70\_tripped and CB79\_tripped and CB97\_tripped and CB727A\_tripped and !CB1\_tripped then CB1\_failstotrip

If CB11\_tripped and CB12\_tripped and CB906B\_tripped and CB918A\_tripped and CB9\_tripped and !CB101\_tripped then CB101\_failstotrip

If CB12\_tripped and CB101\_tripped and CB23\_tripped and CB906B\_tripped and CB918A\_tripped and CB900A\_tripped and !CB11\_tripped then CB11\_failstotrip

If CB11\_tripped and CB101\_tripped and CB23\_tripped and CB906B\_tripped and CB918A\_tripped and CB900A\_tripped and !CB12\_tripped then CB12\_failstotrip

If CB54\_tripped and CB61\_tripped and CB36\_tripped and CB916A\_tripped and !CB13\_tripped then CB13\_failstotrip

If CB28\_tripped and CB916B\_tripped and CB917A\_tripped and CB911B\_tripped and !CB21\_tripped then CB21\_failstotrip

If CB11\_tripped and CB12\_tripped and CB900A\_tripped and CB81\_tripped and !CB23\_tripped then CB23\_failstotrip

If CB21\_tripped and CB916B\_tripped and CB917A\_tripped and CB911B\_tripped and !CB28\_tripped then CB28\_failstotrip

If CB52\_tripped and CB62\_tripped and CB13\_tripped and CB3A\_tripped and  
!CB36\_tripped then CB36\_failstotrip

If CB52\_tripped and CB62\_tripped and CB36\_tripped and CB3B\_tripped and  
!CB3A\_tripped then CB3A\_failstotrip

If CB70\_tripped and CB79\_tripped and CB3A\_tripped and CB55\_tripped and  
!CB3B\_tripped then CB3B\_failstotrip

If CB54\_tripped and CB61\_tripped and CB60\_tripped and CB906A\_tripped and  
!CB40\_tripped then CB40\_failstotrip

If CB88\_tripped and CB76\_tripped and CB755A\_tripped and CB45\_tripped and  
!CB44\_tripped then CB44\_failstotrip

If CB63\_tripped and CB5\_tripped and CB44\_tripped and CB929A\_tripped and  
CB901A\_tripped and !CB45\_tripped then CB45\_failstotrip

If CB63\_tripped and CB45\_tripped and CB80\_tripped and CB900B\_tripped and  
CB929A\_tripped and CB901A\_tripped and !CB5\_tripped then CB5\_failstotrip

If CB62\_tripped and CB36\_tripped and CB60\_tripped and CB797A\_tripped and  
CB3A\_tripped and !CB52\_tripped then CB52\_failstotrip

If CB61\_tripped and CB906A\_tripped and CB916A\_tripped and CB13\_tripped and  
CB40\_tripped and !CB54\_tripped then CB54\_failstotrip

If CB70\_tripped and CB79\_tripped and CB3B\_tripped and CB6\_tripped and

!CB55\_tripped then CB55\_failstotrip

If CB73\_tripped and CB77\_tripped and CB55\_tripped and CB929B\_tripped and  
!CB6\_tripped then CB6\_failstotrip

If CB52\_tripped and CB62\_tripped and CB40\_tripped and CB797A\_tripped and  
!CB60\_tripped then CB60\_failstotrip

If CB54\_tripped and CB906A\_tripped and CB916A\_tripped and CB13\_tripped and  
CB40\_tripped and !CB61\_tripped then CB61\_failstotrip

If CB52\_tripped and CB36\_tripped and CB60\_tripped and CB797A\_tripped and  
CB3A\_tripped and !CB62\_tripped then CB62\_failstotrip

If CB5\_tripped and CB45\_tripped and CB80\_tripped and CB900B\_tripped and  
CB929A\_tripped and CB901A\_tripped and !CB63\_tripped then CB63\_failstotrip

If CB79\_tripped and CB1\_tripped and CB55\_tripped and CB3B\_tripped and  
CB727A\_tripped and !CB70\_tripped then CB70\_failstotrip

If CB8\_tripped and CB95\_tripped and CB9\_tripped and CB717B\_tripped and  
!CB717A\_tripped then CB717A\_failstotrip

If CB88\_tripped and CB76\_tripped and CB717A\_tripped and CB74\_tripped and  
!CB717B\_tripped then CB717B\_failstotrip

If CB77\_tripped and CB917B\_tripped and CB918B\_tripped and CB929B\_tripped  
and CB97\_tripped and CB6\_tripped and !CB73\_tripped then CB73\_failstotrip

If CB88\_tripped and CB76\_tripped and CB717B\_tripped and CB80\_tripped and  
!CB74\_tripped then CB74\_failstotrip

If CB88\_tripped and CB717B\_tripped and CB755A\_tripped and CB44\_tripped and  
CB74\_tripped and !CB76\_tripped then CB76\_failstotrip

If CB73\_tripped and CB917B\_tripped and CB918B\_tripped and CB929B\_tripped  
and CB97\_tripped and CB6\_tripped and !CB77\_tripped then CB77\_failstotrip

If CB70\_tripped and CB1\_tripped and CB55\_tripped and CB3B\_tripped and  
CB727A\_tripped and !CB79\_tripped then CB79\_failstotrip

If CB757B\_tripped and CB717A\_tripped and CB9\_tripped and CB81\_tripped and  
CB95\_tripped and !CB8\_tripped then CB8\_failstotrip

If CB63\_tripped and CB5\_tripped and CB900B\_tripped and CB74\_tripped and  
!CB80\_tripped then CB80\_failstotrip

If CB8\_tripped and CB95\_tripped and CB757B\_tripped and CB23\_tripped and  
!CB81\_tripped then CB81\_failstotrip

If CB76\_tripped and CB717B\_tripped and CB755A\_tripped and CB44\_tripped and  
CB74\_tripped and !CB88\_tripped then CB88\_failstotrip

If CB8\_tripped and CB95\_tripped and CB717A\_tripped and CB101\_tripped and  
!CB9\_tripped then CB9\_failstotrip

If CB11\_tripped and CB12\_tripped and CB900B\_tripped and CB23\_tripped and



!CB900A\_tripped then CB900A\_failstotrip

If CB63\_tripped and CB5\_tripped and CB900A\_tripped and CB80\_tripped and  
!CB900B\_tripped then CB900B\_failstotrip

If CB54\_tripped and CB61\_tripped and CB906B\_tripped and CB40\_tripped and  
!CB906A\_tripped then CB906A\_failstotrip

If CB11\_tripped and CB12\_tripped and CB906A\_tripped and CB918A\_tripped and  
CB101\_tripped and !CB906B\_tripped then CB906B\_failstotrip

If CB54\_tripped and CB61\_tripped and CB916B\_tripped and CB13\_tripped and  
!CB916A\_tripped then CB916A\_failstotrip

If CB21\_tripped and CB28\_tripped and CB916A\_tripped and !CB916B\_tripped  
then CB916B\_failstotrip

If CB21\_tripped and CB28\_tripped and CB917B\_tripped and CB911B\_tripped and  
!CB917A\_tripped then CB917A\_failstotrip

If CB73\_tripped and CB77\_tripped and CB97\_tripped and CB917A\_tripped and  
CB918B\_tripped and !CB917B\_tripped then CB917B\_failstotrip

If CB11\_tripped and CB12\_tripped and CB906B\_tripped and CB918B\_tripped and  
CB101\_tripped and !CB918A\_tripped then CB918A\_failstotrip

If CB73\_tripped and CB77\_tripped and CB97\_tripped and CB917B\_tripped and  
CB918A\_tripped and !CB918B\_tripped then CB918B\_failstotrip

If CB63\_tripped and CB5\_tripped and CB929B\_tripped and CB45\_tripped and  
CB901A\_tripped and !CB929A\_tripped then CB929A\_failstotrip

If CB73\_tripped and CB77\_tripped and CB929A\_tripped and CB6\_tripped and  
!CB929B\_tripped then CB929B\_failstotrip

If CB757B\_tripped and CB717A\_tripped and CB9\_tripped and CB81\_tripped and  
CB8\_tripped and !CB95\_tripped then CB95\_failstotrip

If CB73\_tripped and CB77\_tripped and CB1\_tripped and CB917B\_tripped and  
CB918B\_tripped and !CB97\_tripped then CB97\_failstotrip

## APPENDIX D

### Listing of the rules used in speed comparison

The following are the 40 rules used in the search speed comparison between the proposed and Prolog/V's search algorithm. The first three, first ten, first 20, first 30, and 40 rules are used in the first, second, third, fourth, and fifth comparison, respectively.

If a and b then A

If c and d then B

If B and e then C

If a and d and f then D

If g and h and i and j then C

If C and k and l then D

If a and m then E

If m and n and o then F

If p and q and r and s and t then G

If u and v and w and E then G

If D and G and x and y then H

If a and t and D then I

If z and aa and H and bb then J

If cc and F and I then K

If A and dd and B then L

If ee and ff and L then M

If gg and hh and ii and E then M

If p and aa and K then M

If e and ll and w then M

If jj and kk and u and bb then N

If M and G and ll and mm then O

If L and ll and p then O

If x and z and mm then P

If oo and O then Q

If P and pp and qq then R

If R and w and ss and rr then S

If a and bb and o then S

If g and ll and tt then Q

If S and M then T

If I and tt and rr then S

If uu and vv and ww and xx then T

If yy and zz then U

If U and aaa then V

If V and bbb and T then W

If W and X and Y then Z

If ccc and ddd and eee then X

If fff and ggg and oo then Y

If hhh and S then X

If uuu and zzz then YY

If vvv and www and xxx and yyy then ZZ