California State University, San Bernardino

CSUSB ScholarWorks

Theses Digitization Project

John M. Pfau Library

2003

A tabular propositional logic: and/or Table Translator

Chen-Hsiu Lee

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd-project

Part of the Systems Architecture Commons

Recommended Citation

Lee, Chen-Hsiu, "A tabular propositional logic: and/or Table Translator" (2003). *Theses Digitization Project*. 2409.
https://scholarworks.lib.csusb.edu/etd-project/2409

This Project is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

A TABULAR PROPOSITIONAL LOGIC: AND/OR TABLE TRANSLATOR

A Project

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

Chen-Hsiu Lee

June 2003

A TABULAR PROPOSITIONAL LOGIC:
AND/OR TABLE TRANSLATOR

A Project

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Chen-Hsiu Lee

June 2003

Approved by:

[redacted]

Dr. Richard J. Botting, Chair, Computer Science

May/13/03

Date

[redacted]

Dr. Ernesto Gomez

[redacted]

Dr. Kerstin Voigt

© 2003 Chen-Hsiu Lee

ABSTRACT

The goal of this masters project is to use Java-based and Web-based technologies to develop a tool to translate Boolean expressions into the And/Or table notation.

Introduced by Leveson, Heimdahl, and Reese [1], And/Or Tables are a new tabular notation which makes it easier to express complex mathematical expressions by using two dimensional grids of values or expressions. By developing this tabular notation, the And/Or Table Translator can be used in logic design, discrete mathematics, Artificial Intelligence, and formal methods, etc.

The And/Or Table Translator is object oriented. Java Server Pages, (JSP) make it run over Internet.

And/Or Table Translator system has been carefully tested by Unit test plan, integrated test plan, and system test plan. Also, students in Computer Science 656 & 556 - Formal Methods have tested this system.

The And/Or Table Translator is one prototype in the exploration of tools that make formal methods more accessible to software engineers by Dr. Botting.

ACKNOWLEDGMENTS

I would like to thank Dr. Botting, my advisor, for guiding me in the steps to get this master project done. He also taught me through the steps of my earning a master degree of computer science in the classes of data structure and formal methods in 2002 fall, 2002 winter respectively. In addition, I would thanks to Dr. Voigt and Dr. Gomez for their participating in every step of my project.

The support of the National Science Foundation under award 9810708 is gratefully acknowledged.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

SOFTWARE REQUIREMENTS SPECIFICATION

1.1 Introduction

A proposition is a statement that is either true or false. It is a formal presentation of the commonest logic in computer science. However, long and complicated propositions are annoying and unreadable. Fortunately, even logic sentences can be minimized and expressed in normal forms. Disjunctive normal form (DNF) is a common format, because it is more readable, systematic, and convenient. DNF is very important when simplifying the design of circuits, and switching theory. For example, propositional logic sentences, also known as logic equations, are translated into minimized DNF through Karnaugh Maps in order to design the logic circuit with the most basic logic components.

Alternatively, an AND/OR TABLE is an approach to express DNF. Developed by Leveson, Heimdahl, and Reese [1], an AND/OR TABLE is a particular tabular notation that uses True (T) and False (F) to denote the logic of statement. It is a new tabular notation which has found it easier to express complex mathematical expressions by using two dimensional grids of values or expressions. The data can be manipulated as precisely as other formula, however, they are often quicker to write and easier to read. Therefore, And/Or table is specially used to define, describe, and analyze complex conditions. All the standard propositions can be expressed using AND/OR TABLE.

An AND/OR TABLE is like the table below.

Table 1. Example of an And/Or Table

Condition	1	2
P	F	---
Q	F	---
R	----	F

The first column contains predicates and the remaining columns contain "T", "F", "-" (don't care). "T" means the predicates is true and "F" means it is false. Each column denotes the conjunction (&) of the rows. The table is the disjunction (|) of the meanings of the columns. So the table above stands for: ((~P & ~Q) | ~R).

1.2 Purpose of the Project

The goal of this projects is to design a facilitate tool to help user translate any logic statement into

Disjunctive Normal Form and present the result as an AND/OR TABLE, which makes the logic relation easier to express by using two-dimensional grid of values or expressions. This tool is implemented through a web-based and Java-based application. Thus, the user can utilize this tool via World Wide Web.

This project can be considered as an academic tool in areas, like logic design, discrete mathematics, Artificial Intelligent, and formal method. Users can do logic translation or just verify the answer by the AND/OR TABLE Translator. Furthermore, AND/OR TABLE shows the result of a Karnaugh Map, because it is the best way to represent a simple disjunction normal form.

This project is one step towards the exploration of tools that make formal methods more accessible to software engineers by Dr. Botting.

1.2.1 System Deployment

A deployment diagram shows the physical relationship among software and hardware components in delivered system. The And/Or Table Translator could be divided into server and client sides. TCP/IP connects server and client. However, the communication between them is the Hyper Text Transfer Protocol (HTTP) and goes through a web browser. The deployment diagram is shown below:

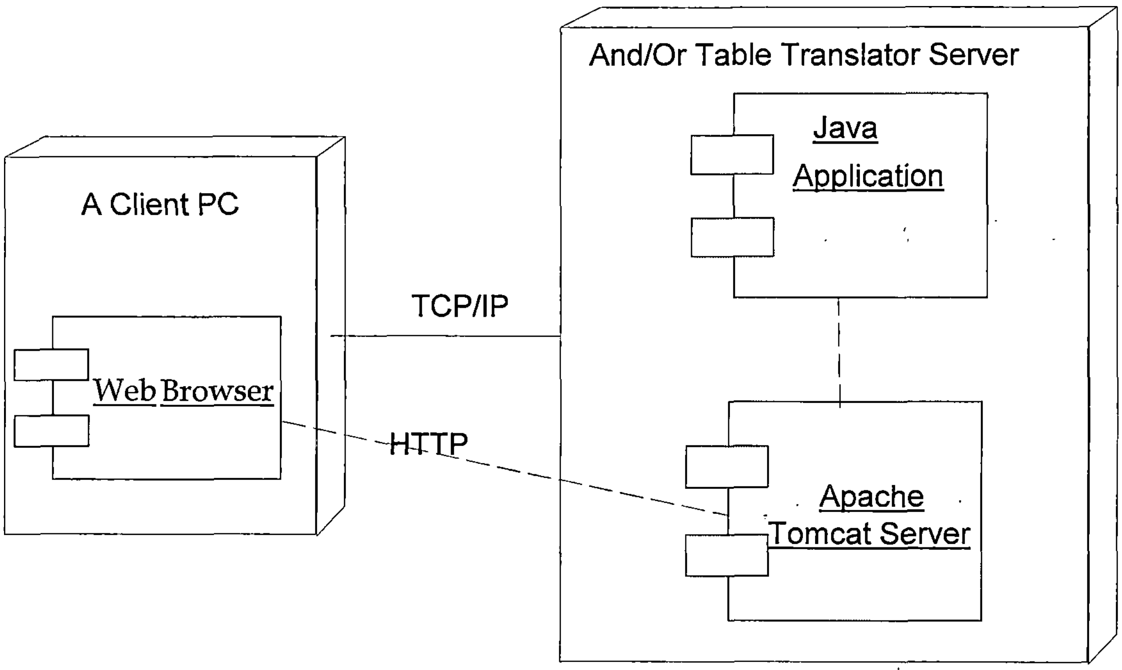


Figure 1. Deployment Diagram

1.2.2 Scenarios and Use Case

A typical scenario will be like the following case:

1. User loads the Java server page.
2. User inputs a proposition logic expression or clauses.
3. System will check the syntax of the proposition expression.
4. If not correct, show the error page which can indicate the exception exist and ask user to edit and re-input.
5. Else translate the proposition into And/Or Table.

6. Show the result in proposition expression as well.
7. User can print out the data or save the data into HTML file (Phase II).
8. Drawing the original expression and result with Node-Tree structure by Java Applets (Phase III).

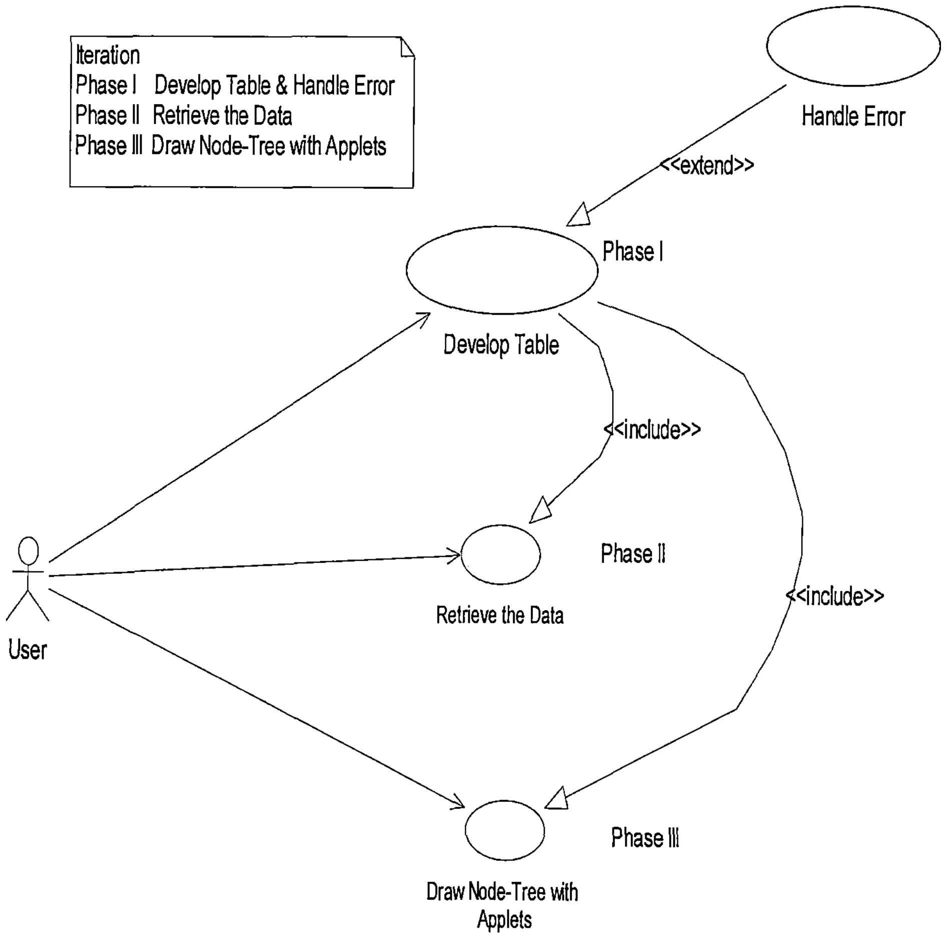

Iteration
Phase I Develop Table & Handle Error
Phase II Retrieve the Data
Phase III Draw Node-Tree with Applets
Handle Error
<<extend>>
Phase I
Develop Table
<<include>>
Phase II
Retrieve the Data
<<include>>
User
Phase III
Draw Node-Tree with Applets

Figure 2. Use Case Diagram

1.2.3 Sequential Diagram

Sequential diagrams are models that describe groups of objects collaborate in some behavior. This diagram shows how three phases interact with each other.

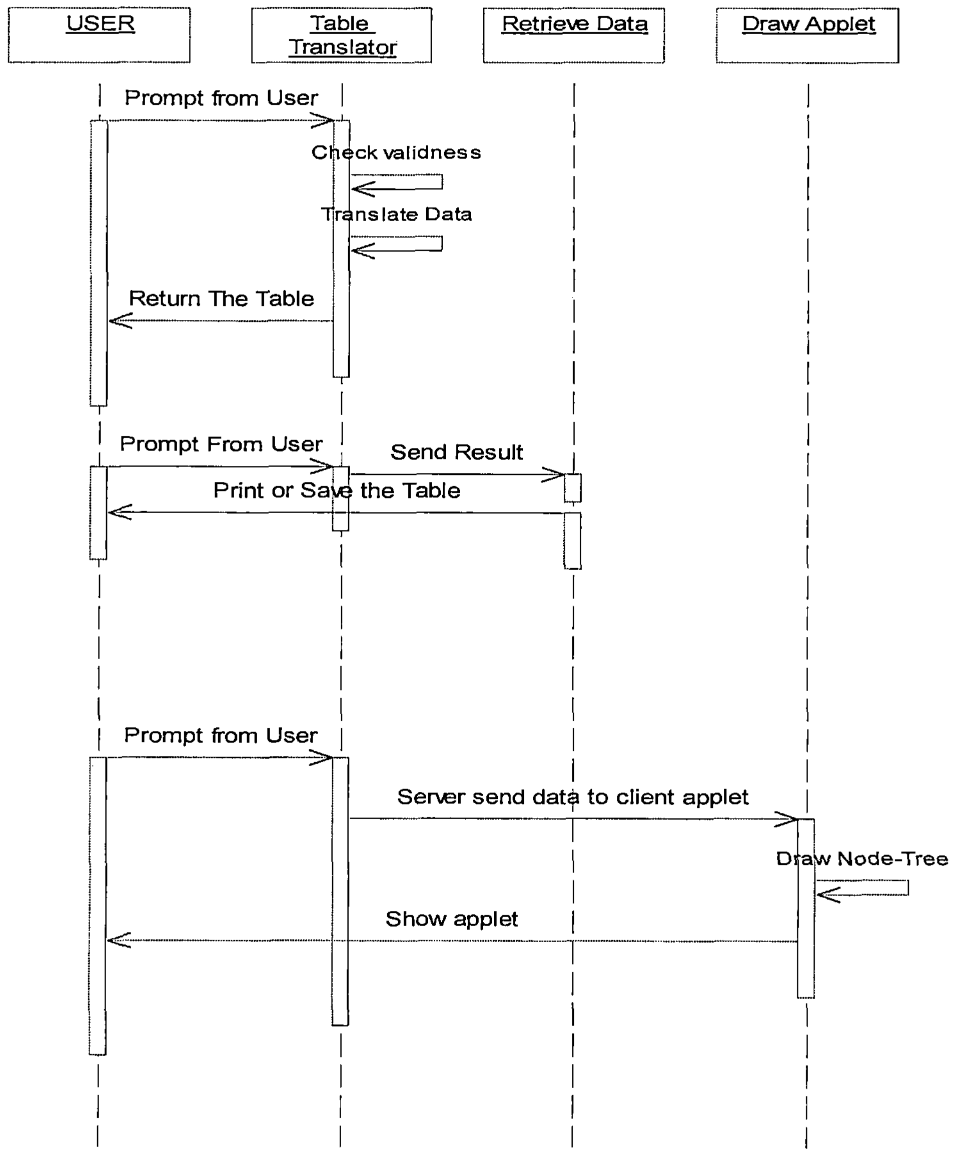


Figure 3. Sequential Diagram

1.3 Context of the Problem

According to Experiences and Lessons from the Analysis TCAS II [3], AND/OR TABLE was first introduced by Leveson, Heimdahl, and Reese [1], and found to be readable, When Leveson et al. [1] were analyzing the requirements specification for a commercial avionics system called TCAS II (Traffic Collision Avoidance System II) for consistency and completeness. At first they used the propositional logic notation to define the guarding conditions in a state unit. However, it did not scale well to complex expressions, and quickly became unreadable. The clients could not use them. Thus, they introduced AND/OR Tables. In this form the conditions were more readable and understandable.

1.4 Limitations

A technological problem was encountered, when developing the client side applet to draw the node tree structure - there is a bug in Apache Tomcat version 4.1.12. It did not present the string value it is supposed to be. Instead, the original expression source code was showed in the applet, which means the Java applet does not recognize the instructions. Other than version 4.1.12, version 4.1.18 or 4.0.6 works perfectly with applet. In summary, Apache Tomcat version 4.1.12 is no good to implement client side applet when passing a string value from server.

1.5 Definition of Terms, Acronyms, and Abbreviations

The following terms are defined as they apply to the project.

Table 2. Definitions of Terms, Acronyms, and Abbreviations

Term	Definition, acronym, and abbreviation
And/Or Table	Developed by Leveson, Heimdahl, and Reese [1], an AND/OR TABLE is a particular tabular notation that uses True (T) and False (F) to denote the logic of statement. Each column in the table denotes the conjunction (&) of the rows. The table is the disjunction (|) of the meanings of the columns.
Applet	An applet is a small program that can be sent along with a Web page to user. Java applets can perform interactive animations, immediate calculations, or other simple tasks without having to send a user request back to the server.
Browser	A graphical display application used to display World Wide Web pages. Basic functions include navigation and printing.
DNF	Disjunctive Normal Form. A formula is a disjunction of one or more conjunctive clauses.

Term	Definition, acronym, and abbreviation
Fuzzy Logic	Fuzzy logic is a departure from classical Boolean logic in that it implements soft linguistic variables on a continuous range of truth values which allows intermediate values to be defined between conventional binary.
GUI	Graphical User Interface. Interface system that relies on graphics as well as test to display and convey information to user.
HTML v. 3.2	HTML (Hypertext Markup Language) is the set of markup symbols or codes inserted in a file intended for display on a World Wide Web browser. The markup tells the Web browser how to display a Web page's words and images.
Hyperlink	A highlighted portion of text that causes browser to take action such as displaying another page or graphic when clicked.
Java Bean	A specification developed by Sun Microsystems that defines how Java objects interact. An object that conforms to this specification is called a JavaBean. It can be used by any application that understands the JavaBeans format.
Java	Java is a programming language expressly designed for use in the distributed environment of the Internet. It was designed to have the "look and feel" of the C++ language, but it is simpler to use than C++ and enforces a completely object-oriented view of programming. Java can be used to create complete applications that may run on a single computer or be distributed among servers and clients in a network. It can also be used to build small application modules or applets for use as part of a Web page. Applets make it possible for a Web page user to interact with the page.

Term	Definition, acronym, and abbreviation
Java Servlet	Java Servlet technology provides Web developers with a simple, consistent mechanism to extend the functionality of a Web server and access existing business systems. A servlet can almost be thought of as an applet that runs on the server side.
JSP	Java Server Page. Java Server Pages technology uses XML-like tags that encapsulate the logic that generates the content for the page. Additionally, the application logic can reside in server-based resources (Such as JavaBeans) that the page accesses with these tags. All formatting (HTML or XML) tags are passed directly back to the response page. By separating the page logic from its design and supporting a reusable component-based structure, JSP technology makes it faster and easier than ever to build Web-based applications.
Karnaugh Maps	A Karnaugh map provides a pictorial method of grouping together expressions with common factors and therefore eliminating unwanted variables.
Propositional logic	Propositional logic is an expression connected with the operands "&", "|", "=>", "⇔" and" ~", and results in True and False.
Tomcat	Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies.

Term	Definition, acronym, and abbreviation
UML	The Unified Modeling Language (UML) is the industry-standard language for specifying visualizing, constructing, and documenting the artifacts of software systems. It simplifies the complex of software design, making a "blueprint" for construction. Rational software's industry-leading methodologists led the UML definition: Grady Booch, Ivar Jacobson, and Jim Rumbaugh.

1.6 Organization of the Project Report

This report is divided into six chapters. Chapter One provides software requirements specification, an introduction to the context of the problem, purpose of the project, limitations, and definitions of terms, acronyms, and Abbreviations. Chapter Two consists of the software design. Chapter Three documents the steps used in testing the project. Chapter Four presents the maintenance required from the project. In Chapter Five, User's manual was presented by a sequential of screen shots. Chapter Six presents conclusions drawn from the development of the project. The Appendix contains the project follows Chapter Six. Finally, the references for the project are presented.

CHAPTER TWO

SOFTWARE DESIGN

2.1 Introduction

Chapter Two consists of a discussion of the software design. Design is a creative process. Software design quality is depended on understandability, verification and adaptability, therefore, Unified Modeling Language (UML) was used to facilitate the software analysis and design. In this chapter UML diagrams, such as class diagram, will be used to demonstrate the software design.

2.2 Preliminary Design

The And/Or Table Translator consists of three phases: Develop Table, Draw Applet, and print or retrieve the data.

2.2.1 Develop Table

This phase is the most complexity and important part. It handles the validation checking of user input, throwing the corresponding exceptions, and translations and manipulations the Tree-Node data structure. Six steps are applied to translate and tabulate the original data. They are:

1) Eliminating Biconditionals and Conditionals

Example: (a $\leftrightarrow$ b) becomes (a $\rightarrow$ b) & (a $\leftarrow$ b), and (a $\rightarrow$ b) becomes (~a | b).

2) Driving in Negations

Every negation in the sentence must be driven in until it has scope only over a single sentence letter.

Example: ~(a & b) will transfer to (~ a | ~b) by DeMorgan's Law.

3) Distributing Conjunctions over Disjunctions

Every conjunction which has scope over a disjunction must be distributed out. Example: (a & (b | c)) becomes (a & b) | (a & c).

4) Simplifying

Any conjunct which contains both a sentence letter and its negation can be eliminated.

Example: (a & b & ~a) | (a & ~b) becomes (a & ~b).

5) Tabulation

Each column denotes the conjunction (&) of the rows. The table is the disjunction (|) of the meanings of the columns.

6) Use "Don't Care" To combine Columns

Repeatedly combine columns that differ by a T vs. F only! Example: a column with data "TTF" and another column "FTF" becomes "-TF".

2.2.2 Draw Applet

Java Applet is a small Java program running on the client side. Two applets are added to draw the diagram of Node-Tree structure. Applets access the data from Java Bean object which is implemented in phase one. Java Bean object passes the value through JSP to client side. By showing the Node-Tree structure diagram, user will understand the precedence level of operators more. Precedence is an important factor to operate propositional statements, because wrong ordered the precedence will cause a wrong result.

2.2.3 Retrieve the Data

After translating, the data can be print out directly from the printer or the user can save them as a HTML file.

2.3 System Analysis

Apache Tomcat performs the web server. The Java server page (JSP) is the server-side language, runs on tomcat. JSP consists of powerful JAVA programming language integrated with the server-side script language based on extra markup language or XML. These make JSP easy for dynamic Web programming. Compared to the other server side language such as, Active Server Pages (ASP), PERL or PHP, JSP has the major advantages of real object-oriented, run faster, and highly reusable. JSP is a language of choice for creating Web-based interaction. Since And/Or Table Translator contains interactions to get data in or to display result out, it is good idea to use JSP in this system.

2.4 Architecture Design

There are twelve classes: DNF, Draw, Node, BinaryNode, ElementNode, InputApplet, NegationNode, NodeException, setNodeTree, SimTable, TableBean, and ValidCheck (see Table 3).

The process of translation begins with when the user clicks the submit button, and that will trigger the JSP to create a TableBean object. The input "String data" will be sent to the TableBean object. Then all the Translation process will be done inside the TableBean object. The ValidCheck class is first doing error checking, if the user's input does not follow the logic rule or key in a proper operator, ValidCheck object will throw a corresponding exception. By doing this, the And/Or Table

Table 3. Classes in And/Or Table Translator

Class Name	Purpose
DNF	Do the translation into DNF.
Draw	An Applet to draw the result with Node Tree Structure.
Node	Abstract class of Tree Node Data Structure
BinaryNode	Derived class of Node.
ElementNode	Derived class of Node.
NegationNode	Derived class of Node.
Class Name	Purpose
InputApplet	An Applet to draw the Node Tree structure of Input formula.
NodeException	Define all the exceptions happen in the translation process.
setNodeTree	Set String logic statement into Tree Node structure.
SimTable	Simplify table. Combine two columns contained both an operand and its negation in each column.
TableBean	The driver of these program, and functioned as a JavaBean.
ValidCheck	Check the validation of user input.

corresponding exception. By doing this, the And/Or Table can ensure the validation of the following steps of translation.

Next step the Node, ElementNode, BinaryNode, and NegationNode class are associated with setNodeTree class to manipulate the valid input data into appropriated structure. Node-Tree structure is an ideal structure to present a logical expression, because any well-formed logical formula can be composed by many atoms (operands) or operators to form a tree structure data type.

After two steps of preparations, the data now can progress the process of translations. The DNF class mainly does the transferred function. The in-order search method is applied to access the node data structure, when the DNF class manipulates the data. Recursive call is also largely used to perform in-order search.

Finally, TableBean class acts as the bridge between JavaBean object and JSP. It provides many member functions to read the user's input data and to allow JSP access the data field of JavaBean. This makes JSP be able to display the result on the web pages. Java Bean object are composed by twelve classes to perform translation function. It encapsulates all these classes in one object, thus the complexity translated procedure will not combine with XML in JSP and make the whole file unreadable.

Figure 4 shows the class diagram of And/Or Table Translator.

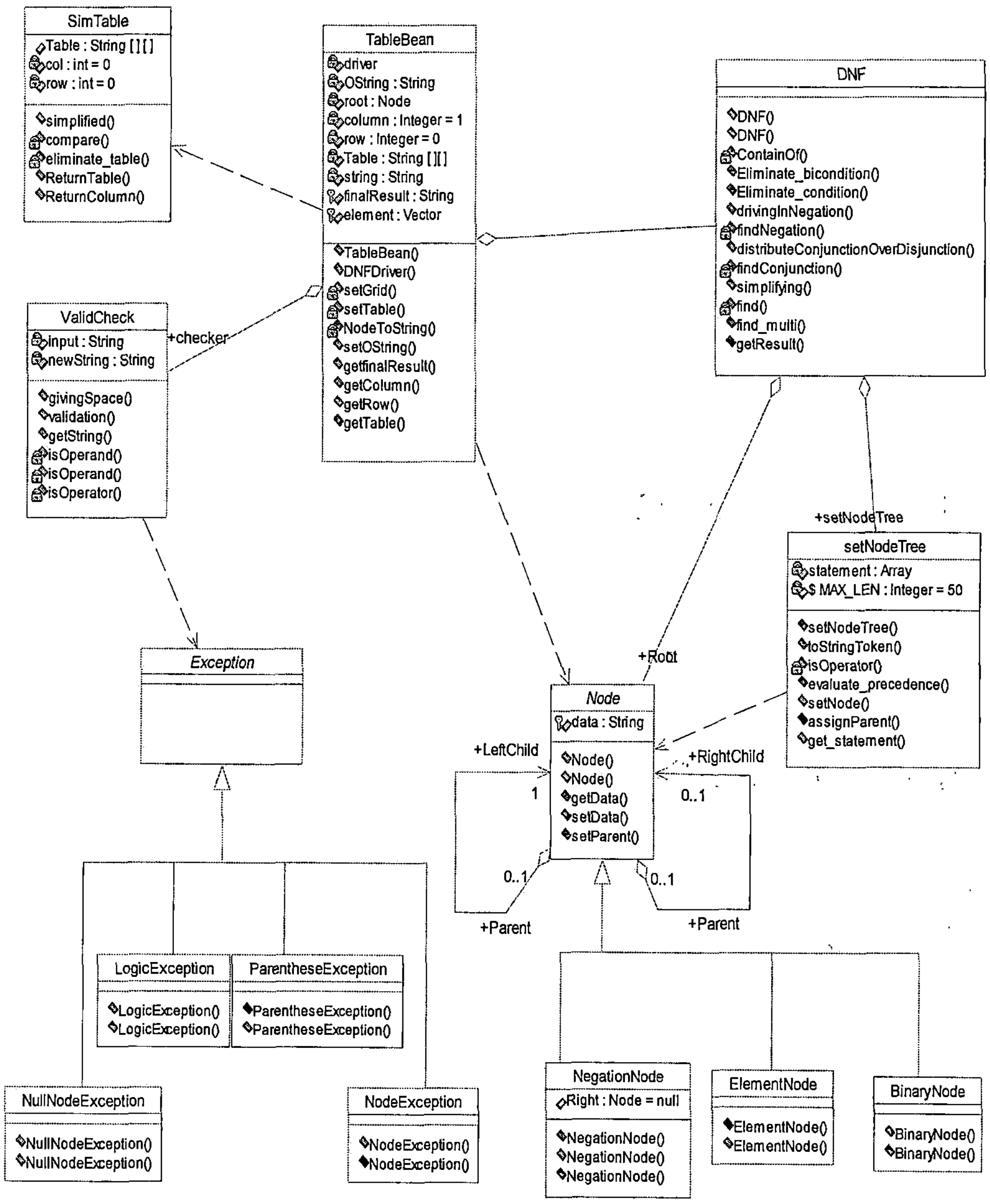


Figure 4. Class Diagram

2.5 Detail Design

Detail design shows the logical algorithms. In other words, detail design shows the program instruction in plain English language. The developed gets benefit from detail design since it is easier to read and detect error. The next section shows the covering all the classes that are listed in the class diagram on Figure 4.

2.5.1 Node Class

Figure 5 has the pseudocode for the Node Class section of the project.

```
Class Name: Node
Purpose          : Define the basic structure of tree node.

Package mybean
Begin class
   Node Left-Child
   Node Right-Child
   Node Parent
   String data

Constructor Node
Begin
        Left-Child          = null
        Right-Child         = null
        Parent              = null
End

Constructor Node with initial value
Parameter String in
Begin
        Data = in
        Left-Child = null
        Right-Child= null
End

Member function getData: return String
Begin
        Return data
End
```

```
Member function setData: no return value
Parameter String newData
Begin
        Data = newData
End

Member function setParent: no return value
Parameter Node par
Begin
        Parent = par
End

Class End
```

Figure 5. Node Class Pseudocode

2.5.2 ElementNode Class

Figure 6 has the pseudocode for the ElementNode Class section of the project.

```
Class Name: ElementNode
Purpose         : Define the sub class of Node.

Package mybean
Begin class

Constructor ElementNode
Parameter String operand
Begin
        Data = operand
        Left-Child = null
        Right-Child= null
End

Class End
```

Figure 6. ElementNode Class Pseudocode

2.5.3 BinaryNode Class

Figure 7 has the pseudocode for the BinaryNode Class section of the project.

```
Class Name: BinaryNode
Purpose         : Define the sub class of Node.

Package mybean
Begin class

Constructor BinaryNode
Parameter String in
Begin
        Data = in
        Left-Child = null
        Right-Child= null
End

Constructor BinaryNode with initial value
Parameter: String in, Node left, Node right
Begin
        Left-Child          = left
        Right-Child         = right
        Data                = in
End

Class End
```

Figure 7. BinaryNode Class Pseudocode

2.5.4 NegationNode Class

Figure 8 has the pseudocode for the NegationNode Class section of the project.

```
Class Name: NegationNode
Purpose         : Define the sub class of Node.

Package mybean
Begin class
          Final Node Right = null
          /* Overwrite the Right-Child defined in
based class */

Constructor NegationNode(Default)
Begin
        Data = "~"
        Left = null
End

Constructor NegationNode with initial value
Parameter: String operand
Begin
```

```
        Data = "~"
        Left = new ElementNode(operand)
End

Class End
```

Figure 8. NegationNode Class Pseudocode

2.5.5 SetNodeTree Class

Figure 9 has the pseudocode for the SetNodeTree Class section of the project.

```
Class Name: setNodeTree
Purpose         : Set every single operand and operator to Node, and construct the
                    Node-Tree.

Package mybean
Begin class
     String [] statement = new String[50];

Constructor setNodeTree (Default)
Begin
End

Member function toStringToken return no value
Parameter: String st
Begin
        Create a StringTokenizer object.
        Separate String st and save them into the String
        Vector.
        Check if the input statement is over the capacity
        of this program (set  50).
End

Member function isOperator return Boolean
Parameter: String in
Begin
        Return true if String in is "&", "|", "=>",
        "⇔", or "~". Else return false.
End

Member function evaluate_precendence return int
Parameter: String [] op
Begin
        Assign values to each operator ("&", "|", "=>", "⇔", "~") to rank them in order to
        decide precedence.
        Return the highest precedence value.
End
```

```
Member function setNode return Node
Parameter: String [] array
Begin
        Int pos = evaluate_precedence(array).
        For loop for size of array
        If array[pos] = "~", create new NegationNode.
              Recursively call setNode function.
        Else if array[pos] = "&", "|", "⇔", or "=>",
                  Create a new BinaryNode.
                  Recursively call setNode function.
        Else create new ElementNode.
                  Recursively call setNode function.
                  /* Recursively call itself to assign all operator and operands. */
        End loop
        Return root-Node
End

Member function assignParent return no value
Parameter: Node root
Begin
          If root = null, return nothing.
          If the left-child of Node root = null,
                    return nothing.
          Else assign the parent of left-child.
          If the right-child of Node root = null,
                    return nothing.
          Else assign the parent of right-child.
          Recursively call assignParent() with parameter:
                    Left-child of root.
          Recursively call assignParent() with parameter:
                    Right-child of root.
End

Class End
```

Figure 9. SetNodeTree Class Pseudocode

2.5.6 ValidCheck

Figure 10 has the pseudocode for the ValidCheck section of the project.

```
Class Name: ValidCheck
Purpose           : Get the input and check if any invalid logic
                      Statement exists.

Package mybean
Begin class
      String input="";
      String newString ="";
Constructor ValidCheck (Default)
Begin
End

Constructor ValidCheck with initial input value
Parameter: String s
Begin
          Input = s
End

Member function givingSpace return no value
throw Exception
Begin
              Give space between every operands and operators.
End

Member function validation return no value
Throw Exception
Begin
          And check the existence of every possible logic error.
          If there is, throw Exception. Else doing nothing.
End

Member function getString return String
Begin
          Return newString
End

Class End
```
Figure 10. ValidCheck Pseudocode

2.5.7 Disjunctive Normal Form

Figure 11 has the pseudocode for the Disjunctive Normal Form section of the project.

```
Class Name: DNF
Purpose         : This class does the translation process to form a statement into DNF.
                  There five steps:
                  1.   Eliminate biconditions and conditions.
                  2.   Drive in Negation.
                  3.   Distributing conjunction over disjunction
                  4.   Simplifying.
Package mybean
Begin class
    Node root
    setNodeTree M

Constructor DNF (Default)
Begin
End

Constructor DNF
Parameter: String input
Begin
        M = new setNodeTree(input).
        Root = M.setNode()
End

Member function ContainOf return Node
Parameter: Node root, String find
Begin
        If root = null, return null;
        Else if data of root = find, return root
        Else ContainOf(left-child of root, find)
                ContainOf(right-child of root, find)
                Return Node
End

Member function Eliminate_bicondition return no value
Begin
        If ContainOf() function find "⇔" operator,
        Replace it with Node(&) with left-Child Node (|),
        Right-child(|).
        assignParent of root.
End


Member function Eliminate_condition return no value
Begin
            If ContainOf() function find =>" operator,
            Replace it with Node(&) with left-Child
            NegationNode(~).
            Re-assign Parent of node-tree.
End

Member function drivingInNegation return No value
Throw NodeException
Begin
```

```
        Node found = findNegation(root)
        Loop while found != null.
                If the left-child of Node found = "&"
                    Set Node found data = "|"
                    // According to Demorgen's Law
                Else if the left-child Node found = "|"
                    Set Node found data = "&"
                Else the left-child of Node found= "~"
                    Eliminate the "~" operator.
                M.assignParent of Node root.
                // Because the structure of this tree has
                // been modified.
                Found = findNegation(root).
        End Loop
End

Member function findNegation return Node
Parameter: Node root
Begin
        If root = null return null
        Else if root = "~" and left-child of root is
                    Instanceof ElementNode
                    Return null
        // Because this one is the leaf of Tree.
        Else if root = "~" and left-child of root is
                    Instanceof BinaryNode
                    Return root
        Else if root = "~" and left-child of root is
                    Instanceof NegationNode
                    Return root
        Else
                    Recursive call of itself with both children
End

Member function distributeConjuctionOverDisjunction
Return no value
Begin
        Node node = findConjuction(root)
        While loop for nood != null
        Begin
        If left-child of node = "|" and right-child of node
          = "|"
          Set data of node = "|"
          Set data of left-child node = "|"
          Set data of right-child node = "|"
          Create new BinaryNode (&) as left-left-child
          Of node.
          Create new Binarynode (&) as left-right-child
          Of node.
        Else if left-child of node = "|"
             Set data of node = "|"
             Create new BinaryNode (&) as left-child of node.
             Create new Binarynode (&) as right-child of
```

```
                node.
            Else if right-child = "|"
                Set data of node = "|"
                Create new BinaryNode (&) as left-child of node.
                Create new Binarynode (&) as right-child of
                node.
            End of Loop
AssginParent of Tree Node root.
End

Member function findConjunction return Node
Parameter: Node node
Begin
            If node = null return null
            Else if node = "&" and left-child of node = "|"
                    Return node
            Else if node = "&" and right-child of node = "|"
                    Return node
            Else return recursive call itself with parameter:
left-child and right-child
End



Member function simplifying return no value
Begin
            Node found = find(root)
            While loop for found != null
            Begin
                    If found is the root of node-tree
                        Eliminate the root node and replace the
                            Root with right-child.
                    Else eliminate the node and replace with
                            Another child.
                    Re-assign Parent of this node-tree.
                    Find next candidate node.
            Loop end
End
    Class End
```
Figure 11. Disjunctive Normal Form Pseudocode

2.5.8 NodeException

Figure 12 has the pseudocode for the NodeException section of the project.

```
Class Name: NodeException
Purpose          : Define all the Exceptions that could happen during the process of the
                   translation.

Package mybean
Begin class NodeException extends Exception

Constructor NodeException
Parameter: String s
Begin
        Super(s)
End
Class End
```

Figure 12. NodeException Pseudocode

2.5.9 ParentheseException

Figure 13 has the pseudocode for the ParentheseException section of the project.

```
Class Name: ParentheseException
Purpose          : Define parentheses Exceptions that could happen during the process
                   of the translation.
Package mybean
Begin class ParentheseException extends Exception

Constructor ParentheseException
Parameter: String s
Begin
        Super(s)
End
Class End
```

Figure 13. ParentheseException Pseudocode

2.5.10 LogicException

Figure 14 has the pseudocode for the LogicException section of the project.

```
Class Name: LogicException
Purpose        : Define logic Exceptions that could happen during the process of the
                 translation.

Package mybean
Begin class LogicException extends Exception

Constructor LogicException
Parameter: String s
Begin
        Super(s)
End

Class End
```

Figure 14. LogicException Pseudocode

2.5.11 TableBean

Figure 15 has the pseudocode for the TableBean section of the project.

```
Class Name: TableBean
Purpose        : This class is the driver of DNF translator. Also, it performs as a Java
                 Bean.

Package mybean
Begin class
     ValidCheck          checker
     DNF                 driver
     String              OString
     Node                root
     int                 column = 1
     int                 row = 0
     String[][] Table
     Vector element = new Vector
     String finalString = ""

Constructor TableBean (Default)
Begin
End

Member function DNFDriver return no value
Throw Exception
Begin
  Checker = new ValidChecker(OString)
  Check the validation of input String.

Driver = new DNF(OString)
```

```
 //Doing the translation process here
End

Member function setGrid return no value
Parameter: Node node
Begin
If node = null return nothing
Else
        If node = "|" then column +1
        Else put into element Vector Container
Recursively call setGrid with parameter left-child of node.
Recursively call setGrid with parameter right-child of node.
End

Member function setTable return no value
Begin
            Sort all the operands in Vector element.
            For loop with column number
                      Fill the operands into the table
                      Fill in "T" with positive condition
                      Fill in "F" with negative condition
            End for loop
            Fill in "—" as "Don't care condition"
End

Member function setOString return no value
Parameter: String OString
Begin
          Trigger the setGrid member function
          Trigger the settable member function
End

Member function getTable return String [][]
Begin
              Return Table
End

Member function getColumn return int
Begin
              Return column
End

Member function getRow return int
Begin
              Return row
End

Class End
```

Figure 15. TableBean Pseudocode

2.6 Summary

In conclusion, the And/Or Table Translator was developed according to the software requirement specification. It provides an easy and quick way to transfer proposition statement into disjunction Normal form in tabular format. Developed by Java and JSP, this system is more reliable and runs faster because of the advantages -- object-oriented and high reusable. One more advantage of this system is flexibility. Uses can add more classes to perform other functionalities. Simply inherit the DNF class to add more functions, and also can perform basic translation in this class. This enhances the future development of And/Or Table Translator system.

CHAPTER THREE

SOFTWARE QUALITY ASSURANCE

3.1 Introduction

Software quality assurance is the testing process to ensure that the program meets the expectation of the user. The purpose of testing And/Or Table Translator is to have assurance about the software quality and functionalities. Also this guarantees system performance and reliability. The testing process contains three parts: unit test plan, integration test plan, and system test plan.

3.2 Unit Test Plan

Unit test plan is the basic level of testing where individual components are tested to ensure that they operate correctly. The individual components can be object, class, program and etc. Several phases of translation are divided to complete the translation of DNF form. They are also considered as units to be tested. The following phases are defined to check that each component meet specification:

Table 4. Unit Test Plan Result

Units	Tests Performed	Results
Validation Check	Check if user's input is valid or following the logic rule	Pass
Set Node-Tree	Check if all nodes are assigned to corresponding position.	Pass
Eliminate Biconditions	Check the node-tree if bicondition has been eliminated properly. Ensure the result is correct.	Pass
Eliminate Conditions	Check the node-tree if condition has been eliminated properly. Ensure the result is correct.	Pass
Driving In Negations	Verify the negation node has been driven in unit. Ensure the result is correct.	Pass
Distributing Conjunctions Over Disjunctions	Verify every &-node has distributed over |-node. Ensure the result is correct.	Pass
Simplifying	Verify every &-node with both an operand and its negation has been eliminated. Ensure the result is correct.	Pass

3.3 Integration Test Plan

Integration test plan is the next step up in the testing process where all related units form a subsystem to do a certain task. Thus, the integration test plan integrates three phases and encapsulates all classes into a Java Bean object. In another word, integration test is to test the Java Bean works properly, and to verify the data transferred correctly.

Table 5. Integration Test Plan Result

Tests	Test Performed	Result
Java Bean	Ensure Java Bean integrates all classes without any exception happened.	Pass
	Ensure the output result correct.	
	Check all the classes are in same package	
Exception Handling	Test if input an invalid statement, Java Bean throws a corresponding Exception.	Pass
	Check if the exception points out the error.	

3.4 System Test Plan

System test plan is the testing process that runs the And/Or Table Translator in the server. Testing processes are executed through web pages or Java server pages. Then test the system by using a variety of data to see the overall performance and displaying final result.

And/Or Table Translator system test plan begins with the following steps (Table 6):

Table 6. System Test Plan Result

	System Test Plan	Result
1.	Install And/Or Table Translator system into Apache tomcat server	Pass
2.	Start up JSP engine	Pass
3.	Log in main page	Pass
4.	Input any valid statement, and click submit button. Test if it links to result page and shows all the information.	Pass
5.	Input any invalid statement, and click submit button. Test if it links to Error Page, and shows relative information.	Pass
6.	In result page, when clicks "Back" button, test if the page links to main page, and clears the previous data in the cookie.	Pass
7.	Check if the "Print" button works.	Pass
8.	Check if the Java Applets load properly.	Pass
9.	In result page, clicks "applet 1". Check if the applet shows the node-tree diagram of original input.	Pass
10.	In result page, clicks "applet 2". Check if the applet shows the node-tree diagram of final result.	Pass
11.	Check the link to source code page.	Pass
12.	Check link to copy right page.	Pass
13.	Check the mail to function works.	Pass

3.5 Summary

And/Or Table Translator system has been carefully tested by all these test plans - Unit test plan, integrated test plan, and system test plan, it passed without detecting any system failure or procedural error. Also, this system has tested by students in Formal Method, class of Computer Science in 2003 winter, conducted by Dr. Botting.

CHAPTER FOUR

MAINTENANCE

4.1 Introduction

System maintenance is an important step to ensure that the system runs smoothly and meets the customer's expectation. There are four major maintenance issues: files and directories, installation, and program modification.

4.2 Files and Directories

There are three main directories, six JSP pages, three html pages and thirteen classes, see Table 7. All directories are under c:\tomcat\webapps\tpl\

Table 7. Files in Main Directories

	Files under main directories
1.	AOTable.jsp
2.	ErrorMsg.jsp
3.	NullNode.jsp
4.	applet1.jsp
5.	applet2.jsp
6.	table.jsp
7.	copyright.html
8.	main.html
9.	source.html
10.	JavaCode/
11.	icon/
12.	global/

Java codes are all under directories (see Table 8).

Table 8. Files in JavaCode Directories

	Java files under JavaCode directories
1.	BinaryNode.java
2.	DNF.java
3.	Draw.java
4.	ElementNode.java
5.	InputApplet.java
6.	LogicException.java
7.	NegationNode.java
8.	Node.java
9.	NodeException.java
10.	NullNodeException.java
11.	ParentheseException.java
12.	TableBean.java
13.	ValidCheck.java
14.	setNodeTree.java
15.	SimTable.java

The entire compiled files are under WEB-INF\classes\ybean directories (see Table 9).

4.3 Software Installation

This section shows how to install And/Or Table Translator into window platform.

1. Download JSP Tomcat server and install into your system.

Table 9. Files in WEB-INF\classes\mybean Directories

	Compiled File in mybean directories
1.	BinaryNode.class
2.	DNF.class
3.	Draw.class
4.	ElementNode.class
5.	InputApplet.class
6.	LogicException.class
7.	NegationNode.class
8.	Node.class
9.	NodeException.class
10.	NullNodeException.class
11.	ParentheseException.class
12.	TableBean.class
13.	ValidCheck.class
14.	setNodeTree.class
15.	SimTable.class

2. Create a folder named tpl under CATALINDA_HOME\webapps, and download all JSP files under http://139.182.137.37:8080/tpl/
3. Under tpl directories, create a directory: WEB-INF\classes\mybean, and download Java classes under http://139.182.137.37:8080/tpl/WEB-INF/classes/mybean/
4. Start up JSP tomcat server.
5. Run a web browser with URL: localhost:8080/tpl/table.jsp

4.4 Recompilation

To enhance the reusability of And/Or Table Translator, modifications and recompilations are necessarily included in maintenance manual. Because these modifications may involve simple changes to correct code error, adding more functions to enhance this system, or implementation of some other logic translations based on this system.

JSP keeps all complied java programs (filename.class) in "TOMCAT_HOME\webapps\tpl\WEB-INF\classes\mybean". Notice that in And/Or Table Translator system, all classes are included in mybean package. Therefore, compiled classes must be put in the folder of mybean under classes.

Modified or added java programs must recompile by using "javac filename.java" command at Java 2 Software Development Kit (J2SDK) environment. After compiled, J2SDK will create "filename.class", which will be moved to the folder mentioned above.

CHAPTER FIVE

USERS MANUAL

5.1 Introduction

Included in Chapter Five was a presentation of the user's manual of the project. To run this system, users must first install Java runtime environment to be able to run Java applets. Sequentially, a presentation of this project is shown to introduce every function by screen shots.

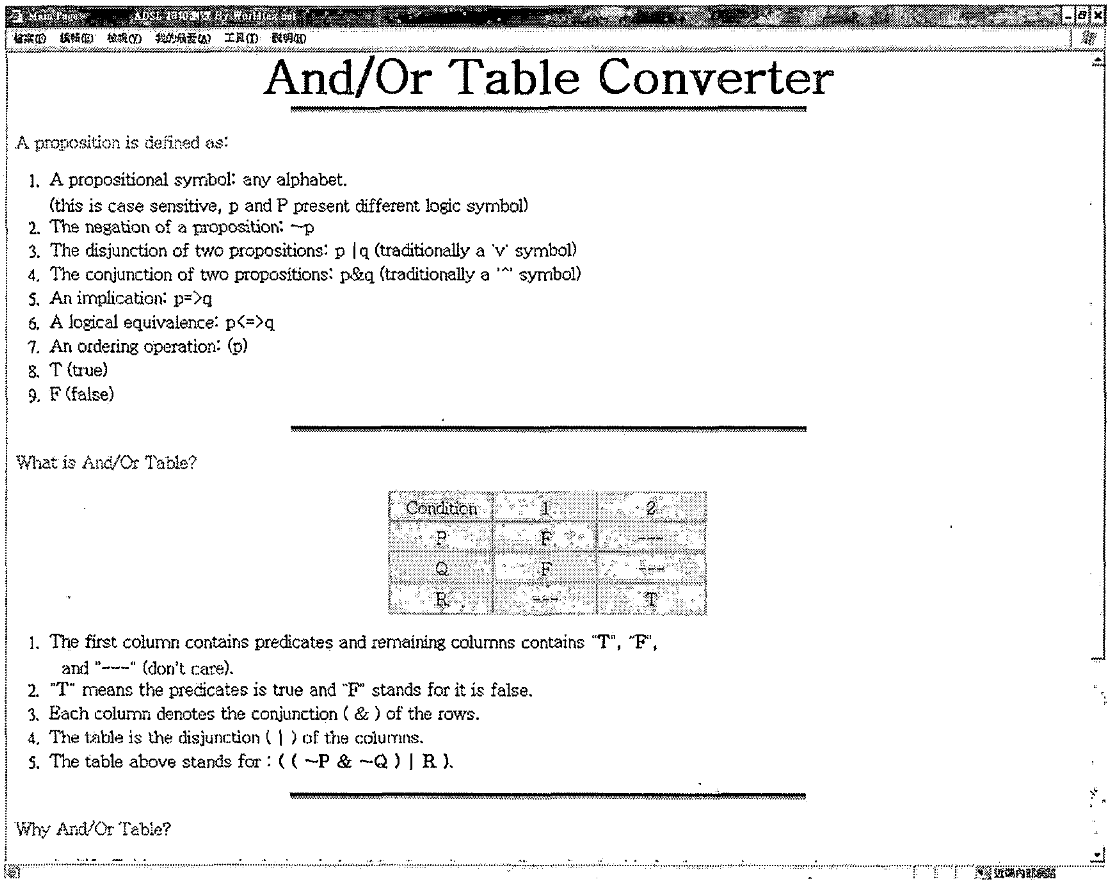
5.2 Java Runtime Environment Installation

Java runtime environment software version 1.1 or later is required to access this system. A user can download this free software from Sun Website or automatically download by opening the Java Applet.

5.3 Project Implementation

5.3.1 Main Page

This is the main page of and/or Table Translator. The definition of Propositional logic, and and/or Table are introduced here. Also, the reasons why and/or table is developed are well explained in the first page of this project, which can give users some basic information, if they are new with formal method. See Figure 16 below.



And/Or Table Converter

A proposition is defined as:

1. A propositional symbol: any alphabet.
   (this is case sensitive, p and P present different logic symbol)
2. The negation of a proposition: ~p
3. The disjunction of two propositions: p |q (traditionally a 'v' symbol)
4. The conjunction of two propositions: p&q (traditionally a '^' symbol)
5. An implication: p=>q
6. A logical equivalence: p<=>q
7. An ordering operation: (p)
8. T (true)
9. F (false)

What is And/Or Table?

Condition	1	2
P	F	---
Q	F	---
R	---	T

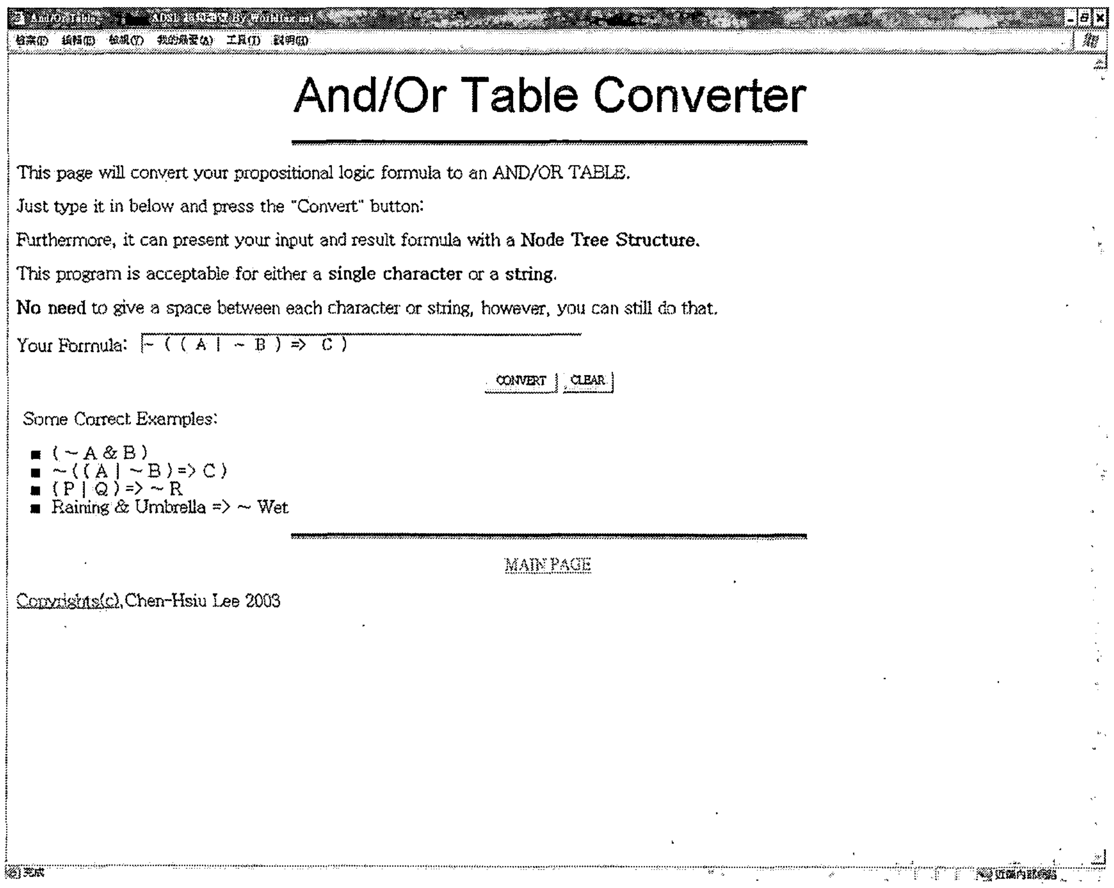
1. The first column contains predicates and remaining columns contains "T", "F", and "---" (don't care).
2. "T" means the predicates is true and "F" stands for it is false.
3. Each column denotes the conjunction (&) of the rows.
4. The table is the disjunction (|) of the columns.
5. The table above stands for : ((~P & ~Q) | R).

Why And/Or Table?

Figure 16. Main Page

5.3.2 Input Page

A user's input form is included in this page along with some valid formula which can give users some good examples. As users input the formula, click the submit button, the system will check the validation of this formula. If the formula is invalid, the system will link to ErrorMsg.jsp page. If not, system will link to AOtable.jsp which shows the result - And/Or Table. See Figure 17 below.


And/Or Table Converter

This page will convert your propositional logic formula to an AND/OR TABLE.

Just type it in below and press the "Convert" button:

Furthermore, it can present your input and result formula with a **Node Tree Structure.**

This program is acceptable for either a **single character** or a **string.**

No need to give a space between each character or string, however, you can still do that.

Your Formula: ~ ((A | ~ B) => C)

Some Correct Examples:

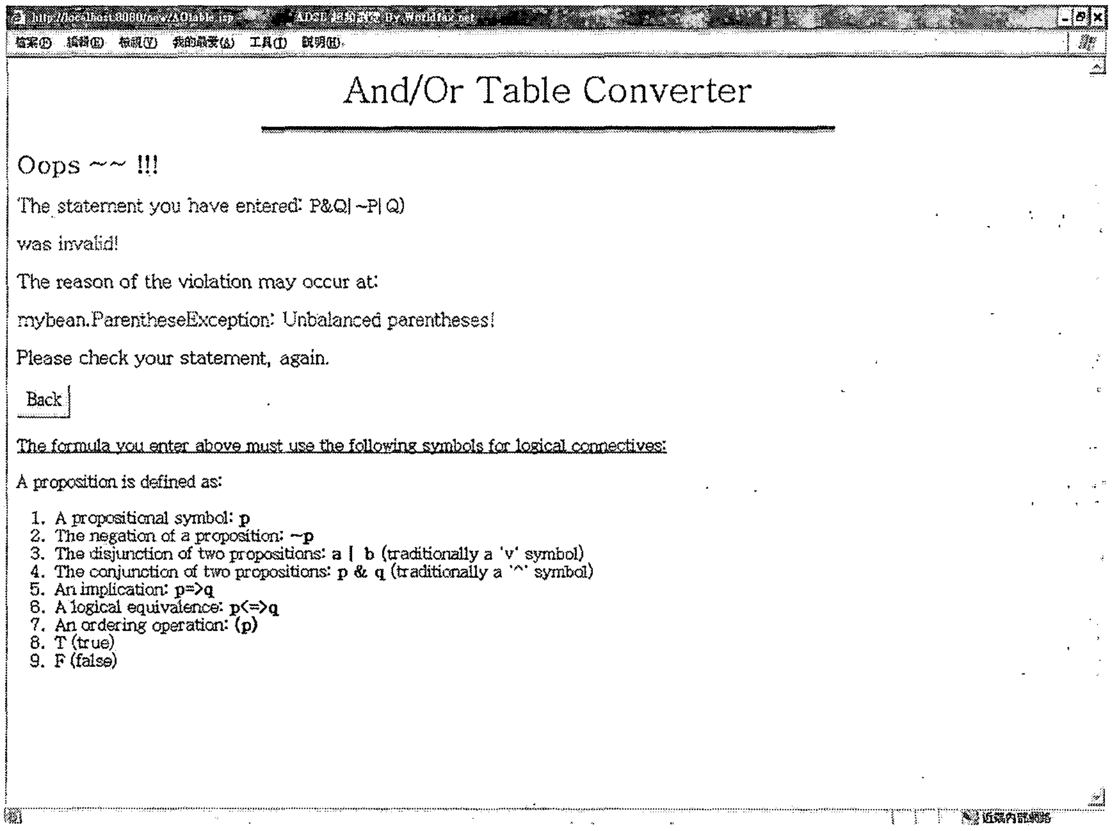
- (~ A & B)
- ~ ((A | ~ B) => C)
- (P | Q) => ~ R
- Raining & Umbrella => ~ Wet

Copyrights(c),Chen-Hsiu Lee 2003

Figure 17. Input Page

5.3.3 Error Message Page

The Error Message page indicates the error where it takes place in user's input. Users may see his/her original formula printed in this page, and compare with errors this system indicating out to make some corrections. See Figure 18 below.


And/Or Table Converter

Oops ~~ !!!

The statement you have entered: P&Q|~P|Q)

was invalid!

The reason of the violation may occur at:

mybean.ParentheseException: Unbalanced parentheses!

Please check your statement, again.

The formula you enter above must use the following symbols for logical connectives:

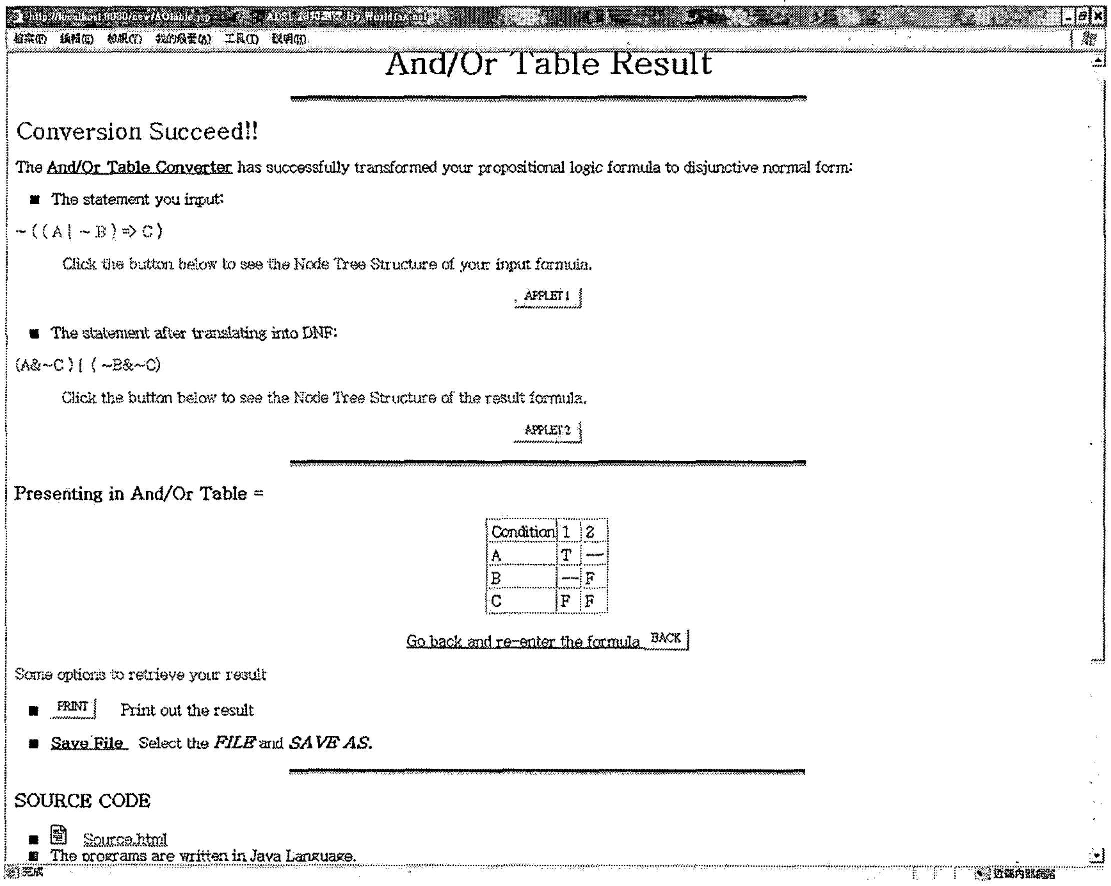
A proposition is defined as:

1. A propositional symbol: **p**
2. The negation of a proposition: **~p**
3. The disjunction of two propositions: **a | b** (traditionally a 'v' symbol)
4. The conjunction of two propositions: **p & q** (traditionally a '^' symbol)
5. An implication: **p=>q**
6. A logical equivalence: **p<=>q**
7. An ordering operation: **(p)**
8. T (true)
9. F (false)

Figure 18. Error Message Page

5.3.4 And/Or Table Page

This page shows the result, And/Or table, and some useful links, such as Apache Tomcat, and Sun Java. Also there are two links of applet which draw the node-tree data structure of original input and result of DNF. The source code of this program can be viewed by linking to Source.html page. Furthermore, in this page some basic java script functions are provided, like "print", and "mail to". Finally, of course, user can go back to input page and re-input a new formula. See Figure 19 below.



And/Or Table Result

Conversion Succeed!!

The And/Or Table Converter has successfully transformed your propositional logic formula to disjunctive normal form:

- The statement you input:

~((A|~B)=>C)

Click the button below to see the Node Tree Structure of your input formula.

APPLET1

- The statement after translating into DNF:

(A&~C)|(~B&~C)

Click the button below to see the Node Tree Structure of the result formula.

APPLET2

Presenting in And/Or Table =

Condition	1	2
A	T	—
B	—	F
C	F	F

Go back and re-enter the formula BACK

Some options to retrieve your result

- PRINT Print out the result
- Save File Select the *FILE* and *SAVE AS.*

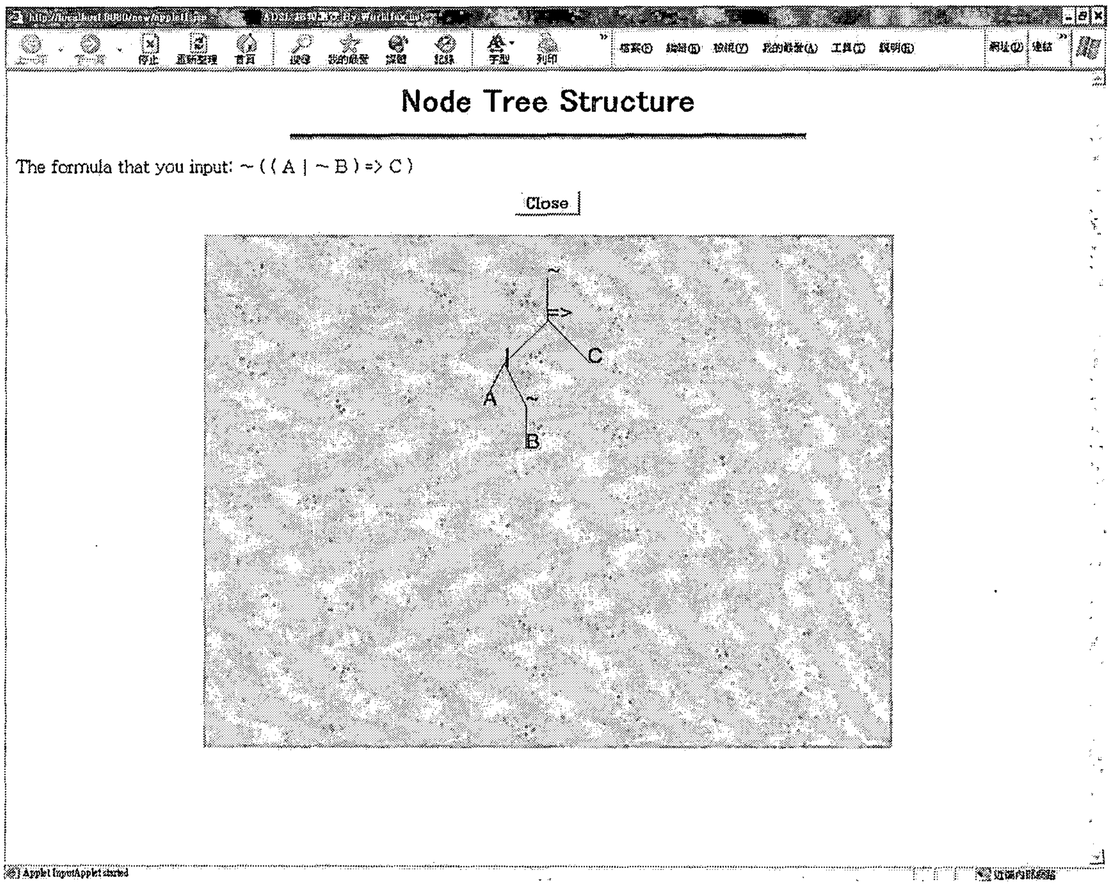
SOURCE CODE

- Source.html
- The programs are written in Java Language.

Figure 19. And/Or Table Page

5.3.5 Applet of Input Page

By clicking the "Applet1" button in And/Or table page, browser will open a new window attached with a applet. The applet draws the original user's input formula in node-tree data structure. To close this window, simply click the "close" button. See Figure 20 below.



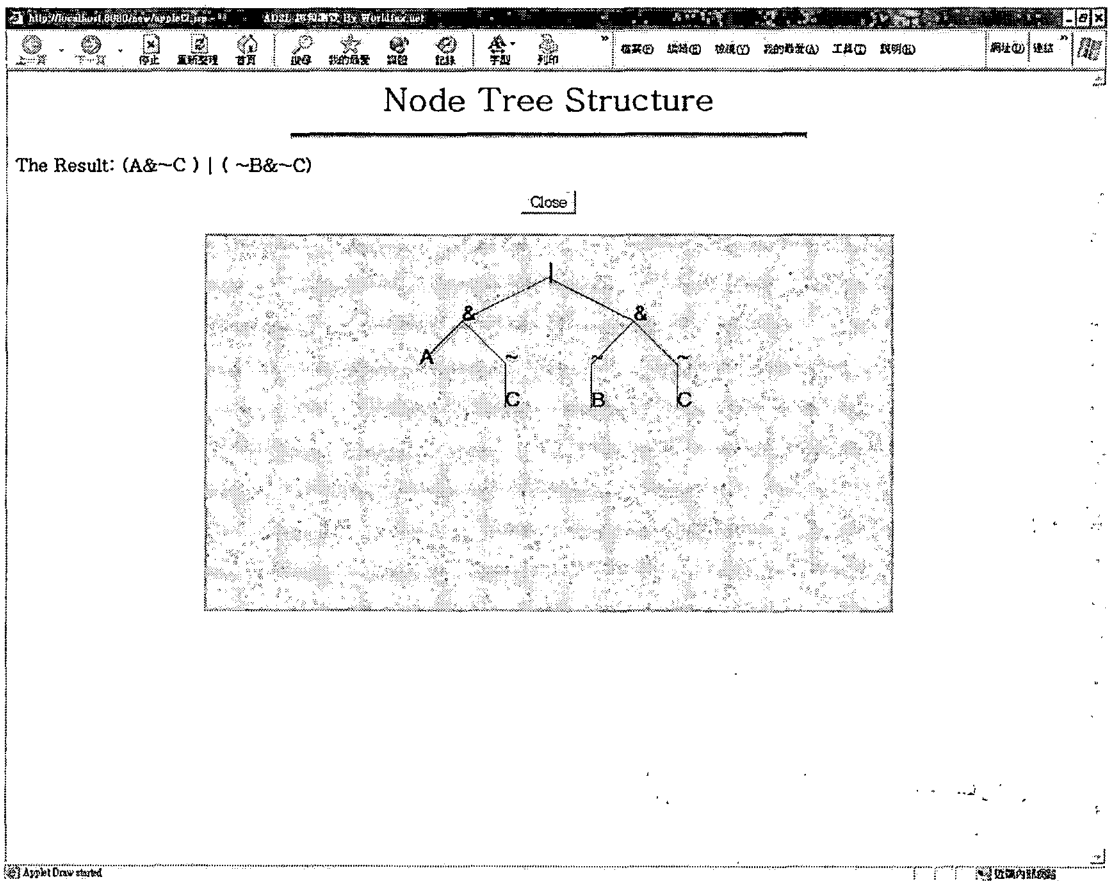
Node Tree Structure

The formula that you input: ~ ((A | ~ B) => C)

Figure 20. Applet of Input Page

5.3.6 Applet of Result Page

Click "Applet2" button will open this window. See Figure 21 below.

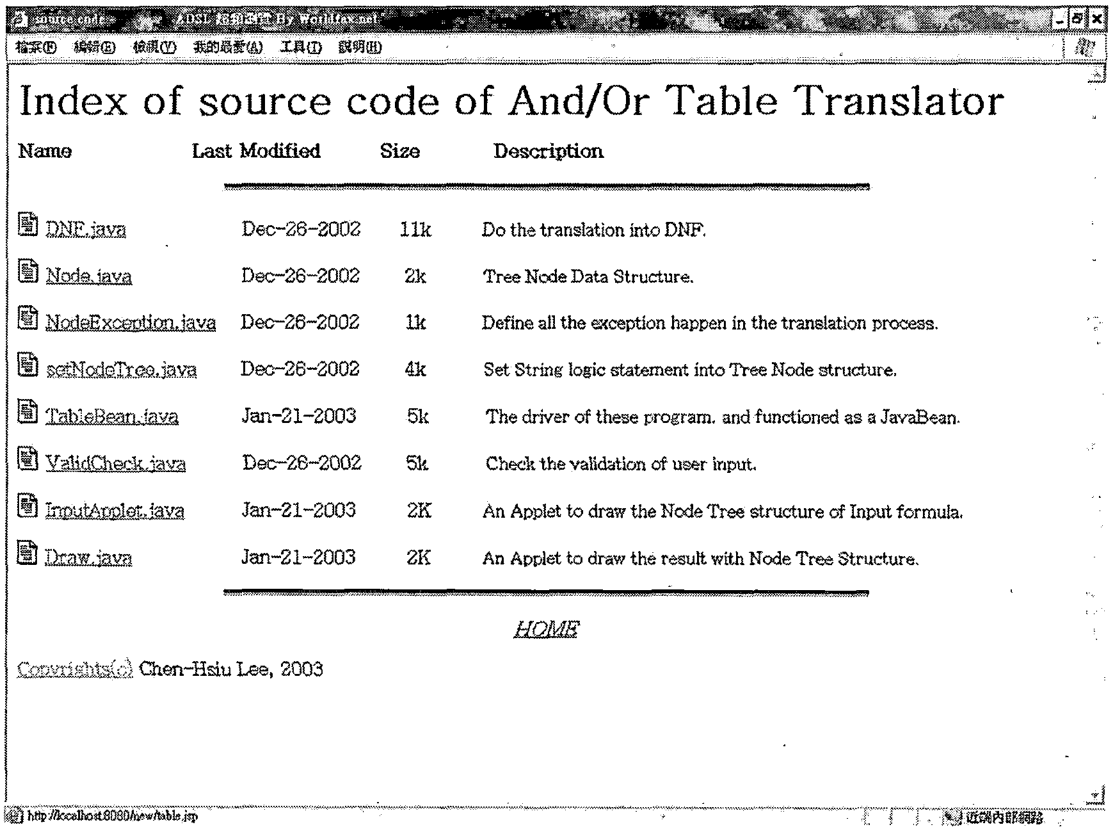


The Result: (A&~C) | (~B&~C)

Figure 21. Applet of Result Page

5.3.7 Source Code Page

Source code page shows the index of source code of And/Or table translator. It contains some necessary information: last modified time, size, and description. User can link to or save each file by clicking the file name. Also it links to the copy right page which declares the copy right of author. See Figure 22 below.



Index of source code of And/Or Table Translator

Name	Last Modified	Size	Description
DNF.java	Dec-26-2002	11k	Do the translation into DNF.
Node.java	Dec-26-2002	2k	Tree Node Data Structure.
NodeException.java	Dec-26-2002	1k	Define all the exception happen in the translation process.
setNodeTree.java	Dec-26-2002	4k	Set String logic statement into Tree Node structure.
TableBean.java	Jan-21-2003	5k	The driver of these program. and functioned as a JavaBean.
ValidCheck.java	Dec-26-2002	5k	Check the validation of user input.
InputApplet.java	Jan-21-2003	2K	An Applet to draw the Node Tree structure of Input formula.
Draw.java	Jan-21-2003	2K	An Applet to draw the result with Node Tree Structure.

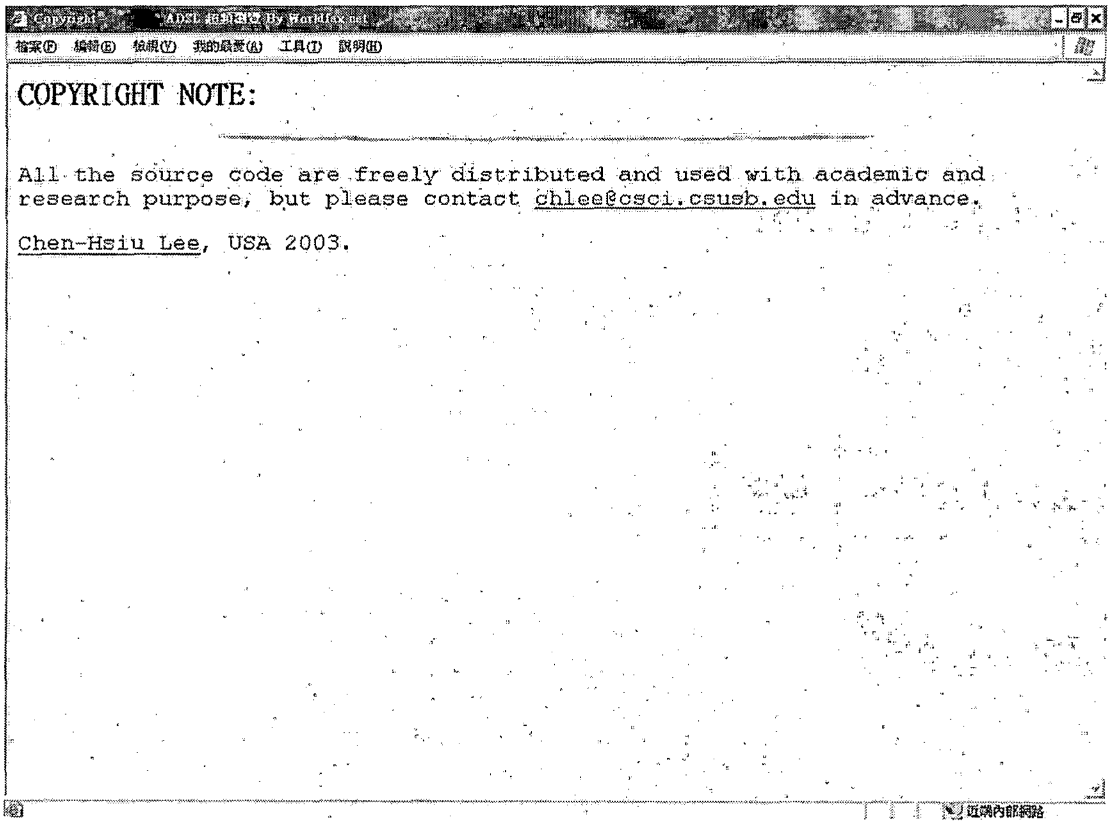
HOME

Copyrights(c) Chen-Hsiu Lee, 2003

Figure 22. Source Code Page

5.3.8 Copy Right Page

Copy right page declares the copy right of this program. See Figure 23 below.


COPYRIGHT NOTE:

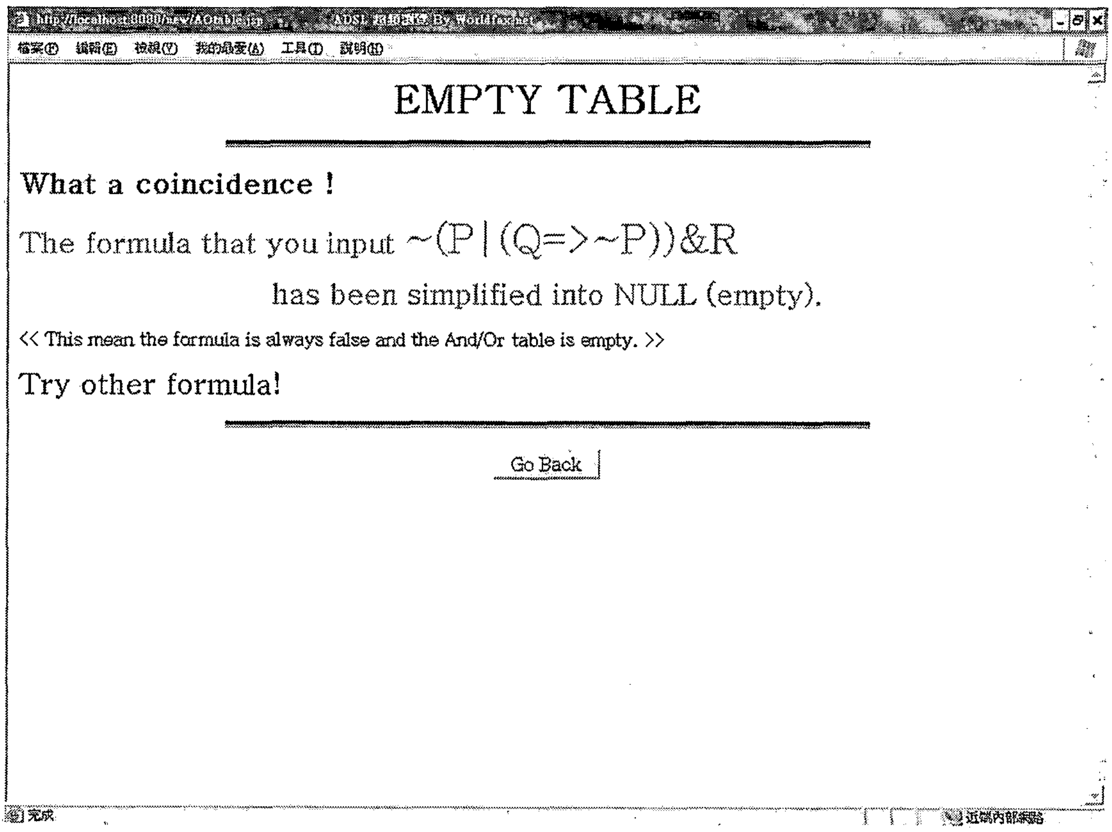
All the source code are freely distributed and used with academic and research purpose, but please contact chlee@csci.csusb.edu in advance.

Chen-Hsiu Lee, USA 2003.

Figure 23. Copy Right Page

5.3.9 Null Node Page

This is rarely happened, however, all situations are considered in this program. When simplifying formula, it happens to eliminate all operand nodes, and winds up with null data. Usually system will refer to error message page. Unfortunately, this is not an error; therefore, this page is specially created for this situation. See Figure 24 below.


EMPTY TABLE

What a coincidence !

The formula that you input ~(P|(Q=>~P))&R

has been simplified into NULL (empty).

<< This mean the formula is always false and the And/Or table is empty. >>

Try other formula!

Figure 24. Null Node Page

CHAPTER SIX

CONCLUSIONS

6.1 Introduction

And/Or Table, developed by Leveson, Heimdahl, and Reese [1], is a particular tabular notation that use true and false to denote the logic of statement. The And/Or Table Translator is one step in the exploration of tools that make formal methods more accessible to software engineers by Dr. Botting.

6.2 Conclusions

This project not only developed a great tool to translate any logic statement into either disjunction normal form or its tabular expression - And/Or Table, but also implement a prototype tool, first introduced by Leveson et al [1] in "Experiences and Lessons from the Analysis of TCAS II" in 1996. The beauty of this project is that users can utilize it to translate a logic statement into a DNF or And/Or Table presentation. This is a complexity process during which higher percentages of errors could be injected. Consequently, this system handles the annoying translation procedure with applets to show the node-tree structure of the logic statement, which makes users have the concept of translation process.

Furthermore, user can simply use web browser anywhere to access this system. With Java object-oriented design, and developed Java Server Page, this project delivers fast, reliable and highly portable web-base system.

In summary, And/Or Table Translator is a designed to be: fast, simple, reusable, portable, and accessible.

6.3 Future Directions

Although And/Or Table Translator developed a usable tool to help users in many aspects, additional and improved features may be added to enhance this system or to explore a further steps in software engineering ideas and tools.

6.3.1 Graphical Animation of Translation

By using Java Applet or Java Swing Package to draw an animation of the translations steps by steps, user can interact with this system to understand the process of translation more. Animation Window provides "next step" and "previous step" button which makes user be able to control the procedure in detail.

6.3.2 Use Binary Decision Diagram

Binary Decision Diagram (BDD) is an alternative way to construct the container of the logic statement and translate into DNF. This could be another project presenting the formal method.

6.3.3 Editing And/Or Table

A future enhancement is to allow AND/OR table to be input and edited directly. These can then be translated into DNF and other normal forms.

6.3.4 Combining And/Or Tables

And/Or tables represent propositions and so can inherit the operations of the propositional calculus. Dr. Botting has developed an algorithm for calculating the combination of two And/Or tables. A future project should explore the implementation of this algorithm.

6.3.5 State Charts

And/Or table was invented to express conditions in complex state charts. A future project should integrate the And/Or objects with a state chart tool.

6.3.6 Extended And/Or Table

Dr. Botting's research indicates that same requirement can be expressed better by replacing "T", "F", and "--" by symbols represented sets of possible values. For example, a variable might be "Too high", "Normal", or "Too low". See Reference [9].

6.3.7 Fuzzy And/Or Tables

There is research needed on using And/Or table to represent fuzzy logic. [9]

APPENDIX

SOURCE CODE

```
/*******************************************************
NAME          : Node.java, BinaryNode.java,
ElementNode.java, NegationNode.java
DESCRIPTION: Define the basic structure of Tree Node,
              which contains BinaryNode, ElementNode,
              NegationNode.
REMARK       : Node class is the base class of BinaryNode, ElementNode,
NegationNode class.
*******************************************************/
package mybean;

import java.io.*;
import java.util.*;

class Node
{
        public Node Left;
        public Node Right;
            public Node Parent;

        protected String data;

        public Node(){
              Parent = null;
              Left = null;
              Right =null;
        }
        public Node(String in){
               data = in;
               Left=null;
               Right=null;
        }
        public String getData(){
               return data;
        }
            public void setData(String newData){
               data = newData;
            }
        public void setParent(Node par){
              Parent = par;
        }
 }
class BinaryNode extends Node
{
        public BinaryNode(String in, String l, String r){
                data = in;
                Left = null;
                Right = null;
        }
          // User can set a Binarynode with both children
          public BinaryNode(String in, Node left, Node right){
                  data = in;
                  Left = left;
                  Right = right;
          }
}

final class ElementNode extends Node
{
        public ElementNode(){
              super();
        }
        public ElementNode(String operand)
        {
                  data = operand;
                  Left =null;
                  Right =null;
        }
}

final class NegationNode extends Node
{
        final Node Right = null;
        public NegationNode(String operand){
                data = "~";
                Left = new ElementNode(operand);
        }
          public NegationNode(Node operand){
                data = "~";
                Left = operand;
           operand.Parent = null;
          }
        public NegationNode(){
              data = "~";
              Left = null;
        }
}
```

```
/*****************************************************
Name:        NodeException.java
Description: Define all the Exception that could happen in the process of
             translating the logic statement into And/Or table
Remark:      All classes here are inherited from
             Object.Throwable.Exception
*****************************************************/
package mybean;

class NodeException extends Exception
{
        public NodeException()
        {}
        public NodeException(String s)
        {
        super(s);
        }
}


class ParentheseException extends Exception
{
        public ParentheseException() {}

        public ParentheseException(String s)
        {
            super(s);
        }
}

class LogicException extends Exception
{
        public LogicException() {}

        public LogicException(String s)
        {
            super(s);
        }
}
```

```
/*****************************************************
*Name:        setNodeTree.java
*Description: Set every single Operand and Operator to
*             the Node, and construct the Tree Node.
*****************************************************/
package mybean;
import java.util.*;

class setNodeTree
{
    private String [] statement = new String[MAX_LEN];
     static final int MAX_LEN = 500;

        // DEFAULT CONSTRUCTOR
     public setNodeTree() { }

     public void toStringToken(String st)
     {
          st.trim();
          StringTokenizer tokenizer =
                    new StringTokenizer(st);
     // THE NUMBER OF STRING TOKEN
          int i=0;
          while(tokenizer.hasMoreTokens())
          {
            statement[i]=(String)tokenizer.nextToken();
            i++;
          }
// CHECK IF THE INPUT STATEMENT IS OVER THE
// CAPACITY OF THIS PROGRAM
          if(i > MAX_LEN)
          {
             throw new IndexOutOfBoundsException("Sorry,
             statement over the limit capacity of this
             program!");
          }
     }// END OF FUNCTION

     public boolean isOperator(String in)
     {
          return ( in.equals("&") ) || (in.equals("|") )
               ||(in.equals("=>"))|| (in.equals("<=>")
               || (in.equals("~")) );
     }
```

```
/*****************************************************
*THIS MEMBER FUNCTION DECIDE THE PRECEDENCE OF THE
*OPERATOR BY GIVING THE VALUE OF EACH OPERATOR.
*****************************************************/
public int evaluate_precedence(String[] op)
{
        String temp ="";
        int braceNumber = 0;
        int value =0;
        int max = -100;
        int return_pos = -200;
        for(int i= 0; op[i]!=null; i++)
        {
             temp = op[i];
             if(temp.equals("("))
             {
                braceNumber++;
                value = -200;
             }
             else if(temp.equals(")"))
             {
                braceNumber--;
                value = -200;
              }
              else if(temp.equals("<=>"))
                   value = 5 + braceNumber*(-10);
              else if(temp.equals("=>"))
                   value = 4 + braceNumber*(-10);
              else if(temp.equals("|"))
                   value = 3 + braceNumber*(-10);
              else if(temp.equals("&"))
                   value = 2 + braceNumber*(-10);
              else if(temp.equals("~"))
                   value = 1 + braceNumber*(-10);
              else value = -200;
        // COULD BE EMPTY STRING,OR "a" ... ALPHABET
             if(value > max)
             {
                max = value;
                return_pos = i;
             }
        }
        return return_pos;
}
```

```
// ASSIGN THE HIGHEST PRECEDENCE OPERATORS
// OR OPERANDS NODE
public Node setNode(String[] array)
{
   int pos = evaluate_precedence(array);
   if(pos==-200)
   {
        for(int i=0; array[i]!=null; i++)
        {
                String operand = array[i];
                // IF THE OPERAND IS "(", ")" OR SPACE
                // THEN DOING NOTHING
                if(operand.equals("(")||operand.equals(")"))
                {}
                else if(operand.charAt(0)>='A'
                                        && operand.charAt(0)<='z')
                {
                Node element = new ElementNode(operand);
                        return element;
                }
        }
        return null;
   }
   else
   {
        String op = array[pos];
        if(op.equals("~"))
        {
             String next = array[pos+1];
             // BECAUSE THIS IS LIKE: ~ (( a & b )........)
             if( next.equals("(") )
             {
                   Node Negate = new NegationNode(op);
                   String[] leftArray=new String[MAX_LEN/2];
                // COPY THE REST OF STRING TO LEFT ARRAY
                // (LEFT CHILD)
                   System.arraycopy(array,(++pos),leftArray,
                                          0,array.length-(++pos));
             // BY DEFINITION OF NEGATION-NODE
             // LEFT CHILD CONTAIN DATA, AND RIGHT CHILD
             // IS NULL
                        Negate.Left = setNode(leftArray);
                        return Negate;
```

```
                    }
                    else
                            // THIS WILL BE THE ELEMENT NODE
                            // EXAMPLE:  ~a, ~d
                    {
                    Node element=new ElementNode(array[pos+1]);
                    Node Negate = new NegationNode(element);
                            array[pos+1] = "";
                            return Negate;
                    }
            }
            else
            // OTHERWISE IT IS A BINARY NODE
            // EXAMPLE: &, | , =>, OR <=> CONTAIN WITH BOTH
// CHILDREN
            {
                    String l="", r="";
                    Node root = new BinaryNode(op,l,r);
                    // COPY LEFT-HAND-SIDE STRING,
                    //AND RECURRSIVE CALL THIS FUNCTION
                    String[] leftArray = new String[MAX_LEN/2];
                    // COPY RIGHT-HAND-SIDE STRING,
                    //AND RECURRSIVE CALL THIS FUNCTION
                    String[] rightArray = new String[MAX_LEN/2];
                   System.arraycopy(array,0,leftArray,0,pos);
                    System.arraycopy(array,(++pos),rightArray,
0, array.length-(++pos));
                    root.Left = setNode(leftArray);
                    root.Right = setNode(rightArray);
                    return root;
             }
            }
    }

    // TO ASSIGN THE PARENT OF EVERY NODE, MAKE
// CONNECTION LIKE LINK LIST.
    public void assignParent(Node root)
    {
            if(root == null) return;
            if(root.Left == null);
            else
                    root.Left.Parent = root;
            if(root.Right == null);
            else
```

```
                    root.Right.Parent = root;
                assignParent(root.Left);
                assignParent(root.Right);
        }
        public String[] get_statement()
            {
                    return statement;
            }
} // END OF CLASS  SETNODETREE
```

```
/*********************************************
*Name:        ValidCheck.java
*Description: Get the input, and check if any
*             invalid logic statement exist. If
*             so, throw the corresponding
*             exception of the error.
*             If not, doing nothing.
*********************************************/

package mybean;

import java.util.StringTokenizer;
import java.io.*;


public class ValidCheck
{
        // String input is the original input from user
        private String input="";
//

        private String newString ="";

        public ValidCheck(){}
        public ValidCheck(String s)
        {
           this.input = s;
        }

        public void givingSpace() throws Exception
        {
           int len = input.length();
           int lBrace = 0;
           int rBrace = 0;
           char pre = ' ';

           for(int i=0; i<len; i++)
           {
              char a = input.charAt(i);

              if(a==' ') newString+=" ";

                else
                        {
```

```
// ACCEPT OPERATOR "<=>"
if(a=='<')
{
     if(input.charAt(++i)=='='
             &&input.charAt(++i)=='>'
             &&(isOperand(pre)|| pre==')'))
     {
        newString+=" <=> ";
        pre = '>';
     }
     else throw new LogicException("Not
     a valid logic operand or perator!");
}
// ACCEPT OPERATOR "=>"
else if(a=='='&&input.charAt(++i)=='>')
{
     newString+=" => ";
     pre = '>';
}
else if(a=='~')
{
     newString = newString+" ~ ";
     pre ='~';
}
else if(a=='(')
{
     lBrace++;
     newString+= " ( ";
     pre = '(';
}
else if(a==')')

{
   rBrace++;
     newString+= " ) ";
     pre = ')';
}
else if( isOperand(a) )
{
   if(isOperand(pre))
   {
        newString+=a;
        pre = a;
     }
```

```
                    else{
                        newString +=" "+a;
                        pre = a;
                    }
                }
                else if(a =='&' || a=='|')
                {
                    if( isOperand(pre)|| pre==')')
                    {
                        newString += " "+a;
                        pre = a;
                    }
                }
                else
                    throw new LogicException("Not a
                      valid logic operand or operator!");
                    }// End of elseif
            } // End of for loop
            if(lBrace != rBrace)
                    throw new ParentheseException("
                    Unbalanced parentheses!");
        }

        private boolean isOperand(char a)
        {
            return (a>=65&&a<=90)||(a>=97&&a<=122);
        }
        private boolean isOperator(char a)
        {

            return a=='&'||a=='|'||a =='<'||a=='>'
    ||a=='=';
        }
        /********************************************
        * MEMBER FUNCTION validation()
        * DESCRPTION: ACCORDING TO THE LOGIC RULE,
        * CHECK THE RELATIONSHIP BETWEEN EVERY
        * OPERATOR AND OPERAND. IF IT BREAKS RULES,
        * THROWS CORRESPONDING EXCEPTIONS.
        ********************************************/
        public void validation() throws Exception
        {
            String temp="";
            String pre ="";
```

```
   StringTokenizer  tokenizer =
                    new StringTokenizer(newString);
   int NumToken = tokenizer.countTokens();
   int count = 0;

   if(NumToken==0) throw
                   new Exception("Empty input!");
  while(tokenizer.hasMoreTokens())
   {
      String token = tokenizer.nextToken();
       count++;

if(token.equals("<=>")||token.equals("=>"))
       {
       if(pre.equals("~")||pre.equals("<=>")
              ||pre.equals("=>")|| pre.equals("&")
              ||pre.equals("|")|| pre.equals("("))
              throw new LogicException("There
              must be an operand right before the <=> (imply) operator
              or => operator!");
       else if(count==NumToken) throw
                     new LogicException("lost an
                     operand in the last position");
        else if(pre.equals(")")||isOperand(pre))
          {}
       else throw new LogicException("Wrong
              logic statement, Check <=> or =>
              operator!");
       }
       else if(token.equals("&")||
                            token.equals("|"))
       {
        if(count==NumToken) throw new
              LogicException("lost an operand in
              the last position");
          if(pre.equals(")") || isOperand("pre"))
          {}
        else throw new LogicException("Wrong
              logic statement,Check &(and),|(or)
              operator!");
       }
       else if(token.equals("~"))
       {
          if(pre.equals(")")|| isOperand(pre)
```

```
                    || pre.equals("~"))
              throw new LogicException("Negation
              error!");
           else if(count == NumToken)
               throw new LogicException("lost an
               operand in the last position");
        }
        else if(token.equals("("))
        {
           if(pre.equals(")")|| isOperand(pre))
              throw new LogicException("Must have
              an operand right before right-
              parenthese!");
           else if(count == NumToken)
              throw new LogicException("lost an
              operand in the last position");
        }
        else if(token.equals(")"))
        {
           if(pre.equals("(")||pre.equals("<=>")
              ||pre.equals("|")||pre.equals("&")
              ||pre.equals("=>")||pre.equals("~"))
              throw new LogicException("Must have
              an operand right before left-
              parenthese!");
        }
        else if(isOperand(token))
        {
           if(isOperand(pre) || pre.equals(")"))
             throw new LogicException("Wrong
             logic statement! Lost an operator!");
        }
        else throw new LogicException("Not a
              valid operand or operand!");
        pre = token;
   } //End of while loop
}

private boolean isOperand(String s)
{
        if(s.equals("")) return false;
        else{
              for(int i=0; i<s.length(); i++){
                    if(! isOperand(s.charAt(i)) )
```

```
                                        return false;
                              }
                              return true;
                    }
          }
          public String getString(){
                    return newString;
          }
} // END OF CLASS VALIDCHECK
```

```
/*****************************************************
 * DNF.java
 * This class is basically doing translation to form a DNF
 * There are 4 steps
 *      1: Eliminate biconditionals and conditional
 *      2: Drive in negation
 *      3: Distributing conjunctions over disjunctions
 *      4: Simplifying.
 *****************************************************/
package mybean;

import java.util.*;

public class DNF
{
        public Node root;
        private setNodeTree M;

        /*********************************************
        * CONSTRUCT THE TREE-NODE BEFORE
        * TRANSLATING INTO DNF INITIALIZATION
        * TAKES PLACE INSIDE THE CLASS DNF CONSTRUCTOR
        *********************************************/
        public DNF(){}
        public DNF(String input)
        {
            M = new setNodeTree();
            M.toStringToken(input);
            root = M.setNode( M.get_statement() );
               M.assignParent(root);
        }

        /************************************************
        PRIVATE MEMBER FUNCTION: ContainOf()
        RETURN: Node object
        This function helps another member function,
        Eliminate_bicondition(), to find the Node contains
        specific String data, and return this node.
        ************************************************/
        private Node ContainOf(Node root, String find)
        {
           if(root == null) return null;
           else if(root.getData().equals(find) )
                return root;
```

```
  else
  {
        Node temp = ContainOf(root.Left, find);
        if (temp != null)       return temp;
        else return ContainOf(root.Right, find);
  }
}
/************************************************
* MEMBER FUNCTION : ELIMINATE_BICONDITION()
* DESCRIPTION      : THE FIRST STEP OF TRANSLATION
************************************************/
public void Eliminate_bicondition()
{
    // CHECK IF THIS NODE-TREE CONTAIN "<=>"
   Node temp = ContainOf(root,"<=>");
   // IF NOT, DOING NOTHING;
   if(temp == null)    return;
    // IF FIND, MANIPULATE DATA
    Node temp_left;
   Node temp_right;

   while(temp != null)
   {
          temp_left = temp.Left;
          temp_right = temp.Right;
          // "A<=>B" NODE CAN BECOME " ~(A|B)&~(A|B)"
          temp.setData("&");
      temp.Left = new BinaryNode("|",
             new NegationNode(temp_left),temp_right);
      temp.Right = new BinaryNode("|", temp_left,
                   new NegationNode(temp_right) );
      temp = ContainOf(root,"<=>");
   }
   // RE-ASSIGN PARENT OF THIS TREE-NODE
     M.assignParent(root);
}
/************************************************
  * MEMBER FUNCTION : Eliminate_condition()
  * DESCRIPTION:    THE SCEOND STEP OF TRANSLATION
  ************************************************/
  public void Eliminate_condition()
  {
       // CHECK IF THIS NODE-TREE CONTAIN NODE "=>"
       Node temp = ContainOf(root,"=>");
```

```
         // IF NOT, DOING NOTHING
       if(temp == null)    return;
       // IF FOUND, MANIPULATE DATA.
       // EXAMPLE: A=>B BECOMES " (~A|B)
       Node temp_left;
         while(temp != null)
       {
         temp_left = temp.Left;
         temp.setData("|");
         Node child = new NegationNode(temp_left);
         child.setParent(temp);
         temp.Left = child;
         temp_left.setParent(child);
         temp = ContainOf(root,"=>");
       }
    }
    /**************************************************
    * MEMBER FUNCTION : drivingInNegation()
    * DESCRIPTION: THE THIRD STEP OF TRANSLATION
    * THROW NodeException
    **************************************************/
    public void drivingInNegation() throws NodeException
    {
       // FIND ANY NEGATION NODE BEFORE BINARYNODE
      Node found = findNegation(root);
      // WHILE FIND, PROCESS DATA
      while(found != null)
      {
         // TEMPORARY STORAGE OF DATA
        Node found_left;
        Node found_right;
         Node found_parent;

         // IF FOUND NODE IS "&"
         // EXAMPLE: ~(A&B) BECOMES ~A|~B
        if(found.Left.getData().equals("&"))
        {
           // SAVE TEMPORARY DATA BEFORE MANIPULATE IT
            found_parent = found.Parent;
            found_left = found.Left.Left;
            found_right = found.Left.Right;

            // IF THE NODE FOUND IS ROOT NODE
            if(found == root)
        {
            root = new BinaryNode("|",
                            new NegationNode(found_left),
                            new NegationNode(found_right));
        }
        // ELSE IF THE NODE FOUND IS A LEFT-CHILD OF
        // A PARENT-NODE
      else if(found == found_parent.Left)
      {
            found_parent.Left = new BinaryNode("|",
                                  new NegationNode(found_left),
                           new NegationNode(found_right));
      }
        // ELSE IF THE NODE FOUND IS A RIGHT-CHILD OF
        // A PARENT-NODE
      else if(found == found_parent.Right)
      {
          found_parent.Right = new BinaryNode("|",
                           new NegationNode(found_left),
                           new NegationNode(found_right));  }
      else {}
   }
   // IF FOUND NODE IS "|"
      // EXAMPLE: ~(A|B) BECOMES ~A&~B
   else if(found.Left.getData().equals("|"))
   {
          found_left = found.Left.Left;
           found_right = found.Left.Right;
          found_parent = found.Parent;

          if(found == root)
                root = new BinaryNode("&",
                           new NegationNode(found_left),
                           new NegationNode(found_right));
          else if(found == found_parent.Left)
            found_parent.Left = new BinaryNode("&",
                           new NegationNode(found_left),
                           new NegationNode(found_right));
          else if(found == found_parent.Right)
            found_parent.Right = new BinaryNode("&",
                           new NegationNode(found_left),
                           new NegationNode(found_right));
          else {}
    }
```

```
// IF FOUND NODE IS " ~ "
  // EXAMPLE: ~ (~A) BECOMES A
else if(found.Left.getData().equals("~"))
{
    found_parent = found.Parent;

    if(found == root)
    {
          Node newChild = found.Left.Left;
          // IF THE NEW CHILD IS A BINARY NODE
          if(newChild instanceof BinaryNode)
                    root=new BinaryNode(newChild.getData(),
                              newChild.Left, newChild.Right);
          // IF THE NEW CHILD IS A NEGATION NODE
          else if(newChild instanceof NegationNode)
                    root = new NegationNode(newChild.Left);
    }
    // ELSE IF THE NODE FOUND IS A LEFT-CHILD OF
    // A PARENT-NODE
    else if(found == found_parent.Left)
    {
        found_left = found.Left.Left;
        if(found_left instanceof ElementNode)
                found_parent.Left = new
                              ElementNode(found_left.getData());
        else if(found_left instanceof NegationNode)
                found_parent.Left = found_left;
          else if(found_left instanceof BinaryNode)
                found_parent.Left =
                    new BinaryNode(found_left.getData(),
                    found_left.Left, found_left.Right);
          else {}
    }
    // ELSE IF THE NODE FOUND IS A RIGHT-CHILD OF
    // A PARENT-NODE
    else if(found == found_parent.Right)
    {
        found_right = found.Left.Left;
        if(found_right instanceof ElementNode)
                found_parent.Right =
                    new ElementNode(found_right.getData());
        else if(found_right instanceof NegationNode)
                found_parent.Right = found_right;
          else if(found_right instanceof BinaryNode)
```

```
                        found_parent.Right =
                            new BinaryNode(found_right.getData(),
                          found_right.Left, found_right.Right);
                    else {}
                  }
              }
              else{ }
              // re-assign parent node
             M.assignParent(root);
             // find the next candidate node
             found = findNegation(root);
              }// End of while loop
         }


         /***********************************************
         * MEMBER FUNCTION: findNegation()
         * DESCRIPTION    : FIND THE NEGATION NODE WITH
         *                             LEFT-CHILD CONTAIN "&","|",
         *                             OR "~". USE RECURRSIVE CALL.
         ***********************************************/
         private Node findNegation(Node root)
        {
           if(root == null)    return null;
          else if( root.getData().equals("~")
                      && root.Left instanceof ElementNode)
                   return null;
           else if(root.getData().equals("~")
                          && root.Left instanceof BinaryNode)
                   return root;
           else if(root.getData().equals("~")
                          && root.Left instanceof NegationNode)
              return root;
           else
           {
              Node temp = findNegation(root.Left);
              if (temp != null) return temp;
              else return findNegation(root.Right);
           }
        }
      /***********************************************
      * FUNCTION: distributeConjunctionOverDisjunction()
      * DESCRIPTION  : THE FOURTH STEP OF TRANSLATION
      ***********************************************/
```

```
public void distributeConjunctionOverDisjunction()
{
 Node node = findConjunction(root);
 while(node!=null)
 {
        if(node.getData().equals("&")
            && node.Left.getData().equals("|")
            && node.Right.getData().equals("|") )
        // this will translate a disjunction with
        // both conjuction children
        {
          // TEMPORARY STORAGE
          Node ll = node.Left.Left;
          Node lr = node.Left.Right;
          Node rl = node.Right.Left;
          Node rr = node.Right.Right;

          node.setData("|");
          node.Left = new BinaryNode("|",
                                new BinaryNode("&",ll,rl),
                                new BinaryNode("&", ll, rr));
          node.Right = new BinaryNode("|",
                                new BinaryNode("&",lr, rl),
                                new BinaryNode("&", lr, rr));
        }
        else if(node.getData().equals("&")
                    && node.Left.getData().equals("|"))
        {
          Node ll = node.Left.Left;
          Node lr = node.Left.Right;
          Node rightChild = node.Right;
          node.setData("|");
          node.Left=new BinaryNode("&", ll,rightChild);
          node.Right=new BinaryNode("&", lr,rightChild);
        }
        else if(node.getData().equals("&")
                 &&node.Right.getData().equals("|"))
        {
          Node rl = node.Right.Left;
          Node rr = node.Right.Right;
          Node leftChild = node.Left;
          node.setData("|");
          node.Left= new BinaryNode("&",leftChild,rl);
          node.Right= new BinaryNode("&",leftChild,rr);
```

```
            }
            node = findConjunction(root);
      }// End of while loop
        M.assignParent(root);
    }

    private Node findConjunction(Node node)
    {
        if(node==null||node.Left==null||node.Right==null)
            return null;
      else if((node.getData().equals("&")
                  && node.Left.getData().equals("|"))
                  ||(node.getData().equals("&")
                  && node.Right.getData().equals("|")) )
            return node;
        else
        {
            Node temp = findConjunction(node.Left);
            if (temp != null)        return temp;
            else return findConjunction(node.Right);
        }
    }
/*************************************************
* MEMBER FUNCTION : simplifying()
* DESCRIPTION  : THE FIFTH STEP OF TRANSLATION
* THROW NullNodeException, and Exception
*************************************************/
    public void simplifying()
    throws NullNodeException, Exception
    {
        Node found = find(root);
        while(found != null)
        {
          // THE IF STATEMENT BELOW WILL HANDLE
          // THE EMPLTY RESULT WHEN SIMPLIFYING,
          // THE FORMULA COULD BE ELIMINATED INTO NULL.
          if(found.Parent == null)
            throw new NullNodeException();

          Node found_parent = found.Parent;
          if(found_parent.Left == found)
          {
              if(found_parent == root)
              {
                // IF THE PARENT OF NODE "FOUND" IS ROOT
                    root = found_parent.Right;
                      root.Parent = null;
                }
                else
                {
                      Node grandpa = found_parent.Parent;
                      if(grandpa.Left == found_parent)
                      grandpa.Left = found_parent.Right;
                      else grandpa.Right = found_parent.Right;
                  }
            }
            else
            {
                if(found_parent == root)
                {
                // IF THE PARENT OF NODE "FOUND" IS ROOT
                root = found_parent.Left;
                root.Parent = null;
                }
                else
                {
                      Node grandpa = found_parent.Parent;
                      if(grandpa.Left == found_parent)
                              grandpa.Left = found_parent.Left;
                      else grandpa.Right = found_parent.Left;
                }
            }
            M.assignParent(root);
          found = find(root);
        } //END OF WHILE LOOP
    }// END OF FUNCTION

    public Node find(Node found)
    {
        if(found == null)   return null;
        else if(found.getData().equals("&")
                                    && find_multi(found))
                    return found;
        else
        {
            Node temp = find(found.Left);
            if(temp != null)        return temp;
            else return find(found.Right);
        }
    }

    public boolean find_multi(Node multiNode)
    {
     final Vector op = new Vector();
       final Vector nop = new Vector();

       // INNER CLASS SearchLeft
       // THIS CLASS WILL GO OVER WHOLE NODE TREE
       // AND FIND THE DATA OF ITS OFFSPRING.
       class SearchLeaf
     {
        public SearchLeaf(){     }
        public void search(Node node)
        {
             if(node == null) return;
             else
             {
              // HANDLE LEFT SIDE
                if(node.Left.getData().equals("&"))
                        search(node.Left);
                else
                {
                     if(node.Left.getData().equals("~"))
                          nop.add(node.Left.Left.getData());
                     else op.add(node.Left.getData());
              }
              // HANDLE RIGHT SIDE
                     if(node.Right.getData().equals("&"))
                            search(node.Right);
                     else
                     {
                        if(node.Right.getData().equals("~"))
                               nop.add(node.Right.Left.getData());
                        else op.add(node.Right.getData());
                   }
          }
        }
     } // End of inner class SearchLeaf

new SearchLeaf().search(multiNode);
```

```
          for(int i=0; i<op.size(); i++)
          {
                  if(nop.contains(op.get(i)))
                          return true;
          }
          return false;
          }// Enf of find_multi() member function

        public Node getResult()
        {
          return root;
        }

  }// End of class DNF
```

```
/*******************************************************
Name: InputApplet.java
Description: To perform drawing of Node-Tree
in the applet.
*******************************************************/
import java.awt.*;
import java.util.*;
import java.util.StringTokenizer;
import java.applet.Applet;

public class InputApplet extends Applet
{
        String s ="";
        public void init()
        {
                s = getParameter("appletString");
        }

        public void paint(Graphics g)
        {
                StringTokenizer tokenizer =
new StringTokenizer(s);

                Vector coordinator  = new Vector();
                String token = "";
                int x=0, y=0;
                int x1=0, y1=0, x2=0, y2=0;
                int number=0;
                while(tokenizer.hasMoreTokens())
                {
                        token = tokenizer.nextToken();

                        x = Integer.parseInt(tokenizer.nextToken());
                        y = Integer.parseInt(tokenizer.nextToken());

                        g.setColor(Color.red);
                        g.setFont(new Font("Ariel",Font.BOLD,24));
                        g.drawString(token, x, y);
                        g.setFont(new Font("Ariel",Font.PLAIN,12));
                        if(token.equals("~"))
                        {
                        x1=Integer.parseInt(tokenizer.nextToken());
                        y1=Integer.parseInt(tokenizer.nextToken());
                                g.setColor(Color.blue);
                                g.drawLine(x,y,x1,y1);
                        }
                        else{
                                token = tokenizer.nextToken();
                                if(token.equals("END")){}
                                else{
                                        x1 = Integer.parseInt(token);
                                        y1 =
Integer.parseInt(tokenizer.nextToken());
                                        g.setColor(Color.blue);
                                        g.drawLine(x,y,x1,y1);
                                        x2 =
Integer.parseInt(tokenizer.nextToken());
                                        y2 =
Integer.parseInt(tokenizer.nextToken());

g.drawLine(x,y,x2,y2);
                                }
                        }
                }// END Of while

        }
}// END OF CLASS InputApplet
```

```
/*****************************************************
* Name: TableBean.java
* Description: This class is the driver of DNF
*               translator. Also, it performs as a
*               JavaBean.
* Remark:      This class implement Serializable
*              interface.
*****************************************************/

package mybean;

import java.io.*;
import java.util.*;

public class TableBean implements Serializable
{
        public ValidCheck checker;
        public DNF driver;
        public String OString = ""; // Original input logic statement

      private Node root;
        private int column = 1;
      private int row = 0;
        private String[][] Table;
        private String  s = "";

        String  finalResult="";
        Vector element = new Vector();
        // Container element contains all the operands in the statement

        public TableBean()
        {}

        public void DNFDriver() throws
                                NullNodeException, Exception
        {
                checker = new ValidCheck(OString);
                checker.givingSpace();
                checker.validation();

                OString = checker.getString();
             driver = new DNF(OString);

        // To get the original input node tree
```

```
    // structure in order to draw in the applet1
            NodeToAppletString(driver.root);

            // fIVE STEPS TO TRANSLATE A DATA INTO DNF
         driver.Eliminate_bicondition();
         driver.Eliminate_condition();
         driver.drivingInNegation();
          driver.distributeConjunctionOverDisjunction();
          driver.simplifying();
            root = driver.getResult();
    }

// TWO STEPS TO TRANSFER A TREE-NODE DATA STRUCTURE
// INTO TABULAR STRUCTURE THIS IS FIRST STEP. USE
// RECURSIVE CALL TO DECOMPOSED THE TREE-NODE.
    private void setGrid(Node node)
    {
            if(node == null) return;
            else
            {
                    if(node.getData().equals("|"))
                            column++;
                    else if(node instanceof ElementNode)
                    {
                            if(!element.contains(node.getData()))
                            element.add(node.getData());
                    }
                    setGrid(node.Left);
                    setGrid(node.Right);
            }
    }
// SECOND STEP OF TRASLATION. WRITE IN THE "T", "F",
// AND "--" TO REPRESENT THE DATA
    private void setTable()
    {
            Table[0][0] = "Condition";
            for(int i=1; i<column+1; i++)
                    Table[i][0] = ""+i;
            // SORT THE ELEMENT ALPHABETICALLY.
            Collections.sort(element);
            for(int i=1; i< row; i++)
                    Table[0][i] =(String)element.get(i-1);
            Vector v = new Vector();
            StringTokenizer tokenizer = new
```

```
                                        StringTokenizer(s);
                    int columnindex = 1;
                    String check;
                    while(tokenizer.hasMoreTokens())
                    {
                         String token = tokenizer.nextToken();
                         check ="";

                         for(int j=1; j<row; j++)
                         {
                            for(int i=0; i<token.length(); i++)
                            {
                              int datalen=Table[0][j].length();

                              if(token.length()< i+datalen);
                              // DOING NOTHING
                              else if (token.substring(i,
                                i+datalen).equals(Table[0][j]))
                              {
                               if(token.length()>(i+datalen)
                               && token.charAt(i+datalen)== '~')
                                    {
                                     Table[columnindex][j]="F";
                                     check+="F";
                                    }
                                    else{
                                         Table[columnindex][j] = "T";
                                         check+="T";
                                     }
                              }
                             }// END OF FOR LOOP

                             if(Table[columnindex][j] == null){
                              Table[columnindex][j] = "--";
                                   // indicate DON'T CARE
                              check+="-";
                         }// END OF WHILE LOOP
          }
          /****************************************************
           * This section is for checking the logic unit
           * (a single conjunction)
           ****************************************************/
                          if(v.contains(check)){
                              if(columnindex == column)
```

```
                                column = column-1;
                            // else overwrite this column by next
                            // new column data
                    }
                    else{
                            v.add(check);
                            columnindex++;
                    }
            }
            column = columnindex-1;
    }
/***************************************************
Translate the result from Node Tree structrue into
String format. The String s is ready for tabulating to
AND/OR Table. The other String finalResult is for user
to check the final answer before tabulation.
***************************************************/
        private void NodeToString(Node node)
        {
            if(node == null) return;
            if(node.Left != null)
            {
                if(node.getData().equals("~"))
                finalResult+="~";
                NodeToString(node.Left);
            }
            if(node.getData().equals("|"))
            {
                s += "  ";
                finalResult+=" ) | ( ";
            }
            else
            {
                s+= node.getData();
                if(node.getData().equals("~")) finalResult+="";
                else finalResult+= node.getData();
            }

            if(node.Right != null){
                        if(node.getData().equals("~"))
                                finalResult+="~";
                        NodeToString(node.Right);
             }
        }
```

```
/******************************************************
* THESE TWO FUNCTIONS COLLECT THE TREE-NODE, AND
* TRANSFER IT INTO STRING. PASS IT TO APPLET.
* BECAUSE APPLET RECEIVE ONLY STRING WITHOUT USING RMI.
******************************************************/
    public String DrawString ="";
    private int x = 400;
    private int y = 0;
    private int range = 400;

    public void NodeToDrawString(Node node)
    {
            y += 50;
            range = range/2;
            if(node == null) return;
            if(node != null){
                    DrawString+= " "+node.getData();
                    DrawString+= " "+x;
                    DrawString+= " "+y;
                    if(node.Left != null)
                    {
                            int x1;
                            int y1 = (y + 50);
                            if(node.getData().equals("~"))
                                    x1 = x;
                            else x1 = x-range/2;
                            DrawString+= " "+x1;
                            DrawString+= " "+y1;
                    }
                    else DrawString+=" END";
                    if(node.Right != null)
                    {
                            int x2 = (x + range/2);
                            int y2 = (y + 50);
                            DrawString+= " "+x2;
                            DrawString+= " "+y2;
                    }
            }
            if(node.Left != null){
                    if(node.getData().equals("~"))
                    {}
                    else x = x-range/2;
                    NodeToDrawString(node.Left);
                    if(node.getData().equals("~"))
                    {}
                    else x+=range/2;
                    }

            if(node.Right != null)
            {
                    x+=range/2;
                    NodeToDrawString(node.Right);
                    x-=range/2;
                    }
            y-= 50;
            range = range*2;
    }
    public String appletString ="";
    private int a = 400;
    private int b = 0;
    private int range2 = 400;

    public void NodeToAppletString(Node node)
    {
            b += 50;
            range2 = range2/2;
            if(node == null) return;
            if(node != null){
                    appletString+= " "+node.getData();
                    appletString+= " "+a;
                    appletString+= " "+b;
                    if(node.Left != null)
                    {
                            int x1;
                            int y1 = (b + 50);
                            if(node.getData().equals("~"))
                                    x1 = a;
                            else x1 = a-range2/2;
                            appletString+= " "+x1;
                            appletString+= " "+y1;
                    }
                    else appletString+=" END";
                    if(node.Right != null)
                    {
                            int x2 = (a + range2/2);
                            int y2 = (b + 50);
                            appletString+= " "+x2;
                            appletString+= " "+y2;
```

```
                }
            }
            if(node.Left != null){
                if(node.getData().equals("~"))
                {}
                else a = a-range2/2;
                NodeToAppletString(node.Left);
                if(node.getData().equals("~"))
                {}
                else a+=range2/2;
                }
            if(node.Right != null)
            {
                a+=range2/2;
                NodeToAppletString(node.Right);
                a-=range2/2;
                }
            b-= 50;
            range2 = range2*2;
    }
/*******************************************************
* MEMBER FUNCTION : setOString
* THIS FUNCTION IS LIKE THE "MAIN FUNCTION" OF ALL THE
* CLASS IN mybean PACKAGE. IT TRIGGER ALL THE CLASS BY
* INPUT A STRING.
*****************************************************/

    public void setOString(String OString)
            throws Exception
    {
            this.OString = OString;
            DNFDriver();
            setGrid(root);
            row = element.size()+1;
            Table = new String[column+1][row];

            finalResult+="(";
            NodeToString(root);
            finalResult+=")";

            // ELIMINATE THE DUPLICATED COLUMN AND TWO
            // COLUMNS CONTAIN "T" AND "F" IN SAME POSITION.
            setTable();
            SimTable Sim = new SimTable(Table,column+1,row);
```

```
			Sim.simplified();
			Table =Sim.ReturnTable();
			column = Sim.ReturnColumn();
			// DRAW THE RESULT IN A SECOND APPLET
			NodeToDrawString(root);
		}

		public String getOString(){
			return OString;
		}
		public String getfinalResult(){
			return finalResult;
		}
		public int getcolumn(){
			return column;
		}
		public int getrow(){
			return row;
		}
		public String[][] getTable(){
			return Table;
		}
		public String getDrawString(){
			return DrawString;
		}
		public String getAppletString(){
			return appletString;
		}
	}
```

```
/*****************************************************
* Name:             SimTable.java
* Description:      This class can simplify the data after
*                   tabulated. For example, if a column in
*                   the table is "TTF", it can eliminate
*                   with another column with data "FTF".
*                   And the result become "--TF".
*                   Also, this class check the duplicated
*                   data, and merge into a single data.
*****************************************************/
package mybean;

import java.util.*;

class  SimTable
{
        public String [][] table;
        private int col, row;

        public SimTable()
        {       }
        //      Constructor with input of two-dimension String
        //      and two integers: column number and row number.
        public SimTable(String[][] T, int C, int R)
        {
                table = T;
                col = C;
                row = R;
        }
        public void simplified()
        {
                for(int i=0; i<col; i++)
                {
                        if(i!=col-1)
                        {
                                for(int j=i+1; j<col; j++)

                                        if(compare(i, j))
                                        {
                                                eliminate_table(j);
                                                i--;
                                        }
                        }
                }
```

```
}

private boolean compare(int a, int b)
{
        int NumOfMatch =0;
        int NotMatch   =0;

        for(int j=0; j<row; j++)
        {
                if(table[a][j]==table[b][j])
                        NumOfMatch++;
                else NotMatch = j;
                // RECORD THE ROW NUMBER
                // IF IT MEETS THE ELIMINATE CONDITION
                // THIS IS THE ONE TO BECOME "DON'T CARE"
        }
        if(NumOfMatch == row-2)
        // This indicate that two columns can be merged
        // into one columns.
        {
                table[a][NotMatch] = "--";
                return true;
        }
        else if(NumOfMatch == row-1)
        // This indicate a duplicate column.
                return true;
        else return false;
}

private void eliminate_table(int deletCol)
{
        String [][] newTable = new String[col][row];
        int newCol=0;
        // USE newCol, BECAUSE COLUMN NUMBER MAY SHRINK.
        // IF ANY TWO COLUMNS CAN BE ELIMINATED.
        for(int i=0; i<col; i++)
        {
                if(i==deletCol && i!=0)newCol--;
                else
                for(int j=0; j<row; j++)
                {
                        newTable[newCol][j] = table[i][j];
                }
                newCol++;
```

```
		}

		table = newTable;
		col = newCol;
	}

	public String[][] ReturnTable()
	{
		return table;
	}
	public int ReturnColumn()
	{
		return col;
	}
}
```

REFERENCES

[1] "Requirements Specification for Process-Control System" by Nancy G Leveson, Mats Per Erik Heimdahl, Holly Hildreth, Jon Damon Reese, IEEE Trans SE VSE-20n9, Sep 1994, pp684-707.

[2] "Designing Specification Language for Process Control System" by Nancy G Leveson, Mats Per Erik Heimdahl, Jon Damon Reese, Oscar Nierstrasz, and Michel Lemoine, 7th European Software engineering Coference, and 7th ACM SIGSOFT Symposium on the Foundations ACM SIGSOFT Software eng Notes V24n6, Nov 1999.

[3]. "Experiences & Lessons from the Analysis of TCAS II", by Mats P E Heimdahl, Porc 1996 Int Symposium on Software Testing & Analysis(ISSTA) and ACM SIGSOFT SENote V21n3, 1996 May, pp79-83.

[4] "Logic in Computer Science" by Michael R A Huth and Mark D Ryan, Cambridge University Press, ISBN 0- 521-65602-8.

[5] "Computer Architecture and Organization", by John P. Hayes, McGraw-Hill, ISBN 0-07-115997-5.

[6] "Java How To Program" by Deitel and Deitel, Prentice Hall, ISBN 0-13-304151-7.

[7] "UML Distilled 2nd Edition: A Brief Guide to The Standard Object Modeling Language", by Martin Fowler and Kendall Scott, Addison Wesley Longman, Inc. ISBN 0-201-65783-X.

[8] "Computing Concepts with JAVA2 Essetials 2nd" by Cay Horstmann, John Wiley & Sons, Inc. ISBN 0-471-34609-8.

[9] "Beginning JSP Web Development" by Jayson Falkner, Ben Galbraith, Romin Irani, ISBN 1861002092.

[10] "Tabular Notations", by Richard J Botting, <http://www.csci.csusb.edu/dick/maths/notn_9_Tables.html>