Ain Shams University
Faculty of Engineering
Discipline Programs

EDA Project 2 Report

Computer Engineering and Software Systems (CESS)

Submitted to:

Dr. Eman El Mandouh

Eng. Adham

Submitted by:

Mohamad Ahmed Mohamad Abdelmoniem	19P5170
Mohamed Elsayed Abdelkader	1901757
Jumana Yasser Mahrous Mostafa	20P8421
Salma Nasrledin aboelela ahmed hendawy	20p7105
Ahmed Hesham Mohamed Sakr	19p8052
Mohamed Ibrahim Elsayed Ibrahim	20P8449
Nadine Hisham Hassan	20P9880
Habiba Yasser AbdelHalim	20P3072

1.0 Table of Contents

2.0 Introduction

A rooted, directed, acyclic graph with multiple (decision) nodes and two terminal nodes can be used to describe a Boolean function. The terminal nodes are marked with the labels 0 (FALSE) and 1. (TRUE). Each decision node (u) has two child nodes, low child and high child, and is denoted by a Boolean variable (xi). The assignment of the value FALSE or TRUE, (as appropriate) to the variable (xi) is represented by the edge from node (u) to a low (or high) child. If various variables appear in the same order on all paths leading away from the root, a BDD is said to be "ordered."

If the graph of a BDD has been subjected to the following two rules, the BDD is "reduced":

- **Merge any isomorphic subgraphs.**
- **Eliminate any node whose two children are isomorphic.**

Reduced Ordered Binary Decision Diagram is nearly always referred to as BDD in common usage (ROBDD in the literature, used when the ordering and reduction aspects need to be emphasized). A ROBDD has the benefit of being canonical (unique) for a specific function and variable order. [1] Functional equivalency checking and other processes, such as functional technology mapping, benefit from this characteristic.

A variable assignment (potentially partial) represented by a path from the root node to the 1-terminal and for which the represented Boolean function is true. A node's variable is set to 0 as the route falls to a low (or high) child from that node (respectively 1).

3.0 Code Implementation

3.1 RobddNode Class

First, we declared the left and right nodes of the tree. Second, we declared the variable (x) and (y) to be used in the GUI to draw the tree later on. Then for the Robdd constructor, we initialized 4 variables: nodeNumber, variable, count, level. The variable, nodeNumber, is a unique integer that differentiates each node from any other node. The variable (level), stores the level of the node. The variable (variable), stores the variable that the node holds. The variable (count), identifies if the Node has been counted during a traversal.

```java

public class RobddNode {
    private final int nodeNumber;
    private int level;
    private int variable;

    // Links to the child RobddNodes.
    private int leftLink;
    private int rightLink;
    private RobddNode leftChild;
    private RobddNode rightChild;

    private int x;
    private int y;
    private int count;
    /*
     * @param nodeNum: The unique number of the node.
     * @param variable: The variable the node holds.
     * @param level: The level of the node, determined by the variable (user should check the varible
     * ordering and assign the value based on that.)
     */
    public RobddNode(int i, int l, int h, int nodeNumber, RobddNode left, RobddNode right) {
        this.nodeNumber = nodeNumber;
        this.variable = i;
        this.count = 0;
        this.level = -1;
        this.x = -1;
        this.y = -1;

        this.leftLink = l;
        this.rightLink = h;
        this.leftChild = left;
        this.rightChild = right;
    }
```

```java
    /** Increases number of times node has been counted.
     * @return the number of times the node has been counted.
     */
     public void incCount() {
         this.count++;
     }

    public int getCount() {
        return this.count;
    }

    public int getLevel() {
        return this.level;
    }

    public int getLeftLink() {
        return this.leftLink;
    }

    public int getNodeNum() {
        return this.nodeNumber;
    }

    public int getRightLink() {
        return this.rightLink;
    }

    public RobddNode getLeftChild() {
        return this.leftChild;
    }

    public RobddNode getRightChild() {
        return this.rightChild;
    }
```

```java
    public int getVar() {
        return this.variable;
    }

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }

    public void setLevel(int level) {
        this.level = level;
    }

    public void setLeftChild(RobddNode l) {
        this.leftChild = l;
    }

    public void setRightChild(RobddNode r) {
        this.rightChild = r;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }
}
```

3.2 Stack Class

We implemented multiple functions in the Stack class to be used later on in storing the Boolean expression

- **The function isEmpty():** Returns true if empty, false otherwise.
- **The function peek():** Returns the top of the stack, but does not remove it
- **The function pop():** Returns and removes the top of the stack.
- **The function push():** Adds the element to the top of the stack.
- **The function size():** Returns the number of elements in the stack.

```java

public interface Stack<E> {


    boolean isEmpty();

    E peek();

    E pop();

    void push(E element);

    int size();
}
```

3.3 NodeStack Class

This class has almost all of the same functions as the normal Stack class except for a minor change in the implementation of the whole class by using Nodes. For example, in pop() function, instead of just removing and returning the top of the stack, we set the top of the stack to be the next node.

```java
import java.util.NoSuchElementException;
public class NodeStack<T> implements Stack<T> {
    private Node<T> top;
    private int size;

    public NodeStack() {
        this.top = null;
        this.size = 0;
    }
    public boolean isEmpty() {
        return (this.size == 0);
    }
    public T peek() {
        if(size > 0) {
            return this.top.getValue();
        } else {
            throw new NoSuchElementException("Stack is Empty.");
        }
    }
    public T pop() {
        if(size > 0) {
            // Temp variables to hold top values and list
            Node<T> temp = top.getNext();
            T value = top.getValue();

            // Delete the top, set new top and return
            this.top.setNext(null);
            this.top = temp;
            this.size--;
            return value;
        } else {
            throw new NoSuchElementException("Stack is Empty.");
        }
    }
    public void push(T value) {
        if(this.size == 0) {
```

```java
    public void push(T value) {
        if(this.size == 0) {
            this.top = new Node<>(value, null);
        } else {
            this.top = new Node<>(value, this.top);
        }
        this.size++;
    }

    public int size() {
        return this.size;
    }
}
```

3.4 ExpressionError Class

ExpressionError class extends Exception class to be used in other classes to display error messages.

```
public class ExpressionError extends Exception {
    public ExpressionError(String message) {
        super(message);
    }
}
```

3.5 RobddNodeTable Class

RobddNodeTable holds the RobddNodes as the Robdd is constructed. Table supports adding a node, getting a node, and increasing the table size.The following functions were declared inside this class:

- **The constructor RobddNodeTable():** Constructs an RobddNodeTable and adds the zero and one terminal nodes to the table.
- **The function add():** Adds a node to the table as mapped by its variable(i), low path(l), and high path(h).
- **The function increaseTable():** Increases the table size when necessary to ensure nodes can fit.
- **The function get**(): Gets the robdd node stored at index.

```java
public class RobddNodeTable {
    RobddNode[] table;
    int nextSpot;
    int tableSize;

    public RobddNodeTable(int startSize) {
        this.table = new RobddNode[startSize];
        this.tableSize = startSize;

        // Zero Node
        this.table[0] = new RobddNode(-2, -1, -1, this.nextSpot, null, null);
        this.nextSpot++;
        // One Node
        this.table[1] = new RobddNode(-1, -1, -1, this.nextSpot, null, null);
        this.nextSpot++;
    }
    /*
     * @param i The variable of the node.
     * @param l The low path of the node.
     * @param h The high path of the node.
     * @return The index the node was stored at.
     */
    public int add(int i, int l, int h) {
        int u = this.nextSpot;
        RobddNode left = get(l);
        RobddNode right = get(h);

        RobddNode tmp = new RobddNode(i, l, h, u, left, right);

        if((this.tableSize - 2) > u) {
            this.table[u] = tmp;
            this.nextSpot++;
        } else {
            increaseTable(u);
            this.table[u] = tmp;
            this.nextSpot++;
        }
        return u;
    }
    /*
     * @param u The index currently needing to be accessed - ensures that the index is present.
     */
    private void increaseTable(int u) {
        RobddNode[] newTable = new RobddNode[2 * tableSize + u];

        for(int i = 0; i < this.table.length; i++) {
            newTable[i] = this.table[i];
        }

        this.table = newTable;
        this.tableSize = 2 * tableSize + u;
    }
    /*
     * @param u The index to be accessed.
     * @return The robdd node stored at the index.
     */
    public RobddNode get(int u) {
        return this.table[u];
    }
}
```

3.6 UniqueTable Class

UniqueTable class maps a node with a variable i, a low path l, and a high path h to an index. Table resolves collisions by linking. Supports searching for a node, retreiving a node, and inserting. Holds the uNode class which is used to store information in the table. We initialized the variable numberOfNodes, to store the number of nodes in the tree. We also initialized the variable tableSize, to hold the size of the table. Then we declared the following functions:

- **The function isMember():** Checks if a node is in the table.
- **The function findMember():** Returns the index in the RobddNodeTable of the node.
- **The function insert():** Inserts the node into the table.
- **The function hashcode():** Returns the hashCode for a node using the following equation: 37*(i + l2 + h3)

In the class uNode the following functions were declared:

- **The function equals():** Checks if two nodes are equal
- **The function getU():** Returns RobddNodeTable index.
- **The function getI():** Returns the variable.
- **The function getL():** Returns the low path.
- **The function getH():** Returns the high path.

- **The function getNext():** Returns the next link.

```java
public class UniqueTable {
    private uNode[] table;
    private int numberOfNodes;
    private int tableSize;
    /*
     * @param startSize The size the table should be initialized to. Recommend power of 2.
     */
    public UniqueTable(int startSize) {
        this.table = new uNode[startSize];
        this.tableSize = startSize;
        this.numberOfNodes =0;
    }
    /*.
     * @param i The variable of the node.
     * @param l The low path of the node.
     * @param h The high path of the node.
     * @return True if a member, false otherwise.
     */
    public boolean isMember(int i, int l, int h) {
        int hash = hashCode(i, l, h);
        int index = hash % this.tableSize;
        uNode tmp = this.table[index];

        while(tmp != null) {
            if(tmp.equals(i, l, h)) {
                return true;
            }
            tmp = tmp.getNext();
        }
        return false;
    }
    /*
     * @param i The variable of the node.
     * @param l The low path of the node.
     * @param h The high path of the node.
```

```java
    public int findMember(int i, int l, int h) {
        int hash = hashCode(i, l, h);
        int index = hash % this.tableSize;
        uNode tmp = this.table[index];

        while(tmp != null) {
            if(tmp.equals(i, l, h)) {
                return tmp.getU();
            }
            tmp = tmp.getNext();
        }
        return -1;
    }
    /*
     * @param u The index of the node in the RobddNodeTable.
     * @param i The variable of the node.
     * @param l The low path of the node.
     * @param h The high path of the node.
     */
    public void insert(int u, int i, int l, int h) {
        uNode newNode = new uNode(u, i, l, h);
        int hash = hashCode(i, l, h);
        int index = hash % this.tableSize;
        uNode tmp = this.table[index];

        if(tmp == null) {
            this.table[index] = newNode;
        } else {
            while(tmp.getNext() != null) {
                tmp = tmp.getNext();
            }
            tmp.setNext(newNode);
        }
    }
```

```java
    /*
     * @param i The variable of the node.
     * @param l The low path of the node.
     * @param h The high path of the node.
     * @return The node's hashCode.
     */
    private int hashCode(int i, int l, int h) {
        return 37 * ((int)(i + Math.pow(l, 2) + Math.pow(h, 3)));
    }

    public class uNode {
        private int u;
        private int i;
        private int l;
        private int h;
        private uNode next;

        /*
         * @param u The index of the node in the RobddNodeTable.
         * @param i The variable of the node.
         * @param l The low path of the node.
         * @param h The high path of the node.
         */
        public uNode(int u, int i, int l, int h) {
            this.u = u;
            this.i = i;
            this.l = l;
            this.h = h;
            this.next = null;
        }
```

```java
        /*
         * @param i The variable of the node.
         * @param l The low path of the node.
         * @param h The high path of the node.
         * @return True if equal, false otherwise.
         */
        public boolean equals(int i, int l, int h) {
            return (this.i == i && this.l == l && this.h == h);
        }

        public int getU() {
            return this.u;
        }

        public int getI() {
            return this.i;
        }

        public int getL() {
            return this.l;
        }

        public int getH() {
            return this.h;
        }

        public uNode getNext() {
            return this.next;
        }

        public void setNext(uNode next) {
            this.next = next;
        }
    }
}
```

3.7 Operators Class

Used to pass a set of operators.Three arrays were initialized.Operators array is used to store the operators. A precedence array is used to store the precedence of the operators in the function get precedence. Arity array stores the valid operators. The following functions were declared in the class:

- **The function getPrecedence():** Method returns the precedence of an operator. Also allows for checking for valid operators. The method will throw an illegalArgumentException if passed an operator not contained in the array of valid operators.
- **The function getArity():** Stores the valid operators in an array.
- **The function performOperation():** Performs the operation in the Boolean expression

```java
public class Operators {
    public char[] operators;
    private int[] precedence;
    private int[] arity;

    public Operators() {
        this.operators = new char[]{'+', '&', '^', '\''};
        this.precedence = new int[]{2, 2, 2, 1};
        this.arity = new int[]{2, 2, 2, 1};
    }
    /*
     * @param c The operator for which to validate and determine precedence.
     * @return The precedence of the operator.
     */
    public int getPrecedence(char c) throws IllegalArgumentException {
        int index = 0;
        for(char o: operators) {
            if(o == c) {
                return precedence[index];
            }
            index++;
        }
        // If code reaches here, the passed argument was invalid.
        throw new IllegalArgumentException("Passed character is not a valid operator.");
    }

    public int getArity(char c) throws IllegalArgumentException {
        int index = 0;
        for(char o: operators) {
            if(o == c) {
                return arity[index];
            }
            index++;
        }
```

```java
        // If code reaches here, the passed argument was invalid.
        throw new IllegalArgumentException("Passed character is not a valid operator.");
    }

    public char performOperation(char operator, char... args) throws ExpressionError{
        switch (operator) {
            case '+': return (args[0] == '1' || args[1] == '1') ? '1':'0';
            case '&': return (args[0] == '1' && args[1] == '1') ? '1':'0';
            case '^': return (args[0] != args[1]) ? '1':'0';
            case '\'': return(args[0] == '1') ? '0':'1';
            default: throw new ExpressionError("Unsupported Operator");
        }
    }
}
```

3.8 ShuntingYardAlgorithm Class

Implements Dijkstra's Shunting Yard algorithm ,which acts as our parser, converts infix expressions to postfix by doing the following :

1. **For all the input tokens:**
    1. **Read the next token**
    2. **If token is an operator (x)**
        1. **While there is an operator (y) at the top of the operators stack and either (x) is left-associative and its precedence is less or equal to that of (y), or (x) is right-associative and its precedence is less than (y)**
            1. **Pop (y) from the stack**
            2. **Add (y) output buffer**
        2. **Push (x) on the stack**
    3. **Else if token is left parenthesis, then push it on the stack**
    4. **Else if token is a right parenthesis**
        1. **Until the top token (from the stack) is left parenthesis, pop from the stack to the output buffer**
        2. **Also pop the left parenthesis but don't include it in the output buffer**
    5. **Else add token to output buffer**
2. **Pop any remaining operator tokens from the stack to the output**

The following functions were declared inside the class:

- **The function infixToPostfix():** Converts any valid infix expression to a validpostfix expression
- **The function checkArray():** Checks if a char is in a char array and is used to check for valid variables.

```java
import java.util.NoSuchElementException;

public class ShuntingYardAlgorithm {

    /** The method that carries out the algorithm.
     * @param input The infix expression to be converted.
     * @param variables The valid variables of the expression.
     * @param ops An Operators object containing the valid operators and operator precedence.
     * @return The postfix expression.
     */
    public static char[] infixToPostfix(String input, char[] variables, Operators ops) throws ExpressionError {
        // Reject empty expressions.
        int inputLength = input.length();
        if(inputLength == 0) {
            throw new ExpressionError("Error: Input an equation.");
        }
        if(variables.length == 0) {
            throw new ExpressionError("Error: Input a variable ordering.");
        }

        // Arrays which define the parenthesis used in the expression.
        final char[] parens = {'(', '[', '{', ')', ']', '}'};
        final char[] leftParens = {'(', '[', '{'};
        final char[] rightParens = {')', ']', '}'};

        NodeStack<Character> stack = new NodeStack<>();

        // Keep track of the new output length; will be different if input contains parens.
        int outputLen = 0;
        char[] output = new char[inputLength];
        int outputIndex = 0;
        int inputIndex = 0;

        // Initialize to random chars.
        char currChar = 'a';
        char tempChar  = 'a';
```

s

```java
        // While there is input to be read
        while(inputIndex < inputLength) {
            try {
                // Pull a char from the input string.
                currChar = input.charAt(inputIndex);
            } catch(IndexOutOfBoundsException e) {
                System.err.println("charAt() method error." + e.getMessage());
            }

            if(currChar == ' ') {
                // Skip whitespace
            } else {
                if(checkArray(currChar, variables)) {
                    output[outputIndex] = currChar;
                    outputIndex++;
                    outputLen++;
                } else if(checkArray(currChar, ops.operators)) {
                    try {
                        tempChar = stack.peek().charValue();

                        /* While the top of the stack is an operator, and the precedence of the
                         * input token is less than or equal to the precedence of the top operator.
                         */
                        while(ops.getPrecedence(currChar) <= ops.getPrecedence(tempChar)) {
                            // Pop the top of the stack and append to the output string.
                            output[outputIndex] = stack.pop().charValue();
                            outputIndex++;
                            outputLen++;
                            tempChar = stack.peek().charValue();
                        }
                    } catch(IllegalArgumentException e) {
                        // This is fine, tempChar was not an operator
                    } catch(NoSuchElementException e) {
                        // This is fine - stack is empty; while loop is done.
                    }
```

```java
                    stack.push(new Character(currChar));
                } else if(checkArray(currChar, leftParens)) {
                    // Push left parentheses onto stack.
                    stack.push(new Character(currChar));
                } else if(checkArray(currChar, rightParens)) {
                    // If a right parenthesis.
                    try {
                        tempChar = stack.peek().charValue();

                        // While the top of the stack is not a left parentheses.
                        while(!checkArray(tempChar, leftParens)) {
                            output[outputIndex] = stack.pop().charValue();
                            outputIndex++;
                            outputLen++;
                            tempChar = stack.peek().charValue();
                        }
                    } catch(NoSuchElementException e) {
                        // Stack empty before a left parenthesis found; there are mismatched parens.
                        throw new ExpressionError("Mismatched parentheses in expression"
                                                  + ", expected a left parenthesis.");
                    }

                    // Pop left parenthesis; do not append to output
                    stack.pop();
                } else {
                    String msg = "Error: " + currChar + " is not a variable, operator, or parenthesis.";
                    throw new ExpressionError(msg);
                }
            }
            inputIndex++;
        }

        // While there are still tokens on the stack.
        while(!stack.isEmpty()) {
            tempChar = stack.peek().charValue();
```

```java
        // While there are still tokens on the stack.
        while(!stack.isEmpty()) {
            tempChar = stack.peek().charValue();

            // If left parenthesis, there are mismatched parentheses.
            if(checkArray(tempChar, leftParens)) {
                throw new ExpressionError("Mismatched parentheses in expression" +
                                          ", there are too many left parentheses.");
            }

            // Append the top of the stack to the outpu string.
            output[outputIndex] = stack.pop().charValue();
            outputIndex++;
            outputLen++;
        }

        // Now copy output to an appropriately-sized char array; if necessary
        if(outputLen < output.length) {
            char[] tmp = new char[outputLen];
            System.arraycopy(output, 0, tmp, 0, outputLen);
            output = tmp;
        }

        return output;
    }
```

```java
    /** Checks if a char is in a char array. Method used to check for valid variables.
     * @param a The char to check for.
     * @param array The char array to search.
     * @return True if found, false otherwise.
     */
    public static boolean checkArray(char a, char[] array) {
        // Traverse array.
        for(char c: array) {
            // If a is equal, return true.
            if(a == c) {
                return true;
            }
        }
        return false;
    }
}
```

3.9 RobddBuilder Class

RobddBuilder Class builds the robbd. The following functions were declared in the class:

- **The function build():** Builds the robdd using Shannon Expansion,which will be explained in the shannonExpansion,function and the variable order array.
- **The function buildHelper():** Helper Class for build.
- **The function make():** Makes the ROBDD nodes.
- **The function postfixEvaluator():** Evaluates a postfix expression when the variables have been replaced with either 1 or 0.
- **The function shannonExpansion():** Performs Shannon Expansion on expression by replacing variables with 1 or 0 to correctly expand the expression.

```java
public class RobddBuilder {
    /*
     * @param expression The postfix boolean expression to be converted to an ROBDD.
     * @param variableOrder The char array containing the variable order.
     * @return The RobddNode that holds the root of the ROBDD.
     */
    public static RobddNode build(char[] exp, char[] variableOrder, Operators ops) throws ExpressionError {
        UniqueTable h = new UniqueTable(500);
        RobddNodeTable t =  new RobddNodeTable(200);
        int root = buildHelper(exp, 0, variableOrder, t, h, ops);
        return t.get(root);
    }
    /*
     * @param expression The postfix boolean expression to be converted to an ROBDD.
     * @param variableOrder The char array containing the variable order.
     * @param i The current variable being expanded; i refers to its index in the variable array.
     * @param t The table holding the ROBDD nodes.
     * @param h The unique table used to look up the ROBDD nodes.
     * @return The int value for table index that holds the root of the ROBDD.
     */
    private static int buildHelper(char[] exp, int i, char[]varOrder, RobddNodeTable t,
                                   UniqueTable h, Operators ops) throws ExpressionError {
        if(i < varOrder.length) {
            int v0 = buildHelper(shannonExpansion(exp, varOrder[i], '0'), (i + 1), varOrder, t, h, ops);
            int v1 = buildHelper(shannonExpansion(exp, varOrder[i], '1'), (i + 1), varOrder, t, h, ops);
            return make(i, v0, v1, t, h);
        } else {
            return postfixEvaluator(exp, ops);
        }
    }
```

```java
    /*
     * @param i The variable number.
     * @param l The index for the low path.
     * @param h The index for the high path.
     * @param t The table holding the ROBDD nodes.
     * @param h The unique table used to look up the ROBDD nodes.
     * @return The int value for table index that holds the root of the ROBDD.
     */
    private static int make(int i, int l, int h, RobddNodeTable t, UniqueTable hTable) {
        if(l == h) {
            return l;
        } else if(hTable.isMember(i, l, h)) {
            return hTable.findMember(i, l, h);
        } else {
            int u = t.add(i, l, h);
            hTable.insert(u, i, l, h);
            return u;
        }
    }

    /*
     * @param exp The boolean expression to be evaluated.
     * @return The result of the evaluation.
     */
    public static int postfixEvaluator(char[] exp, Operators ops) throws ExpressionError {
        NodeStack<Character> stack = new NodeStack<>();
        int index = 0;
        char currChar = 'a';

        while(index < exp.length) {
            currChar = exp[index];
            if(currChar == '0' || currChar =='1') {
                // It is a variable.
                stack.push(new Character(currChar));

            } else if(ShuntingYardAlgorithm.checkArray(currChar, ops.operators)) {
                // It is an operator.
                int operatorArgs = ops.getArity(currChar);

                if(operatorArgs > stack.size()) {
                    throw new ExpressionError("Insufficient number of variables in expression.");
                }

                if(operatorArgs == 2) {
                    stack.push(new Character(ops.performOperation
                                        (currChar, stack.pop().charValue(), stack.pop().charValue())));
                } else {
                    stack.push(new Character(ops.performOperation(currChar, stack.pop().charValue())));
                }
            } else {
                throw new ExpressionError("Not all variables have been initialized to values.");
            }
            index++;
        }

        if(stack.size() == 1) {
            return Character.getNumericValue(stack.pop().charValue());
        } else {
            throw new ExpressionError("There are too many variables in the Expression.");
        }
    }

    /** Performs Shannon Expansion on expression by replacing variables with replacement.
     * Replacement should be either 0 or 1 to correctly expand the expression.
     * @param exp The boolean expression to be expanded.
     * @param variable The variable to be replaced.
     * @param replacement The value to take the variable's place.
     * @return The result of the expansion.
     */
    private static char[] shannonExpansion(char[] exp, char variable, char replacement) {
        int index = 0;
        char currChar = 'a';
```

```java
        char[] newExp = new char[exp.length];

        while(index < newExp.length) {
            // Take char from old expression
            currChar = exp[index];

            // Use the replacement in position i if variable matches, else use value from exp
            if(currChar == variable) {
                newExp[index] = replacement;
            } else {
                newExp[index] = currChar;
            }
            index ++;
        }
        return newExp;
    }
}
```

3.10 Robdd Class

Robdd class holds an ROBDD. The following functions were declared in the class:

- **The function RobddFactory():** Constructs and returns an ROBDD. It creates a new Robdd, gets information about its levels, and then adds its nodes to an array in level order to allow for drawing the Robdd.
- **The function setLevelsRobdd():** Counts the number of nodes at each level; adds that info to an array.

```java
public class Robdd {
    public RobddNode root;
    public int[] levelsCount;
    public RobddNode[][] nodes;

    public Robdd() {
        this.root = null;
        this.levelsCount = null;
        this.nodes = null;
    }

     * @param postfixExpression The postfixExpression the ROBDD will represent.
     * @param variableOrder The variable Ordering being used.
     * @return The Robdd
     */
    public static Robdd RobddFactory(char[] postfixExpression, char[] variableOrder, Operators ops)
    throws ExpressionError {
        Robdd newRobdd = new Robdd();

        newRobdd.root = RobddBuilder.build(postfixExpression, variableOrder, ops);
        newRobdd.levelsCount = new int[variableOrder.length + 1];

        newRobdd.setLevelsRobdd(newRobdd.root, (newRobdd.root.getCount() + 1), 0);

        newRobdd.nodes = new RobddNode[newRobdd.levelsCount.length][];
        // Construct the levelOrder array to hold the nodes.
        for(int i = 0; i < newRobdd.levelsCount.length; i++) {
            newRobdd.nodes[i] = new RobddNode[newRobdd.levelsCount[i]];
        }

        newRobdd.getLevels(newRobdd.root, (newRobdd.root.getCount() + 1));

        return newRobdd;
    }
```

```java
     * @param t The RobddNode to be added.
     * @param levelsInfo The array to be filled with the number of nodes at each level.
     * @param count The value for which a node should be counted. Indicates if the node
     * has been visited during traversal.
     */
    private void setLevelsRobdd(RobddNode n, int count, int level) {
        if(n == null) {
            return;
        }
        if(n.getCount() == (count - 1)) {
            // Node hasn't been visited yet
            n.incCount();
            n.setLevel(level);
            this.levelsCount[level]++;
        } else {
            /* Node has been visited.
             * If the current level is higher than node's level, set node's level to current level.
             * If lower, leave unchanged.
             */
            int nodeLevel = n.getLevel();
            if(nodeLevel < level) {
                this.levelsCount[nodeLevel]--;
                n.setLevel(level);
                this.levelsCount[level]++;
            }
        }

        setLevelsRobdd(n.getLeftChild(), count, (level + 1));
        setLevelsRobdd(n.getRightChild(), count, (level + 1));
    }

    private void getLevels(RobddNode n, int count) {
        if(n == null) {
            return;
        }
```

```java
        if(n.getCount() == (count - 1)) {
            // Node hasn't been visited yet
            n.incCount();

            for(int i = 0; i < this.nodes[n.getLevel()].length; i++) {
                if(this.nodes[n.getLevel()][i] == null) {
                    this.nodes[n.getLevel()][i] = n;
                    break;
                }
            }

        }

        getLevels(n.getLeftChild(), count);
        getLevels(n.getRightChild(), count);
    }
}
```

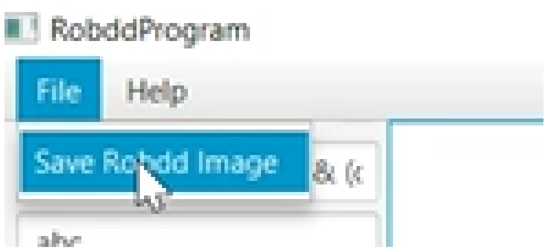
4.0 GUI

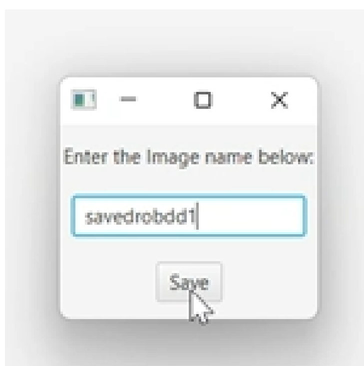
A user can enter 2 Boolean function into RobddProgram using a graphical user interface. The ROBDD (Reduced Ordered Binary Decision Diagram) for these Boolean function will subsequently be created and displayed by the application.

As a directed, acyclic digraph, the Boolean functions are represented by a ROBDD. A user-supplied Boolean expression in infix notation and a variable ordering are both supported by the application.

The program generates a ROBDD for the Boolean expressions using the variable ordering. The design shows circular nodes for the variables and terminal nodes. Between nodes, the zero paths (also known as low paths) are shown as dashed lines, while the one paths (also known as high paths) are shown as solid lines. Both invalid expressions and variable orderings will be rejected by the program.

The program supports saving an Robdd image to the current folder in the PNG format as shown in the following pictures.





We checked the equivalence of both boolean equations entered using the following function

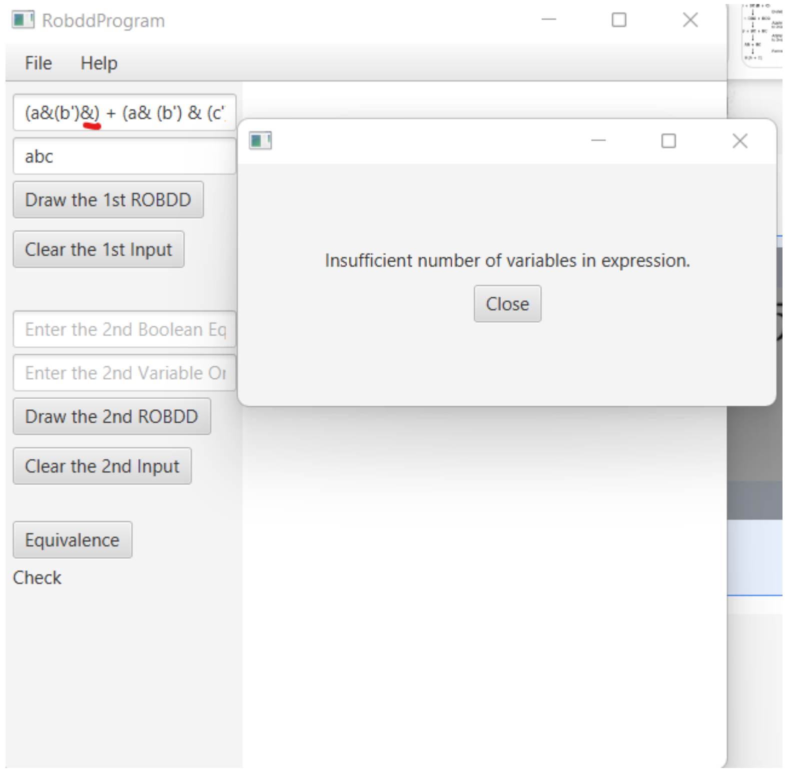
```
    ///FUNCTION NEW
public static boolean checkEquivalence (Robdd a, Robdd b){

        StringBuffer alla= new StringBuffer("f");
        StringBuffer allb= new StringBuffer("f");
        int numOfLevelsa = a.levelsCount.length;
        int numOfLevelsb = b.levelsCount.length;
        boolean isequal= true;


        for(int i = 0; i < numOfLevelsa; i++) {
                    for(int j = 0; j < a.levelsCount[i]; j++){
                        alla.append( a.nodes[i][j].getVar());
                    }
        }

        for(int i = 0; i < numOfLevelsb; i++) {
                    for(int j = 0; j < b.levelsCount[i]; j++){
                        allb.append( b.nodes[i][j].getVar());
                    }
        }

        if ( alla.length() != allb.length())
        {
            isequal= false;
            return isequal;
        }


        for(int i=0; i< allb.length(); i++){
            if (alla.charAt(i) != allb.charAt(i)){
                isequal= false;
            }

}
                return isequal;
```

How the Program Works:

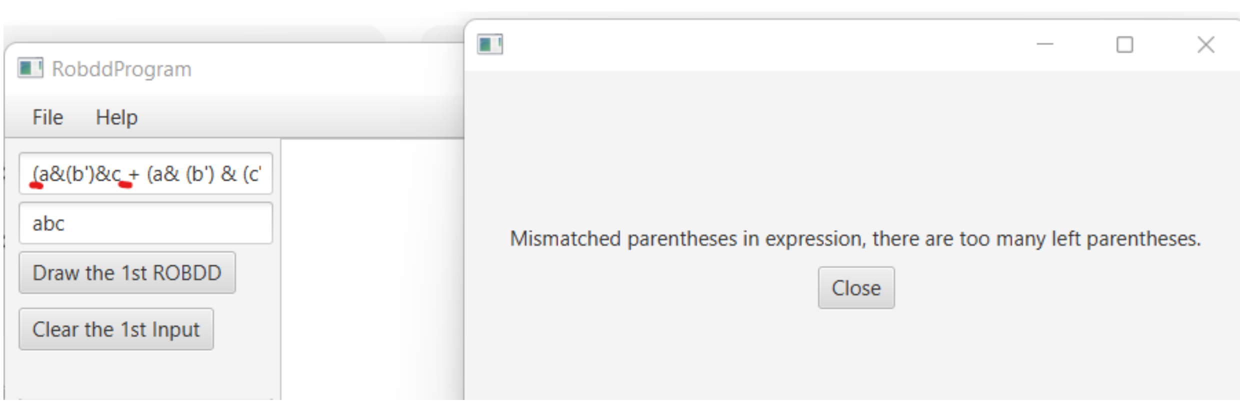
- Enter a Boolean expression in infix notation into the text box
- The expression can contain any amount of whitespace separating the operators, parentheses, and variables.
- The expression can contain any number of parentheses, but the parentheses must be balanced.
- The program will reject expressions with unbalanced parentheses.( ), { }, or [ ] are all accepted as parenthesis and the program does not distinguish among them.
- The characters '0' and '1' are reserved for program use and the program will reject expressions containing them.
- The characters '+', '&', '^', and ' ' ' are reserved for the OR, AND, XOR, and NEGATION operators. The expression can contain these as operators but not as variables.
- Expressions where all operators have the same precedence will be evaluated left to right.
- The first three operators are placed between the two variables or subexpressions they operate on. The negation operator is placed to the right of the variable or subexpression it operates on. The negation operator has first precedence. All other operators have second precedence.
- Enter a variable ordering for the ROBDD to use into the second text box.
- The variables are ordered from left to right, with the leftmost variable being the top variable in the ROBDD and the rightmost being the bottom variable in the ROBDD.
- All variables in the Boolean expression must be in the variable ordering.
- The program will reject expressions which contain variables that are not in the variable ordering.
- Once the Boolean expression and the variable input have been entered, click the 'Build ROBDD' to build it.

5.0 Parser Test Cases with GUI

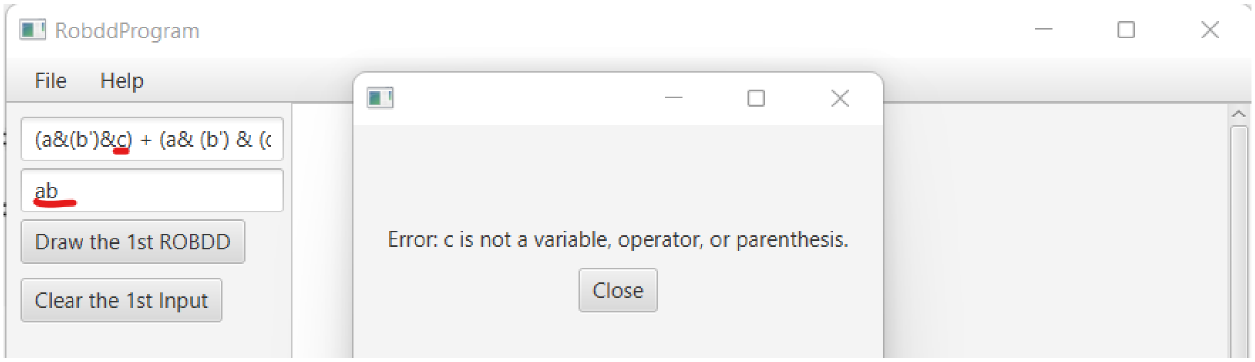
5.1 Parser test 1



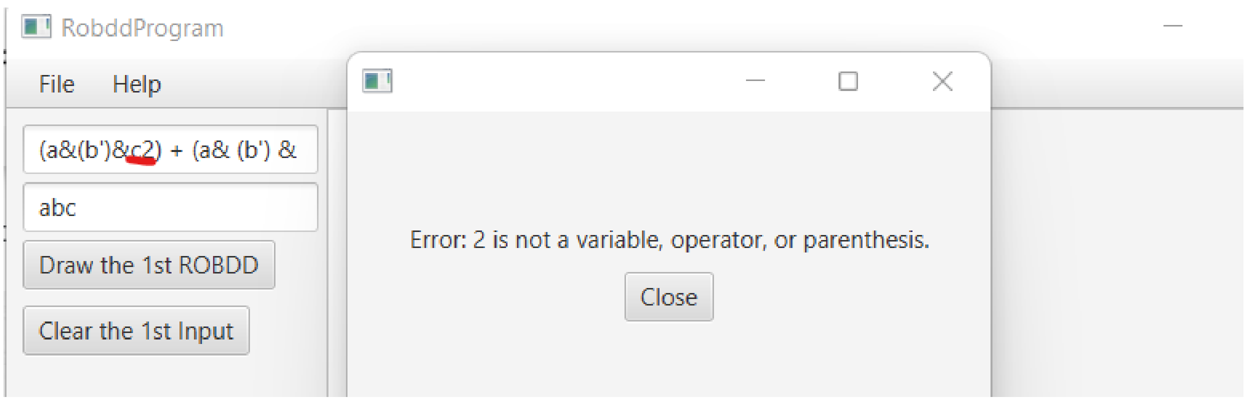
5.2 Parser test 2



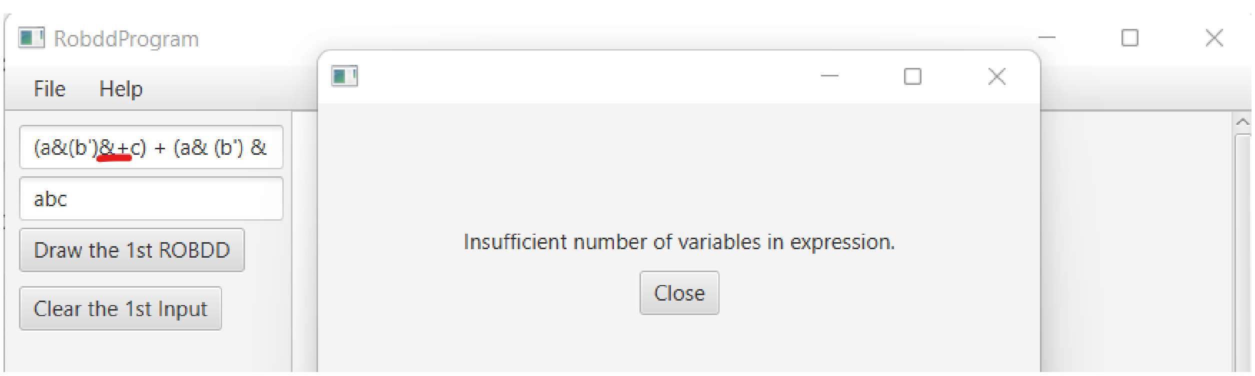
5.3 Parser test 3



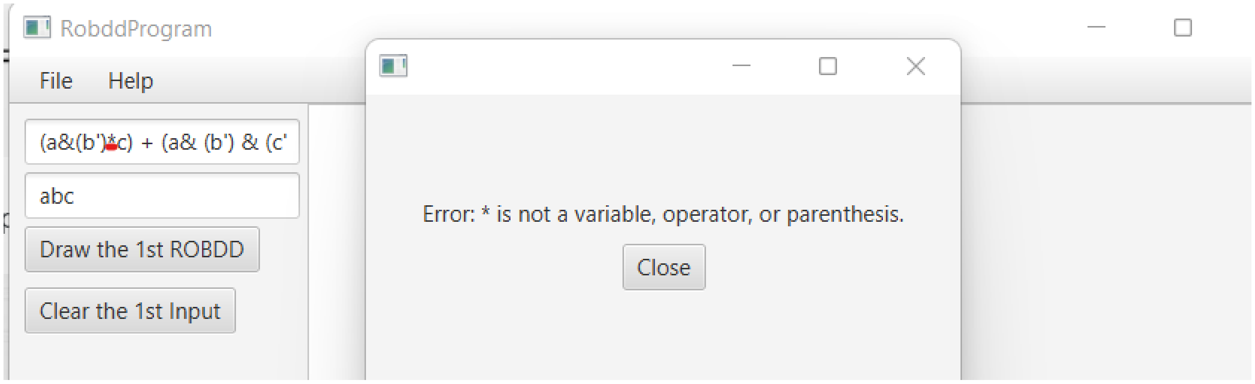
5.4 Parser test 4



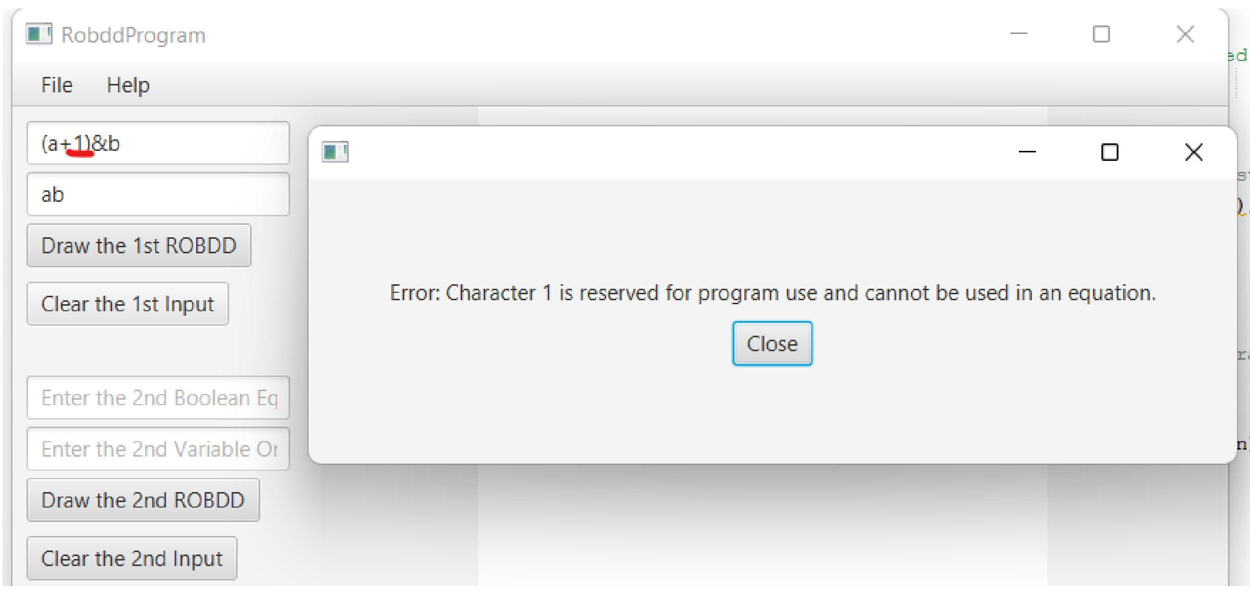
5.5 Parser test 5



5.6 Parser test 6

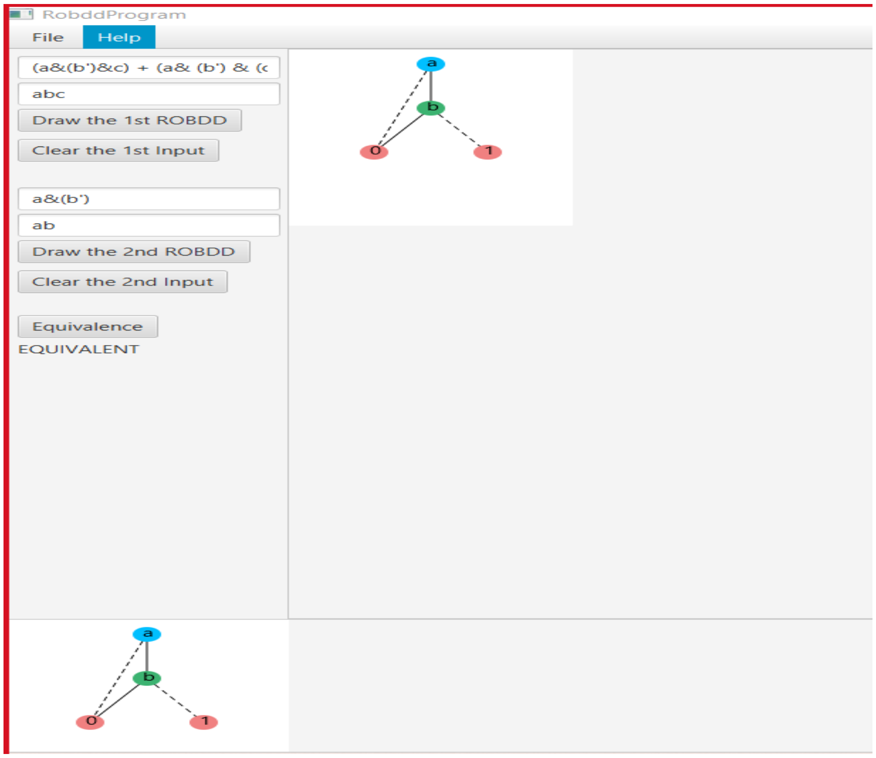


5.7 Parser test 7

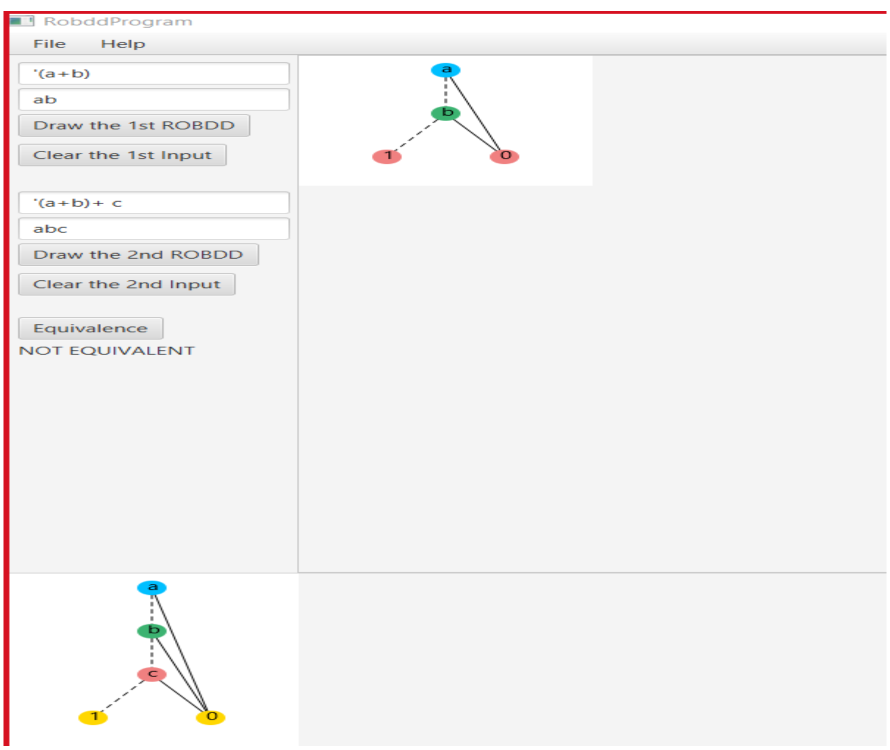


6.0 Equivalence Test Cases with GUI

6.1 Equivalence test 1

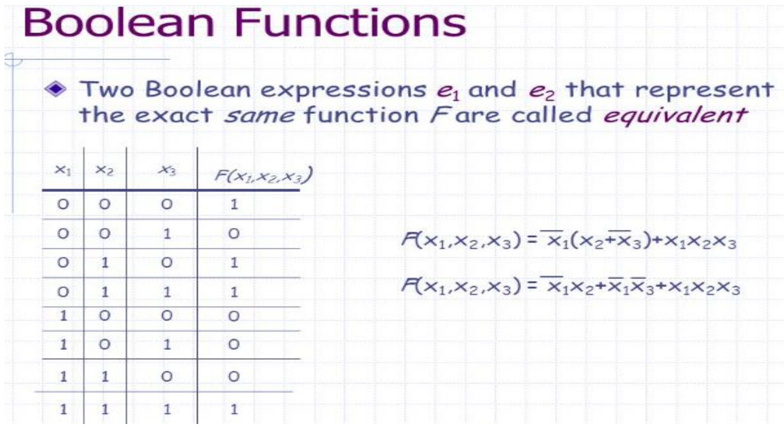


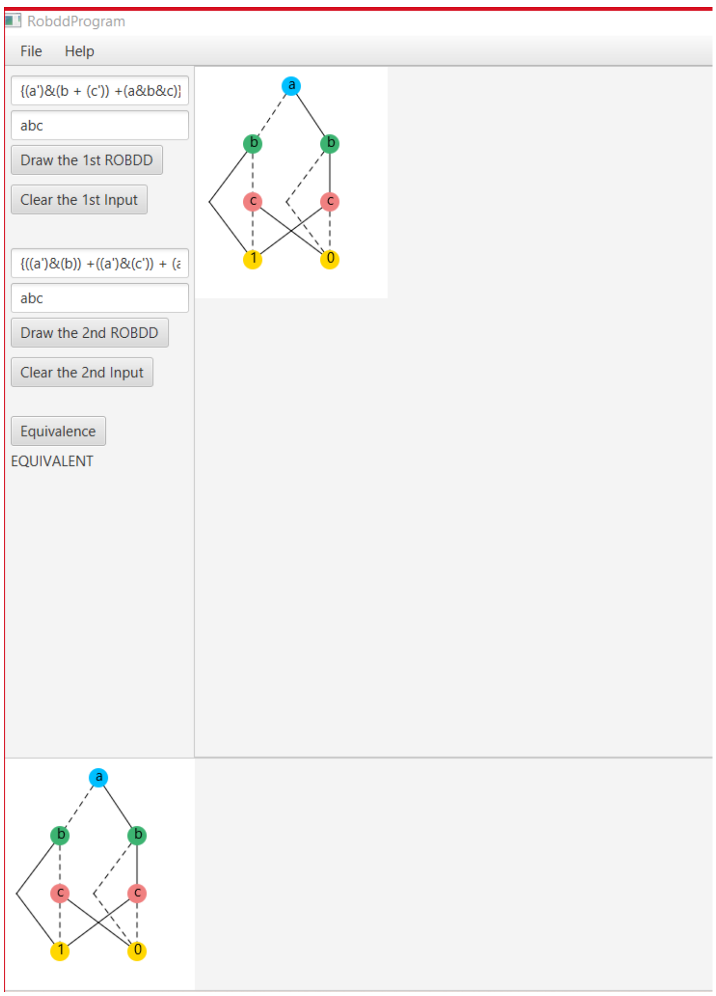
6.2 Equivalence test 2



6.3 Equivalence test 3

The boolean expressions in the picture below were used to test the equivalnce





7.0 Refrences

- "An Introduction to Binary Decision Diagrams", author Henrik Reif Andersen
- *Shunting yard animation*. (n.d.). https://somethingorotherwhatever.com/shunting-yard-animation/
- Wikipedia contributors. (2022, November 29). Binary decision diagram. Wikipedia. https://en.wikipedia.org/wiki/Binary_decision_diagram
- *Aqua Architect*. (n.d.). https://aquarchitect.github.io/swift-algorithm-club/Shunting%20Yard/#:~:text=The%20shunting%20yard%20algorithm%20was,being%20entered%20to%20postfix%20form.&text=The%20following%20table%20describes%20the%20precedence%20and%20the%20associativity%20for%20each%20operator.
- http://users.encs.concordia.ca/~tahar/coen7501/notes/2-equ-05.11-4p.pdf

8.0 Recorded Video Link and QR Code

8.1 Link

https://drive.google.com/drive/folders/1gMBSgRYipHwupl1h4nvUniztEAElaZwu

8.2 QR Code

Scan Me