Національний технічний університет України

«Київський політехнічний інститут»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Лабораторна робота №7

з курсу «Автоматизація проектування комп'ютерних систем»

Виконав

студент групи ІО-73

Захожий Ігор

Номер залікової книжки: 7308

Київ-2010

Тема роботи

Автоматизація генерації аналітичних форм булевих функцій з табличної форми.

Мета роботи

Здобуття навичок автоматизації перетворення представлення булевих функцій з табличної до аналітичної форми для заданого елементного базису.

Завдання

1. Представити номер залікової книжки в двійковому вигляді: $7308_{10} = 1110010001100_2$.

2. В залежності від молодших розрядів номера залікової книжки визначити елементний базис:

n_3	n_2	n_1	Елементний базис
1	0	0	NOT, 3AND

3. Розробити модуль генерації аналітичної форми мінімізованих булевих функцій з попередньої роботи (Лаб. робота 6).

4. Реалізувати засоби збереження результатів у файл формату VHDL.

Опис програми

У результаті виконання даної лабораторної роботи мною був реалізований модуль для генерації аналітичної форми мінімізованих булевих функцій в заданому елементному базисі. Для приведення функцій до базису (NOT, 3AND) я використав правило де Моргана і привів їх до другої нормальної форми (I-HE/I-HE), а потім погрупував елементи по 3 та каскадував їх. Ці дії реалізовані в статичному методі convertFromAndOrTo3AndNotBasis() класу FunctionsWorker. Для приведення функцій до заданого базису необхідно натиснути кнопку «Convert Functions To Basis NOT, 3AND» (Рис. 1). Результат для функцій (Рис. 1) показаний на рисунку 2.

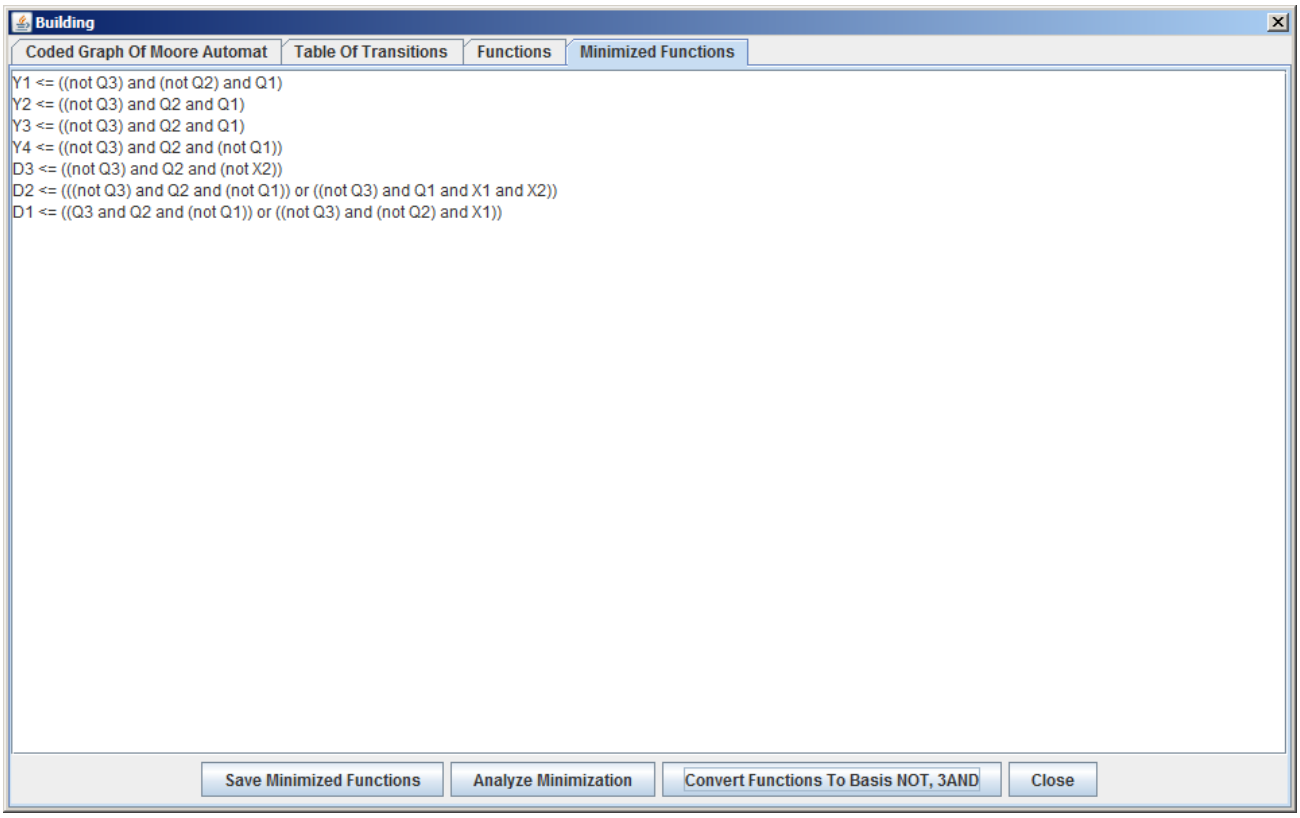

Рисунок 1

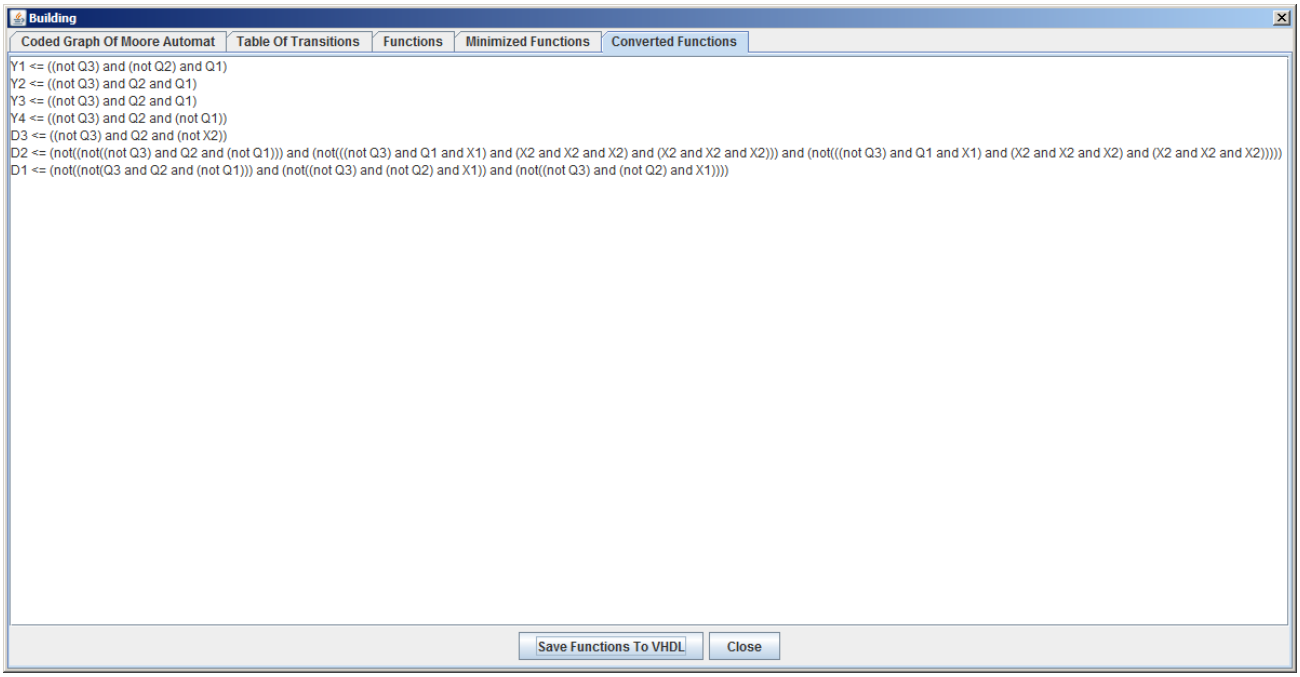

Рисунок 2

Також мною була реалізована можливість збереження аналітичного представлення функцій у заданому базисі у файлі формату VHDL. Ця дія реалізована у статичному методі getVHDLDescriptionOfFunctions() класу FunctionsWorker, що використовує методи toString() класів Function, Implicant, CompositeImplicant. Для збереження функцій у файлі формату VHDL необхідно натичнути кнопку «Save Functions To VHDL» (Рис. 2). Вміст файлу для функцій з рисунку 2 представлено на рисунку 3.

```
entity automat is
    port (Q3, Q2, Q1, X2, X1: in BIT; Y1, Y2, Y3, Y4, D3, D2, D1: out BIT);
end automat;
architecture functions of automat is
begin
    Y1 <= ((not Q3) and (not Q2) and Q1);
    Y2 <= ((not Q3) and Q2 and Q1);
    Y3 <= ((not Q3) and Q2 and Q1);
    Y4 <= ((not Q3) and Q2 and (not Q1));
    D3 <= ((not Q3) and Q2 and (not X2));
    D2 <= (not((not((not Q3) and Q2 and (not Q1))) and (not(((not Q3) and Q1 and X1) and (X2 and X2 and X2) and (X2 and X2 and X2))) and (not(((not
    D1 <= (not((not(Q3 and Q2 and (not Q1))) and (not((not Q3) and (not Q2) and X1)) and (not((not Q3) and (not Q2) and X1))));
end functions;
```

test2.vhd

Рисунок 3

Лістинг програми

```
package automat.functions;

import java.util.ArrayList;

/**
 * Created by IntelliJ IDEA.
 * User: Zak
 * Date: 16.11.2010
 * Time: 19:29:13
 * To change this template use File | Settings | File Templates.
 */
class Implicant implements Cloneable {

    protected ArrayList<String> names;
    protected ArrayList<Boolean> values;
    protected int boolFunction;
```

```
    Implicant() {}

    public Implicant(ArrayList<String> names, ArrayList<Boolean> values, int boolFunction) {
        this.names = names;
        this.values = values;
        this.boolFunction = boolFunction;
    }

    public ArrayList<String> getNames() {
        return names;
    }

    public ArrayList<Boolean> getValues() {
        return values;
    }

    public int getBoolFunction() {
        return boolFunction;
    }

    public void setNames(ArrayList<String> names) {
        this.names = names;
    }

    public void setValues(ArrayList<Boolean> values) {
        this.values = values;
    }

    public String toString() {
        StringBuilder builder = new StringBuilder();
        String boolFunctionString;
        if (boolFunction <= 2) {
            boolFunctionString = BoolFunction.getBoolFunctionString(boolFunction);
        } else {
            builder.append(BoolFunction.getBoolFunctionString(4));
            boolFunctionString = BoolFunction.getBoolFunctionString(boolFunction - 2);
        }
        builder.append("(");
        for (int i = 0; i < names.size() - 1; i++) {
            if (!values.get(i)) {
                builder.append("(");
                builder.append(BoolFunction.getBoolFunctionString(4));
                builder.append(" ");
                builder.append(names.get(i));
                builder.append(")");
            }
            else {
                builder.append(names.get(i));
            }
            builder.append(" ");
            builder.append(boolFunctionString);
            builder.append(" ");
        }
        if (!values.get(names.size() - 1)) {
            builder.append("(");
            builder.append(BoolFunction.getBoolFunctionString(4));
            builder.append(" ");
            builder.append(names.get(names.size() - 1));
            builder.append(")");
        }
        else {
            builder.append(names.get(names.size() - 1));
        }
        builder.append(")");
        return builder.toString();
    }

    public Implicant clone() {
        ArrayList<String> cloneNames = new ArrayList<String>();
        for (int i = 0; i < names.size(); i++) {
            cloneNames.add(new String(names.get(i)));
        }
        ArrayList<Boolean> cloneValues = new ArrayList<Boolean>();
        for (int i = 0; i < values.size(); i++) {
            if (values.get(i) != null) {
                cloneValues.add(new Boolean(values.get(i)));
            }
            else {
                cloneValues.add(null);
            }
        }
        return new Implicant(cloneNames, cloneValues, boolFunction);
    }

    public ArrayList<String> getAllNames() {
        ArrayList<String> allNames = new ArrayList<String>();
        for (int i = 0; i < names.size(); i++) {
            allNames.add(new String(names.get(i)));
        }
        return allNames;
    }

}

package automat.functions;

import java.util.ArrayList;

/**
 * Created by IntelliJ IDEA.
 * User: Zak
 * Date: 23.11.2010
 * Time: 23:36:31
 * To change this template use File | Settings | File Templates.
 */
class CompositeImplicant extends Implicant {

    private ArrayList<Implicant> implicants;
```

```
public CompositeImplicant(ArrayList<Implicant> implicants, int boolFunction) {
    this.implicants = implicants;
    this.boolFunction = boolFunction;
}

public CompositeImplicant(Implicant simpleImplicant, ArrayList<Implicant> implicants) {
    this.names = simpleImplicant.names;
    this.values = simpleImplicant.values;
    this.boolFunction = simpleImplicant.boolFunction;
    this.implicants = implicants;
}

public CompositeImplicant(ArrayList<String> names, ArrayList<Boolean> values, ArrayList<Implicant> implicants,
                          int boolFunction) {
    this.names = names;
    this.values = values;
    this.implicants = implicants;
    this.boolFunction = boolFunction;
}

public ArrayList<Implicant> getImplicants() {
    return implicants;
}

@Override
public String toString() {
    StringBuilder builder = new StringBuilder();
    String boolFunctionString;
    if (boolFunction <= 2) {
        boolFunctionString = BoolFunction.getBoolFunctionString(boolFunction);
    } else {
        builder.append("(");
        builder.append(BoolFunction.getBoolFunctionString(4));
        boolFunctionString = BoolFunction.getBoolFunctionString(boolFunction - 2);
    }
    if (boolFunction != 4) {
        builder.append("(");
    }
    if (names != null) {
        for (int i = 0; i < names.size() - 1; i++) {
            if (!values.get(i)) {
                builder.append(BoolFunction.getBoolFunctionString(4));
            }
            builder.append(names.get(i));
            builder.append(" ");
            builder.append(boolFunctionString);
            builder.append(" ");
        }
        if (!values.get(names.size() - 1)) {
            builder.append(BoolFunction.getBoolFunctionString(4));
        }
        builder.append(names.get(names.size() - 1));
        if (implicants == null) {
            builder.append(" ");
            builder.append(BoolFunction.getBoolFunctionString(4));
        }
    }
    if (implicants != null) {
        for (int i = 0; i < implicants.size() - 1; i++) {
            if (i != 0) {
                builder.append(" ");
            }
            builder.append(implicants.get(i).toString());
            builder.append(" ");
            builder.append(boolFunctionString);
        }
        if (!implicants.isEmpty()) {
            if (implicants.size() > 1) {
                builder.append(" ");
            }
            builder.append(implicants.get(implicants.size() - 1));
        }
    }
    builder.append(")");
    return builder.toString();
}

@Override
public Implicant clone() {
    Implicant simpleImplicant = super.clone();
    ArrayList<Implicant> cloneImplicants = new ArrayList<Implicant>();
    for (Implicant i : implicants) {
        cloneImplicants.add(i.clone());
    }
    return new CompositeImplicant(simpleImplicant, cloneImplicants);
}

public ArrayList<String> getAllNames() {
    ArrayList<String> allNames = new ArrayList<String>();
    if (names != null) {
        allNames.addAll(super.getAllNames());
    }
    for (Implicant i : implicants) {
        ArrayList<String> iNames = i.getAllNames();
        for (String s1 : iNames) {
            boolean isAlready = false;
            for (String s2 : allNames) {
                if (s1.compareTo(s2) == 0) {
                    isAlready = true;
                }
            }
            if (!isAlready) {
                allNames.add(s1);
            }
        }
    }
    return allNames;
}
```

```
}
```

```
package automat.functions;

import java.util.ArrayList;

/**
 * Created by IntelliJ IDEA.
 * User: Zak
 * Date: 16.11.2010
 * Time: 19:20:26
 * To change this template use File | Settings | File Templates.
 */
public class Function {

    private String name;
    private ArrayList<Implicant> implicants;
    private int boolFunction;

    public Function(String name, ArrayList<Implicant> implicants, int boolFunction) {
        this.name = name;
        this.implicants = implicants;
        this.boolFunction = boolFunction;
    }

    public String getName() {
        return name;
    }

    public ArrayList<Implicant> getImplicants() {
        return implicants;
    }

    public int getBoolFunction() {
        return boolFunction;
    }

    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append(name);
        builder.append(" <= ");
        String boolFunctionString;
        if (boolFunction <= 2) {
            boolFunctionString = BoolFunction.getBoolFunctionString(boolFunction);
        } else {
            builder.append(BoolFunction.getBoolFunctionString(4));
            boolFunctionString = BoolFunction.getBoolFunctionString(boolFunction - 2);
        }
        if (implicants.size() > 1) {
            builder.append("(");
        }
        for (int i = 0; i < implicants.size() - 1; i++) {
            builder.append(implicants.get(i).toString());
            builder.append(" ");
            builder.append(boolFunctionString);
            builder.append(" ");
        }
        builder.append(implicants.get(implicants.size() - 1).toString());
        if (implicants.size() > 1) {
            builder.append(")");
        }
        return builder.toString();
    }

    public ArrayList<String> getAllNames() {
        ArrayList<String> allNames = new ArrayList<String>();
        for (Implicant i : implicants) {
            ArrayList<String> iNames = i.getAllNames();
            for (String s1 : iNames) {
                boolean isAlready = false;
                for (String s2 : allNames) {
                    if (s1.compareTo(s2) == 0) {
                        isAlready = true;
                    }
                }
                if (!isAlready) {
                    allNames.add(s1);
                }
            }
        }
        return allNames;
    }

}
```

```
package automat.functions;

import automat.moore.AutomatTableModel;

import java.util.ArrayList;

/**
 * Created by IntelliJ IDEA.
 * User: Zak
 * Date: 17.11.2010
 * Time: 1:11:19
 * To change this template use File | Settings | File Templates.
 */
public class FunctionsWorker {

    public static ArrayList<Function> getFunctions(AutomatTableModel tableModel) {
        ArrayList<Function> functions = new ArrayList<Function>();
        String[][] table = tableModel.getTable();
        int index1 = tableModel.getyStartIndex();
        for (int i = 0; i < tableModel.getyCount(); i++) {
            ArrayList<Implicant> implicants = new ArrayList<Implicant>();
            for (int j = 1; j < table.length; j++) {
                if (table[j][index1].compareTo("1") == 0) {
```

```
                ArrayList<String> names = new ArrayList<String>();
                ArrayList<Boolean> values = new ArrayList<Boolean>();
                int index2 = tableModel.getqStartIndex();
                for (int k = 0; k < tableModel.getqCount(); k++) {
                    names.add(table[0][index2].substring(0, table[0][index2].length() - 3));
                    if (table[j][index2].compareTo("0") == 0) {
                        values.add(false);
                    }
                    else {
                        values.add(true);
                    }
                    index2++;
                }
                boolean isAlready = false;
                for (int k = 0; k < implicants.size(); k++) {
                    if (names.size() == implicants.get(k).getNames().size()) {
                        boolean equal = true;
                        for (int z = 0; z < implicants.get(k).getNames().size(); z++) {
                            if ((names.get(z).compareTo(implicants.get(k).getNames().get(z)) != 0) ||
                                        (((names.get(z) != implicants.get(k).getNames().get(z)) &&
                                        (values.get(z) != implicants.get(k).getValues().get(z))))) {
                                equal = false;
                            }
                        }
                        if (equal) {
                            isAlready = true;
                        }
                    }
                }
                if (!isAlready) {
                    implicants.add(new Implicant(names, values, 0));
                }
            }
        }
        functions.add(new Function(table[0][index1], implicants, 1));
        index1++;
    }
    index1 = tableModel.getdStartIndex();
    for (int i = 0; i < tableModel.getdCount(); i++) {
        ArrayList<Implicant> implicants = new ArrayList<Implicant>();
        for (int j = 1; j < table.length; j++) {
            if (table[j][index1].compareTo("1") == 0) {
                ArrayList<String> names = new ArrayList<String>();
                ArrayList<Boolean> values = new ArrayList<Boolean>();
                int index2 = tableModel.getqStartIndex();
                for (int k = 0; k < tableModel.getqCount(); k++) {
                    names.add(table[0][index2].substring(0, table[0][index2].length() - 3));
                    if (table[j][index2].compareTo("0") == 0) {
                        values.add(false);
                    }
                    else {
                        values.add(true);
                    }
                    index2++;
                }
                index2 = tableModel.getxStartIndex();
                for (int k = 0; k < tableModel.getxCount(); k++) {
                    if (table[j][index2].compareTo("-") != 0) {
                        names.add(table[0][index2]);
                        if (table[j][index2].compareTo("0") == 0) {
                            values.add(false);
                        } else {
                            values.add(true);
                        }
                    }
                    index2++;
                }
                implicants.add(new Implicant(names, values, 0));
            }
        }
        functions.add(new Function(table[0][index1], implicants, 1));
        index1++;
    }
    return functions;
}

public static ArrayList<Function> prepareFunctionsToMinimization(ArrayList<Function> functions) {
    ArrayList<Function> newFunctions = new ArrayList<Function>();
    for (Function f : functions) {
        ArrayList<Implicant> implicants = f.getImplicants();
        ArrayList<String> allNames = new ArrayList<String>();
        for (int i = 0; i < implicants.size(); i++) {
            ArrayList<String> names = implicants.get(i).getNames();
            for (int j = 0; j < names.size(); j++) {
                boolean contains = false;
                for (int k = 0; k < allNames.size(); k++) {
                    if (allNames.get(k).compareTo(names.get(j)) == 0) {
                        contains = true;
                    }
                }
                if (!contains) {
                    allNames.add(new String(names.get(j)));
                }
            }
        }
        ArrayList<Implicant> newImplicants = new ArrayList<Implicant>();
        for (int i = 0; i < implicants.size(); i++) {
            ArrayList<String> names = implicants.get(i).getNames();
            ArrayList<Boolean> values = implicants.get(i).getValues();
            ArrayList<String> newNames = new ArrayList<String>();
            ArrayList<Boolean> newValues = new ArrayList<Boolean>();
            for (int j = 0; j < allNames.size(); j++) {
                newNames.add(new String(allNames.get(j)));
                newValues.add(null);
            }
            for (int j = 0; j < names.size(); j++) {
                for (int k = 0; k < newNames.size(); k++) {
                    if (names.get(j).compareTo(newNames.get(k)) == 0) {
                        newValues.set(k, new Boolean(values.get(j)));
```

```
                    }
                }
            }
            newImplicants.add(new Implicant(newNames, newValues, implicants.get(i).getBoolFunction()));
        }
        newFunctions.add(new Function(new String(f.getName()), newImplicants, f.getBoolFunction()));
    }
    return newFunctions;
}

public static ArrayList<Function> minimizeFunctions(ArrayList<Function> functions) {
    ArrayList<Function> minimizedFunctions = new ArrayList<Function>();
    for (Function f : functions) {
        ArrayList<ArrayList<Implicant>> implicants = new ArrayList<ArrayList<Implicant>>();
        ArrayList<ArrayList<Boolean>> isCovered = new ArrayList<ArrayList<Boolean>>();
        ArrayList<Implicant> originalImplicats = f.getImplicants();
        ArrayList<Implicant> startImplicants = new ArrayList<Implicant>();
        ArrayList<Boolean> startIsCovered = new ArrayList<Boolean>();
        for (int i = 0; i < originalImplicats.size(); i++) {
            startImplicants.add(originalImplicats.get(i).clone());
            startIsCovered.add(false);
        }
        implicants.add(startImplicants);
        isCovered.add(startIsCovered);
        boolean flag = false;
        while (!flag) {
            ArrayList<Implicant> coveringImplicants = implicants.get(implicants.size() - 1);
            ArrayList<Boolean> coveringIsCover = isCovered.get(isCovered.size() - 1);
            ArrayList<Implicant> coverImplicants = new ArrayList<Implicant>();
            for (int i = 0; i < coveringImplicants.size() - 1; i++) {
                if (!coveringIsCover.get(i)) {
                    for (int j = i + 1; j < coveringImplicants.size(); j++) {
                        int difference = 0;
                        int differenceIndex = -1;
                        ArrayList<String> names1 = coveringImplicants.get(i).getNames();
                        ArrayList<Boolean> values1 = coveringImplicants.get(i).getValues();
                        ArrayList<String> names2 = coveringImplicants.get(j).getNames();
                        ArrayList<Boolean> values2 = coveringImplicants.get(j).getValues();
                        for (int k = 0; k < names1.size(); k++) {
                            if (names1.get(k).compareTo(names2.get(k)) != 0) {
                                difference++;
                                differenceIndex = k;
                            }
                            else {
                                if ((values1.get(k) != null) && (values2.get(k) != null) &&
                                            (values1.get(k).compareTo(values2.get(k)) != 0)) {
                                    difference++;
                                    differenceIndex = k;
                                }
                            }
                        }
                        if (difference < 2) {
                            ArrayList<String> newNames = new ArrayList<String>();
                            ArrayList<Boolean> newValues = new ArrayList<Boolean>();
                            for (int k = 0; k < names1.size(); k++) {
                                if (k != differenceIndex) {
                                    newNames.add(new String(names1.get(k)));
                                    if (values1.get(k) != null) {
                                        newValues.add(new Boolean(values1.get(k)));
                                    }
                                    else {
                                        if (values2.get(k) != null) {
                                            newValues.add(new Boolean(values2.get(k)));
                                        }
                                        else {
                                            newValues.add(null);
                                        }
                                    }
                                }
                            }
                            boolean isAlready = false;
                            for (int k = 0; k < coverImplicants.size(); k++) {
                                if (newNames.size() == coverImplicants.get(k).getNames().size()) {
                                    boolean equal = true;
                                    for (int z = 0; z < coverImplicants.get(k).getNames().size(); z++) {
                                        if ((newNames.get(z).compareTo(coverImplicants.get(k).getNames().get(z)) != 0) ||
                                                (((newNames.get(z) != coverImplicants.get(k).getNames().get(z)) &&
                                                        (newValues.get(z) != coverImplicants.get(k).getValues().get(z))))) {
                                            equal = false;
                                        }
                                    }
                                    if (equal) {
                                        isAlready = true;
                                    }
                                }
                                else {
                                    isAlready = true;
                                }
                            }
                            if (!isAlready) {
                                coverImplicants.add(new Implicant(newNames, newValues,
                                            coveringImplicants.get(i).getBoolFunction()));
                            }
                            coveringIsCover.set(i, true);
                            coveringIsCover.set(j, true);
                        }
                    }
                }
            }
            implicants.add(coverImplicants);
            isCovered.add(new ArrayList<Boolean>());
            for (int i = 0; i < coverImplicants.size(); i++) {
                isCovered.get(isCovered.size() - 1).add(false);
            }
            flag = true;
            for (int i = 0; i < isCovered.get(isCovered.size() - 2).size(); i++) {
                if (isCovered.get(isCovered.size() - 2).get(i)) {
                    flag = false;
                }
```

```
            }
        }
        ArrayList<Implicant> minimizedImplicants = new ArrayList<Implicant>();
        for (int i = 0; i < implicants.size(); i++) {
            for (int j = 0; j < implicants.get(i).size(); j++) {
                if (!isCovered.get(i).get(j)) {
                    ArrayList<String> names = implicants.get(i).get(j).getNames();
                    ArrayList<Boolean> values = implicants.get(i).get(j).getValues();
                    boolean hasNull = true;
                    while (hasNull) {
                        hasNull = false;
                        for (int k = 0; k < values.size(); k++) {
                            if (values.get(k) == null) {
                                hasNull = true;
                                names.remove(k);
                                values.remove(k);
                            }
                        }
                    }
                    implicants.get(i).get(j).setNames(names);
                    implicants.get(i).get(j).setValues(values);
                    minimizedImplicants.add(implicants.get(i).get(j));
                }
            }
        }
        minimizedFunctions.add(new Function(new String(f.getName()), minimizedImplicants, f.getBoolFunction()));
    }
    return minimizedFunctions;
}

public static MinimizationEfficiencyObject analyzeMinimization(ArrayList<Function> functions,
                                                                ArrayList<Function> minimizedFunctions) {
    int elementCount1 = 0;
    int elementCount2 = 0;
    int entryCount1 = 0;
    int entryCount2 = 0;
    int exitCount1 = 0;
    int exitCount2 = 0;
    for (Function f : functions) {
        elementCount1++;
        exitCount1++;
        for (Implicant i : f.getImplicants()) {
            elementCount1++;
            entryCount1 += i.getNames().size();
            exitCount1++;
        }
    }
    for (Function f : minimizedFunctions) {
        elementCount2++;
        exitCount2++;
        for (Implicant i : f.getImplicants()) {
            elementCount2++;
            entryCount2 += i.getNames().size();
            exitCount2++;
        }
    }
    return new MinimizationEfficiencyObject(elementCount1, elementCount2, entryCount1, entryCount2, exitCount1,
                                            exitCount2);
}

public static ArrayList<Function> convertFromAndOrTo3AndNotBasis(ArrayList<Function> originalFunctions) {
    ArrayList<Function> functions = new ArrayList<Function>();
    for (Function f : originalFunctions) {
        ArrayList<Implicant> implicants = new ArrayList<Implicant>();
        ArrayList<Implicant> originalImplicants = f.getImplicants();
        for (Implicant i : originalImplicants) {
            if (i.getNames().size() > 3) {
                double temp = Math.log(i.getNames().size()) / Math.log(3);
                if (temp > (int) temp) {
                    temp = (int) temp + 1;
                }
                ArrayList<Implicant> tempImplicants = new ArrayList<Implicant>();
                int counter = 1;
                ArrayList<String> names = i.getNames();
                ArrayList<Boolean> values = i.getValues();
                ArrayList<String> newNames = null;
                ArrayList<Boolean> newValues = null;
                for (int j = 0; j < names.size(); j++) {
                    if (counter == 1) {
                        newNames = new ArrayList<String>();
                        newValues = new ArrayList<Boolean>();
                    }
                    newNames.add(new String(names.get(j)));
                    newValues.add(new Boolean(values.get(j)));
                    if (counter % 3 == 0) {
                        tempImplicants.add(new Implicant(newNames, newValues, 0));
                        newNames = null;
                        newValues = null;
                        counter = 1;
                    } else {
                        counter++;
                    }
                }
                if (counter != 1) {
                    while (counter < 4) {
                        newNames.add(new String(newNames.get(newNames.size() - 1)));
                        newValues.add(new Boolean(newValues.get(newValues.size() - 1)));
                        counter++;
                    }
                    tempImplicants.add(new Implicant(newNames, newValues, 0));
                }
                if (temp > 2) {
                    counter = 0;
                    while (counter < temp - 2) {
                        ArrayList<Implicant> newImplicants = new ArrayList<Implicant>();
                        ArrayList<Implicant> simpleImplicants = null;
                        int subCounter = 1;
                        for (int j = 0; j < implicants.size(); j++) {
                            if (subCounter == 1) {
```

```
                                simpleImplicants = new ArrayList<Implicant>();
                            }
                            simpleImplicants.add(implicants.get(j));
                            if (subCounter % 3 == 0) {
                                newImplicants.add(new CompositeImplicant(simpleImplicants, 0));
                                simpleImplicants = null;
                                subCounter = 1;
                            } else {
                                subCounter++;
                            }
                        }
                        if (subCounter != 1) {
                            while (subCounter < 4) {
                                simpleImplicants.add(simpleImplicants.get(simpleImplicants.size() - 1).clone());
                                subCounter++;
                            }
                            newImplicants.add(new CompositeImplicant(simpleImplicants, 0));
                        }
                        implicants = newImplicants;
                        counter++;
                    }
                } else {
                    while (tempImplicants.size() % 3 != 0) {
                        tempImplicants.add(tempImplicants.get(tempImplicants.size() - 1));
                    }
                    implicants.add(new CompositeImplicant(tempImplicants, 0));
                }
            } else {
                Implicant cloneImplicant = i.clone();
                ArrayList<String> names = cloneImplicant.getNames();
                ArrayList<Boolean> values = cloneImplicant.getValues();
                if (names.size() > 1) {
                    while (names.size() % 3 != 0) {
                        names.add(names.get(names.size() - 1));
                        values.add(values.get(values.size() - 1));
                    }
                }
                implicants.add(new Implicant(names, values, 0));
            }
        }
        ArrayList<Implicant> newImplicants;
        if (implicants.size() > 1) {
            newImplicants = new ArrayList<Implicant>();
            for (int j = 0; j < implicants.size(); j++) {
                ArrayList<Implicant> tempList = new ArrayList<Implicant>();
                tempList.add(implicants.get(j));
                newImplicants.add(new CompositeImplicant(tempList, 4));
            }
            implicants = newImplicants;
        }
        if (implicants.size() > 3) {
            double temp = Math.log(implicants.size()) / Math.log(3);
            if (temp > (int) temp) {
                temp = (int) temp + 1;
            }
            while (temp > 1) {
                newImplicants = new ArrayList<Implicant>();
                ArrayList<Implicant> subImplicants = null;
                int counter = 1;
                for (int j = 0; j < implicants.size(); j++) {
                    if (counter == 1) {
                        subImplicants = new ArrayList<Implicant>();
                    }
                    subImplicants.add(implicants.get(j));
                    if (counter % 3 == 0) {
                        newImplicants.add(new CompositeImplicant(subImplicants, 0));
                        newImplicants = null;
                        counter = 1;
                    } else {
                        counter++;
                    }
                }
                if (counter != 1) {
                    while (counter < 4) {
                        subImplicants.add(subImplicants.get(subImplicants.size() - 1).clone());
                        counter++;
                    }
                    newImplicants.add(new CompositeImplicant(subImplicants, 0));
                }
                implicants = newImplicants;
                temp = temp - 1;
            }
        }
        else {
            if (implicants.size() > 1) {
                while (implicants.size() % 3 != 0) {
                    implicants.add(implicants.get(implicants.size() - 1));
                }
            }
        }
        ArrayList<Implicant> tempList = new ArrayList<Implicant>();
        if (implicants.size() > 1) {
            ArrayList<Implicant> tempSubList = new ArrayList<Implicant>();
            tempSubList.add(new CompositeImplicant(implicants, 0));
            tempList.add(new CompositeImplicant(tempSubList, 4));
        } else {
            tempList.add(implicants.get(0));
        }
        functions.add(new Function(f.getName(), tempList, 0));
    }
    return functions;
}

public static String getVHDLDescriptionOfFunctions(ArrayList<Function> functions) {
    ArrayList<String> inNames = new ArrayList<String>();
    ArrayList<String> outNames = new ArrayList<String>();
    for (Function f : functions) {
        outNames.add(f.getName());
        ArrayList<String> fNames = f.getAllNames();
```

```
            for (String s1 : fNames) {
                boolean isAlready = false;
                for (String s2 : inNames) {
                    if (s1.compareTo(s2) == 0) {
                        isAlready = true;
                    }
                }
                if (!isAlready) {
                    inNames.add(s1);
                }
            }
        }
        StringBuilder builder = new StringBuilder();
        builder.append("entity automat is\n");
        builder.append("\tport (");
        for (int i = 0; i < inNames.size() - 1; i++) {
            builder.append(inNames.get(i));
            builder.append(", ");
        }
        builder.append(inNames.get(inNames.size() - 1));
        builder.append(": in BIT; ");
        for (int i = 0; i < outNames.size() - 1; i++) {
            builder.append(outNames.get(i));
            builder.append(", ");
        }
        builder.append(outNames.get(outNames.size() - 1));
        builder.append(": out BIT);\nend automat;\narchitecture functions of automat is\nbegin\n\t");
        for (int i = 0; i < functions.size() - 1; i++) {
            builder.append(functions.get(i).toString());
            builder.append(";\n\t");
        }
        builder.append(functions.get(functions.size() - 1).toString());
        builder.append(";\n");
        builder.append("end functions;");
        return builder.toString();
    }

}
```

```
package face;

import automat.functions.Function;
import automat.functions.FunctionsWorker;
import automat.functions.VHDLFileFilter;
import automat.moore.*;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;

/**
 * Created by IntelliJ IDEA.
 * User: Zak
 * Date: 20.10.2010
 * Time: 1:17:35
 * To change this template use File | Settings | File Templates.
 */
class BuildFrame extends JDialog {

    private MainFrame mainFrame;

    private JTabbedPane tabbedPane;
    private GraphPanel graphPanel;
    private CodedGraphPanel codedGraphPanel;
    private JButton codeGraphButton;
    private AutomatTableModel tableModel;
    private JButton buildTableButton;
    private String functionsString;
    private ArrayList<Function> functions;
    private JButton buildFunctionsButton;
    private ArrayList<Function> minimizedFunctions;
    private String minimizedFunctionsString;
    private JButton minimizeFunctionsButton;
    private JButton convertToBasisButton;
    private ArrayList<Function> convertedFunctions;
    private String convertedFunctionsString;

    public BuildFrame(MainFrame frame, Rectangle bounds, MooreAutomat automat) {
        super(frame);
        mainFrame = frame;
        setBounds(bounds);
        setMinimumSize(bounds.getSize());
        setResizable(true);
        setModal(true);
        setTitle("Building");
        tabbedPane = new JTabbedPane();
        add(tabbedPane);
        JPanel mooreGraphPanel = new JPanel();
        mooreGraphPanel.setLayout(new BorderLayout());
        graphPanel = new GraphPanel(new GraphModel(automat));
        JPanel mooreGraphButtonsPanel = new JPanel();
        JButton saveGraphButton = new JButton(new SaveGraphAction(this));
        saveGraphButton.setText("Save Graph");
        codeGraphButton = new JButton(new CodeGraphAction(this));
        codeGraphButton.setText("Code Graph");
        JButton closeButton = new JButton(new AbstractAction() {
            public void actionPerformed(ActionEvent e) {
                setVisible(false);
            }
        });
        closeButton.setText("Close");
        mooreGraphButtonsPanel.add(saveGraphButton);
        mooreGraphButtonsPanel.add(codeGraphButton);
        mooreGraphButtonsPanel.add(closeButton);
```

```
        mooreGraphPanel.add(mooreGraphButtonsPanel, BorderLayout.SOUTH);
        mooreGraphPanel.add(graphPanel);
        tabbedPane.addTab("Graph Of Moore Automat", mooreGraphPanel);
    }

    public BuildFrame(MainFrame frame, Rectangle bounds, CodedMooreAutomat automat) {
        super(frame);
        mainFrame = frame;
        setBounds(bounds);
        setMinimumSize(bounds.getSize());
        setResizable(true);
        setModal(true);
        setTitle("Building");
        tabbedPane = new JTabbedPane();
        add(tabbedPane);
        JPanel mooreCodedGraphPanel = new JPanel();
        mooreCodedGraphPanel.setLayout(new BorderLayout());
        codedGraphPanel = new CodedGraphPanel(new GraphModel(automat));
        JPanel mooreGraphButtonsPanel = new JPanel();
        JButton saveCodedGraphButton = new JButton(new SaveCodedGraphAction(this));
        saveCodedGraphButton.setText("Save Graph");
        buildTableButton = new JButton(new BuildTableAction(this));
        buildTableButton.setText("Build Table Of Transitions");
        JButton closeButton = new JButton(new AbstractAction() {
            public void actionPerformed(ActionEvent e) {
                setVisible(false);
            }
        });
        closeButton.setText("Close");
        mooreGraphButtonsPanel.add(saveCodedGraphButton);
        mooreGraphButtonsPanel.add(buildTableButton);
        mooreGraphButtonsPanel.add(closeButton);
        mooreCodedGraphPanel.add(mooreGraphButtonsPanel, BorderLayout.SOUTH);
        mooreCodedGraphPanel.add(codedGraphPanel);
        tabbedPane.addTab("Coded Graph Of Moore Automat", mooreCodedGraphPanel);
    }

    private class SaveGraphAction extends AbstractAction {

        private BuildFrame frame;

        public SaveGraphAction(BuildFrame frame) {
            this.frame = frame;
        }

        public void actionPerformed(ActionEvent e) {
            JFileChooser chooser = mainFrame.getChooser();
            chooser.resetChoosableFileFilters();
            chooser.addChoosableFileFilter(new GraphFileFilter());
            int result = chooser.showSaveDialog(frame);
            if (result == JFileChooser.APPROVE_OPTION) {
                if (!chooser.getSelectedFile().getName().endsWith(GraphFileFilter.GRAPH_EXTENSION)) {
                    chooser.setSelectedFile(new File(chooser.getSelectedFile().getAbsolutePath() + GraphFileFilter.GRAPH_EXTENSION));
                }
                try {
                    MooreAutomat.writeToFile(chooser.getSelectedFile(), graphPanel.getModel().getAutomat());
                } catch (IOException e1) {
                    JOptionPane.showMessageDialog(frame, "Error! Can't create file.",
                            "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
        }

    }

    private class SaveCodedGraphAction extends AbstractAction {

        private BuildFrame frame;

        public SaveCodedGraphAction(BuildFrame frame) {
            this.frame = frame;
        }

        public void actionPerformed(ActionEvent e) {
            JFileChooser chooser = mainFrame.getChooser();
            chooser.resetChoosableFileFilters();
            chooser.addChoosableFileFilter(new CodedGraphFileFilter());
            int result = chooser.showSaveDialog(frame);
            if (result == JFileChooser.APPROVE_OPTION) {
                if (!chooser.getSelectedFile().getName().endsWith(CodedGraphFileFilter.CODED_GRAPH_EXTENSION)) {
                    chooser.setSelectedFile(new File(chooser.getSelectedFile().getAbsolutePath() + CodedGraphFileFilter.CODED_GRAPH_EXTENSION));
                }
                try {
                    CodedMooreAutomat.writeToFile(chooser.getSelectedFile(), (CodedMooreAutomat) codedGraphPanel.getModel().getAutomat());
                } catch (IOException e1) {
                    JOptionPane.showMessageDialog(frame, "Error! Can't create file.",
                            "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
        }

    }

    private class CodeGraphAction extends AbstractAction {

        private BuildFrame frame;

        public CodeGraphAction(BuildFrame frame) {
            this.frame = frame;
        }

        public void actionPerformed(ActionEvent e) {
            JPanel mooreCodedGraphPanel = new JPanel();
            mooreCodedGraphPanel.setLayout(new BorderLayout());
            codedGraphPanel = new CodedGraphPanel(new GraphModel(graphPanel.getModel().getAutomat()));
            JPanel mooreGraphButtonsPanel = new JPanel();
            JButton saveCodedGraphButton = new JButton(new SaveCodedGraphAction(frame));
            saveCodedGraphButton.setText("Save Graph");
            buildTableButton = new JButton(new BuildTableAction(frame));
```

```
        buildTableButton.setText("Build Table Of Transitions");
        JButton closeButton = new JButton(new AbstractAction() {
            public void actionPerformed(ActionEvent e) {
                setVisible(false);
            }
        });
        closeButton.setText("Close");
        mooreGraphButtonsPanel.add(saveCodedGraphButton);
        mooreGraphButtonsPanel.add(buildTableButton);
        mooreGraphButtonsPanel.add(closeButton);
        mooreCodedGraphPanel.add(mooreGraphButtonsPanel, BorderLayout.SOUTH);
        mooreCodedGraphPanel.add(codedGraphPanel);
        tabbedPane.addTab("Coded Graph Of Moore Automat", mooreCodedGraphPanel);
        tabbedPane.setSelectedIndex(1);
        codeGraphButton.setEnabled(false);
    }

}

private class SaveTableAction extends AbstractAction {

    private BuildFrame frame;

    public SaveTableAction(BuildFrame frame) {
        this.frame = frame;
    }

    public void actionPerformed(ActionEvent e) {
        JFileChooser chooser = mainFrame.getChooser();
        chooser.resetChoosableFileFilters();
        chooser.addChoosableFileFilter(new TextFileFilter());
        int result = chooser.showSaveDialog(frame);
        if (result == JFileChooser.APPROVE_OPTION) {
            if (!chooser.getSelectedFile().getName().endsWith(TextFileFilter.TEXT_FILE_EXTENSION)) {
                chooser.setSelectedFile(new File(chooser.getSelectedFile().getAbsolutePath() + TextFileFilter.TEXT_FILE_EXTENSION));
            }
            try {
                tableModel.writeToFile(chooser.getSelectedFile());
            } catch (IOException e1) {
                JOptionPane.showMessageDialog(frame, "Error! Can't create file.",
                        "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

}

private class BuildTableAction extends AbstractAction {

    private BuildFrame frame;

    public BuildTableAction(BuildFrame frame) {
        this.frame = frame;
    }

    public void actionPerformed(ActionEvent e) {
        JPanel tablePanel = new JPanel();
        tablePanel.setLayout(new BorderLayout());
        tableModel = new AutomatTableModel((CodedMooreAutomat) codedGraphPanel.getModel().getAutomat());
        JTable table = new JTable(tableModel);
        JPanel tableButtonsPanel = new JPanel();
        JButton saveTableButton = new JButton(new SaveTableAction(frame));
        saveTableButton.setText("Save Table");
        buildFunctionsButton = new JButton(new BuildFunctionsAction(frame));
        buildFunctionsButton.setText("Build Functions");
        JButton closeButton = new JButton(new AbstractAction() {
            public void actionPerformed(ActionEvent e) {
                setVisible(false);
            }
        });
        closeButton.setText("Close");
        tableButtonsPanel.add(saveTableButton);
        tableButtonsPanel.add(buildFunctionsButton);
        tableButtonsPanel.add(closeButton);
        tablePanel.add(tableButtonsPanel, BorderLayout.SOUTH);
        tablePanel.add(table);
        tabbedPane.addTab("Table Of Transitions", tablePanel);
        tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);
        buildTableButton.setEnabled(false);
    }

}

private class SaveFunctionsAction extends AbstractAction {

    private BuildFrame frame;

    public SaveFunctionsAction(BuildFrame frame) {
        this.frame = frame;
    }

    public void actionPerformed(ActionEvent e) {
        JFileChooser chooser = mainFrame.getChooser();
        chooser.resetChoosableFileFilters();
        chooser.addChoosableFileFilter(new TextFileFilter());
        int result = chooser.showSaveDialog(frame);
        if (result == JFileChooser.APPROVE_OPTION) {
            if (!chooser.getSelectedFile().getName().endsWith(TextFileFilter.TEXT_FILE_EXTENSION)) {
                chooser.setSelectedFile(new File(chooser.getSelectedFile().getAbsolutePath() + TextFileFilter.TEXT_FILE_EXTENSION));
            }
            try {
                PrintWriter output = new PrintWriter(new FileWriter(chooser.getSelectedFile()));
                output.print(functionsString);
                output.close();
            } catch (IOException e1) {
                JOptionPane.showMessageDialog(frame, "Error! Can't create file.",
                        "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
```

```
    }

}

private class BuildFunctionsAction extends AbstractAction {

    private BuildFrame frame;

    public BuildFunctionsAction(BuildFrame frame) {
        this.frame = frame;
    }

    public void actionPerformed(ActionEvent e) {
        JPanel functionsPanel = new JPanel();
        functionsPanel.setLayout(new BorderLayout());
        JTextArea functionsArea = new JTextArea();
        functionsArea.setEditable(false);
        functions = FunctionsWorker.getFunctions(tableModel);
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < functions.size(); i++) {
            builder.append(functions.get(i).toString());
            builder.append("\n");
        }
        functionsString = builder.toString();
        functionsArea.setText(functionsString);
        JPanel functionsButtonPanel = new JPanel();
        JButton saveFunctionsButton = new JButton(new SaveFunctionsAction(frame));
        saveFunctionsButton.setText("Save Functions");
        minimizeFunctionsButton = new JButton(new MinimizeFunctionsAction(frame));
        minimizeFunctionsButton.setText("Minimize Functions");
        JButton closeButton = new JButton(new AbstractAction() {
            public void actionPerformed(ActionEvent e) {
                setVisible(false);
            }
        });
        closeButton.setText("Close");
        functionsButtonPanel.add(saveFunctionsButton);
        functionsButtonPanel.add(minimizeFunctionsButton);
        functionsButtonPanel.add(closeButton);
        functionsPanel.add(functionsButtonPanel, BorderLayout.SOUTH);
        functionsPanel.add(new JScrollPane(functionsArea));
        tabbedPane.addTab("Functions", functionsPanel);
        tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);
        buildFunctionsButton.setEnabled(false);
    }

}

private class SaveMinimizedFunctionsAction extends AbstractAction {

    private BuildFrame frame;

    public SaveMinimizedFunctionsAction(BuildFrame frame) {
        this.frame = frame;
    }

    public void actionPerformed(ActionEvent e) {
        JFileChooser chooser = mainFrame.getChooser();
        chooser.resetChoosableFileFilters();
        chooser.addChoosableFileFilter(new TextFileFilter());
        int result = chooser.showSaveDialog(frame);
        if (result == JFileChooser.APPROVE_OPTION) {
            if (!chooser.getSelectedFile().getName().endsWith(TextFileFilter.TEXT_FILE_EXTENSION)) {
                chooser.setSelectedFile(new File(chooser.getSelectedFile().getAbsolutePath() + TextFileFilter.TEXT_FILE_EXTENSION));
            }
            try {
                PrintWriter output = new PrintWriter(new FileWriter(chooser.getSelectedFile()));
                output.print(minimizedFunctionsString);
                output.close();
            } catch (IOException e1) {
                JOptionPane.showMessageDialog(frame, "Error! Can't create file.",
                        "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

}

private class MinimizeFunctionsAction extends AbstractAction {

    private BuildFrame frame;

    public MinimizeFunctionsAction(BuildFrame frame) {
        this.frame = frame;
    }

    public void actionPerformed(ActionEvent e) {
        JPanel minimizedFunctionsPanel = new JPanel();
        minimizedFunctionsPanel.setLayout(new BorderLayout());
        JTextArea minimizedFunctionsArea = new JTextArea();
        minimizedFunctionsArea.setEditable(false);
        minimizedFunctions = FunctionsWorker.minimizeFunctions(FunctionsWorker.prepareFunctionsToMinimization(functions));
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < minimizedFunctions.size(); i++) {
            builder.append(minimizedFunctions.get(i).toString());
            builder.append("\n");
        }
        minimizedFunctionsString = builder.toString();
        minimizedFunctionsArea.setText(minimizedFunctionsString);
        JPanel minimizedFunctionsButtonPanel = new JPanel();
        JButton saveMinimizedFunctionsButton = new JButton(new SaveMinimizedFunctionsAction(frame));
        saveMinimizedFunctionsButton.setText("Save Minimized Functions");
        JButton analyzeButton = new JButton(new AbstractAction() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(frame, FunctionsWorker.analyzeMinimization(functions, minimizedFunctions).toString(),
                                              "Analysys Of Efficiency Of Minimization", JOptionPane.INFORMATION_MESSAGE);
            }
        });
        analyzeButton.setText("Analyze Minimization");
```

```
            convertToBasisButton = new JButton(new ConvertToBasisAction(frame));
            convertToBasisButton.setText("Convert Functions To Basis NOT, 3AND");
            JButton closeButton = new JButton(new AbstractAction() {
                public void actionPerformed(ActionEvent e) {
                    setVisible(false);
                }
            });
            closeButton.setText("Close");
            minimizedFunctionsButtonPanel.add(saveMinimizedFunctionsButton);
            minimizedFunctionsButtonPanel.add(analyzeButton);
            minimizedFunctionsButtonPanel.add(convertToBasisButton);
            minimizedFunctionsButtonPanel.add(closeButton);
            minimizedFunctionsPanel.add(minimizedFunctionsButtonPanel, BorderLayout.SOUTH);
            minimizedFunctionsPanel.add(new JScrollPane(minimizedFunctionsArea));
            tabbedPane.addTab("Minimized Functions", minimizedFunctionsPanel);
            tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);
            minimizeFunctionsButton.setEnabled(false);
        }

    }

    private class SaveToVHDLAction extends AbstractAction {

        private BuildFrame frame;

        public SaveToVHDLAction(BuildFrame frame) {
            this.frame = frame;
        }

        public void actionPerformed(ActionEvent e) {
            JFileChooser chooser = mainFrame.getChooser();
            chooser.resetChoosableFileFilters();
            chooser.addChoosableFileFilter(new VHDLFileFilter());
            int result = chooser.showSaveDialog(frame);
            if (result == JFileChooser.APPROVE_OPTION) {
                if (!chooser.getSelectedFile().getName().endsWith(VHDLFileFilter.VHDL_FILE_EXTENSION)) {
                    chooser.setSelectedFile(new File(chooser.getSelectedFile().getAbsolutePath() + VHDLFileFilter.VHDL_FILE_EXTENSION));
                }
                try {
                    PrintWriter output = new PrintWriter(new FileWriter(chooser.getSelectedFile()));
                    output.print(FunctionsWorker.getVHDLDescriptionOfFunctions(convertedFunctions));
                    output.close();
                } catch (IOException e1) {
                    JOptionPane.showMessageDialog(frame, "Error! Can't create file.",
                            "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
        }

    }

    private class ConvertToBasisAction extends AbstractAction {

        private BuildFrame frame;

        public ConvertToBasisAction(BuildFrame frame) {
            this.frame = frame;
        }

        public void actionPerformed(ActionEvent e) {
            JPanel convertedFunctionsPanel = new JPanel();
            convertedFunctionsPanel.setLayout(new BorderLayout());
            JTextArea convertedFunctionsArea = new JTextArea();
            convertedFunctionsArea.setEditable(false);
            convertedFunctions = FunctionsWorker.convertFromAndOrTo3AndNotBasis(minimizedFunctions);
            StringBuilder builder = new StringBuilder();
            for (int i = 0; i < convertedFunctions.size(); i++) {
                builder.append(convertedFunctions.get(i).toString());
                builder.append("\n");
            }
            convertedFunctionsString = builder.toString();
            convertedFunctionsArea.setText(convertedFunctionsString);
            JPanel convertedFunctionsButtonPanel = new JPanel();
            JButton saveToVHDLButton = new JButton(new SaveToVHDLAction(frame));
            saveToVHDLButton.setText("Save Functions To VHDL");
            JButton closeButton = new JButton(new AbstractAction() {
                public void actionPerformed(ActionEvent e) {
                    setVisible(false);
                }
            });
            closeButton.setText("Close");
            convertedFunctionsButtonPanel.add(saveToVHDLButton);
            convertedFunctionsButtonPanel.add(closeButton);
            convertedFunctionsPanel.add(convertedFunctionsButtonPanel, BorderLayout.SOUTH);
            convertedFunctionsPanel.add(new JScrollPane(convertedFunctionsArea));
            tabbedPane.addTab("Converted Functions", convertedFunctionsPanel);
            tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);
            convertToBasisButton.setEnabled(false);
        }

    }

}
```

Висновки

В результаті виконання даної лабораторної роботи я здобув навички з автоматизації генерації аналітичних форм булевих функцій в заданому елементному базисі з табличної форми. Я реалізував модуль для приведення функцій до заданого елементного базису та збереження результатів у файлі формату VHDL. Модуль був реалізований на мові програмування Java. Також в процесі виконання роботи я освоїв основи мови опису апаратури інтегральних схем VHDL.