**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**
**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**
**«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук
Департамент программной инженерии

СОГЛАСОВАНО
Старший научный сотрудник научно-учебной лаборатории ПОИС

_____ А.А. Каленкова
«___» _____ 2018 г.

УТВЕРЖДАЮ
Академический руководитель образовательной программы «Программная инженерия»

_____ В.В. Шилов
«___» _____ 2018 г.

| Подп. и дата | |
|---|---|
| Инв. № дубл. | |
| Взам. инв. № | |
| Подп. и дата | |
| Инв. № подл | RU.17701729.507140-01 12 01-1 |

# ПРОГРАММА ПОСТРОЕНИЯ РЕГУЛЯРНОГО ВЫРАЖЕНИЯ ПО КОНЕЧНОМУ АВТОМАТУ

## Текст программы

## ЛИСТ УТВЕРЖДЕНИЯ

## RU.17701729.04.13-01 12 01-1-ЛУ

Исполнитель:
студент группы БПИ173
_____ / Таранцова П.Д. /
«____»_____ 2018 г.

**2018**

**УТВЕРЖДЕНО**
**RU.17701729.04.13-01 12 01-1-ЛУ**

# ПРОГРАММА ПОСТРОЕНИЯ РЕГУЛЯРНОГО ВЫРАЖЕНИЯ ПО КОНЕЧНОМУ АВТОМАТУ

**Текст программы**

**RU.17701729.04.13-01 12 01-1**

**Листов 109**

| Инв. № подл | Подп. и дата | Взам. инв. № | Инв. № дубл. | Подп. и дата |
|---|---|---|---|---|
| RU.17701729.507140-01 12 01-1 | | | | |

**2018**

# Аннотация

В данном документе приведет текст «Программы построения регулярного выражения по конечному автомату». Программа разработана на языке C# 4.0 в среде разработки - Microsoft Visual Studio 2017.

Функциональным назначением программы является работа с загруженным в программу из внешних источников или построенным пользователем конечным автоматом, предполагающая последовательное приведение конечного автомата к соответствующему регулярному выражению посредством метода удаления состояний. В качестве выходных данных программа предоставляет регулярное выражение, полученное с применением алгоритма минимизации регулярного выражения и без него, а также анализ эффективности данного алгоритма.

Настоящий документ разработан в соответствии с требованиями:

1) ГОСТ 19.101-77 Виды программ и программных документов [1];
2) ГОСТ 19.102-77 Стадии разработки [2];
3) ГОСТ 19.103-77 Обозначения программ и программных документов [3];
4) ГОСТ 19.104-78 Основные надписи [4];
5) ГОСТ 19.105-78 Общие требования к программным документам [5];
6) ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом[6];
7) ГОСТ 19.401-78 Текст программы. Требования к содержанию и оформлению [7].

Изменения к данному документу оформляются согласно ГОСТ 19.603-78 [8], ГОСТ 19.604-78 [9].

Перед прочтением данного документа рекомендуется ознакомиться с терминологией, приведенной в Приложении 1.

## СОДЕРЖАНИЕ

Аннотация
1. Текст программы....4
1.1 Код библиотеки DeletingAlgo....4
1.1.1 Код класса StateRemoval....4
1.2. Код библиотеки DrawingSupporter....19
1.2.1. Код класса DrawingSupporter....19
1.3. Код библиотеки LibGraph....29
1.3.1. Код класса GraphBuilder....29
1.4. Код библиотеки Project....34
1.4.1. Код класса ChoiceFSMForm....34
1.4.2. Код класса Draw....36
1.4.3. Код класса FormBefore....53
1.4.4. Код класса FormInfo....58
1.4.5. Код класса FormProcessDefinitions....58
1.4.6. Код класса FormRegularExpression....62
1.4.7. Код класса FormRemoveStateWithAnimation....69
1.4.8. Код класса FormSavingRules....76
1.4.9. Код класса FormTransitionName....79
1.4.10. Код класса FormUsersFSM....81
1.4.11. Код класса Minimizator....88
1.4.12. Код класса TSMLtransformer....97
1.4.13. Код класса UsersFiles....104
ПРИЛОЖЕНИЕ 1. Терминология....108
ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ....109

# 1. Текст программы

- Программа включает в себя 4 библиотеки: DeletingAlgo, DawingSupporter, LibGraph и Project. DeletingAlgo хранит алгоритм удаления состояний, DrawingSupporter – инструменты, используемые при отрисовке пользовательского конечного автомата, LibGraph – представление конечного автомата, как графа, с которым работает программа, Project – окна, наследуемые от Windows.Forms, а также другие алгоритмы, используемые программой (минимазация конечного автомата, работа с файлами, преобразование в формат .tsml). Код, хранимый в этих библиотеках, приведен ниже. Кроме того, в программе используется функционал двух сторонних библиотек: Graphviz.2.38.0.2, 2016, авторы: Arif Bilgin, John Ellson, Emden Gansner, Yifan Hu, Arif Bilgin, Stephen North (визуализация графа); QuickGraph.3.6.61119.7, 2011, автор: Jonathan Peli de Halleux (представление графа).

## 1.1 Код библиотеки DeletingAlgo

### 1.1.1 Код класса StateRemoval

```
using System;
using System.Collections.Generic;
using System.Linq;
using QuickGraph;
using LibGraph;

namespace DeletingAlgo
{
    /// <summary>
    ///  Класс, представляющий пару состояние - переход,
используется в списке исходящих дуг.
    /// </summary>
    public class OutEdgeList
    {
        /// <summary>
        /// Регулярное выражение, задающее переход.
        /// </summary>
        public string EdgeLabel { get; set; }
        /// <summary>
        /// Состояние входа.
        /// </summary>
        public string TargetVertex { get; set; }
    }
    /// <summary>
    ///  Класс, представляющий пару состояние - переход,
используется в списке входящих дуг.
    /// </summary>
    public class InEdgeList
```

```
        {
            /// <summary>
            /// Регулярное выражение, задающее переход.
            /// </summary>
            public string EdgeLabel { get; set; }
            /// <summary>
            /// Состояние исхода.
            /// </summary>
            public string SourceVertex { get; set; }
        }
        /// <summary>
        /// Класс, который оперирует удалением состояний.
        /// </summary>
        public class StateRemoval
        {
            Random rand = new Random();
            /// <summary>
            /// Поле, задающее список исходящих из состояния дуг
            /// </summary>
            List<OutEdgeList>      exitingEdges      =      new
List<OutEdgeList>();
            /// <summary>
            /// Поле, задающее список входящих в состояние дуг
            /// </summary>
            List<InEdgeList>      enteringEdges      =      new
List<InEdgeList>();
            /// <summary>
            /// Поле, задающее список неудаляемых состояний.
            /// </summary>
            List<string> untouchedStatesList;
            /// <summary>
            /// Поле, задающее начальное состояние.
            /// </summary>
            string startingState;
            /// <summary>
            /// Поле, задающее единое конечное состояние.
            /// </summary>
            string uniqueFinal;
            /// <summary>
            ///  Поле,  задающее  автомат,  у  которого  удаляются
состояния.
            /// </summary>
            GraphBuilder graph;
            /// <summary>
            /// Конструктор шага удаления состояния.
            /// </summary>
            ///  <param  name="changhedGraph">текущее  состояние
графа</param>
```

```
            ///     <param     name="list">список     неудаляемых
состояний</param>
            ///         <param         name="headState">начальное
состояние</param>
            public                        StateRemoval(GraphBuilder
changhedGraph,List<string>   list,   string   headState,   string
unqueFinal)
            {
                uniqueFinal = unqueFinal;
                startingState = headState;
                untouchedStatesList = list;
                graph = changhedGraph;
            }
            /// <summary>
            /// Метод, который проверяет, является ли состояние
конечным или начальным.
            /// </summary>
            /// <param name="vertexName">имя состояния</param>
            /// <returns></returns>
            public bool isUntouched(string vertexName)
            {
                return  untouchedStatesList.Exists(x  =>  x  ==
vertexName);
            }
            /// <summary>
            ///  Метод, добавляющий в автомат единое конечное
состояние.
            /// </summary>
            /// <param name="uniqueFinal">Имя единого конечного
состояния.</param>
            /// <returns></returns>
            public   GraphBuilder   TransformGraph(out   string
uniqueFinal)
            {
                graph.UserAddFinalVertix(uniqueFinal          =
RandomVertexName());
                foreach(string vertex in untouchedStatesList)
                {
                    if  (vertex  !=  String.Empty  &&  vertex  !=
startingState)
                    {
                        graph.UserRemoveFinalVertix(vertex);
                        graph.UserAddEdge(vertex,   uniqueFinal,
"&");
                    }
                }
                return graph;
            }
```

```
            /// <summary>
            /// Метод, проверяющий, остались ли в автомате неконечные состояния, при положительном результате возвращает имя состояния, которое необходимо удалить.
            /// </summary>
            /// <param name="name">Имя неконечного состояния.</param>
            /// <returns>Наличие неконечных состояния.</returns>
            public bool NonFinalLeft(out string name)
            {
                bool result = false;
                name = "";
                graph.ConcatSelfLoops();
                Dictionary<string, int> income = new Dictionary<string, int>();//ключ - имя состояния, значение - длины всех регулярных выражений, задающих переходы входящих в состояние дуг
                Dictionary<string, int> outcome = new Dictionary<string, int>();//ключ - имя состояния, значение - длины всех регулярных выражений, задающих переходы исходящих из состояния дуг
                Dictionary<string, int> inrange = new Dictionary<string, int>();//ключ- имя состояния, значение - количество входящих в состояние дуг
                Dictionary<string, int> outrange = new Dictionary<string, int>();//ключ- имя состояния, значение - количество исходящих из состояния дуг
                Dictionary<string, int> self = new Dictionary<string, int>();//ключ - имя состояния, значение - длина регулярного выражения, задающего петлю
                Dictionary<string, int> damage = new Dictionary<string, int>();//ключ - имя состояния, значение - урон, который нанесет удаление состояния регулярному выражению, задающему автомат
                foreach(var edge in graph.myGraph.Edges)//обход всех дуг конечного автомата и заполнение словарей
                {
                    if (!self.Keys.Contains(edge.Source))
                        self.Add(edge.Source, 0);
                    if (!self.Keys.Contains(edge.Target))
                        self.Add(edge.Target, 0);
                    if (!income.Keys.Contains(edge.Target))
                    {
                        income.Add(edge.Target, 0);
                        inrange.Add(edge.Target, 0);
                    }
                    if (!income.Keys.Contains(edge.Source))
                    {
```

```
                    income.Add(edge.Source, 0);
                    inrange.Add(edge.Source, 0);
                }
                if (!outcome.Keys.Contains(edge.Source))
                {
                    outcome.Add(edge.Source, 0);
                    outrange.Add(edge.Source, 0);
                }
                if (!outcome.Keys.Contains(edge.Target))
                {
                    outcome.Add(edge.Target, 0);
                    outrange.Add(edge.Target, 0);
                }
                if (edge.Source == edge.Target)
                    self[edge.Source] += edge.Tag.Length;
                else
                {
                        income[edge.Target]                += 
edge.Tag.Length;
                        inrange[edge.Target]++;
                        outcome[edge.Source]               += 
edge.Tag.Length;
                        outrange[edge.Source]++;
                }
            }
            foreach(string key in income.Keys)
                damage.Add(key, income[key] * outcome[key] + 
inrange[key] * outrange[key] * self[key]);
            int min = 1000000000;
            foreach(string   key   in   damage.Keys)//поиск 
состояния с минимальным уроном
            {
                if (!isUntouched(key))
                {
                    if (damage[key] < min)
                    {
                        min = damage[key];
                        name = key;
                    }
                    result = true;
                }
            }
            return result;
        }

        /// <summary>
        ///    Метод,    реализующий    алгоритм    удаления 
недопускающего состояния.
```

```
            /// </summary>
            ///   <param   name="nameOfVertex">Имя   удаляемого
состояния.</param>
            /// <returns></returns>
            public    GraphBuilder    RemoveNonFinalState(string
nameOfVertex)
            {
                int   numOfOutEdges;//количество   исходящих   из
состояния дуг
                int    numOfInEdges;//количество    входящих    в
состояние дуг
                string selfEdge = String.Empty;//петля
                bool  existInEdges=false;//индикатор,  есть  ли
входящие дуги
                enteringEdges  =  new  List<InEdgeList>();//список
входящих дуг
                exitingEdges  =  new  List<OutEdgeList>();//список
исходящих дуг
                if (graph.myGraph.OutEdges(nameOfVertex).Count()
== 0)//исключение, если из неконечного состояния нет исходящих дуг
                    throw  new  Exception("The  automata  is  not
finite!");
                foreach           (var           edge           in
graph.myGraph.OutEdges(nameOfVertex))//формируем список исходящих
из состояния дуг
                {
                    if (edge.Target == nameOfVertex)
                        selfEdge = edge.Tag;
                    else
                        exitingEdges.Add(new    OutEdgeList    {
EdgeLabel = edge.Tag, TargetVertex = edge.Target });
                }
                foreach(var                edge                in
graph.myGraph.Edges)//формируем список входящих в состояние дуг
                {

if(edge.Target==nameOfVertex&&edge.Source!=nameOfVertex)
                    {
                            enteringEdges.Add(new  InEdgeList  {
EdgeLabel = edge.Tag, SourceVertex = edge.Source });
                            existInEdges = true;
                    }
                }
                if (!existInEdges)//исключение, если в состояние
нет входящих дуг
                    throw  new  Exception("The  automata  is  not
finite!");
                numOfOutEdges = exitingEdges.Count;
```

```
                numOfInEdges = enteringEdges.Count;
                List<string> numberOfUniqChangedEdges = new
List<string>();//список получившихся регулярных выражений
                Dictionary<string,int> prefixes = new
Dictionary<string, int>();//словарь префиксов, ключ которого -
регулярное выражение, задающее префикс, а значение - количество
повторений префикса
                for (int
enterIndex=0;enterIndex<numOfInEdges;enterIndex++)//формируем
регулярные выражения для всех пар (входящая дуга) - (исходящая
дуга)

//это выражения, соединяющие состояния, связанные удаляемым
                    for(int
exitIndex=0;exitIndex<numOfOutEdges;exitIndex++)
                    {
                        graph.ConcatSelfLoops();
                        string label = "";//регулярное выражение,
для текущей пары (входящая дуга) - (исходящая дуга)
                        bool existsEdge=false;//индикатор, есть
ли дуга, связывающая состояния
                        TaggedEdge<string, string> edge;//дуга,
соединяющая
                        string source =
enteringEdges[enterIndex].SourceVertex;//состояние,из которого
есть переход в удаляемое
                        string target =
exitingEdges[exitIndex].TargetVertex;//состояние,в которое есть
переход из удаляемого
                        if (graph.myGraph.TryGetEdge(source,
target, out edge) && !(source == startingState & target ==
uniqueFinal))//проверяем, есть ли дуга, связывающая состояния
source и target
                        {
                            label = edge.Tag + "+";
                            existsEdge = true;
                        }
                        if (label != "" && label.Last() != '+')
                            label += '+';
                        string addedLabel = "";//регулярное
выражение, получившееся после удаления текущего состояния
                        if
(enteringEdges[enterIndex].EdgeLabel.Contains("+"))
                            addedLabel += "(" +
enteringEdges[enterIndex].EdgeLabel + ")";
                        else
                            addedLabel +=
enteringEdges[enterIndex].EdgeLabel;//добавляем в регулярное
```

```
выражение символ, по которому осуществляется переход из состояния
source в удаляемое
                                if (selfEdge != String.Empty)//если
существует петля, добавляем ее к регулярному выражению
                                {
                                    if    (addedLabel    !=    ""    &&
addedLabel.Last() != '.')
                                        addedLabel += '.';
                                    addedLabel += "("+selfEdge+")*";
                                }

if(!prefixes.Keys.Contains(addedLabel)&&addedLabel.Length>2)//ec
ли префикс не был добавлен в словарь ранее, добавляем его
                                    prefixes.Add(addedLabel,0);
                                if     (addedLabel     !=     ""     &&
addedLabel.Last() != '.')
                                    addedLabel += '.';
                                if
(exitingEdges[exitIndex].EdgeLabel.Contains("+"))//добавляем    к
регулярному выражению конкатенацию с переходом исходящей дуги
                                    addedLabel      +=      "("      +
exitingEdges[exitIndex].EdgeLabel + ")";
                                else
                                    addedLabel                        +=
exitingEdges[exitIndex].EdgeLabel;
                                if (existsEdge)//если дуга уже есть,
удаляем ее
                                    graph.myGraph.RemoveEdge(edge);
                                //добавляем   дугу   с   получившимся
регулярным выражением

graph.UserAddEdge(enteringEdges[enterIndex].SourceVertex,
exitingEdges[exitIndex].TargetVertex, label+ addedLabel);
                               if
(!numberOfUniqChangedEdges.Contains(label + addedLabel))
                                numberOfUniqChangedEdges.Add(label +
addedLabel);
                              else
if(!graph.finalDefinitions.Values.Contains(label+ addedLabel))
                              {
                                int                num                =
graph.finalDefinitions.Values.Count;
                                    string str = ((char)(num % 26 +
65)).ToString();
                                    if (num >= 26)
                                        str += num / 26;
                                graph.finalDefinitions.Add(str,
label + addedLabel);
```

```
                        }
                    }
                    graph.ConcatSelfLoops();
                    graph.myGraph.RemoveVertex(nameOfVertex);
                DefineSubprocesses();
                DefinePrefixes(prefixes);
                return graph;
            }
            /// <summary>
            /// Метод извлечения подпроцессов по префиксам.
            /// </summary>
            ///          <param          name="prefixes">Словарь
префиксов.</param>
            void     DefinePrefixes(Dictionary<string,     int>
prefixes)
            {
                List<string>     keys     =     SortByLength(new
List<string>(prefixes.Keys));//сортируем  список  префисксов  по
убыванию их длины
                //замена по подпроцессам
                foreach (var edge in graph.myGraph.Edges)
                {
                    string newLabel = "";
                    string[] stringArray = edge.Tag.Split('+');
                    int i = 0;
                    foreach (string str in stringArray)
                    {
                        i++;
                        if
(graph.finalDefinitions.Values.Contains(str))
                        {
                            foreach      (string      key      in
graph.finalDefinitions.Keys)
                                if  (graph.finalDefinitions[key]
== str)
                                    newLabel += key;
                        }
                        else
                            newLabel += str;
                        if (i != stringArray.Length)
                            newLabel += "+";
                    }
                    edge.Tag = newLabel;
                    foreach       (string       prefix       in
keys)//подсчитывается  количество  вхождений  префиксов  в  состав
перехода
                        prefixes[prefix]                      +=
CountOccurance(newLabel, prefix);
```

```
                }
                Dictionary<string, string> frequentPrefixes = new
Dictionary<string,   string>();//словарь   часто   встречающиеся
префиксов
                foreach (string prefix in keys)
                    if (prefixes[prefix] > 1)
                    {
                        int                 num                 =
graph.finalDefinitions.Values.Count + frequentPrefixes.Count;
                        string  strVal  =  ((char)(num  %  26  +
65)).ToString();
                        if (num >= 26)
                            strVal += num / 26;
                        frequentPrefixes.Add(strVal, prefix);
                    }
                //замена по префиксам
                foreach (var edge in graph.myGraph.Edges)
                    foreach       (string       prefix       in
frequentPrefixes.Keys)
                        edge.Tag    =    RemoveIdentic(edge.Tag,
frequentPrefixes[prefix], prefix);
                foreach (string key in frequentPrefixes.Keys)
                    graph.finalDefinitions.Add(key,
frequentPrefixes[key]);
            }
            /// <summary>
            /// Метод извлечения подпроцессов в составе переходов
на дугах конечного автомата.
            /// </summary>
            void DefineSubprocesses()
            {
                List<string>     numberOfUniqParts     =     new
List<string>();//список подпроцессов
                foreach (var edge in graph.myGraph.Edges)
                {
                    foreach        (string        key        in
graph.finalDefinitions.Keys)//заменяем подпроцессы всех переходов
на ключевые символы подпроцессов
                        if    (graph.finalDefinitions[key]    ==
edge.Tag)
                            edge.Tag = key;
                    string[] strArray = (edge.Tag).Split('+');
                    List<string> strArr = new List<string>();
                    int i = 0;
                    while  (i  <  strArray.Length)//находим  все
подпроцессы, входящие в состав данного перехода
                    {
```

```
                        char[]              symbs               =
strArray[i].ToCharArray();
                        int open = 0;
                        int close = 0;
                        foreach (char c in symbs)
                            if (c == '(')
                                open++;
                            else if (c == ')')
                                close++;
                        if (open == close)
                            strArr.Add(strArray[i]);
                        else
                        {
                            string makeNormal = strArray[i];
                            do
                            {
                                i++;
                                makeNormal += "+" + strArray[i];
                                open = 0;
                                close = 0;
                                symbs                           =
makeNormal.ToCharArray();
                                foreach (char c in symbs)
                                    if (c == '(')
                                        open++;
                                    else if (c == ')')
                                        close++;
                            } while (open != close);
                            strArr.Add(makeNormal);
                        }
                        i++;
                    }
                    foreach (string str in strArr)//добавляем все
подпроцессы, входящие в состав данного перехода, длина которых
более одного символа, в словарь подпроцессов
                        if (str.Length != 1)
                        {
                            if
(!numberOfUniqParts.Contains(str))
                                numberOfUniqParts.Add(str);
                            else                                if
(!graph.finalDefinitions.Values.Contains(str))
                            {
                                int             num             =
graph.finalDefinitions.Values.Count;
                                string strVal = ((char)(num % 26
+ 65)).ToString();
                                if (num >= 26)
```

```
                                    strVal += num / 26;

graph.finalDefinitions.Add(strVal, str);
                                }
                            }
                        }
                    }
                    /// <summary>
                    ///  Метод, подсчитывающий количество повторений
подстроки в строке.
                    /// </summary>
                    /// <param name="str"></param>
                    /// <param name="substr"></param>
                    /// <returns></returns>
                    int CountOccurance(string str, string substr)
                    {
                        int count = (str.Length - str.Replace(substr,
"").Length) / substr.Length;
                        return count;
                    }
                    /// <summary>
                    /// Метод, сортирующий список строк по убыванию их
длины.
                    /// </summary>
                    /// <param name="list">Сортируемый список.</param>
                    /// <returns>Отсортированный список.</returns>
                    List<string> SortByLength(List<string> list)
                    {
                        List<string> result = new List<string>();
                        int max = 0;
                        do
                        {
                            max = 0;
                            foreach (string str in list)
                                if (str.Length > max)
                                    max = str.Length;
                            bool found = false;
                            do
                            {

                                found = false;
                                foreach (string str in list)
                                {
                                    if (str.Length == max)
                                    {
                                        found = true;
                                        list.Remove(str);
                                        result.Add(str);
```

```
                                break;
                            }
                        }
                    } while (found == true);
                } while (list.Count > 0);
                return result;
            }
            /// <summary>
            ///  Метод, заменяющий подпроцессы в регулярном
выражении на их ключевые символы.
            /// </summary>
            /// <param name="str">Регулярное выражение.</param>
            /// <param name="substr">Подпроцесс.</param>
            ///     <param     name="change">Ключевой     символ
подпроцесса.</param>
            ///          <returns>Измененное          регулярное
выражение.</returns>
            string  RemoveIdentic(string  str,  string  substr,
string change)
            {
                string cleanedStr="";
                if (str.Contains(substr))
                    do
                    {
                        int index = str.IndexOf(substr);
                        cleanedStr      +=      str.Substring(0,
index)+change;
                        str    =    str.Remove(0,    index    +
substr.Length);
                    } while (str.Contains(substr));
                cleanedStr += str;
                return cleanedStr ;
            }
            /// <summary>
            /// Поиск свободного имени для состояния.
            /// </summary>
            /// <returns>Имя состояния.</returns>
                public string RandomVertexName()
            {
                string name;
                name = rand.Next(65, 97).ToString();
                if (graph.myGraph.Vertices.Contains(name))
                    do
                        name += rand.Next(65, 97).ToString();
                    while
(!graph.myGraph.Vertices.Contains(name));
                return name;
            }
```

```
            /// <summary>
            /// Метод, удаляющий лишние скобки в регулярных
выражениях.
            /// </summary>
            /// <param name="expressions">Список регулярных
выражений, к которым требуется применить алгоритм.</param>
            /// <returns>Список измененных регулярных
выражений.</returns>
            public List<string> DeleteBrackets(List<string>
expressions)
            {
                int count;
                for (int i=0;i<expressions.Count;i++)
                {
                    expressions[i].Remove(0,1);
                    expressions[i].Remove(expressions[i].Length
- 1,1);
                }
                count = expressions.Count;
                for(int index=0;index<count;index++)
                {
                    foreach(string expression in expressions)
                    {
                        if(expression=="")
                        {
                            expressions.Remove(expression);
                            break;
                        }
                    }
                }
                return expressions;
            }
            /// <summary>
            /// Метод, осуществляющий финальное преобразование
конечного автомата в регулярное выражение
            /// </summary>
            /// <param name="graph">Конечный автомат(представлен
как начальное и конечное состояние, множество переходов).</param>
            /// <returns>Регулярные выражения.</returns>
            public List<string> FinalRemoval(GraphBuilder graph)
            {
                string regularExpression = "";
                int count = 0;
                List<string> expressions = new List<string>();
                foreach (var edge in
graph.myGraph.OutEdges(startingState))
                {
```

```
                    regularExpression                                   =
Change(edge.Tag.Replace('\n', ' '));
                    count++;
                    expressions.Add(regularExpression);

                }
                expressions = DeleteBrackets(expressions);
                for(int index=0;index<expressions.Count;index++)
                {
                    if (index != count - 1)
                        expressions[index] += "+";
                }
                return expressions;
            }
            /// <summary>
            /// Метод, удаляющий пустые переходы.
            /// </summary>
            ///       <param       name="expression">Изменяемое
выражение.</param>
            /// <returns>змененное выражение.</returns>
            public string Change(string expression)
            {
                string result="";
                char[] expArr = expression.ToCharArray();
                foreach(char c in expArr)
                {
                    if (c == '&' && result[result.Length - 1] ==
'.')
                        result  =  result.Remove(result.Length  -
1);
                    else if (c == '+' && result[result.Length -
1] == '&')
                    {
                        result  =  result.Remove(result.Length  -
1);
                        result += (char)400;
                        result += c;
                    }
                    else
                        result += c;
                }
                return result;
            }
        }
    }
```

## 1.2. Код библиотеки DrawingSupporter

### 1.2.1. Код класса DrawingSupporter

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Drawing2D;

namespace DrawingSupporter
{
    /// <summary>
    ///   Класс,   оперирующий   отрисовкой   элементов
создаваемого пользователем конечного автомата.
    /// </summary>
    public class DrawAssistant
    {
        Bitmap bitmap;
        Pen blackPen = new Pen(Color.Black);
        Pen bluePen = new Pen(Color.Blue);
        Pen redPen = new Pen(Color.Red);
        Pen greenPen = new Pen(Color.Green);
        Pen brownPen = new Pen(Color.Brown);
        Graphics graphics;
        Point point;
        Font vertexFont;
        Font edgeFont;
        /// <summary>
        /// Список ребер графа.
        /// </summary>
        List<Edge> edgeList;
        /// <summary>
        /// Список вершин графа.
        /// </summary>
        List<Vertex> vertexList;
        /// <summary>
        /// Список конечных состояний графа.
        /// </summary>
        List<string> finalsList;
        /// <summary>
        /// Начальное состояние графа.
        /// </summary>
        string startingState="";
        /// <summary>
        /// Радиус вершины графа.
        /// </summary>
        int Rad = 20;
        /// <summary>
```

```
        /// Свойство, возвращающее начальное состояние
автомата.
        /// </summary>
        public string StartingState { set { startingState
= value; } }
        /// <summary>
        /// Свойство, возвращающее радиус состояния
автомата.
        /// </summary>
        public int Radius { get { return Rad; } }
        /// <summary>
        /// Свойство, хранящее изображение построенного
автомата.
        /// </summary>
        public Bitmap Image { get { return bitmap; } }
        /// <summary>
        /// Свойство, хранящее список конечных состояний
автомата.
        /// </summary>
        public List<string> Finals { get { return
finalsList; } }
        public DrawAssistant(int width, int height,
List<Edge> edgeList, List<Vertex> vertexList,
List<string> finals)
        {
            bitmap = new Bitmap(width, height);
            graphics = Graphics.FromImage(bitmap);
            vertexFont = new Font("Times New Roman", 16,
FontStyle.Bold);
            edgeFont = new Font("Times New Roman", 16);
            this.edgeList = edgeList;
            this.vertexList = vertexList;
            finalsList = finals;
        }
        /// <summary>
        /// Метод, изменяющий список конечных состояний
автомата.
        /// </summary>
        /// <param name="name">Имя состояния</param>
        public void ChangeFinalsList(string name)
        {
            if (name != startingState)
            {
                if (finalsList.Contains(name))
                    finalsList.Remove(name);
                else
                    finalsList.Add(name);
            }
```

```
        }
        /// <summary>
        /// Метод, выделяющий вершину, выбранную как
исходную для ребра.
        /// </summary>
        /// <param name="x">Координата вершины по оси
X.</param>
        /// <param name="y">Координата вершины по оси
Y.</param>
        /// <param name="name">Название вершины.</param>
        public void DrawSelectedVertex(int x, int
y,string name)
        {
            graphics.FillEllipse(Brushes.Aqua, x - Rad, y
- Rad, 2 * Rad, 2 * Rad);
            graphics.DrawEllipse(bluePen, x - Rad, y -
Rad, 2 * Rad, 2 * Rad);
            point = new Point(x - 5 * Rad / 12, y - 6 *
Rad / 12);
            graphics.DrawString(name, vertexFont,
Brushes.Blue, point);
        }
        /// <summary>
        /// Метод, выделяющий вершину, выбранную для
удаления.
        /// </summary>
        /// <param name="x">Координата вершины по оси
X.</param>
        /// <param name="y">Координата вершины по оси
Y.</param>
        /// <param name="name">Название вершины.</param>
        public void DrawVertexToDelete(int x, int y,
string name)
        {
            graphics.FillEllipse(Brushes.Pink, x - Rad, y
- Rad, 2 * Rad, 2 * Rad);
            graphics.DrawEllipse(redPen, x - Rad, y -
Rad, 2 * Rad, 2 * Rad);
            point = new Point(x - 5 * Rad / 12, y - 6 *
Rad / 12);
            graphics.DrawString(name, vertexFont,
Brushes.Red, point);
        }
        /// <summary>
        /// Метод, выполняющий отрисовку вершины.
        /// </summary>
        /// <param name="x">Координата вершины по оси
X.</param>
```

```
        /// <param name="y">Координата вершины по оси
Y.</param>
        /// <param name="name">Название вершины.</param>
        public void DrawVertex(int x, int y, string name)
        {
            if (name == startingState)//если начальная
вершина - рисуем зеленым
            {
                graphics.FillEllipse(Brushes.LightGreen,
x - Rad, y - Rad, 2 * Rad, 2 * Rad);
                graphics.DrawEllipse(greenPen, x - Rad, y
- Rad, 2 * Rad, 2 * Rad);
            }
            else if (finalsList.Contains(name))//если
вершина из списка конечных состояний - рисуем желтым
            {

graphics.FillEllipse(Brushes.LightYellow, x - Rad, y -
Rad, 2 * Rad, 2 * Rad);
                graphics.DrawEllipse(brownPen, x - Rad, y
- Rad, 2 * Rad, 2 * Rad);
            }
            else//если обычная вершина - рисуем в черно-
белом стиле
            {
                graphics.FillEllipse(Brushes.White, x -
Rad, y - Rad, 2 * Rad, 2 * Rad);
                graphics.DrawEllipse(blackPen, x - Rad, y
- Rad, 2 * Rad, 2 * Rad);
            }
            point = new Point(x - 5 * Rad / 12, y - 6 *
Rad / 12);
            graphics.DrawString(name,        vertexFont,
Brushes.Black, point);
        }
        /// <summary>
        /// Метод, выполняющий отрисовку ребра.
        /// </summary>
        /// <param name="x1">Координата вершины исхода по
оси X.</param>
        /// <param name="y1">Координата вершины исхода по
оси Y.</param>
        /// <param name="x2">Координата вершины входа по
оси X.</param>
        /// <param name="y2">Координата вершины входа по
оси Y.</param>
        /// <param name="name">Символ перехода.</param>
```

```
        /// <param name="edge">Само ребро, как линия,
соединяющая две вершины.</param>
        public void DrawEdge(int x1, int y1, int x2, int
y2, string name, Edge edge)
        {
            Pen pen = new Pen(Color.Black, 1);
            pen.StartCap = LineCap.RoundAnchor;
            pen.EndCap = LineCap.ArrowAnchor;
            if (x1 != x2 && y1 != y2)
            {
                int startX = 0, startY = 0, finX = 0, finY
= 0;
                Point p1 = new Point(x2 - x1, y2 - y1);
                int k1 = (int)(Rad * (x2 - x1) /
Math.Sqrt(Math.Pow(x2 - x1, 2) + Math.Pow(y2 - y1, 2)));
                int k2 = (int)(Rad * (y2 - y1) /
Math.Sqrt(Math.Pow(x2 - x1, 2) + Math.Pow(y2 - y1, 2)));
                if (!ReverseEdge(edge))
                {
                    startX = x1 + k1;
                    startY = y1 + k2;
                    finX = x2 - k1;
                    finY = y2 - k2;
                    graphics.DrawLine(pen,       startX,
startY, finX, finY);
                    point = new Point((x1 + x2 - k1) / 2,
(y1 + y2 - k1) / 2);
                }
                else
                {
                    if (y1 < y2)
                    {
                        startY = y1;
                        finY = y2;
                        if (x1 > x2)
                        {
                            startX = x1 - Rad;
                            finX = x2 + Rad;
                        }
                        if (x2 >= x1)
                        {
                            startX = x1 + Rad;
                            finX = x2 - Rad;
                        }
                    }
                    if (y1 > y2)
                    {
                        if (x1 == x2)
```

```
                    {
                        startY = y1;
                        finY = y2;
                        startX = x1 + Rad;
                        finX = x2 - Rad;
                    }
                    else
                    {
                        startX = x1;
                        finX = x2;
                        startY = y1 - Rad;
                        finY = y2 + Rad;
                    }
                }
                if (y1 == y2)
                {
                    startX = x1;
                    finX = x2;
                    startY = y1 - Rad;
                    finY = y2 + Rad;
                }
                graphics.DrawLine(pen,       startX,
startY, finX, finY);
                point = new Point(startX + (finX -
startX) / 3, startY + (finY - startY) / 3);
                }
            }
            else
            {
                int x = x1 - 20 * Rad / 12;
                int y = y1 - Rad;
                float width = 2 * Rad;
                float height = 22 * Rad / 12;
                graphics.DrawArc(pen, x,y, width, height,
70, 360);
                point = new Point(x1 - 5 * Rad / 2, y1 -
Rad);
            }
            graphics.DrawString(name,          edgeFont,
Brushes.Black, point);
        }
        /// <summary>
        /// Метод, выполняющий отрисовку графа по списку
ребер и вершин.
        /// </summary>
        public void Draw()
        {
            ClearSheet();
```

```
            foreach (Edge edge in edgeList)
                DrawEdge(edge.Source.X,   edge.Source.Y,
edge.Target.X, edge.Target.Y, edge.Transition, edge);
            foreach (Vertex vert in vertexList)
                DrawVertex(vert.X, vert.Y, vert.Name);
        }
        /// <summary>
        ///  Метод, определяющий, есть ли обратное к
данному ребро.
        /// </summary>
        ///  <param name="edge">Само ребро, как линия,
соединяющая две вершины.</param>
        /// <returns></returns>
        bool ReverseEdge(Edge edge)
        {
            foreach (Edge e in edgeList)
                if   (edge.Source   ==   e.Target   &&
edge.Target == e.Source)
                    return true;
            return false;
        }
        /// <summary>
        /// Метод, очищающий лист для рисования.
        /// </summary>
        public void ClearSheet()
        {
            graphics.Clear(Color.White);
        }
        /// <summary>
        /// Метод, удаляющий вершину из списка вершин.
        /// </summary>
        /// <param name="vertex">Вершина графа, как имя и
пара координат.</param>
        public void DeleteVertex(Vertex vertex)
        {
            vertexList.Remove(vertex);
            bool found = false;
            do
            {
                found = false;
                foreach (var edge in edgeList)
                    if (edge.Source.Name == vertex.Name
|| edge.Target.Name == vertex.Name)
                    {
                        found = true;
                        edgeList.Remove(edge);
                        break;
                    }
```

```
            } while (found);
            Draw();
        }
        /// <summary>
        /// Метод, удаляющий ребро из списка ребер.
        /// </summary>
        ///  <param name="edge">Само ребро, как линия,
соединяющая две вершины.</param>
        public void DeleteEdge(Edge edge)
        {
            edgeList.Remove(edge);
            Draw();
        }
        /// <summary>
        /// Метод, определяющий, нажал ли пользователь на
ребро.
        /// </summary>
        ///  <param name="edge">Ребро, на которое нажал
пользователь. (Имеет значение null, если не было нажато
ни одно ребро).</param>
        ///  <param name="x">Координата мыши по оси
X.</param>
        ///  <param name="y">Координата мыши по оси
Y.</param>
        /// <returns></returns>
        public bool EdgeHitten(ref Edge edge, int x, int
y)
        {
            foreach (Edge e in edgeList)
            {
                if (CheckEdgeHitten(e,x,y))
                {
                    edge = e;
                    return true;
                }
            }
            return false;
        }
        /// <summary>
        /// Метод, определяющий, принадлежит ли данная
точка ребру.
        /// </summary>
        /// <param name="edge">Проверяемое ребро.</param>
        ///  <param name="x">Координата мыши по оси
X.</param>
        ///  <param name="y">Координата мыши по оси
Y.</param>
        /// <returns></returns>
```

```
        public bool CheckEdgeHitten(Edge edge, int x, int
y)
        {
            int x1 = edge.Source.X;
            int y1 = edge.Source.Y;
            if (edge.Source==edge.Target)
            {
                if (Math.Pow(x - x1, 2) + Math.Pow(y -
y1, 2) <= 4 * Rad*Rad)
                    return true;
            }
            else
            {
                int x2 = edge.Target.X;
                int y2 = edge.Target.Y;
                float catheter = (y2 - y1) * (x - x1) /(x2
- x1) + y1;
                if (catheter>=y-4&&catheter<=y+4)
                {
                    if (x >= x2 && x <= x1 || x >= x1 &&
x <= x2)
                        return true;
                }
            }
            return false;
        }
        /// <summary>
        /// Метод, определяющий, нажал ли пользователь на
вершину.
        /// </summary>
        ///  <param name="vertex">Вершина, на которую
нажал пользователь. (Имеет значение null, если не была
нажата ни одна вершина).</param>
        ///  <param name="x">Координата мыши по оси
X.</param>
        ///  <param name="y">Координата мыши по оси
Y.</param>
        /// <param name="vertices">Список вершин.</param>
        /// <returns></returns>
        public bool VertexHitten(ref Vertex vertex, int
x, int y, List<Vertex> vertices)
        {
            foreach (var vert in vertices)
                if   (Math.Pow(vert.X   -   x,   2)   +
Math.Pow(vert.Y - y, 2) <= Rad * Rad)
                {
                    vertex = vert;
                    return true;
```

```
                }
            return false;
        }
    }
        /// <summary>
        /// Класс, задающий вершину графа, как имя и пару
координат.
        /// </summary>
        public class Vertex
        {
        /// <summary>
        /// Имя состояния.
        /// </summary>
            public string Name { get; set; }
        /// <summary>
        /// Координата вершины графа по оси X.
        /// </summary>
            public int X { get; set; }
        /// <summary>
        /// Координата вершины графа по оси Y.
        /// </summary>
        public int Y { get; set; }

            public Vertex(string name, int x, int y)
            {
                Name = name;
                X = x;
                Y = y;
            }
        }
    /// <summary>
    /// Класс, задающий переход, как символ перехода и
два состояния, как вершины графа.
    /// </summary>
        public class Edge
        {
        /// <summary>
        /// Символ перехода.
        /// </summary>
            public string Transition { get; set; }
        /// <summary>
        /// Вершина исхода.
        /// </summary>
            public Vertex Source { get; set; }
        /// <summary>
        /// Вершина входа.
        /// </summary>
            public Vertex Target { get; set; }
```

```
            public Edge(Vertex source, Vertex target,
string tag)
            {
                Transition = tag;
                Source = source;
                Target = target;
            }
        }
}
```

### 1.3. Код библиотеки LibGraph

#### 1.3.1. Код класса GraphBuilder

```
using System;
using System.Collections.Generic;
using System.Linq;
using QuickGraph;
using QuickGraph.Graphviz;
using QuickGraph.Graphviz.Dot;
using System.IO;
using System.Diagnostics;

namespace LibGraph

{
    /// <summary>
    /// Класс, участвующий в получении изображения из
конечного автомата.
    /// </summary>
    public class FileDotEngine : IDotEngine
    {
        /// <summary>
        /// Получение изображения автомата.
        /// </summary>
        /// <param name="imageType"></param>
        /// <param name="dot"></param>
        /// <param name="outputFileName"></param>
        /// <returns></returns>
        public string Run(GraphvizImageType imageType,
string dot, string outputFileName)
        {
            using (StreamWriter writer = new
StreamWriter(outputFileName))
            {
                writer.Write(dot);
            }
            return Path.GetFileName(outputFileName);
```

```
        }
    }
    /// <summary>
    /// Класс, задающий конечный автомат.
    /// </summary>
    public class GraphBuilder

    {
        /// <summary>
        ///   Количество   совершенных   перобразований
автомата.
        /// </summary>
        public int NumberOfGraphs { get; private set; }
        /// <summary>
        /// Поле, список конечных состояний.
        /// </summary>
        List<string> finalStates=new List<string>();
        /// <summary>
        /// Свойство, список конечных состояний.
        /// </summary>
        public List<string> FinalStates { get { return
finalStates; } }
        /// <summary>
        /// Словарь процессов.
        /// </summary>
        public        Dictionary<string,        string>
basicDefinitions = new Dictionary<string, string>();
        /// <summary>
        /// Словарь подпроцессов.
        /// </summary>
        public        Dictionary<string,        string>
finalDefinitions = new Dictionary<string, string>();
        /// <summary>
        /// Пользовательский конечный автомат.
        /// </summary>
        public AdjacencyGraph<string, TaggedEdge<string,
string>>   myGraph   =   new   AdjacencyGraph<string,
TaggedEdge<string, string>>();
        /// <summary>
        ///  Метод,  инициализирующий  список  конечных
состояний.
        /// </summary>
        ///   <param   name="finStates">Список   конечных
состояний.</param>
        public  void  InitializeFinalStates(List<string>
finStates)
        {
            finalStates = finStates;
```

```
        }
        /// <summary>
        /// Метод, объединяющий все петли состояния.
        /// </summary>
        public void ConcatSelfLoops()
        {
            foreach (string vertex in myGraph.Vertices)
            {
                List<string> loopsText = new
List<string>();
                int outDegree =
myGraph.OutDegree(vertex);
                for (int i = 0; i < outDegree; i++)
                {
                    foreach (var edge in
myGraph.OutEdges(vertex))
                    {
                        if (edge.Target == vertex)
                        {
                            loopsText.Add(edge.Tag);
                            myGraph.RemoveEdge(edge);
                            break;
                        }
                    }
                }
                string edgeText = "";
                if (loopsText.Count != 0)
                {
                    if (loopsText.Count == 1)
                        edgeText = loopsText.First();
                    else
                    {
                        int num = 0;
                        foreach (string text in
loopsText)
                        {
                            num++;
                            edgeText += text;
                            if (num != loopsText.Count)
                                edgeText += "+";
                        }
                    }
                    UserAddEdge(vertex, vertex,
edgeText);
                }
            }
        }
        /// <summary>
```

```
        /// Метод, добавляющий в автомат состояние.
        /// </summary>
        /// <param name="nameVert">Имя состояния.</param>
        public void UserAddVertix(string nameVert)
        {
            myGraph.AddVertex(nameVert);
        }
        /// <summary>
        /// Метод, удаляющий конечное состояние автомата
из списка.
        /// </summary>
        /// <param name="nameVert">Имя состояния.</param>
        public     void     UserRemoveFinalVertix(string
nameVert)
        {
            finalStates.Remove(nameVert);
        }
        /// <summary>
        /// Метод, добавляющий в автомат переход.
        /// </summary>
        ///        <param        name="startVert">Вершина
исхода.</param>
        /// <param name="endVert">Вершина входа.</param>
        ///         <param         name="edgeText">Символ
перехода.</param>
        public void UserAddEdge(string startVert, string
endVert, string edgeText)
        {
            myGraph.AddEdge(new        TaggedEdge<string,
string>(startVert, endVert, edgeText));
        }
        /// <summary>
        ///   Метод,   добавляющий   в   автомат   конечное
состояние.
        /// </summary>
        /// <param name="nameVert">Имя состояния.</param>
        public void UserAddFinalVertix(string nameVert)
        {
            myGraph.AddVertex(nameVert);
            finalStates.Add(nameVert);
        }
        /// <summary>
        ///   Метод,   преобразующий   описание   графа   в
картинку.
        /// </summary>
        /// <param name="graphNum"></param>
        /// <returns></returns>
        public string MakeDot(int graphNum)
```

```
        {
            if (graphNum != -1)
                NumberOfGraphs = graphNum;
            var graphViz = new GraphvizAlgorithm<string,
TaggedEdge<string, string>>(myGraph);
            graphViz.FormatVertex += FormVertex;
            graphViz.FormatEdge += FormEdge;
            string                  name                =
String.Format("graphviz{0}",graphNum);
            var                 dotPath                 =
Directory.GetCurrentDirectory()+Path.DirectorySeparatorC
har+"Graphs"+Path.DirectorySeparatorChar+ name + ".dot";
            graphViz.Generate(new       FileDotEngine(),
dotPath);
            var               dotExePath                =
@"Graphviz.2.38.0.2\dot.exe";
             if (File.Exists(dotExePath))
             {
                var          psi          =          new
ProcessStartInfo(dotExePath);
                psi.Arguments = String.Format( "-Tpng -o
\"{0}\"           \"{1}\"",Path.ChangeExtension(dotPath,
"png"),dotPath);
                psi.UseShellExecute = false;
                psi.RedirectStandardOutput = true;
                psi.CreateNoWindow = true;
                var p = Process.Start(psi);
                var               stdout                =
p.StandardOutput.ReadToEnd();
                p.WaitForExit();
            }
            return Path.ChangeExtension(dotPath, "png");
        }
        /// <summary>
        /// Метод, задающий формат вершины графа.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="vert"></param>
        private    void    FormVertex(object    sender,
FormatVertexEventArgs<string> vert)
        {
            bool isFinal = false;
            foreach (string fin in finalStates)
                if (vert.Vertex == fin)
                    isFinal = true;
            vert.VertexFormatter.Label = vert.Vertex;
            if(isFinal)
```

```
                vert.VertexFormatter.Shape              =
GraphvizVertexShape.DoubleCircle;
            else
                vert.VertexFormatter.Shape              =
GraphvizVertexShape.Circle;
                vert.VertexFormatter.StrokeColor        =
GraphvizColor.Black;
                vert.VertexFormatter.FillColor          =
GraphvizColor.LightYellow;
                vert.VertexFormatter.Style              =
GraphvizVertexStyle.Bold;
                vert.VertexFormatter.Font            =new
GraphvizFont("Calibri", 12);

        }
        /// <summary>
        /// Метод, задающий формат ребра графа.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="edge"></param>
        private     void     FormEdge(object     sender,
FormatEdgeEventArgs<string,  TaggedEdge<string,  string>>
edge)
        {
                edge.EdgeFormatter.Label      =      new
GraphvizEdgeLabel { Value = edge.Edge.Tag };
                edge.EdgeFormatter.Font       =       new
GraphvizFont("TimesNewRoman",                          12-
edge.Edge.Tag.Length/60);
                edge.EdgeFormatter.FontGraphvizColor   =
GraphvizColor.Black;
                edge.EdgeFormatter.StrokeGraphvizColor =
GraphvizColor.Black;
        }
    }
}
```

### 1.4. Код библиотеки Project

#### 1.4.1. Код класса ChoiceFSMForm

```
using System;
using System.Windows.Forms;
using System.IO;

namespace Project
{
    /// <summary>
```

```
    /// Класс, представляющий стартовое окно приложения с
возможностью выбрать, каким образом начать работу с
программой.
    /// </summary>
    public partial class ChoiceFSMForm : Form
    {
        public ChoiceFSMForm()
        {
            InitializeComponent();
        }
        /// <summary>
        /// Метод, обрабатывающий желание пользователя
создать конечный автомат и открывающий окно отрисовки
автомата.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void buttonCreate_Click(object sender,
EventArgs e)
        {
            Draw draw = new Draw(this);

Directory.CreateDirectory(Directory.GetCurrentDirectory(
) + Path.DirectorySeparatorChar.ToString() + "Graphs");
            draw.Show();
            Hide();

        }
        /// <summary>
        /// Метод, открывающий окно с информацией о
разработчике.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void aboutToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            FormInfo formInfo = new FormInfo();
            formInfo.ShowDialog();
        }
        /// <summary>
        /// Метод, обрабатывающий желание пользователя
загрузить автомат из стороннего ресурса и открывающий окно
открытия файла.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
```

```
        private void buttonFromFile_Click(object sender,
EventArgs e)
        {
            UsersFiles file = new UsersFiles();
            bool opened = file.OpenFile();

Directory.CreateDirectory(Directory.GetCurrentDirectory(
) + Path.DirectorySeparatorChar.ToString() + "Graphs");
            TSMLtransformer      tSML      =      new
TSMLtransformer(file.FilePath,this, opened);
        }
        /// <summary>
        ///  Метод, завершающий работу приложения при
закрытии окна.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private   void   ChoiceFSMForm_FormClosed(object
sender, FormClosedEventArgs e)
        {
            Application.Exit();
        }
    }
}
```

### 1.4.2. Код класса Draw

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
using DrawingSupporter;
using System.IO;
using LibGraph;

namespace Project
{
    /// <summary>
    /// Класс, представляющий окно с инструментами и полем
для создания пользовательского конечного автомата.
    /// </summary>
    public partial class Draw : Form
    {
        /// <summary>
        /// Инструмент, оперирующий операциями отрисовки.
        /// </summary>
        DrawAssistant drawer;
        /// <summary>
```

```
        /// Построенный граф, представляющий конечный
автомат.
        /// </summary>
        GraphBuilder graph;
        /// <summary>
        ///   Истинно,   если   происходит   добавление
состояний.
        /// </summary>
        bool addingVertices;
        /// <summary>
        ///   Истинно,   если   происходит   добавление
переходов.
        /// </summary>
        bool addingEdges;
        /// <summary>
        ///   Истинно,   если   происходит   передвижение
вершины.
        /// </summary>
        bool movingVertex;
        /// <summary>
        /// Истинно, если происходит выбор начального
состояния.
        /// </summary>
        bool selectingStart;
        /// <summary>
        /// Истинно, если происходит изменение списка
конечных вершин.
        /// </summary>
        bool correctingFinalsList;
        /// <summary>
        /// Имя текущего передвигаемого состояния.
        /// </summary>
        string moovingName = "";
        //Константные значения текстовых полей.
        const   string   textAddVert1   =   "Start   adding
vertices";
        const   string   textAddVert2   =   "Stop   adding
vertices";
        const string textAddEdge1 = "Start adding edges";
        const string textAddEdge2 = "Stop adding edges";
        //Границы, в пределах которых можно перемещать
объекты.
        int leftBound;
        int rightBound;
        int upperBound;
        int lowerBound;
        /// <summary>
        /// Вершина, из которой исходит ребро.
```

```
        /// </summary>
        Vertex sourceVertex;
        /// <summary>
        /// Вершина, в которую входит ребро.
        /// </summary>
        Vertex targetVertex;
        /// <summary>
        /// Перемещаемая вершина.
        /// </summary>
        Vertex movedVertex;
        /// <summary>
        /// Список вершин.
        /// </summary>
        List<Vertex> vertexList;
        /// <summary>
        /// Список ребер.
        /// </summary>
        List<Edge> edgeList;
        /// <summary>
        /// Список ребер, которые нужно переместить.
        /// </summary>
        List<Edge> movingEdgeList;
        /// <summary>
        /// Список конечных состояний.
        /// </summary>
        List<string> finalsList;
        /// <summary>
        /// Начальное состояние.
        /// </summary>
        string startingState;
        /// <summary>
        /// Стартовое окно.
        /// </summary>
        ChoiceFSMForm newProjectForm;
        public Draw(ChoiceFSMForm form)
        {
            InitializeComponent();
            newProjectForm = form;
            finalsList = new List<string>();
            vertexList = new List<Vertex>();
            edgeList = new List<Edge>();
            movingEdgeList = new List<Edge>();
            drawer = new
DrawAssistant(pictureBoxAutomata.Width,
pictureBoxAutomata.Height,edgeList,vertexList,
finalsList);
            pictureBoxAutomata.Image = drawer.Image;
```

```
            Bitmap bit = new Bitmap(Image.FromStream(new
MemoryStream(File.ReadAllBytes("circle1.jpg"))),     new
Size(10   *   buttonVertex.Size.Width   /   12,   10   *
buttonVertex.Size.Width / 12));
            buttonVertex.Image= bit;
            bit   =   new   Bitmap(Image.FromStream(new
MemoryStream(File.ReadAllBytes("transition.jpg"))),  new
Size(10   *   buttonEdge.Size.Width   /   12,   10   *
buttonEdge.Size.Width / 12));
            buttonEdge.Image = bit;
            moovingName = "";
            startingState = "";
            addingVertices = false;
            movingVertex = false;
            addingEdges = false;
            selectingStart = false;
            correctingFinalsList = false;
            labelAddEdge.Text = textAddEdge1;
            labelAddVertex.Text = textAddVert1;
            labelHelpAdd.Text = "";
            leftBound = drawer.Radius;
            rightBound    =    pictureBoxAutomata.Width-
drawer.Radius;
            lowerBound = drawer.Radius;
            upperBound   =   pictureBoxAutomata.Height   -
drawer.Radius;
            Size          =          new          Size(
pictureBoxAutomata.Location.X                           +
pictureBoxAutomata.Width,pictureBoxAutomata.Location.Y  +
pictureBoxAutomata.Height);
        }
        /// <summary>
        ///  Обработка желания пользователя добавить
вершину.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private  void  buttonVertex_Click(object  sender,
EventArgs e)
        {
            drawer.Draw();
            pictureBoxAutomata.Image = drawer.Image;
            addingVertices = !addingVertices;
            if (addingVertices)
            {
                labelHelpAdd.Text = "Adding vertices...";
                labelAddVertex.Text = textAddVert2;
            }
```

```
            else
            {
                labelAddVertex.Text = textAddVert1;
                labelHelpAdd.Text = "";
            }
            movingVertex = false;
            addingEdges = false;
            selectingStart = false;
            correctingFinalsList = false;
            labelAddEdge.Text = textAddEdge1;
        }
        /// <summary>
        ///  Обработка желания пользователя добавить
ребро.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void buttonEdge_Click(object sender,
EventArgs e)
        {
            drawer.Draw();
            pictureBoxAutomata.Image = drawer.Image;
            addingVertices = false;
            movingVertex = false;
            selectingStart = false;
            correctingFinalsList = false;
            labelAddVertex.Text = textAddVert1;
            addingEdges = !addingEdges;
            if (addingEdges)
            {
                labelHelpAdd.Text = "Please, choose
source vertex...";
                labelAddEdge.Text = textAddEdge2;
            }
            else
            {
                labelAddEdge.Text = textAddEdge1;
                labelHelpAdd.Text = "";
            }
        }
        /// <summary>
        ///  Обработка нажатия на поле для рисования.
Выполняет операции рисования в зависимости от того, какое
действие выполняет пользователь в данный момент.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
```

```
        private                                          void
pictureBoxAutomata_MouseClick(object                sender,
MouseEventArgs e)
        {
            Vertex vertex = null;
            if (e.Button == MouseButtons.Left)
            {
                if (!movingVertex && !addingEdges)
                {
                    if   (addingVertices)//если   выбрали
место для добавления состояния, рисуем состояние
                    {
                        int x, y;
                        SetXY(out x, out y, e);
                        string name = (vertexList.Count +
1).ToString();
                        drawer.DrawVertex(x, y, name);
                        vertexList.Add(new   Vertex(name,
x, y));
                        pictureBoxAutomata.Image       =
drawer.Image;
                    }
                    if         (selectingStart         &&
drawer.VertexHitten(ref vertex, e.X, e.Y, vertexList))
                    {
                       startingState=
drawer.StartingState = vertex.Name;
                        drawer.Draw();
                        pictureBoxAutomata.Image       =
drawer.Image;
                    }
                    if      (correctingFinalsList      &&
drawer.VertexHitten(ref vertex, e.X, e.Y, vertexList))
                    {

drawer.ChangeFinalsList(vertex.Name);
                        drawer.Draw();
                        pictureBoxAutomata.Image       =
drawer.Image;
                    }
                    if
(!selectingStart&&!correctingFinalsList&&!addingVertices
&&    drawer.VertexHitten(ref    vertex,    e.X,    e.Y,
vertexList))//начали движение состояния
                    {
                        movingVertex = true;
                        vertexList.Remove(vertex);
                        drawer.Draw();
```

```
                        moovingName = vertex.Name;
                    }

                    }
                if             (addingEdges             &&
drawer.VertexHitten(ref      vertex,      e.X,      e.Y,
vertexList))//выбрали связующее состояние
                {
                    if (sourceVertex == null)
                    {
                        sourceVertex = vertex;

drawer.DrawSelectedVertex(vertex.X,vertex.Y,vertex.Name)
;
                        pictureBoxAutomata.Image       =
drawer.Image;
                        labelHelpAdd.Text   =   "Please,
choose target vertex...";
                    }
                    else
                    {
                        targetVertex = vertex;
                        labelHelpAdd.Text   =   "Please,
choose source vertex...";
                        FormTransitionName  form  =  new
FormTransitionName(this);
                        form.Show();
                    }
                }
            }
            else
            {
                Edge edge=null;
                DialogResult dia = DialogResult.No;
                if (drawer.VertexHitten(ref vertex, e.X,
e.Y, vertexList))
                {
                    drawer.DrawVertexToDelete(vertex.X,
vertex.Y, vertex.Name);
                    pictureBoxAutomata.Image           =
drawer.Image;
                    dia = MessageBox.Show("Delete vertex
" + vertex.Name + "?", "Sure", MessageBoxButtons.YesNo);
                }
                if (dia == DialogResult.Yes)
                {
                    drawer.DeleteVertex(vertex);
```

```
                    pictureBoxAutomata.Image            =
drawer.Image;
                }
                if(drawer.EdgeHitten(ref edge,e.X,e.Y))
                    dia = MessageBox.Show("Delete edge "
+       edge.Transition       +       "?",       "Sure",
MessageBoxButtons.YesNo);
                if (dia == DialogResult.Yes)
                {
                    drawer.DeleteEdge(edge);
                    pictureBoxAutomata.Image            =
drawer.Image;
                }
            }
        }
        /// <summary>
        /// Завершение добавление ребра после введения
текста перехода.
        /// </summary>
        /// <param name="name"></param>
        public void CompleteAddingEdge(string name)
        {
            if (name != "")
            {
                bool found=false;
                foreach(Edge edge in edgeList)
                {

if(edge.Target==targetVertex&&edge.Source==sourceVertex)
                    {
                        found = true;
                        edge.Transition                 =
edge.Transition + "+" + name;
                        break;
                    }
                }
                if(!found)
                    edgeList.Add(new  Edge(sourceVertex,
targetVertex, name));
            }
            drawer.Draw();
            pictureBoxAutomata.Image = drawer.Image;
            sourceVertex = null;
        }
        /// <summary>
        /// Метод выполняющий проверку, не пытается ли
пользователь переместить объект за поле для рисования.
        /// </summary>
```

```
        /// <param name="x"></param>
        /// <param name="y"></param>
        /// <param name="e"></param>
        void SetXY(out int x, out int y, MouseEventArgs
e)
        {
            if (e.X > leftBound && e.X < rightBound)
                x = e.X;
            else if (e.X <= leftBound)
                x = leftBound;
            else
                x = rightBound;
            if (e.Y > lowerBound && e.Y < upperBound)
                y = e.Y;
            else if (e.Y <= lowerBound)
                y = lowerBound;
            else
                y = upperBound;
        }
        /// <summary>
        /// Метод, обрабатывающий разжатие кнопки мыши.
Выполняет операции рисования в зависимости от того, какое
действие выполняет пользователь в данный момент.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void pictureBoxAutomata_MouseUp(object
sender, System.Windows.Forms.MouseEventArgs e)
        {
            if (movingVertex && !addingEdges &&
!correctingFinalsList && !selectingStart &&
!addingVertices)
            {
                int x, y;
                SetXY(out x, out y, e);
                Vertex vertex = new
Vertex(movedVertex.Name, x, y);
                vertexList.Add(vertex);
                bool found = false;
                do
                {
                    found = false;
                    foreach (var edge in movingEdgeList)
                        if (edge.Source.Name ==
movedVertex.Name || edge.Target.Name == movedVertex.Name)
                        {
```

```
                                if    (edge.Source.Name    ==
movedVertex.Name        &&        (edge.Source.X        !=
x||edge.Source.Y!=y))
                                {
                                    found = true;
                                    edge.Source = vertex;
                                }
                                if    (edge.Target.Name    ==
movedVertex.Name        &&        (edge.Target.X        !=
x||edge.Target.Y!=y))
                                {
                                    found = true;
                                    edge.Target = vertex;
                                }
                                break;
                            }
                        } while (found);
                        foreach (var edge in movingEdgeList)
                            edgeList.Add(edge);
                        drawer.Draw();
                        pictureBoxAutomata.Image = drawer.Image;
                        movingEdgeList = new List<Edge>();
                        movedVertex = null;
                        movingVertex = !movingVertex;
                    }
                }

                /// <summary>
                ///   Метод,   обрабатывающий   движение   мыши.
Выполняет операции рисования в зависимости от того, какое
действие выполняет пользователь в данный момент.
                /// </summary>
                /// <param name="sender"></param>
                /// <param name="e"></param>
                private                                    void
pictureBoxAutomata_MouseMove_2(object              sender,
MouseEventArgs e)
                {
                    if
(movingVertex&&!addingEdges&&!correctingFinalsList    &&
!selectingStart&&!addingVertices )
                    {
                        int x, y;
                        SetXY(out x, out y, e);
                        bool found = false;
                        do
                        {
                            found = false;
```

```
                    foreach (var edge in movingEdgeList)
                        if      (edge.Source.Name      ==
movedVertex.Name || edge.Target.Name == movedVertex.Name)
                        {
                            if    (edge.Source.Name    ==
movedVertex.Name&& (edge.Source.X != x || edge.Source.Y
!= y))
                            {
                                found = true;
                                edge.Source.X = x;
                                edge.Source.Y = y;
                            }
                            if    (edge.Target.Name    ==
movedVertex.Name&& (edge.Target.X != x || edge.Target.Y
!= y))
                            {
                                found = true;
                                edge.Target.X = x;
                                edge.Target.Y = y;
                            }
                            break;
                        }
                } while (found);
                movedVertex.X = x;
                movedVertex.Y = y;
                drawer.ClearSheet();
                foreach (Edge edge in movingEdgeList)
                    drawer.DrawEdge(edge.Source.X,
edge.Source.Y,        edge.Target.X,        edge.Target.Y,
edge.Transition,edge);
                foreach (Edge edge in edgeList)
                    drawer.DrawEdge(edge.Source.X,
edge.Source.Y,        edge.Target.X,        edge.Target.Y,
edge.Transition,edge);
                foreach (Vertex vert in vertexList)
                    drawer.DrawVertex(vert.X,     vert.Y,
vert.Name);
                drawer.DrawVertex(x,
y,movedVertex.Name);
                pictureBoxAutomata.Image = drawer.Image;

            }
        }
        /// <summary>
        /// Метод, обрабатывающий зажатие кнопки мыши.
Выполняет операции рисования в зависимости от того, какое
действие выполняет пользователь в данный момент.
        /// </summary>
```

```
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void pictureBoxAutomata_MouseDown(object
sender, MouseEventArgs e)
        {
            Vertex vertex = null;
            if (e.Button == MouseButtons.Left)
            {
                if
(!correctingFinalsList&&!selectingStart&&!addingEdges &&
!addingVertices && drawer.VertexHitten(ref vertex, e.X,
e.Y, vertexList))
                {
                    movingVertex = true;
                    vertexList.Remove(vertex);
                    movedVertex = vertex;
                    bool found = false;
                    do
                    {
                        found = false;
                        foreach (var edge in edgeList)
                            if    (edge.Source.Name    ==
vertex.Name || edge.Target.Name == vertex.Name)
                            {
                                found = true;
                                edgeList.Remove(edge);

movingEdgeList.Add(edge);
                                break;
                            }
                    } while (found);
                }
            }
            else
            {
                DialogResult dia = DialogResult.No;
                if (drawer.VertexHitten(ref vertex, e.X,
e.Y, vertexList))
                {
                    drawer.DrawVertexToDelete(vertex.X,
vertex.Y, vertex.Name);
                    pictureBoxAutomata.Image           =
drawer.Image;
                    dia = MessageBox.Show("Delete vertex
" + vertex.Name + "?", "Sure", MessageBoxButtons.YesNo);
                }
                if (dia == DialogResult.Yes)
                {
```

```
                    drawer.DeleteVertex(vertex);
                    pictureBoxAutomata.Image =
drawer.Image;
                }
            }
        }
        /// <summary>
        /// Метод, обрабатывающий желание пользователя
назначить начальное состояние.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void buttonStartingVertex_Click(object
sender, EventArgs e)
        {
            drawer.Draw();
            pictureBoxAutomata.Image = drawer.Image;
            selectingStart = !selectingStart;
            if(!selectingStart)
                labelHelpAdd.Text = "";
            else
                labelHelpAdd.Text = "Selecting starting
state...";
            addingVertices = false;
            movingVertex = false;
            addingEdges = false;
            correctingFinalsList = false;
            labelAddEdge.Text = textAddEdge1;
            labelAddVertex.Text = textAddVert1;
        }
        /// <summary>
        /// Метод, обрабатывающий желание пользователя
отредвктировать список конечных состояний.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void buttonAddFinal_Click(object sender,
EventArgs e)
        {
            drawer.Draw();
            pictureBoxAutomata.Image = drawer.Image;
            correctingFinalsList =
!correctingFinalsList;
            if(correctingFinalsList==false)
                labelHelpAdd.Text = "";
            else
                labelHelpAdd.Text = "Correcting finals
list...";
```

```
            addingVertices = false;
            movingVertex = false;
            addingEdges = false;
            selectingStart = false;
            labelAddEdge.Text = textAddEdge1;
            labelAddVertex.Text = textAddVert1;
        }
        /// <summary>
        /// Метод, обрабатывающий желание пользователя
закончить построение автомата.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void buttonFinish_Click(object sender,
EventArgs e)
        {
            if (startingState == String.Empty)
                MessageBox.Show(   "Please,   choose   a
starting state...", "Error!",MessageBoxButtons.OK);
            else if (drawer.Finals.Count == 0)
                MessageBox.Show( "Please, choose at least
one final state...", "Error!",MessageBoxButtons.OK);
            else if (edgeList.Count == 0)
                MessageBox.Show("Please, add at least one
edge", "Error!",MessageBoxButtons.OK);
            else
            {
                graph = new GraphBuilder();
                foreach(Vertex vertex in vertexList)
                {
                    if (vertex.Name == startingState)
                    {
                        graph.UserAddVertix("");

graph.UserAddVertix(vertex.Name);
                        graph.UserAddEdge("",
vertex.Name, "");
                    }
                    else                                  if
(drawer.Finals.Contains(vertex.Name))

graph.UserAddFinalVertix(vertex.Name);
                    else

graph.UserAddVertix(vertex.Name);
                }
                foreach (Edge edge in edgeList)
```

```
                    graph.UserAddEdge(edge.Source.Name,
edge.Target.Name, edge.Transition);
                FormUsersFSM      form      =      new
FormUsersFSM(graph,                          drawer.Finals,
startingState,newProjectForm);
                form.Show();
                Hide();
            }

        }
        /// <summary>
        ///  Метод, открывающий окно с информацией о
разработчике.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void aboutToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            FormInfo formInfo = new FormInfo();
            formInfo.ShowDialog();
        }
        /// <summary>
        /// Метод, открывающий стартовое окно приложения.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private                                      void
newProjectToolStripMenuItem_Click(object         sender,
EventArgs e)
        {
            DialogResult         dialogResult         =
DialogResult.Cancel;
            dialogResult=MessageBox.Show("Are you sure,
that you want to start a new project? Current progress
will be lost.", "Ensure starting new project",
MessageBoxButtons.OKCancel);
            if(dialogResult==DialogResult.OK)
            {
                newProjectForm.Show();
                Hide();
            }
        }
        /// <summary>
        ///  Метод, сохраняющий построенный автомат в
формате .tsml.
        /// </summary>
        /// <param name="sender"></param>
```

```
        /// <param name="e"></param>
        private void saveToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            if (startingState == String.Empty)
                MessageBox.Show("Please,    choose    a
starting state...", "Error!",MessageBoxButtons.OK);
            else if (drawer.Finals.Count == 0)
                MessageBox.Show( "Please, choose at least
one final state...", "Error!",MessageBoxButtons.OK);
            else if (edgeList.Count == 0)
                MessageBox.Show( "Please, add at least
one edge", "Error!",MessageBoxButtons.OK);
            else
            {

                TSMLtransformer transformer;
                UsersFiles file = new UsersFiles();
                string              path              =
file.ChooseSavingPath("tsml");
                if (path == String.Empty)
                    MessageBox.Show("Error! File was not
saved!");
                else
                transformer = new TSMLtransformer(path,
vertexList, finalsList, startingState, edgeList);
                }
        }
        /// <summary>
        ///   Метод,   обрабатывающий   закрытие   формы.
Предупреждает о возможной потери данных.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private  void  Draw_FormClosing(object  sender,
FormClosingEventArgs e)
        {
            DialogResult result;
            if
(e.CloseReason!=CloseReason.ApplicationExitCall)
            {
                result = MessageBox.Show("Do you want to
quit the application?\nCurrent progress will be lost...",
"Ensure        quitting        the        application",
MessageBoxButtons.YesNo);
                if (result == DialogResult.No)
                    e.Cancel = true;
                else
```

```
                    Application.Exit();
                }
            }
            //Далее представлены методы, генерирующие
появление сплывающих подсказок.
            ToolTip toolTip = new ToolTip();
            private void buttonVertex_MouseEnter(object
sender, EventArgs e)
            {
                if(!addingVertices)
                    toolTip.SetToolTip(buttonVertex, "Press
to start adding vertices...");
                else
                    toolTip.SetToolTip(buttonVertex, "Press
to stop adding vertices...");
            }

            private void buttonEdge_MouseEnter(object sender,
EventArgs e)
            {
                if(!addingEdges)
                    toolTip.SetToolTip(buttonEdge, "Press to
start adding edges...");
                else
                    toolTip.SetToolTip(buttonEdge, "Press to
stop adding edges...");
            }

            private                                    void
buttonStartingVertex_MouseEnter(object sender, EventArgs
e)
            {
                if(!selectingStart)
                    toolTip.SetToolTip(buttonStartingVertex,
"Press to select a starting state of the FSM...");
                else
                    toolTip.SetToolTip(buttonStartingVertex,
"Press to stop selecting the starting state...");
            }

            private void buttonAddFinal_MouseEnter(object
sender, EventArgs e)
            {
                if(!correctingFinalsList)
                    toolTip.SetToolTip(buttonAddFinal,
"Press to add or remove a state from the list of final
states of the FSM...");
                else
```

```
                toolTip.SetToolTip(buttonAddFinal,
"Press to stop correcting the list of final states of the
FSM...");
        }

        private void buttonFinish_MouseEnter(object
sender, EventArgs e)
        {
            toolTip.SetToolTip(buttonFinish,      "Press
here, if you finished building the FSM...");
        }
    }
}
```

### 1.4.3. Код класса FormBefore

```
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace Project
{
    /// <summary>
    /// Класс, представляющий окно, содержащее регуляное
выражение, полученное по конечному автомату без
использования алгоритма минимизации регулярного
выражения.
    /// Содержит аналитические данные: длину выражения до
минимизации и максимальную вложенность.
    /// </summary>
    public partial class FormBefore : Form
    {
        /// <summary>
        /// Длина регулярного выражения до минимизации.
        /// </summary>
        public string BeforeCount { get { return
labelNumOfEvents.Text; } }
        /// <summary>
        /// Максимальная вложенность.
        /// </summary>
        public string MaxNesting { get { return
labelMaxNesting.Text; } }
        /// <summary>
        /// Содержимое richTextBoxBefore (регулярное
выражение).
        /// </summary>
        public List<string> textBoxContents
        {
```

```
            get
            {
                List<string>     contents     =     new
List<string>();
                foreach      (string      line      in
richTextBoxBefore.Lines)
                    contents.Add(line);
                return contents;
            }
        }
        public                    FormBefore(List<string>
regularExpressions, Dictionary<string,string> finals)
        {
            InitializeComponent();
            Height   =   richTextBoxBefore.Location.Y   +
richTextBoxBefore.Height + 80;
            Dictionary<string,  string>  contents  =  new
Dictionary<string, string>();
            List<string>     expressions     =     new
List<string>();
            int count = 0;
            //заполняем richTextBoxBefore
            bool found = false;
            do
            {
                foreach (string key in finals.Keys)
                {
                    found = false;
                    string str;
                    if (!contents.Keys.Contains(key))
                        str = Change(finals[key]);
                    else
                        str = Change(contents[key]);
                    char[] array = str.ToCharArray();
                    str = "";
                    string s = "";
                    for(int i=0;i<array.Length;i++)
                    {
                        if (!(array[i] >= 'A' && array[i]
<= 'Z' || array[i] >= 'a' && array[i] <= 'z' || array[i]
>= 0 && array[i] <= 9))
                        {
                            if (s != "")
                            {
                                if
(finals.Keys.Contains(s))
                                {
```

```
                                    if (str.Length != 0
&& str[str.Length - 1] == '.')
                                        str += "(" +
Change(finals[s]) + ")";
                                    else
                                        str           +=
Change(finals[s]);
                                    found = true;
                                }
                                else
                                    str += s;
                            }
                            s = "";
                            str += array[i];
                        }
                        else
                            s += array[i];
                    }
                    if (s != "")
                        {
                        if (finals.Keys.Contains(s))
                        {
                            if (str.Length != 0 &&
str[str.Length - 1] == '.')
                            str     +=     "("     +
Change(finals[s]) + ")";
                            else
                                str                   +=
Change(finals[s]);
                            found = true;
                        }
                        else
                            str += s;
                        }
                    if (!contents.Keys.Contains(key))
                        contents.Add(key, str);
                    else
                        contents[key] = str;
                }
                if (found)
                    count++;
            } while (found);
            labelMaxNesting.Text = count.ToString();
            foreach(string str in regularExpressions)
            {
                string result = "";
                char[] charArr = str.ToCharArray();
                string s = "";
```

```
                for (int i = 0; i < charArr.Length; i++)
                {
                    if (!(charArr[i] >= 'A' && charArr[i]
<= 'Z' || charArr[i] >= 'a' && charArr[i] <= 'z' ||
charArr[i] >= 0 && charArr[i] <= 9))
                    {
                        if (s != "")
                        {
                            if
(contents.Keys.Contains(s))
                            {
                                result += contents[s];
                                found = true;
                            }
                            else
                                result += s;
                        }
                        s = "";
                        result += charArr[i];
                    }
                    else
                        s += charArr[i];
                }
                if (s != "")
                {
                    if (contents.Keys.Contains(s))
                    {
                        result += finals[s];
                        found = true;
                    }
                    else
                        result += s;
                }
                expressions.Add(result);
            }
            count = 0;
            foreach (string expression in expressions)
            {
                richTextBoxBefore.Text += expression;
                char[]charArr                                =
expression.ToCharArray();
                foreach (char c in charArr)
                {
                    if (c != '+' && c != '.' && c != ')'
&& c != '(' && c != '*'&&!(c>=1&&c<=9))
                        count++;
                }
            }
```

```
            labelNumOfEvents.Text = count.ToString();
        }
        /// <summary>
        /// Изменение регулярного выражения. (Удаление
символов,   добавленных   вместе   с   единым   конечным
состоянием).
        /// </summary>
        ///     <param     name="expression">Регулярное
выражение.</param>
        /// <returns></returns>
        string Change(string expression)
        {
            string result = "";
            char[] expArr = expression.ToCharArray();
            for (int i = 0; i < expArr.Length; i++)
            {
                if  (expArr[i]  ==  '&'  &&  i  !=  0  &&
result[result.Length - 1] == '.')
                    result = result.Remove(result.Length
- 1);
                else if (expArr[i] == '+' && i != 0 &&
result[result.Length - 1] == '&')
                {
                    result = result.Remove(result.Length
- 1);
                    result += (char)400;
                    result += expArr[i];
                }
                else
                    result += expArr[i];
            }
            return result;
        }
        /// <summary>
        /// Обработка закрытие формы. (Объект формы не
должен удаляться).
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private    void    FormBefore_FormClosing(object
sender, FormClosingEventArgs e)
        {
            if              (e.CloseReason              !=
CloseReason.ApplicationExitCall)
            {
                e.Cancel = true;
                Visible = false;
            }
```

```
        }
    }
}
```

### 1.4.4. Код класса FormInfo

```
using System.Windows.Forms;

namespace Project
{
    /// <summary>
    ///   Класс,   представляющий   окно,   содержащее
информацию о разработчике.
    /// </summary>
    public partial class FormInfo : Form
    {
        public FormInfo()
        {
            InitializeComponent();
        }
    }
}
```

### 1.4.5. Код класса FormProcessDefinitions

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace Project
{
    /// <summary>
    ///   Класс,   хранящий   определения   процессов   и
подпроцессов.
    /// </summary>
    public partial class FormProcessDefinitions : Form
    {
        /// <summary>
        /// Средняя длина подпроцессов.
        /// </summary>
        public   string   MidLength   {   get   {   return
labelAverageLength.Text; } }
        /// <summary>
        /// Содержимое таблицы - словаря определений.
        /// </summary>
        public List<string> tableContents
        {
            get
            {
```

```
                List<string>     contents     =     new
List<string>();
                for   (int   count   =   0;   count   <
dataGridViewDefinitions.RowCount; count++)
                {
                    string  str  =  $"{count  +  1}.  "  +
dataGridViewDefinitions.Rows[count].Cells[0].Value
                        +      "      =      "      +
Change(dataGridViewDefinitions.Rows[count].Cells[1].Valu
e.ToString());
                    if             (count             !=
dataGridViewDefinitions.RowCount - 1)
                        str += ";";
                    else
                        str += ".";
                    contents.Add(str);
                }
                return contents;
            }
        }
        /// <summary>
        ///  Заполнение  таблицы  словаря  определениями
процессов.
        /// </summary>
        ///         <param         name="basics">Словарь
процессов.</param>
        public FormProcessDefinitions(Dictionary<string,
string> basics)
        {
            InitializeComponent();
            panel1.Width = Width;
            dataGridViewDefinitions.Width = Width - 35;
            dataGridViewDefinitions.Height  =  Height  -
panel1.Height - 65;
            label2.Text = "";
            labelAverageLength.Text = "";
            dataGridViewDefinitions.RowCount           =
basics.Count;
            dataGridViewDefinitions.ColumnCount = 2;
            dataGridViewDefinitions.Columns[0].Name    =
"Process";
            dataGridViewDefinitions.Columns[1].Name    =
"Definition";
            int i = 0;
            foreach (string key in basics.Keys)
            {

dataGridViewDefinitions.Rows[i].Cells[0].Value = key;
```

```
dataGridViewDefinitions.Rows[i].Cells[1].Value          =
Change(basics[key]);
                i++;
            }

        }
        /// <summary>
        ///  Заполнение  таблицы  словаря  определениями
подпроцессов.
        /// </summary>
        ///          <param          name="basics">Словарь
процессов</param>
        ///          <param          name="finals">Словарь
подпроцессов.</param>
        public
FormProcessDefinitions(Dictionary<string,string> basics,
Dictionary<string,string> finals)
        {
            InitializeComponent();
            panel1.Width = Width;
            dataGridViewDefinitions.Width = Width - 35;
            dataGridViewDefinitions.Height  =  Height  -
panel1.Height - 65;
            dataGridViewDefinitions.RowCount            =
finals.Count;
            dataGridViewDefinitions.ColumnCount = 2;
            dataGridViewDefinitions.Columns[0].Name     =
"Process";
            dataGridViewDefinitions.Columns[1].Name     =
"Definition";
            Dictionary<string,  string>  contents  =  new
Dictionary<string, string>();
            foreach (string key in finals.Keys)
                contents.Add(key, Change(finals[key]));
            float count = 0;
            foreach (string key in contents.Keys)
         {
             string str = contents[key];
             char[] array = str.ToCharArray();
                    foreach (char c in array)
                    {
                        if (c != '+' && c != '.' && c !=
')' && c != '(' && c != '*')
                        count++;
                    }
            }
            try
```

```
                {
                    count /= contents.Count;
                }
                catch(DivideByZeroException)
                {
                    count = 0;
                }
                if (dataGridViewDefinitions.RowCount == 0)
                    count = 0;
                labelAverageLength.Text = count.ToString();
                int i = 0;
                foreach (string key in contents.Keys)
                {

dataGridViewDefinitions.Rows[i].Cells[0].Value = key;

dataGridViewDefinitions.Rows[i].Cells[1].Value          =
Change(finals[key]);
                    i++;
                }
            }
            /// <summary>
            /// Изменение регулярного выражения. (Удаление
символов,   добавленных   вместе   с   единым   конечным
состоянием).
            /// </summary>
            ///      <param      name="expression">Регулярное
выражение.</param>
            /// <returns></returns>
            string Change(string expression)
            {
                string result = "";
                char[] expArr = expression.ToCharArray();
                for (int i=0;i<expArr.Length;i++)
                {
                    if        (expArr[i]        ==        '&'
&&i!=0&&result[result.Length - 1] == '.')
                        result = result.Remove(result.Length
- 1);
                    else if (expArr[i] == '+' && i != 0 &&
result[result.Length - 1] == '&')
                    {
                        result = result.Remove(result.Length
- 1);
                        result += (char)400;
                        result += expArr[i];
                    }
                    else
```

| | | | | |
|---|---|---|---|---|
| Изм. | Лист | № докум. | Подп. | Дата |
| RU.17701729.502610-01 12 | | | | |
| Инв. № подл. | Подп. и дата | Взам. инв. № | Инв. № дубл. | Подп. и дата |

```
                    result += expArr[i];
                }
                return result;
            }
            /// <summary>
            /// Обработка закрытие формы. (Объект формы не
должен быть удален).
            /// </summary>
            /// <param name="sender"></param>
            /// <param name="e"></param>
            private                                    void
FormProcessDefinitions_FormClosing(object         sender,
FormClosingEventArgs e)
            {
                if              (e.CloseReason              !=
CloseReason.ApplicationExitCall)
                {
                    e.Cancel = true;
                    Visible = false;
                }
            }
            /// <summary>
            /// Обработка изменения размеров формы.
            /// </summary>
            /// <param name="sender"></param>
            /// <param name="e"></param>
            private                                    void
FormProcessDefinitions_SizeChanged(object         sender,
EventArgs e)
            {
                panel1.Width = Width;
                dataGridViewDefinitions.Width = Width-35;
                dataGridViewDefinitions.Height   =   Height-
panel1.Height-65;
            }
        }
    }
```

### 1.4.6. Код класса FormRegularExpression

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using LibGraph;

namespace Project
{
    /// <summary>
```

```
    /// Класс, представляющий окно, содержащее регулярное
выражение, построенное по пользовательскому конечному
автомату, и предоставляющее доступ к аналитическим
данным.
    /// </summary>
    public partial class FormRegularExpression : Form
    {
        /// <summary>
        /// Конечный автомат, по которому было построено
регулярное выражение.
        /// </summary>
        GraphBuilder graph;
        /// <summary>
        /// Окно, содержащее определение процессов.
        /// </summary>
        FormProcessDefinitions formBasicsDictionary;
        /// <summary>
        /// Окно, содержащее определение подпроцессов.
        /// </summary>
        FormProcessDefinitions formFinalsDictionary;
        /// <summary>
        ///   Окно,   содержащее   регуляное   выражение,
полученное по конечному автомату без использования
алгоритма минимизации регулярного выражения.
        /// </summary>
        FormBefore formBefore;
        /// <summary>
        ///   Окно,   позволяющее   пользователю   задать
параметры сохранения.
        /// </summary>
        FormSavingRules formSavingRules;
        /// <summary>
        /// Строки, хранящие регулярное выражение.
        /// </summary>
        List<string> regularExpression;
        /// <summary>
        /// Стартовое оено приложения.
        /// </summary>
        ChoiceFSMForm newProjectForm;
        /// <summary>
        ///     Свойство,     считывающее     содержимое
richTextBoxRegularExpression. (Регулярное выражение).
        /// </summary>
        List<string> textBoxContents
        {
            get
            {
```

```
                List<string>     contents     =     new
List<string>();
                foreach      (string      line      in
richTextBoxRegularExpression.Lines)
                    contents.Add(line);
                return contents;
            }
        }

        public        FormRegularExpression(List<string>
regularExpression,  GraphBuilder  graph,  ChoiceFSMForm
form)
        {
            InitializeComponent();
            newProjectForm = form;
            Height                                      =
richTextBoxRegularExpression.Location.Y                 +
richTextBoxRegularExpression.Height + 80;
            this.regularExpression = regularExpression;
            this.graph = graph;
            richTextBoxRegularExpression.Text           =
String.Empty;
            foreach      (string      expression      in
regularExpression)
                richTextBoxRegularExpression.Text+=
expression;
            char[]               charArr                =
richTextBoxRegularExpression.Text.ToCharArray();
            int count = 0;
            foreach (char c in charArr)
            {
                if (c != '+' && c != '.' && c != ')' && c
!= '(' && c != '*' && !(c >= 1 && c <= 9))
                    count++;
            }
            labelNoDic.Text = count.ToString();
            formBasicsDictionary          =          new
FormProcessDefinitions(graph.basicDefinitions,
graph.finalDefinitions);
            formFinalsDictionary          =          new
FormProcessDefinitions(graph.basicDefinitions);
            formBefore                =                new
FormBefore(regularExpression, graph.finalDefinitions);
            formSavingRules = new FormSavingRules(this);
        }
        /// <summary>
        ///  Метод,  открывающий  окно  с  определениями
подпроцессов.
```

```
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void buttonDefinitions_Click(object
sender, EventArgs e)
        {
            formBasicsDictionary.Visible=true;
        }
        /// <summary>
        /// Метод, открывающий окно с определениями
процессов.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void buttonBasics_Click(object sender,
EventArgs e)
        {
            formFinalsDictionary.Visible=true;
        }
        /// <summary>
        /// Метод, открывающий окно с регуляным
выражением, полученным по конечному автомату без
использования алгоритма минимизации регулярного
выражения.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void buttonBefore_Click(object sender,
EventArgs e)
        {
            formBefore.Visible=true;
        }
        /// <summary>
        /// Метод, обрабатывающий закрытие формы.
Предупреждает о возможной потери данных.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private                                    void
FormRegularExpression_FormClosing(object          sender,
FormClosingEventArgs e)
        {
            UsersFiles file = new UsersFiles();
            DialogResult result;
            if              (e.CloseReason              !=
CloseReason.ApplicationExitCall)
            {
```

```
                result = MessageBox.Show("Do you want to
quit the application?\nCurrent progress will be lost...",
"Ensure        quitting        the        application",
MessageBoxButtons.YesNo);
                if (result == DialogResult.No)
                    e.Cancel = true;
                else
                {
                    file.DeleteFiles();
                    Application.Exit();
                }
            }
        }
        /// <summary>
        ///  Метод, открывающий окно с информацией о
разработчике.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void aboutToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            FormInfo formInfo = new FormInfo();
            formInfo.ShowDialog();
        }
        /// <summary>
        /// Метод, открывающий стартовое окно приложения.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private                                    void
newProjectToolStripMenuItem_Click(object          sender,
EventArgs e)
        {
            UsersFiles file = new UsersFiles();
            file.DeleteFiles();
            DialogResult          dialogResult          =
DialogResult.Cancel;
            dialogResult = MessageBox.Show("Are you sure,
that you want to start a new project? Current progress
will   be   lost.",   "Ensure   starting   new   project",
MessageBoxButtons.OKCancel);
            if (dialogResult == DialogResult.OK)
            {
                newProjectForm.Show();
                Hide();
            }
        }
```

```
        /// <summary>
        /// Метод, обрабатывающий желание пользователя
сохранить данные, полученные в результате работы
алгоритма, и запускающий выбор параметров сохранения.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void saveToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            formSavingRules = new FormSavingRules(this);
            formSavingRules.Show();
        }
        /// <summary>
        /// Метод, завершающий сохранение данных в
зависимости от выбранных пользователем параметров
сохранения.
        /// </summary>
        public void CompleteSaving()
        {
            if (formSavingRules.SavingCompleted)
            {
                List<string>                textAfter=new
List<string>();
                List<string>    textBefore    =    new
List<string>();
                List<string>    textBasics    =    new
List<string>();
                List<string>   textProcessDef   =   new
List<string>();
                List<string>   textAnalytics   =   new
List<string>();
                if (formSavingRules.AfterAlgo)
                {
                    textAfter.Add("Expression      mined
after applying the algorithm: ");
                    foreach     (string     line     in
textBoxContents)
                        textAfter.Add(line);
                    textAfter.Add(Environment.NewLine);
                }
                if(formSavingRules.BeforeAlgo)
                {
                    textBefore.Add("Expression     mined
without applying the algorithm: ");
                    foreach     (string     line     in
formBefore.textBoxContents)
                        textBefore.Add(line);
```

```
                    textBefore.Add(Environment.NewLine
);
                }
                if(formSavingRules.BasicDefinitions)
                {
                    textBasics.Add("Basic   definitions:
");
                    foreach     (string     line     in
formBasicsDictionary.tableContents)
                        textBasics.Add(line);
                    textBasics.Add(Environment.NewLine);
                }
                if (formSavingRules.ProcessDefinitions)
                {
                    textProcessDef.Add("Basic
definitions: "+Environment.NewLine);
                    foreach     (string     line     in
formFinalsDictionary.tableContents)
                        textProcessDef.Add(line);

textProcessDef.Add(Environment.NewLine );
                }
                if(formSavingRules.Analytics)
                {
                    textAnalytics.Add("Analysis  of  the
effectiveness          of          the          applied
algorithm:"+Environment.NewLine);
                    textAnalytics.Add($"Number of events
after applying the algorithm: " + labelNoDic.Text+".");
                    textAnalytics.Add($"Number of events
before applying the algorithm: "+formBefore.BeforeCount +
".");
                    textAnalytics.Add("Maximum  nesting:
" + formBefore.MaxNesting + ".");
                    textAnalytics.Add("Average length of
the subprocesses: " + formBasicsDictionary.MidLength +
".");
                }
                UsersFiles file = new UsersFiles();
                file.SaveMinedExpression(textAfter,
textBefore, textBasics, textProcessDef, textAnalytics);
            }
            else
                MessageBox.Show("Error!  You  have  not
chosen information to save...");
        }
    }
}
```

### 1.4.7. Код класса FormRemoveStateWithAnimation

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;
using DeletingAlgo;
using LibGraph;
using System.IO;

namespace Project
{
    /// <summary>
    ///    Класс,    представляющий    окно,    являющееся
графической реализацией одного шага анимации к методу
последовательного удаления состояний.
    /// </summary>
    public partial class FormRemoveStatesWithAnimation :
Form
    {
        /// <summary>
        /// Поле - индикатор, было ли добавлено единое
конечное состояние.
        /// </summary>
        bool transformed = false;
        /// <summary>
        /// Поле, хранящее номер открытого шага анимации.
        /// </summary>
        int picShown;
        /// <summary>
        /// Поле, хранящее номер последнего проделанного
шага анимации.
        /// </summary>
        int picsMade;
        /// <summary>
        /// Поле, хранящее текущее состояние автомата.
        /// </summary>
        GraphBuilder graph;
        /// <summary>
        /// Поле, хранящее начальное состояние автомата.
        /// </summary>
        string headState;
        /// <summary>
        ///  Поле,  хранящее  путь  к  изображению  с
иллюстрацией шага анимации.
        /// </summary>
```

```
        string imagePath;
        /// <summary>
        ///  Поле, хранящее имя единого конечного
состояния.
        /// </summary>
        string unqueFinal=String.Empty;
        /// <summary>
        /// Поле, хранящее номер шага анимации, который
будет присвоен.
        /// </summary>
        int count = 1;
        /// <summary>
        ///  Поле, представляющее класс с реализацией
алгоритма удаления состояний.
        /// </summary>
        StateRemoval state;
        /// <summary>
        /// Стартовое окно приложения.
        /// </summary>
        ChoiceFSMForm newProjectForm;
        /// <summary>
        ///  Поле, хранящее список состояний, которые
нельзя удалять (начальное и конечные).
        /// </summary>
        List<string> untouchedStatesList;
        public
FormRemoveStatesWithAnimation(GraphBuilder        graph,
List<string> list, string headState, string imagePath,
ChoiceFSMForm form)
        {
            InitializeComponent();
            transformed = false;
            newProjectForm = form;
            buttonPreviousGraph.Enabled = false;

buttonRemoveNext.Size=buttonRemoveNext.Image.Size;
            buttonPreviousGraph.Size                    =
buttonPreviousGraph.Image.Size;
            buttonSkip.Size = buttonSkip.Image.Size;
            picShown = picsMade = -1;
            List<string> list1 = new List<string>();
            foreach (string elem in list)
                if  (elem  !=  String.Empty  &&  elem  !=
headState)
                    list1.Add(elem);
            if(list1.Count==1)
            {
                transformed = true;
```

```
                unqueFinal = list1.First();
            }
            untouchedStatesList = list;
            state = new StateRemoval(graph, list,
headState,unqueFinal);
            this.graph = graph;
            this.headState = headState;
            pictureBoxGraph.Image = Image.FromStream(new
MemoryStream(File.ReadAllBytes(imagePath)));
            pictureBoxGraph.SizeMode =
PictureBoxSizeMode.Zoom;
        }
        /// <summary>
        /// Метод, открывающий следующий шаг анимации.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void buttonRemoveNext_Click(object
sender, EventArgs e)
        {
            buttonPreviousGraph.Enabled = true;
            if (picsMade != picShown)
            {
                if (picShown == -1)
                    picShown = 1;
                else
                picShown++;
                imagePath = String.Format("Graphs" +
Path.DirectorySeparatorChar.ToString() +
"graphviz{0}.png", picShown);
                pictureBoxGraph.Image =
Image.FromStream(new
MemoryStream(File.ReadAllBytes(imagePath)));
            }
            else
            {
                string stateName;
                if (state.NonFinalLeft(out stateName) &&
!transformed)//если остались недопускающие состояния и не
было добавлено единое конечное состояние,

//запускаем алгоритм удаления недопускающего состояния
                {

state.RemoveNonFinalState(stateName);
                    imagePath = graph.MakeDot(count);
```

```
                    pictureBoxGraph.Image                =
Image.FromStream(new
MemoryStream(File.ReadAllBytes(imagePath)));
                    pictureBoxGraph.SizeMode             =
PictureBoxSizeMode.Zoom;
                    if (picsMade == -1)
                        picsMade = picShown = 1;
                    else
                    {
                        picsMade++;
                        picShown++;
                    }
                    count++;
                }
                else  if  (!transformed)//если  остались
только допускающие состояния и не было добавлено единое
конечное состояние,
                                            //добавляем его
                {
                    state.TransformGraph(out
unqueFinal);
                    transformed = true;
                    untouchedStatesList      =      new
List<string> { String.Empty, headState, unqueFinal };
                    state   =   new   StateRemoval(graph,
untouchedStatesList, headState, unqueFinal);
                    imagePath = graph.MakeDot(count);
                    pictureBoxGraph.Image                =
Image.FromStream(new
MemoryStream(File.ReadAllBytes(imagePath)));
                    pictureBoxGraph.SizeMode             =
PictureBoxSizeMode.Zoom;
                    if (picsMade == -1)
                        picsMade = picShown = 1;
                    else
                    {
                        picsMade++;
                        picShown++;
                    }
                    count++;
                }
                else//было  добавлено  единое  конечное
состояние
                {
                    if            (state.NonFinalLeft(out
stateName))//если  остались  недопускающие  состояния,
удаляем их
                    {
```

```
state.RemoveNonFinalState(stateName);
                        imagePath                       =
graph.MakeDot(count);
                        pictureBoxGraph.Image           =
Image.FromStream(new
MemoryStream(File.ReadAllBytes(imagePath)));
                        pictureBoxGraph.SizeMode        =
PictureBoxSizeMode.Zoom;
                        if (picsMade == -1)
                            picsMade = picShown = 1;
                        else
                        {
                            picsMade++;
                            picShown++;
                        }
                        count++;
                    }
                    else//если осталось только единое
конечное состояние и начальное состояние, строим
регулярное выражение
                    {
                        List<string> regularExpression;
                        regularExpression               =
state.FinalRemoval(graph);
                        FormRegularExpression form = new
FormRegularExpression(regularExpression,
graph,newProjectForm);
                        pictureBoxGraph.SizeMode        =
PictureBoxSizeMode.Zoom;
                        if (picsMade == -1)
                            picsMade = picShown = 1;
                        else
                        {
                            picsMade++;
                            picShown++;
                        }
                        form.Show();
                        Hide();
                    }
                }
            }
        }
        /// <summary>
        /// Метод, открывающий предыдущий шаг анимации.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
```

```
        private void buttonPreviousGraph_Click(object
sender, EventArgs e)
        {
            if (picShown == 1)
            {
                picShown = -1;
                buttonPreviousGraph.Enabled = false;
            }
            else
                picShown--;
            imagePath                                    =
String.Format("Graphs"+Path.DirectorySeparatorChar.ToStr
ing()+"graphviz{0}.png", picShown);
            pictureBoxGraph.Image = Image.FromStream(new
MemoryStream(File.ReadAllBytes(imagePath)));
        }
        /// <summary>
        /// Метод, открывающий стартовое окно приложения.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private                                        void
newProjectToolStripMenuItem_Click(object          sender,
EventArgs e)
        {
            UsersFiles file = new UsersFiles();
            file.DeleteFiles();
            DialogResult          dialogResult          =
DialogResult.Cancel;
            dialogResult = MessageBox.Show("Are you sure,
that you want to start a new project? Current progress
will   be   lost.",   "Ensure   starting   new   project",
MessageBoxButtons.OKCancel);
            if (dialogResult == DialogResult.OK)
            {
                newProjectForm.Show();
                Hide();
            }
        }
        /// <summary>
        ///  Метод,  пропускающий  все  шаги  анимации,
начиная с текущего, и открывающий окно с регулярным
выражением.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private  void  buttonSkip_Click(object  sender,
EventArgs e)
```

```
        {
            FormUsersFSM form = new FormUsersFSM(graph,
untouchedStatesList, headState, true, newProjectForm);
            Hide();
        }
        /// <summary>
        ///   Метод,   обрабатывающий   закрытие   формы.
Предупреждает о возможной потери данных.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private                                        void
FormRemoveStatesWithAnimation_FormClosing(object sender,
FormClosingEventArgs e)
        {
            DialogResult result;
            if              (e.CloseReason              !=
CloseReason.ApplicationExitCall)
            {
                result = MessageBox.Show("Do you want to
quit the application?\nCurrent progress will be lost...",
"Ensure        quitting        the        application",
MessageBoxButtons.YesNo);
                if (result == DialogResult.No)
                    e.Cancel = true;
                else
                    Application.Exit();
            }
        }
        /// <summary>
        /// Метод, сохраняющий картинку с иллюстрацией к
текущему шагу анимации.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private  void  saveToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            UsersFiles file = new UsersFiles();
            string path = file.ChooseSavingPath("png");
            if (path == string.Empty)
                MessageBox.Show("Error!   File   was   not
saved!");
            else
                file.SaveImage(path,
(Bitmap)pictureBoxGraph.Image);
        }
        /// <summary>
```

```
        /// Метод, открывающий окно с информацией о
разработчике.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void aboutToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            FormInfo formInfo = new FormInfo();
            formInfo.ShowDialog();
        }
    }
}
```

### 1.4.8. Код класса FormSavingRules

```
using System;
using System.Windows.Forms;

namespace Project
{
    /// <summary>
    ///   Класс,   представляющий   окно,   позволяющее
пользователю задать параметры сохранения.
    /// </summary>
    public partial class FormSavingRules : Form
    {
        /// <summary>
        /// Поле - индикатор, желает ли пользователь
сохранить регулярное выражение, полученное с применением
алгоритма минимизации.
        /// </summary>
        bool afterAlgo;
        /// <summary>
        /// Поле - индикатор, желает ли пользователь
сохранить регулярное выражение, полученное без применения
алгоритма минимизации.
        /// </summary>
        bool beforeAlgo;
        /// <summary>
        /// Поле - индикатор, желает ли пользователь
сохранить определения процессов.
        /// </summary>
        bool basicDefinitions;
        /// <summary>
        /// Поле - индикатор, желает ли пользователь
сохранить определения подпроцессов.
```

```
        /// </summary>
        bool processDefinitions;
        /// <summary>
        /// Поле - индикатор, желает ли пользователь
сохранить аналитические данные.
        /// </summary>
        bool analytics;
        /// <summary>
        /// Поле - индикатор, было ли завершено
сохранение.
        /// </summary>
        bool savingCompleted;
        /// <summary>
        /// Свойство, делающее поле afterAlgo доступным
для чтения.
        /// </summary>
        public bool AfterAlgo { get { return afterAlgo; }
}
        /// <summary>
        /// Свойство, делающее поле beforeAlgo доступным
для чтения.
        /// </summary>
        public bool BeforeAlgo { get { return beforeAlgo;
} }
        /// <summary>
        /// Свойство, делающее поле basicDefinitions
доступным для чтения.
        /// </summary>
        public bool BasicDefinitions { get { return
basicDefinitions; } }
        /// <summary>
        /// Свойство, делающее поле processDefinitions
доступным для чтения.
        /// </summary>
        public bool ProcessDefinitions { get { return
processDefinitions; } }
        /// <summary>
        /// Свойство, делающее поле analytics доступным
для чтения.
        /// </summary>
        public bool Analytics { get { return analytics; }
}
        /// <summary>
        /// Свойство, делающее поле savingCompleted
доступным для чтения.
        /// </summary>
        public bool SavingCompleted { get { return
savingCompleted; } }
```

```
        /// <summary>
        ///  Окно, содержащее регулярное выражение,
полученное из конечного автомата.
        /// </summary>
        FormRegularExpression formRE;

        public     FormSavingRules(FormRegularExpression
form)
        {
            InitializeComponent();
            formRE = form;
        }
        /// <summary>
        /// Метод, вызываемый при желании пользователя
сохранить некоторые из доступных аналитических данных.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private    void    buttonSaveChosen_Click(object
sender, EventArgs e)
        {
            MarkChecked();
        }
        /// <summary>
        /// Метод, вызываемый при желании пользователя
сохранить абсолютно все аналитические данные.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private  void  buttonAddAll_Click(object  sender,
EventArgs e)
        {
            checkBoxAfterAlgo.Checked = true;
            checkBoxAnalytics.Checked = true;
            checkBoxBasics.Checked = true;
            checkBoxBeforeAlgo.Checked = true;
            checkBoxProcessDefinitions.Checked = true;
            MarkChecked();
        }
        /// <summary>
        ///  Метод, задающий параметры сохранения в
зависимости от выбранных пользователем характеристик, и
запускающий процесс сохранения.
        /// </summary>
        void MarkChecked()
        {
             afterAlgo= checkBoxAfterAlgo.Checked;
             beforeAlgo= checkBoxBeforeAlgo.Checked;
```

```
            basicDefinitions= checkBoxBasics.Checked;
            processDefinitions=
checkBoxProcessDefinitions.Checked;
            analytics= checkBoxAnalytics.Checked;
           if    (afterAlgo    ||    beforeAlgo    ||
basicDefinitions || processDefinitions || analytics)
               savingCompleted = true;
           formRE.CompleteSaving();
           Hide();
        }
    }
}
```

### 1.4.9. Код класса FormTransitionName

```
using System;
using System.Windows.Forms;

namespace Project
{
    /// <summary>
    /// Класс, позволяющий ввести регулярное выражение,
задающее переход.
    /// </summary>
    public partial class FormTransitionName : Form
    {
        /// <summary>
        /// Окно отрисовки.
        /// </summary>
        Draw form;
        /// <summary>
        ///  Введенное  регулярное  выражение,  задающее
переход.
        /// </summary>
        string transition;
        public FormTransitionName(Draw form)
        {
            InitializeComponent();
            this.form = form;
            textBoxMain.Select();
            transition = "";
        }

        /// <summary>
        /// Добавляет переход перед закрытием формы.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
```

```
        private                                          void
FormTransitionName_FormClosed(object                sender,
FormClosedEventArgs e)
        {

form.CompleteAddingEdge(transition);//добавляем  переход
или даем сигнал, что пользователь не завершил добавление
перехода
        }
        /// <summary>
        /// Проверяет корректность введенного перехода.
        /// </summary>
        /// <param name="input">ввод пользователя</param>
        /// <returns></returns>
        bool CheckInput(string input)
        {
            char[] arr = input.ToCharArray();
            foreach (char c in arr)
                if (!(c >= 'a' && c <= 'z' || c >= 'A' &&
c <= 'Z' || c >= 'а' && c <= 'я' || c >= 'А' || c <= 'Я'))
                    return false;
            return true;
        }
        /// <summary>
        ///   Обрабатывает   запрос   пользователя   на
добавление перехода.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private   void   buttonAdd_Click(object   sender,
EventArgs e)
        {
            if (textBoxMain.Text == "")
                MessageBox.Show("Error!   Please,   enter
transition...");
            else if (CheckInput(textBoxMain.Text))
            {
                transition = textBoxMain.Text;
                Close();
            }
            else
                MessageBox.Show("Error!          Invalid
transition value...");
        }
    }
}
```

### 1.4.10. Код класса FormUsersFSM

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;
using LibGraph;
using DeletingAlgo;
using System.IO;

namespace Project
{
    /// <summary>
    /// Класс, представляющий окно с возможностью начать построение регулярного выражения с анимацией и без нее.
    /// </summary>
    public partial class FormUsersFSM : Form
    {
        /// <summary>
        ///  Конечный автомат, с которым необходимо работать.
        /// </summary>
        GraphBuilder graph;
        /// <summary>
        /// Список из состояний, которые нельзя удалять на начальном этапе.
        /// </summary>
        List<string> untouchedStatesList = new List<string>();
        /// <summary>
        /// Путь к изображению графа.
        /// </summary>
        string imagePath;
        /// <summary>
        /// Начальное состояние автомата.
        /// </summary>
        string headState;
        /// <summary>
        /// Индикатор, была ли успешно начата работа с графом.
        /// </summary>
        bool success;
        /// <summary>
        /// Стартовое окно приложения.
        /// </summary>
        ChoiceFSMForm formChoice;
```

```
        /// <summary>
        ///  Свойство, предоставляющее доступ к полю
success.
        /// </summary>
        public bool Success { get { return success; } }
        /// <summary>
        /// Конструктор вызывается из формы с анимацией,
чтобы пропустить дальнейшие шаги анимации.
        /// </summary>
        ///   <param   name="graph">Текущее   состояние
графа.</param>
        ///  <param  name="finalStates">Список  конечных
состояний.</param>
        ///       <param       name="headState">Начальное
состояние.</param>
        ///     <param     name="skip">Индикатор,     что
пропускаются шаги анимации.</param>
        ///     <param     name="form">Стартовое     окно
приложения.</param>
        public      FormUsersFSM(GraphBuilder      graph,
List<string> finalStates, string headState, bool skip,
ChoiceFSMForm form)
        {
            formChoice = form;
            this.graph = graph;
            untouchedStatesList = finalStates;
            this.headState = headState;
            buttonConstructExpression_Click(new
object(), new EventArgs());
        }
        /// <summary>
        ///  Конструктор вызывается, когда пользователь
только начал работу с автоматом.
        /// </summary>
        ///   <param   name="graph">Текущее   состояние
графа.</param>
        ///  <param  name="finalStates">Список  конечных
состояний.</param>
        ///       <param       name="headState">Начальное
состояние.</param>
        ///     <param     name="form">Стартовое     окно
приложения.</param>
        public      FormUsersFSM(GraphBuilder      graph,
List<string>  finalStates,string  headState,ChoiceFSMForm
form)
        {
            InitializeComponent();
```

```
            Height = pictureBoxUsersFSM.Location.Y +
pictureBoxUsersFSM.Height + 80;
            formChoice = form;
            graph.InitializeFinalStates(finalStates);
            imagePath = graph.MakeDot(0);
            pictureBoxUsersFSM.Image =
Image.FromStream(new
MemoryStream(File.ReadAllBytes(imagePath)));
            pictureBoxUsersFSM.SizeMode =
PictureBoxSizeMode.Zoom;
            MinimizeGraph(graph, finalStates,
headState);
        }
        /// <summary>
        /// Метод, запускающий минимизацию автомата.
        /// </summary>
        ///     <param name="graph">Минимизируемый
автомат.</param>
        /// <param name="finalStates">Конечные состония
автомата.</param>
        /// <param name="headState">Начальное состояние
автомата.</param>
        private void MinimizeGraph(GraphBuilder graph,
List<string> finalStates, string headState)
        {

            Minimizator minimizator = new
Minimizator(finalStates, headState, graph);
            this.graph = minimizator.Minimize();
            if (this.graph != null)
            {
                List<string> verticesToRemove = new
List<string>();
                foreach (var vertex in
this.graph.myGraph.Vertices)
                {
                    bool found = false;
                    foreach (var edge in
this.graph.myGraph.Edges)
                        if (edge.Target == vertex)
                        {
                            found = true;
                            break;
                        }
                    if (!found)
                        verticesToRemove.Add(vertex);
                }
                if (verticesToRemove.Count != 0)
```

```
                    foreach     (string     vertex     in
verticesToRemove)

this.graph.myGraph.RemoveVertex(vertex);
                this.headState                          =
minimizator.StartingState;
                foreach      (string      state      in
this.graph.FinalStates)
                    untouchedStatesList.Add(state);
                untouchedStatesList.Add(this.headState);
                untouchedStatesList.Add(String.Empty);
                this.graph.UserAddVertix("");
                this.graph.UserAddEdge("",
this.headState, "");
                foreach        (var        edge        in
this.graph.myGraph.Edges)
                {
                    if
(!this.graph.basicDefinitions.Values.Contains(edge.Tag)
&& edge.Tag != String.Empty)
                    {
                        int             num             =
this.graph.basicDefinitions.Keys.Count;
                        string str = ((char)(num % 26 +
97)).ToString();
                        if (num >= 26)
                            str += num / 26;

this.graph.basicDefinitions.Add(str, edge.Tag);
                    }
                    foreach     (var     element     in
this.graph.basicDefinitions)
                        if (element.Value == edge.Tag)
                            edge.Tag = element.Key;
                }
                imagePath = this.graph.MakeDot(-1);
                success = true;
            }
        }
        /// <summary>
        ///  Метод, запускающий построение регулярного
выражения с анимацией при нажатии соответствующей кнопки.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private                                     void
buttonConstructWithAnimation_Click(object         sender,
EventArgs e)
```

```
        {
            FormRemoveStatesWithAnimation form = new
FormRemoveStatesWithAnimation(graph,untouchedStatesList,
headState, imagePath, formChoice);
            form.Show();
            Hide();
        }
        /// <summary>
        /// Метод, запускающий построение регулярного
выражения без анимации при нажатии соответствующей
кнопки.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private                                    void
buttonConstructExpression_Click(object sender, EventArgs
e)
        {

            List<string>       regularExpression       =
ConstructRE();
            FormRegularExpression     form     =     new
FormRegularExpression(regularExpression,
graph,formChoice);
            form.Show();
            Hide();
        }
        /// <summary>
        /// Метод, производящий построение регулярного
выражения по конечному автомату без предоставления
анимации.
        /// </summary>
        /// <returns>Регулярное выражение.</returns>
        private List<string> ConstructRE()
        {
            StateRemoval state = new StateRemoval(graph,
untouchedStatesList, headState, String.Empty);
            string stateName;
            string  unqueFinal  =  String.Empty;//единое
конечное состояние
            List<string>      listFinals      =      new
List<string>();//список всех допускающих состояний
            foreach (string elem in untouchedStatesList)
                if  (elem  !=  String.Empty  &&  elem  !=
headState)
                    listFinals.Add(elem);
```

```
            if (listFinals.Count == 1)//если допускающее
состояние единственно, оно становится единым конечным
состоянием
                unqueFinal = listFinals.First();
            if  (state.NonFinalLeft(out  stateName)  &&
unqueFinal  ==  String.Empty)//удаляем все недопускающие
состояния
                do
                {

state.RemoveNonFinalState(stateName);
                }     while     (state.NonFinalLeft(out
stateName));
            if (unqueFinal == String.Empty)//определяем
единое конечное состояние, если оно не было определено
                state.TransformGraph(out unqueFinal);
            untouchedStatesList  =  new  List<string>  {
String.Empty, headState, unqueFinal };//формируем новый
список неудаляемых состояний
            state     =     new     StateRemoval(graph,
untouchedStatesList, headState, unqueFinal);
            if                    (state.NonFinalLeft(out
stateName))//удаляем все недопускающие состояния
                do
                {

state.RemoveNonFinalState(stateName);
                }     while     (state.NonFinalLeft(out
stateName));
            return    state.FinalRemoval(graph);//строим
регулярное выражения
        }
        /// <summary>
        /// Метод, открывающий стартовое окно приложения.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private                                      void
newProjectToolStripMenuItem_Click(object         sender,
EventArgs e)
        {
            UsersFiles file = new UsersFiles();
            file.DeleteFiles();
            DialogResult         dialogResult         =
DialogResult.Cancel;
            dialogResult = MessageBox.Show("Are you sure,
that you want to start a new project? Current progress
```

```
will be lost.", "Ensure starting new project",
MessageBoxButtons.OKCancel);
            if (dialogResult == DialogResult.OK)
            {
                formChoice.Show();
                Hide();
            }
        }
        /// <summary>
        ///   Метод, обрабатывающий закрытие формы.
Предупреждает о возможной потери данных.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void FormUsersFSM_FormClosing(object
sender, FormClosingEventArgs e)
        {
            DialogResult result;
            if (e.CloseReason !=
CloseReason.ApplicationExitCall)
            {
                result = MessageBox.Show("Do you want to
quit the application?\nCurrent progress will be lost...",
"Ensure quitting the application",
MessageBoxButtons.YesNo);
                if (result == DialogResult.No)
                    e.Cancel = true;
                else
                    Application.Exit();
            }
        }
        /// <summary>
        ///  Метод, открывающий окно с информацией о
разработчике.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void aboutToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            FormInfo formInfo = new FormInfo();
            formInfo.ShowDialog();
        }
    }
}
```

### 1.4.11. Код класса Minimizator

```
using System;
using System.Collections.Generic;
using System.Linq;
using LibGraph;
using System.Windows.Forms;

namespace Project
{
    /// <summary>
    /// Класс, хранящий символ перехода и состояние назначения.
    /// </summary>
    class OutRange
    {
        /// <summary>
        /// Поле, хранящее символ перехода.
        /// </summary>
        string transition;
        /// <summary>
        /// Поле, хранящее состояние в которое осуществляется переход.
        /// </summary>
        string distinationState;
        /// <summary>
        /// Свойство, делающее поле transition доступным для чтения.
        /// </summary>
        public string Transition { get { return transition; } }
        /// <summary>
        /// Свойство, делающее поле distinationState доступным для чтения.
        /// </summary>
        public string DictinationState { get { return distinationState; } }
        public OutRange(string symb, string state)
        {
            transition = symb;
            distinationState = state;
        }
    }
    /// <summary>
    /// Класс, представляющий сплиттер и хранящий символ разбиения и номер класса эквивалентности.
    /// </summary>
    class Splitter
```

```
    {
        /// <summary>
        ///  Поле, хранящее символ, по которому идет
разбиение.
        /// </summary>
        string symb;
        /// <summary>
        /// Поле, хранящее номер класса эквивалентности.
        /// </summary>
        int equivalenceClass;
        /// <summary>
        /// Свойство, делающее поле symb доступным для
чтения.
        /// </summary>
        public string Symbol
        {
            get { return symb; }
        }
        /// <summary>
        ///  Свойство, делающее поле equivalenceClass
доступным для чтения.
        /// </summary>
        public int EquivalenceClass
        {
            get { return equivalenceClass; }
        }
        public Splitter(string symbol, int classNum)
        {
            symb = symbol;
            equivalenceClass = classNum;
        }
    }
    /// <summary>
    ///   Класс,   выполняющий   минимизацию   конечного
автомата.
    /// </summary>
    class Minimizator
    {
        /// <summary>
        /// Поле, хранящее словарь исходящих дуг, ключом
является состояние, значением - список дуг.
        /// </summary>
        Dictionary<string,               List<OutRange>>
StateInfoDictionary     =     new     Dictionary<string,
List<OutRange>>();
        /// <summary>
```

```
        /// Поле, хранящее словарь классов эквивалентности, ключом является класс, значением - список состояний.
        /// </summary>
        Dictionary<int, List<string>> classMembers = new Dictionary<int, List<string>>();
        /// <summary>
        /// Поле, хранящее список всех возможных переходов (алфавит).
        /// </summary>
        List<string> possibleTransitions = new List<string>();
        const string trap = "trap";
        /// <summary>
        /// Поле, представляющее очередь из пар типа (символ перехода) - (класс).
        /// </summary>
        Queue<Splitter> queue = new Queue<Splitter>();
        /// <summary>
        /// Поле, хранящее первоначальный список допускающих состояний.
        /// </summary>
        List<string> acceptStates;
        /// <summary>
        /// Поле, хранящее начальное состояние.
        /// </summary>
        string startingState;
        /// <summary>
        /// Свойство доступа к полю startingState.
        /// </summary>
        public string StartingState { get; private set; }
        public Minimizator(List<string> finalStates, string startingState, GraphBuilder graph)
        {
            this.startingState = startingState;
            acceptStates = finalStates;
            foreach (var edge in graph.myGraph.Edges)
                if (!possibleTransitions.Contains(edge.Tag)&&edge.Tag!=String.Empty)
                    possibleTransitions.Add(edge.Tag);
            List<OutRange> list = new List<OutRange>();
            foreach (string transition in possibleTransitions)//добавляем в ловушку переходы по всем символам из нее самой
                list.Add(new OutRange(transition, trap));
```

```
            StateInfoDictionary.Add(trap,
list);//добавляем информацию о ловушке
            classMembers.Add(1,                          new
List<string>());//инициализируем  два  начальных  класса
эквивалентности
            classMembers.Add(2, new List<string>());
            classMembers[1].Add(trap);
            foreach       (string       node       in
graph.myGraph.Vertices)//заполняем    информацию    об
исходящих дугах для каждого состояния
                if(node!=String.Empty)
                {
                    list = new List<OutRange>();//список
дуг
                    List<string> usedTransitions = new
List<string>();//использованные символы
                    foreach(var         edge         in
graph.myGraph.OutEdges(node))
                    {
                        list.Add(new  OutRange(edge.Tag,
edge.Target));
                        if
(!usedTransitions.Contains(edge.Tag))

usedTransitions.Add(edge.Tag);
                    }
                    foreach   (string   transition   in
possibleTransitions)//по    каждому    неиспользованному
символу добавляем переход в ловушку
                        if
(!usedTransitions.Contains(transition))
                            list.Add(new
OutRange(transition, trap));
                    if (finalStates.Contains(node))
                    {
                        StateInfoDictionary.Add(node,
list);//доступ по состоянию к списку дуг
                        classMembers[2].Add(node);//если
состояние конечное, принадлежит ко 2 классу
                    }
                    else
                    {
                        StateInfoDictionary.Add(node,
list);
                        classMembers[1].Add(node);//если
состояние конечное, принадлежит к 1 классу
                    }
                }
```

```
            foreach (string symbol in
possibleTransitions)//добавляем в очередь все пары
(символ перехода, класс)
            {
                queue.Enqueue(new Splitter(symbol, 1));
                queue.Enqueue(new Splitter(symbol, 2));
            }
        }
        /// <summary>
        /// Метод, который находит минимальное свободное
число для номера класса разбиения.
        /// </summary>
        /// <returns>имя класс</returns>
        int FindFreeClassName()
        {
            int count = 0;
            foreach (var member in queue)
                if (member.EquivalenceClass > count)
                    count = member.EquivalenceClass;
            return count+1;
        }
        /// <summary>
        ///     Метод,     разбивающий     один     класс
эквивалентности.
        /// </summary>
        ///    <param    name="key">Номер    азбиваемого
класса.</param>
        ///          <param          name="symbol">Символ
разбиения.</param>
        ///                                          <param
name="equivalenceClass">Сплиттер.</param>
        bool   Split(int   key,   string   symbol,   int
equivalenceClass)
        {
            int   createdClass   =   0;//номер   созданного
класса
            bool broken = false;//разбилось ли множество
на подмножества по символу
            List<string>       r1_inSplitter;//множество
состояний, переходящих по данному символу в сплиттер
            List<string>           r2_Others;//множество
состояний,не переходящих по данному символу в сплиттер
            bool  goesToSplitter;//переходит  ли  данное
состояние по данному символу в сплиттер
            r1_inSplitter = new List<string>();
            r2_Others = new List<string>();
```

```
            foreach (string member in
classMembers[key])//для каждого состояния из данного
класса
            {
                goesToSplitter = false;
                foreach (OutRange edge in
StateInfoDictionary[member])//если дуга по символу из
данного состояния переходит в сплиттер, добавляем
состояние в один класс, иначе - в другой
                    if (edge.Transition == symbol)
                    {
                        foreach (string node in
classMembers[equivalenceClass])
                            if (edge.DictinationState ==
node)
                            {
                                goesToSplitter = true;
                                break;
                            }
                        if (goesToSplitter)
                            r1_inSplitter.Add(member);
                        else
                            r2_Others.Add(member);
                    }
            }
            if (r1_inSplitter.Count != 0 &&
r2_Others.Count != 0)//если множество разбилось на
подмножества, добавляем новый класс, изменяем текущий
            {
                broken = true;
                int num = classMembers[key].Count;
                for (int index = 0; index < num;
index++)//удаляем из текущего класса состояния, не
переходящие в сплиттер
                {
                    foreach (string state in
classMembers[key])
                        if (r2_Others.Contains(state))
                        {

classMembers[key].Remove(state);
                            break;
                        }
                }
                classMembers.Add(createdClass =
FindFreeClassName(), r2_Others);//добавляем новый класс с
состояниями, не переходящими в сплиттер
```

```
                List<string> lostTransitions = new
List<string>();
                foreach (string transition in
possibleTransitions)//какими переходами нужно дополнить
разбитый класс
                    lostTransitions.Add(transition);
                foreach(var element in queue)
                    if (element.EquivalenceClass == key)

lostTransitions.Remove(element.Symbol);
                    foreach (string symb in
possibleTransitions)
                        queue.Enqueue(new Splitter(symb,
createdClass));//добавляем в очередь пары из всех
возможных символов и нового класса
                foreach (string symb in
possibleTransitions)
                    if
(lostTransitions.Contains(symb))//дополняем очередь
парами так, чтобы были пары из всех символов и измененного
класса
                    queue.Enqueue(new Splitter(symb,
key));
            }
            return broken;
        }
        /// <summary>
        ///  Метод, реализующий алгоритм разбиения
множества состояний на минимальные подмножества.
        /// </summary>
        void Divide()
        {
            string symbol;//символ перехода
            int equivalenceClass;//номер класса
эквивалентности
            while (queue.Count!=0)//пока в очереди
остались пары символ-класс, делим классы
            {
                symbol = queue.First().Symbol;//первый в
очереди символ
                equivalenceClass =
queue.First().EquivalenceClass;//первый в очереди класс
                int numOfClasses =
classMembers.Keys.Count;//количество классов на данный
момент
                for (int key = 1; key < numOfClasses+1;
key++)//обход всех классов, кроме сплиттера
                {
```

```
                        if (key!=equivalenceClass)//если
класс не сплиттер
                        Split(key,                symbol,
equivalenceClass);
                }

if(!Split(equivalenceClass,symbol,equivalenceClass))//ра
збиваем сам сплиттер
                queue.Dequeue();//отработавшая      пара
сплиттер-символ покидает очередь, если сплиттер не был
разбит на подмножества
            }
        }
        /// <summary>
        ///      Метод,      реализующий      построение
минимизированного автомата.
        /// </summary>
        ///   <returns>автомат   с   минимальным   числом
состояний</returns>
        public GraphBuilder Minimize()
        {
            Divide();//минимальное разбиение
            GraphBuilder       graph       =       new
GraphBuilder();//новый граф
            int trapClass=0;
            foreach (int key in classMembers.Keys)//для
каждого класса
                if (classMembers[key].Contains(trap))
                    trapClass = key;
            foreach (int key in classMembers.Keys)//для
каждого класса
                if(key!=trapClass)//если  в  этом  классе
нет ловушки
                {
                    try
                    {
                        if
(acceptStates.Contains(classMembers[key].First()))//если
в классе допускающие состояния, делаем новое состояние
допускающим

graph.UserAddFinalVertix(key.ToString());

                    else

graph.UserAddVertix(key.ToString());//иначе      обычное
состояние
                    }
```

```
                    catch (InvalidOperationException)
                    {
                        MessageBox.Show("Automata
without final states cannot be minimized...", "Error!",
MessageBoxButtons.OK);
                        return null;
                    }
                    if
(classMembers[key].Contains(startingState))//если      в
множестве есть начальное состояние, делаем новое
состояние начальным
                    {
                        StartingState = key.ToString();

graph.UserAddVertix(String.Empty);
                        graph.UserAddEdge(String.Empty,
key.ToString(), String.Empty);
                    }
                }
            foreach (int key in classMembers.Keys)//для
каждого класса
                if (key != trapClass)//если в этом классе
нет ловушки
                {
                    string            state            =
classMembers[key].First();//смотрим на первое состояние в
классе, потому что все состояния эквивалентны
                    foreach      (var      edge      in
StateInfoDictionary[state])//обход   всех   исходящих   из
данного состояния дуг
                    {
                        string distinationState="";

if(!classMembers[trapClass].Contains(edge.DictinationSta
te))//если дуга не в ловушку
                        {
                            foreach(int   className   in
classMembers.Keys)//ищем, в каком классе лежит состояние,
куда переходит дуга

if(classMembers[className].Contains(edge.DictinationStat
e))
                                {
                                    distinationState   =
className.ToString();
                                    break;
                                }
```

```
graph.UserAddEdge(key.ToString(),      distinationState,
edge.Transition);
                            }
                        }
                    }
            return graph;
        }
    }
}
```

### 1.4.12. Код класса TSMLtransformer

```
using System.Collections.Generic;
using System.Text;
using System.Xml;
using LibGraph;
using DrawingSupporter;

namespace Project
{
    /// <summary>
    /// Класс, задающий вершину графа, как ее имя и
идентификационное имя (необходимо для сохранения в
формате .tsml).
    /// </summary>
    class TSMlVertex
    {
        /// <summary>
        /// Имя вершины.
        /// </summary>
        public string name { get; set; }
        /// <summary>
        /// Идентификационное имя вершины.
        /// </summary>
        public string id { get; set; }
    }
    /// <summary>
    /// Класс, осуществляющий преобразование конечного
автомата, заданного в формате .tsml в формат, используемый
программой, и наоборот.
    /// </summary>
    class TSMLtransformer
    {
        /// <summary>
        /// Словарь состояний. Ключом является имя
состояния, значением - ее идентификационное имя.
```

```
        /// </summary>
        Dictionary<string, string> stateID;
        /// <summary>
        /// Начальное состояние автомата.
        /// </summary>
        string startingState;
        /// <summary>
        /// Список конечных состояний автомата.
        /// </summary>
        List<string> finalStates;
        /// <summary>
        /// Документ в формате .tsml.
        /// </summary>
        XmlDocument doc;
        /// <summary>
        /// Конечный автомат.
        /// </summary>
        GraphBuilder graph;
        /// <summary>
        /// Поле для записи информации в формате .xml.
        /// </summary>
        XmlTextWriter textWritter;
        /// <summary>
        /// Конструктор, вызываемый при необходимости
преобразовать конечный автомат, заданный в формате .tsml
в формат, используемый программой.
        /// </summary>
        /// <param name="filePath">Путь к файлу в формате
.tsml.</param>
        /// <param name="formChoice">Стартовое окно
приложения.</param>
        /// <param name="opened">Индикатор, был ли открыт
файл.</param>
        public TSMLtransformer(string
filePath,ChoiceFSMForm formChoice, bool opened)
        {
            if (opened)
            {
                formChoice.Hide();
                graph = new GraphBuilder();
                doc = new XmlDocument();
                doc.Load(filePath);
                XmlElement root = doc.DocumentElement;
                GetEdgeList(root, GetNodeList(root));
                FormUsersFSM form = new
FormUsersFSM(graph, finalStates,
startingState,formChoice);
                if (form.Success == true)
```

```
                    form.Show();
                else
                    formChoice.Show();
            }
        }
        /// <summary>
        /// Конструктор, вызываемый при необходимости
преобразовать конечный автомат, заданный в формате
формат, используемым программой в формат .tsml.
        /// </summary>
        /// <param name="path">Путь к созданному
файлу.</param>
        /// <param name="stateList">Список
состояний.</param>
        /// <param name="finalStateList">Список конечных
состояний.</param>
        /// <param name="startingState">Начальное
состояние.</param>
        /// <param name="edgeList">Список
переходов.</param>
        public TSMLtransformer(string path, List<Vertex>
stateList,List<string> finalStateList, string
startingState, List<Edge> edgeList)
        {
            stateID = new Dictionary<string, string>();
            doc = new XmlDocument();
            CreateTSML(path,stateList, finalStateList,
startingState, edgeList);
        }
        /// <summary>
        /// Метод, выполняющий сохранение конечного
автомата в формате .tsml.
        /// </summary>
        /// <param name="path">Путь к сохраняемому
файлу.</param>
        /// <param name="stateList">Список
состояний.</param>
        /// <param name="finalStateList">Список конечных
состояний.</param>
        /// <param name="startingState">Начальное
состояние.</param>
        /// <param name="edgeList">Список
переходов.</param>
        void CreateTSML(string path, List<Vertex>
stateList, List<string> finalStateList, string
startingState, List<Edge> edgeList)
        {
```

```
            textWritter = new XmlTextWriter(path,
Encoding.UTF8);
            textWritter.WriteStartDocument();
            textWritter.WriteStartElement("graph");
            textWritter.WriteEndElement();
            textWritter.Close();
            doc.Load(path);
            int count = 1;
            foreach (Vertex state in stateList)
                if (!finalStateList.Contains(state.Name)
&& state.Name != startingState)
                {
                    AppendState(state.Name,       count,
false, false);
                    stateID.Add(state.Name,  "state"  +
count);
                    count++;
                }
            foreach (string state in finalStateList)
                if (state != startingState)
                {
                    AppendState(state,   count,   true,
false);
                    stateID.Add(state, "state" + count);
                    count++;
                }
            AppendState(startingState,   count,   false,
true);
            stateID.Add(startingState, "state" + count);
            count = 1;
            foreach (Edge edge in edgeList)
            {
                AppendEdge(edge.Transition,       count,
stateID[edge.Source.Name], stateID[edge.Target.Name]);
                count++;
            }
            doc.Save(path);
        }
        /// <summary>
        /// Метод, получающий список состояний автомата
из корневого элемента tsml файла.
        /// </summary>
        /// <param name="root">Корневой элемент.</param>
        /// <returns>Список состяний.</returns>
        public  List<TSMlVertex>  GetNodeList(XmlElement
root)
        {
            finalStates = new List<string>();
```

```
            List<TSMlVertex>      nodes      =      new
List<TSMlVertex>();
            foreach (XmlNode xnode in root)
            {
                if (xnode.Name=="state")
                foreach     (XmlNode     childnode     in
xnode.ChildNodes)
                {
                    if (childnode.Name == "name")
                        {
                            XmlNode        attr1        =
xnode.Attributes.GetNamedItem("start");
                            if (attr1 != null)
                            {
                                startingState          =
childnode.InnerText;
                                graph.UserAddVertix("");
                                graph.UserAddEdge("",
startingState, "");
                            }
                            XmlNode        attr2        =
xnode.Attributes.GetNamedItem("accept");
                            if (attr2 != null)

finalStates.Add(childnode.InnerText);
                            XmlNode        attr        =
xnode.Attributes.GetNamedItem("id");
                            //MessageBox.Show("id="    +
attr.Value, childnode.InnerText);
                            nodes.Add(new TSMlVertex { id
= attr.Value, name = childnode.InnerText });

graph.UserAddVertix(childnode.InnerText);
                        }
                }
            }
            return nodes;
        }
        /// <summary>
        /// Метод, получающий список переходов автомата
из корневого элемента tsml файла.
        /// </summary>
        /// <param name="root"></param>
        /// <param name="nodes"></param>
        public    void    GetEdgeList(XmlElement    root,
List<TSMlVertex> nodes)
        {
            string source;
```

```
            string target;
            string text;
            XmlNode attr;
            foreach (XmlNode xnode in root)
            {
                if (xnode.Name == "transition")
                    foreach    (XmlNode    childnode    in
xnode.ChildNodes)
                    {
                        if (childnode.Name == "name")
                        {
                            attr                          =
xnode.Attributes.GetNamedItem("source");
                            source=nodes.Find(item    =>
item.id == attr.Value).name;
                            attr                          =
xnode.Attributes.GetNamedItem("target");
                            target  =  nodes.Find(item  =>
item.id == attr.Value).name;
                            text = childnode.InnerText;
                            //MessageBox.Show("source="
+ source+"target="+target+"text="+text);

graph.UserAddEdge(source,target,text.Replace("+complete"
," ").Replace("+"," "));
                        }
                    }
            }
        }
        /// <summary>
        ///  Метод, записывающий в файл формата .tsml
информацию о состоянии автомата.
        /// </summary>
        /// <param name="name">Имя состояния.</param>
        /// <param name="num">Номер состяония.</param>
        ///     <param     name="isFinal">Конечное     ли
состояние.</param>
        ///    <param    name="isStarting">Начальное    ли
состояние.</param>
        public  void  AppendState(string  name,  int  num,
bool isFinal, bool isStarting)
        {
            XmlAttribute attribute;
            XmlNode element;
            element = doc.CreateElement("state");
            doc.DocumentElement.AppendChild(element);
                attribute = doc.CreateAttribute("id"); //
создаём атрибут
```

```
                attribute.Value                                =
"state"+num.ToString();   //   устанавливаем   значение
атрибута
                element.Attributes.Append(attribute); //
добавляем атрибут
            XmlNode              childNode              =
doc.CreateElement("name");
            childNode.InnerText = name;
            element.AppendChild(childNode);
            if (isFinal)
            {
                attribute                                      =
doc.CreateAttribute("accept"); // создаём атрибут
                attribute.Value     =     "true";     //
устанавливаем значение атрибута
                element.Attributes.Append(attribute); //
добавляем атрибут
            }
            if (isStarting)
            {
                attribute                                      =
doc.CreateAttribute("start"); // создаём атрибут
                attribute.Value     =     "true";     //
устанавливаем значение атрибута
                element.Attributes.Append(attribute); //
добавляем атрибут
            }
            //element.InnerText = name;
        }
        /// <summary>
        ///  Метод, записывающий в файл формата .tsml
информацию о переходе автомата.
        /// </summary>
        /// <param name="name">Символ перехода.</param>
        /// <param name="num">Номер перехода.</param>
        ///        <param        name="source">Состояние
исхода.</param>
        /// <param name="target">Состояние входа.</param>
        public  void  AppendEdge(string  name,  int  num,
string source, string target)
        {
            XmlAttribute attribute;
            XmlNode element;
            element = doc.CreateElement("transition");
            doc.DocumentElement.AppendChild(element);
            attribute  =  doc.CreateAttribute("id");  //
создаём атрибут
```

```
            attribute.Value     =     "transition"     +
num.ToString(); // устанавливаем значение атрибута
            element.Attributes.Append(attribute);     //
добавляем атрибут
            attribute = doc.CreateAttribute("source");
            attribute.Value = source;
            element.Attributes.Append(attribute);
            attribute = doc.CreateAttribute("target");
            attribute.Value = target;
            element.Attributes.Append(attribute);
            XmlNode              childNode              =
doc.CreateElement("name");
            childNode.InnerText = name;
            element.AppendChild(childNode);
            //element.InnerText = name;
        }
    }

}
```

### 1.4.13. Код класса UsersFiles

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
using System.IO;
namespace Project
{
    /// <summary>
    /// Класс, оперирующий работой с файлами.
    /// </summary>
    class UsersFiles
    {
        /// <summary>
        /// Файловый поток.
        /// </summary>
        Stream myStream = null;
        /// <summary>
        /// Поле, хранящее путь к открытому пользоватем
файлу.
        /// </summary>
        string filePath;
        /// <summary>
        /// Диалоговое окно открытия файла.
        /// </summary>
        OpenFileDialog myOpenFileDialog;
        /// <summary>
```

```
        /// Свойство доступа к полю filePath.
        /// </summary>
        public string FilePath { get { return filePath; }
}
        /// <summary>
        /// Метод, удаляющий папку, созданную во время
работы программы.
        /// </summary>
        public void DeleteFiles()
        {
            if(Directory.Exists("Graphs"))
            Directory.Delete("Graphs",true);
        }
        /// <summary>
        ///   Метод,   получающий   путь   к   открытому
пользователем файлу.
        /// </summary>
        /// <returns></returns>
        public bool OpenFile()
        {
            myOpenFileDialog = new OpenFileDialog();
            myOpenFileDialog.InitialDirectory = "c:\\";
            myOpenFileDialog.RestoreDirectory = true;
            myOpenFileDialog.Filter     =     "tsml-файл
(*.tsml)|*.tsml";
            if     (myOpenFileDialog.ShowDialog()     ==
DialogResult.OK)
            {
                filePath                                =
Path.GetFullPath(myOpenFileDialog.FileName);
                try
                {
                    myStream                            =
myOpenFileDialog.OpenFile();
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Error!  File  is  not
found! " + ex.Message);
                }
                return true;
            }
            else
            {
                MessageBox.Show("Error!   File   is   not
opened! ");
                return false;
            }
```

```
        }
        /// <summary>
        /// Метод, получающий путь к файлу, в который
нужно сохранить информацию.
        /// </summary>
        ///  <param  name="format">Формат  сохраняемого
файла.</param>
        /// <returns></returns>
        public string ChooseSavingPath(string format)
        {
            string path = "";
            SaveFileDialog    saveFileDialog    =    new
SaveFileDialog();
            switch (format)
            {
                case "tsml":
                    saveFileDialog.Filter  =  "tsml-file
(*.tsml)|*.tsml";
                    break;
                case "png":
                    saveFileDialog.Filter   =   "png-file
(*.png)|*.png";
                    break;
                case "txt":
                    saveFileDialog.Filter   =   "txt-file
(*.txt)|*.txt";
                    break;
            }
            saveFileDialog.RestoreDirectory = true;
            if      (saveFileDialog.ShowDialog()      ==
DialogResult.OK)//создаем новый файл
            {
                try
                {
                    if            ((myStream            =
saveFileDialog.OpenFile()) != null)
                    {
                        myStream.Close();
                        path                            =
Path.GetFullPath(saveFileDialog.FileName);
                    }
                }
                catch (IOException ex)
                {
                    MessageBox.Show(ex.Message);
                }
            }
            return path;
```

```
        }
        /// <summary>
        /// Метод, осуществляющий сохранение изображения.
        /// </summary>
        /// <param name="path"></param>
        /// <param name="bmp"></param>
        public void SaveImage(string path, Bitmap bmp)
        {
                bmp.Save(path,
System.Drawing.Imaging.ImageFormat.Png);
        }
        /// <summary>
        /// Метод, осуществляющий сохраниние аналиических
данных, полученных в результате работы алгоритма.
        /// </summary>
        /// <param name="afterAlgo"></param>
        /// <param name="beforeAlgo"></param>
        /// <param name="basicDefinitions"></param>
        /// <param name="processDefinitions"></param>
        /// <param name="analytics"></param>
        public   void   SaveMinedExpression(List<string>
afterAlgo,   List<string>   beforeAlgo,   List<string>
basicDefinitions,   List<string>   processDefinitions,
List<string> analytics)
        {
            string path = ChooseSavingPath("txt");
            if (path == "")
                MessageBox.Show("Error!  File  was  not
saved...");
            else
            {
                using   (StreamWriter   stream   =   new
StreamWriter(path))
                {
                    foreach (string str in afterAlgo)
                        stream.WriteLine(str);
                    foreach (string str in beforeAlgo)
                        stream.WriteLine(str);
                    foreach(string        str        in
basicDefinitions)
                        stream.WriteLine(str);
                    foreach(string        str        in
processDefinitions)
                        stream.WriteLine(str);
                    foreach(string str in analytics)
                        stream.WriteLine(str);
                }
            }}}}
```

## ПРИЛОЖЕНИЕ 1

## ТЕРМИНОЛОГИЯ

*Таблица 1*

| Термин (рус) | Термин (англ) | Определение |
|---|---|---|
| **Глубинный анализ процессов** | **Process mining** | Область исследований, занимающаяся построением моделей процессов по логам событий. |
| **Регулярные выражения** | **Regular expressions** | Способ моделирования последовательных процессов, использующий шаблоны выбора, повторения и итерации. |
| **Метод удаления состояний** | **State removal method** | Алгоритм построения регулярного выражения по конечному автомату, предполагающий последовательное удаление состояний автомата и одновременное построение регулярных выражений, соответствующих заменяемым системам переходов. |
| **Конечный автомат** | **Finite state machine (FSM)** | Конечный автомат – это кортеж $FSM = (S, E, T, s_i, S_f)$, где $S$ – конечное множество состояний, $E$– конечное множество событий, $T \subseteq (S \times E \times S)$ – конечное множество переходов, $s_i \subseteq S$ – начальное состояние, и $S_f \subseteq S$ – множество конечных состояний. |
| **Начальное состояние** | **Initial state** | Стартовое состояние автомата. |
| **Конечные (допускающие) состояния** | **Accepting states** | Состояния, в которых автомат может завершить работу. |
| **Событие (Процесс)** | **Event** | Действие, способное изменить состояние автомата. |
| **Переход** | **Transition** | Событие, связывающее два состояния автомата. |
| **Подпроцесс** | **Subprocess** | Регулярное выражение, которое является частью регулярного выражения, задающего автомат, и может быть заменено одним символом. |
| **Метод построения регулярного выражения из конечного автомата с заменой подпроцессов** | **Method of constructing regular expression out of an FSM using subprocess changing** | Метод построения регулярного выражения из конечного автомата, при котором переходы отождествляются с регулярными выражениями, повторяющиеся регулярные выражения именуются подпроцессами и заменяются ключевыми символами. |

# ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

| Лист регистрации изменений | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Номера листов (страниц) | | | | | Всего листов (страниц в докум.) | № документа | Входящий № сопроводительного докум. и дата | Подп. | Дата |
| Изм. | Измененных | Заменных | Новых | Аннулированных | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

| | | | | |
|---|---|---|---|---|
| Изм. | Лист | № докум. | Подп. | Дата |
| RU.17701729.502610-01 12 | | | | |
| Инв. № подл. | Подп. и дата | Взам. инв. № | Инв. № дубл. | Подп. и дата |