ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Департамент программной инженерии

СОГЛАСОВАНО Доцент, старший научный сотрудник научно-учебной лаборатории ПОИС, к.ф.-м.н.	УТВЕРЖДАЮ Академический руководитель образовательной программы «Программная инженерия»
_____ А.А. Каленкова «___» _____ 2018 г.	_____ В.В. Шилов «___» _____ 2018 г.

Инв. № подл	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата
RU.17701729.507140-01 81 01-1				

ПРОГРАММА ПОСТРОЕНИЯ РЕГУЛЯРНОГО ВЫРАЖЕНИЯ ПО КОНЕЧНОМУ АВТОМАТУ

Пояснительная записка

ЛИСТ УТВЕРЖДЕНИЯ

RU.17701729.04.13-01 81 01-1-ЛУ

Исполнитель:
студент группы БПИ173
_____ /Таранцова Полина Дмитриевна/
«____»_____ 2018 г.

2018

УТВЕРЖДЕНО
RU.17701729.04.13-01 81 01-1-ЛУ

ПРОГРАММА ПОСТРОЕНИЯ РЕГУЛЯРНОГО ВЫРАЖЕНИЯ ПО КОНЕЧНОМУ АВТОМАТУ

Пояснительная записка

RU.17701729.04.13-01 81 01-1

Листов 47

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

2018

СОДЕРЖАНИЕ

1.ВВЕДЕНИЕ.....5
1.1. Наименование программы.....5
1.2. Документы, на основании которых ведется разработка.....5
2.НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ……6
2.1. Назначение программы.....6
2.2. Краткая характеристика области применения.....6
3.ТЕХНИЧЕСКИЕ ХАРАКТЕРИСТИКИ……7
3.1. Постановка задачи на разработку программы.....7
3.2. Описание используемого математического аппарата.....7
3.3. Описание алгоритма и функционирования программы.....9
3.4. Обоснование выбора схемы алгоритма решения задачи.....18
3.5. Описание и обоснование выбора метода организации входных и выходных данных……..19
3.6. Описание и обоснование выбора состава технических и программных средств……..19
4.ОЖИДАЕМЫЕ ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ.....20
4.1. Предполагаемая потребность.....20
4.2. Экономические преимущества разработки по сравнению с отечественными и зарубежными образцами или аналогами.....20
5. ИСТОЧНИКИ, ИСПОЛЬЗОВАННЫЕ ПРИ РАЗРАБОТКЕ.....21
ПРИЛОЖЕНИЕ 1. ТЕРМИНОЛОГИЯ.....23
ПРИЛОЖЕНИЕ 2. КЛЮЧЕВЫЕ СИМВОЛЫ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ.....25
ПРИЛОЖЕНИЕ 3. ОПИСАНИЕ И ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ КЛАССОВ.....26
ПРИЛОЖЕНИЕ 4. ОПИСАНИЕ И ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ ПОЛЕЙ, МЕТОДОВ И СВОЙСТВ.....28
ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ.....47

АННОТАЦИЯ

В данном программном документе приведена пояснительная записка к программе «Expression Miner» («Программа построения регулярного выражения по конечному автомату»), несущей научно-образовательный характер и предназначенной для анализа работы алгоритмов.

В разделе «Введение» указано наименование программы, краткое наименование программы и документы, на основании которых ведется разработка.

В разделе «Назначение и область применения» указано функциональное назначение программы, эксплуатационное назначение программы и краткая характеристика области применения программы.

В разделе «Технические характеристики» содержатся следующие подразделы: – постановка задачи на разработку программы; – описание алгоритма и функционирования программы с обоснованием выбора схемы алгоритма решения задачи и возможные взаимодействия программы с другими программами; – описание и обоснование выбора метода организации входных и выходных данных; – описание и обоснование выбора состава технических и программных средств.

В разделе «Ожидаемые технико-экономические показатели» указана предполагаемая потребность и экономические преимущества разработки по сравнению с отечественными и зарубежными образцами или аналогами

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 19.404-79 Пояснительная записка. Требования к содержанию и оформлению [7].

Изменения к Пояснительной записке оформляются согласно ГОСТ 19.603-78 [8], ГОСТ 19.604-78 [9].

Перед прочтением данного документа рекомендуется ознакомиться с терминологией, представленной в Приложении 1 настоящей пояснительной записки.

1. ВВЕДЕНИЕ

1.1. Наименование программы

Наименование программы: «Программа построения регулярного выражения по конечному автомату».

Краткое наименование программы: «Expression Miner».

Условное обозначение темы разработки: «A Program for Constructing Regular Expression from a Finite State Machine (FSM)».

1.2. Документы, на основании которых ведется разработка

Программа выполнена в рамках темы курсовой работы — «Программа построения регулярного выражения по конечному автомату», в соответствии с учебным планом подготовки бакалавров по направлению 09.03.04 «Программная инженерия».

Разработка ведется на основании приказа декана факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики» № 2.3-02/1212-01 от 12.12.2017 «Об утверждении тем, руководителей курсовых работ студентов образовательной программы Программная инженерия факультета компьютерных наук».

2. НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ

2.1. Назначение программы

2.1.1. Функциональное назначение

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.

2.1.2. Эксплуатационное назначение

Возможности программы предполагают ее использование в научных лабораториях для анализа работы алгоритмов, анализа логов событий в рамках теории глубинного анализа процессов, а также в образовательных учреждениях для ознакомления студентов с теорией автоматов и формирования у учащихся навыков построения конечных автоматов, их минимизации и приведения к регулярному выражению. Клиентская часть программы выполняется на персональных компьютерах.

2.2. Краткая характеристика области применения

«Программа построения регулярного выражения по конечному автомату» несет научно-образовательный характер и предназначена для анализа работы алгоритмов.

3. ТЕХНИЧЕСКИЕ ХАРАКТЕРИСТИКИ

3.1. Постановка задачи на разработку программы

Разрабатываемая программа должна:

1) Предоставить инструменты создания конечного автомата;
2) Визуализировать загруженный в программу конечный автомат (в формате .tsml);
3) Строить по пользовательскому конечному автомату регулярное выражение с использованием анимации, демонстрирующей метод удаления состояний;

3.2. Описание используемого математического аппарата

В данной программе реализован алгоритм построения регулярных выражений по логам событий. В программу загружается построенный по логу событий конечный автомат, по которому строится регулярное выражение. (Рис. 2)

Рассмотрим лог $E = \{\langle a, b\rangle, \langle a, b, c\rangle, \langle a, d, b\rangle\}$. Конечный автомат, построенный по данному логу, представлен на рисунке 1. Допускающие состояния выделены белым цветом, каждое состояние соответствует последовательности последних извлеченных событий длины меньше или равной 1. Хотя в этом примере журнал событий и язык соответствующего конечного автомата совпадают, в общем случае гарантируется, что журнал событий является подмножеством языка соответствующего конечного автомата.

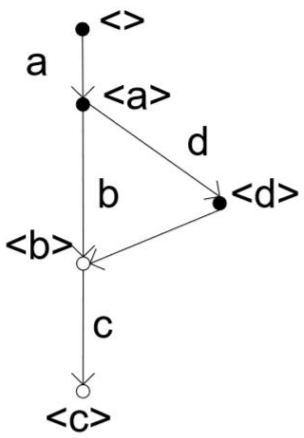


Рис. 1. Конечный автомат, построенный по логу E

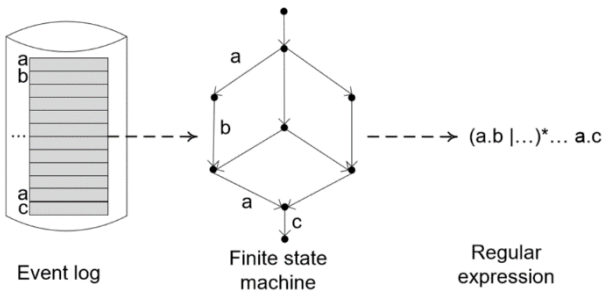


Рис. 2. Разработанный подход.

В программе применяется подход, базирующийся на удалении состояний с целью построения регулярного выражения по заданному конечному автомату [20]. Основная идея

метода удаления состояний заключается в том, что переходы могут быть заданы не событиями, а регулярными выражениями (Рис. 3). На последнем шаге алгоритма результирующие регулярные выражения заключены в переходы, ведущие из начального состояния в конечное. Выбор удаляемого состояния существенно влияет на длину результирующего регулярного выражения, поэтому алгоритм выбора удаляемого состояния [21] работает таким образом, что на каждом шаге выбирается состояние, удаление которого произведет регулярные выражения минимальной длины. Состояние s с минимальным значением

$$W(s) = (l-1)\sum_{i=1}^{m}|r_{s_i s}| + (m-1)\sum_{i=1}^{l}|r_{s s_i}| + (ml-1)\sum_{i=1}^{l}|r_{ss}|,$$

где $m(l)$ - количество входящих (исходящих) переходов для s, и $r_{s_i s_j}$ - регулярное выражение, задающее переход из s_i в s_j, будет удалено первым.

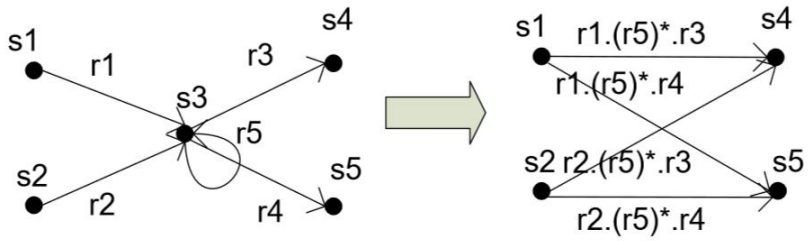


Рис. 3. Технология удаления состояния.

С опорой на свойства конечных автоматов был разработан алгоритм, позволяющий уменьшить длину результирующего регулярного выражения. Рассмотрим часть конечного автомата, представленную на рисунке 4. Так как конечный автомат был построен с использованием префиксов длины 1, каждое состояние соответствует определенному событию, и все входящие в это состояние переходы помечены этим событием. Таким образом, после удаления состояния s будут получены три регулярных выражения $a.b^*.c$ и три регулярных выражения $a.b^*.d$. Повторяющееся регулярное выражение может быть заменено ключевым символом, задающим подпроцесс. Такие замены могут быть произведены не только на первом шаге алгоритма удаления состояний, но и на последующих шагах, потому как регулярные выражения, задающие входящие в состояние переходы, идентичны. Кроме того, если регулярные выражения, задающие входящие в состояние переходы, не идентичны, такие префиксы, как $a.b^*$ (для автомата на Рис. 3), так же могут задавать подпроцессы.

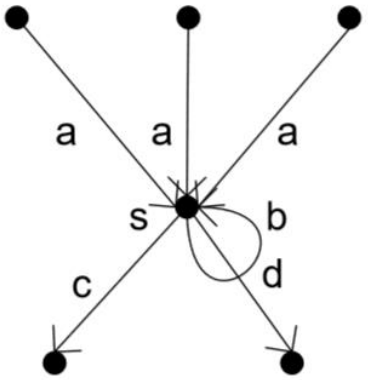


Рис. 4. Часть конечного автомата.

3.3. Описание алгоритма и функционирования программы

3.3.1. Последовательность запускаемых алгоритмов

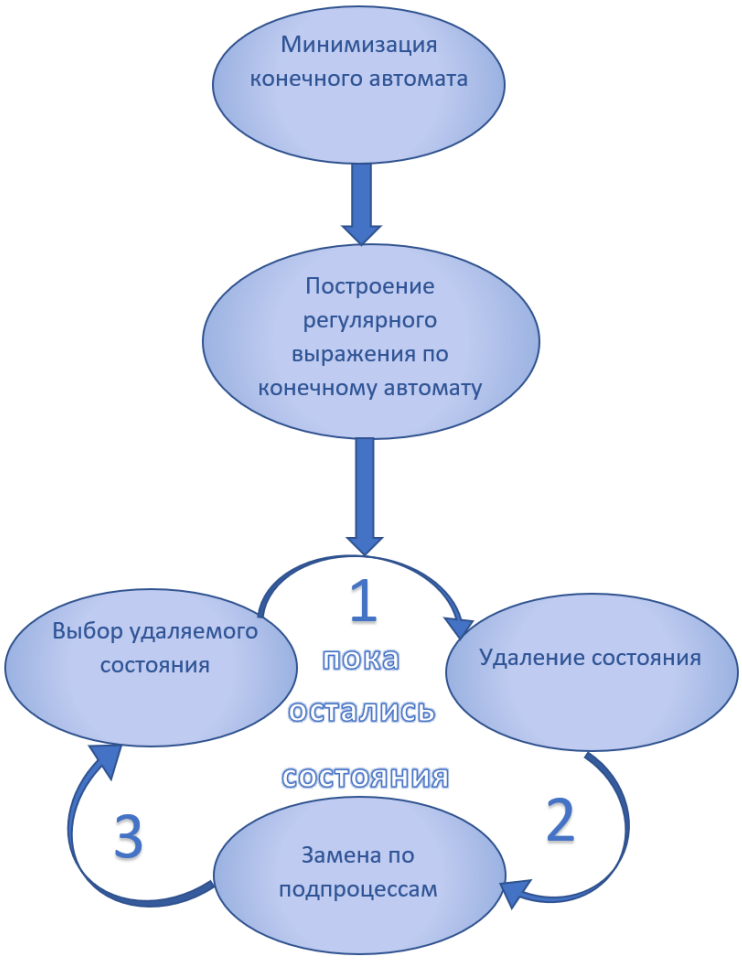


Рис. 5. Последовательность запускаемых алгоритмов

3.3.2. Алгоритм минимизации конечного автомата

Минимизация конечного автомата производится с использованием алгоритма Мура. Для конечного автомата $M' = (S', E, T', s_i, S_f')$ строим эквивалентный всюду определенный автомат $M = (S, E, T, s_i, S_f)$;

*Автомат $M = (S, E, T, s_i, S_f)$ является всюду определенным, если $T'(q, a) \neq \emptyset$ для $\forall q \in S, \forall a \in E$. $S = \{S', trap\}$, $S_f = \{S_f', trap\}$, trap (ловушка), для $D(q, a) = \emptyset, q \in S', a \in E$ автомата M' в автомате M строится $T(trap, a) \neq \emptyset$, $T \supseteq T'$.

1) Строим начальное разбиение X множества состояний S, состоящее из двух классов эквивалентности: множества допускающих состояний S_f и всех остальных $S - S_f$;
2) Создаем очередь из пар $(G; k)$, именуемых сплиттерами, для $\forall k \in E$, для $\forall G \in X$, где G – класс эквивалентности, а k – символ перехода;
3) Для всех классов эквивалентности $J \in X/G$ создаем два класса $J_1 = \{\emptyset\}$ и $J_2 = \{\emptyset\}$ и выполняем разбиение по первому сплиттеру (G_1, k_1) из очереди так, что состояние $m \in J$:
   - переходит в J_1, если имеет переход по k_1 в G_1;
   - переходит в J_2, если не имеет такого перехода.
   Очередь дополняется парами $(J_1; k) \forall k \in E$ (при условии, что $J_1 \neq \emptyset$), $(J_2; k) \forall k \in E$ (при условии, что $J_2 \neq \emptyset$);
4) Создаем класс $G_2 = \{\emptyset\}$ и выполняем разбиение сплиттера, согласно следующим правилам:
   состояние $m \in G_1$
   - остается в G_1, если имеет переход по k_1 в G_1;
   - переходит в G_2, если не имеет такого перехода;
5) Если $G_2 \neq \{\emptyset\}$, то очередь дополняется парами $(G_1; k), (G_2; k) \forall k \in E$ (если пара $(G_1; k)$ уже существует в очереди, ее добавлять не нужно), иначе сплиттер (G_1, k_1) покидает очередь;
6) Шаги 4-6) повторяются до тех пор, пока в очереди остались сплиттеры;
7) Класс эквивалентности, содержащий trap, и все переходы в него, удаляются;
8) Строится минимизированный автомат, состояниями которого являются классы эквивалентности, а переходами – символы, входящие в эти классы.

Реализацию алгоритма разбиения производит метод `Divide` класса `Minimizator`.

```
        /// <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))//разбиваем
сам сплиттер
                queue.Dequeue();//отработавшая пара сплиттер-символ покидает
очередь, если сплиттер не был разбит на подмножества
            }
        }
```

Описание использованных в алгоритме методов доступно в Приложении 4.

Результатом применения алгоритма минимизации является объект класса `GraphBuilder` – граф, задающий конечный автомат с минимальным числом состояний. Данный алгоритм используется в коде программы, так как он позволяет получить конечный автомат, эквивалентный исходному, но с минимальным возможным количеством состояний, соответственно для построения регулярного выражения, задающего конечный автомат, отныне требуется удалить меньшее количество состояний.

3.3.3. Алгоритм удаления состояния конечного автомата с использованием алгоритма замены по подпроцессам

Алгоритм удаления состояния s конечного автомата реализуется следующим образом:

1) Для s формируется список входящих дуг `enteringEdges` (здесь и далее дугой именуется переход, связывающий два состояния автомата), список исходящих дуг `exitingEdges`
2) Для всех пар $(n, m), n \in N, m \in M$, где $N -$ множество состояний, из которых есть переходы в s, $M -$ множество состояний, в которые есть переходы из s (в коде $n - source, m - target$, строится регулярное выражение, связывающее эти состояния, таким образом:
    a. Проверяется, существует ли дуга между состояниями $source$ и $target$, если такая дуга существует, то символ перехода этой дуги поступает в поле label;
    b. Строится регулярное выражение, полученное посредством удаления состояния s, оно хранится в поле addedLabel
        i. addedLabel присваивается значение символа перехода из source в s;

ii. если у состояния s существует петля, присваиваем addedLabel конкатенацию addedLabel и итерации петли;
iii. addedLabel на данном этапе является префиксом, если его длина больше двух символов, если addedLabel присутствует в словаре `Dictionary<string,int> prefixes` (словарь префиксов, ключ которого - регулярное выражение, задающее префикс, а значение - количество повторений префикса), увеличиваем количество таких префиксов, иначе добавляем этот префикс в словарь префиксов;
iv. addedLabel присваивается конкатенация addedLabel и дуг из s в target;

c. Добавляем дугу между состояниями source и target, переходом является объединение label и addedLabel, если получившийся переход содержится в списке всех регулярных выражений, получивших посредством удаления s `numberOfUniqChangedEdges`, добавляем его в словарь подпроцеессов графа `graph.finalDefinitions`, иначе добавляем в список;

3) Состояние s удаляется из автомата;
4) Производится обход всех дуг конечного автомата, для каждой дуги:
   a. Если переход, задающий данную дугу, присутствует в словаре подпроцессов `graph.finalDefinitions`, заменяем подпроцесс на соответствующий ему ключевой символ;
   b. Добавляем все подпроцессы, входящие в состав данного перехода, длина которых более одного символа, в список подпроцессов `numberOfUniqParts`, если такой подпроцесс был добавлен ранее, добавляем его в словарь подпроцессов графа `graph.finalDefinitions`;
5) Производится сортировка словаря префиксов по убыванию длины регулярного выражения, задающего префикс;
6) Производится второй обход всех дуг конечного автомата, для каждого перехода, задающего дугу, подпроцессы в ее составе, входящие в словарь подпроцессов `graph.finalDefinitions`, заменяются на соответствующие им ключевые символы. После замены подсчитывается количество вхождений префиксов в состав перехода, повторения фиксируются в словаре префиксов;
7) Производится обход словаря префиксов, префиксы, входящие в состав регулярных выражений более одного раза, добавляются в словарь часто встречающихся префиксов `Dictionary<string, string> frequentPrefixes` (ключ – ключевой символ префикса, значение -сам префикс)
8) Производится второй обход всех дуг конечного автомата, для каждого перехода, задающего дугу, подпроцессы в ее составе, входящие в словарь часто встречающихся префиксов `frequentPrefixes`, заменяются на соответствующие им ключевые символы.

Реализацию алгоритма удаления недопускающего состояния конечного автомата с использованием алгоритма минимизации регулярных выражений производит метод `RemoveNonFinalState` класса `StateRemoval.`

```
                /// <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)//если 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;
    }
```

Описание использованных в алгоритме методов доступно в Приложении 4.

Результатом применения алгоритма минимизации является объект класса `GraphBuilder` – граф, получившийся из исходного после удаления состояния s.

Данный алгоритм используется в коде программы, так как он позволяет реализовать метод удаления состояний в удобном и наглядном для пользователя формате, за счет того, что повторяющиеся подпроцессы заменяются, что предотвращает появление громоздких выражений.

3.3.4. Алгоритм выбора удаляемого состояния

Алгоритм выбора удаляемого состояния реализуется следующим образом:

1) Создаются словари:
   a. `Dictionary<string, int> income`: ключ - имя состояния, значение - длины всех регулярных выражений, задающих переходы входящих в состояние дуг;
   b. `Dictionary<string, int> outcome`: ключ - имя состояния, значение - длины всех регулярных выражений, задающих переходы исходящих из состояния дуг;
   c. `Dictionary<string, int> inrange`: ключ- имя состояния, значение - количество входящих в состояние дуг;
   d. `Dictionary<string, int> outrange`: ключ- имя состояния, значение - количество исходящих из состояния дуг;
   e. `Dictionary<string, int> self`: ключ - имя состояния, значение - длина регулярного выражения, задающего петлю;
   f. `Dictionary<string, int> damage`: ключ - имя состояния, значение - урон, который нанесет удаление состояния регулярному выражению, задающему автомат;

   ключами словарей являются все состояния автомата;
2) Производится обход всех дуг конечного автомата и заполнение словарей a-e;
3) Для каждого состояния производится подсчет урона по формуле income * outcome + inrange * outrange * self, заполняется словарь f;
4) По словарю damage производится поиск недопускающее состояние с минимальным уроном.

Реализацию алгоритма удаления недопускающего состояния конечного автомата с использованием алгоритма минимизации регулярных выражений производит метод `NonFinalLeft` класса `StateRemoval.`

```
/// <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;
              }
```

Описание использованных в алгоритме методов доступно в Приложении 4.

Результатом применения выбора удаляемого состояния является значение true/false – индикатор, было ли найдено недопускающее состояние. При положительном результате на выход подается недопускающее состояние, удаление которого нанесет минимальным урон длине результирующего регулярного выражения.

Данный алгоритм используется в коде программы, так как он позволяет реализовать метод удаления состояний в удобном и наглядном для пользователя формате, за счет того, что на очередном шаге удаления выбирается удаляемое состояние, которое увеличит регулярное выражение на минимальное число символов.

3.3.5. Алгоритм построения регулярного выражения по конечному автомату

Алгоритм построения регулярного выражения по конечному автомату реализуется следующим образом:

1) Удаляются все состояния автомата с помощью алгоритма 3.2.2, кроме начального и допускающих (выбор состояния производится по алгоритму 3.2.3);
2) Добавляется единое конечное состояние s, из всех допускающих состояний строится переход в s по символу &, эти состояния исключаются из множества допускающих, таким образом, в множестве допускающих состояний остается только состояние s;
3) Повторяется шаг 1;
4) Автомат принимает вид: начальное состояние, единое конечное состояния и множество переходов, заданных регулярными выражениями. Строится регулярное выражение, задающее автомат, как объединение всех этих переходов.

Реализацию алгоритма построения регулярного выражения по конечному автомату производит метод `ConstructRE` класса `FormUsersFSM`.

```
        /// <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
        }
```

Описание использованных в алгоритме методов доступно в Приложении 4.

Результатом применения алгоритма является регулярное выражение, эквивалентное поданному на вход конечному автомату.

Данный алгоритм используется в коде программы, так как он прост для визуализации и эффективно решает поставленную задачу.

3.4. Обоснование выбора схемы алгоритма решения задачи

Используемый в программе алгоритм был протестирован на реальных логах (TS – tracking system, FS – banking financial system, BS – flight booking system), были взяты метрики с регулярных выражений (количество состояний, длина регулярного выражения,

длина регулярного выражения после замены подпроцессов, длина корневого процесса, средняя длина подпроцессов, максимальная вложенность), построенных с использованием эвристики, основанной на префиксах и без нее. Результаты приведены в таблице.

Таблица 1.

Лог	Количество состояний	Длина регулярного выражения	Длина регулярного выражения после замены подпроцессов	Длина корневого процесса	Средняя длина подпроцессов	Максимальная вложенность
ST	5	48	39	16	2.75	2
FS	7	22	21	14	2.33	1
FS	9	66	54	32	2.2	2
FS	12	397	175	99	3.8	3
FS	13	747	207	97	3.93	5
FS	19	2480	490	214	6.27	6
BS	5	19	16	11	2.5	2
BS	7	569	106	16	5.625	4
BS	9	11000	337	114	5.87	7
BS	11	41953	643	135	9.24	8

3.5. Описание и обоснование выбора метода организации входных и выходных данных

На вход программы получает конечный автомат. Входные данные могут быть загружены из файла формата .tsml (xml – подобный формат, в котором описывается граф). Конечный автомат также может быть построен пользователем во время работы программы.

Выходными данными программы являются:

- построенный пользователем граф, сохраняемый в формате .tsml;
- шаги анимации к методу удаления состояний, сохраняемые в формате .png;
- аналитические данные (регулярное выражение, построенное с использованием метода замены подпроцессов и без него, метрики взятые с этих регулярных выражений), сохраняемые в формате .txt.

Выбор входных и входных данных обусловлен установленным функционалом программы. В частности, формат .tsml используется как наиболее популярный формат описания графов (используется в программе ProM).

3.6. Описание и обоснование выбора состава технических и программных средств

Для корректной работы программы рекомендуется использовать персональный компьютер со следующими техническими характеристиками:

- процессор с частотой 2,2 Ггц или выше;
- 0.5 гигабайт (ГБ) (для 32-разрядной системы) или 1 ГБ (для 64-разрядной системы) оперативной памяти (ОЗУ);

- 0.5 гигабайт (ГБ) (для 32-разрядной системы) или 1 ГБ (для 64-разрядной системы) пространства на жестком диске;
- монитор с разрешением 1024×640 или более высоким;
- клавиатура и мышь.

Требования к программным средствам, используемым программой:

- операционная система Windows 7 или более поздняя версия;
- установленный Microsoft .NET Framework версии не ниже 4.0.

Программа разработана на языке C#, как на кроссплатформенном объектно-ориентированном языке.

4. ОЖИДАЕМЫЕ ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ

В рамках данной работы расчет экономической эффективности не предусмотрен.

4.1. Предполагаемая потребность

Предполагается, что данный продукт будет использоваться в научных лабораториях для анализа работы алгоритмов, а также в образовательных учреждениях для ознакомления студентов с теорией автоматов и формирования у учащихся навыков построения конечных автоматов, их минимизации и приведения к регулярному выражению.

4.2. Экономические преимущества разработки по сравнению с отечественными и зарубежными образцами или аналогами

Данное приложение:
1) На данный момент не имеет аналогов;
2) распространяется бесплатно;
3) не требует вложения денежных средств во время использования;
4) имеет неограниченный срок службы;
5) интуитивно просто в использовании.

5. ИСТОЧНИКИ, ИСПОЛЬЗОВАННЫЕ ПРИ РАЗРАБОТКЕ

1. ГОСТ 19.101-77 Виды программ и программных документов. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
2. ГОСТ 19.102-77 Стадии разработки. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
3. ГОСТ 19.103-77 Обозначения программ и программных документов. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
4. ГОСТ 19.104-78 Основные надписи. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
5. ГОСТ 19.105-78 Общие требования к программным документам. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
6. ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
7. ГОСТ 19.201-78 Техническое задание. Требования к содержанию и оформлению. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
8. ГОСТ 19.603-78 Общие правила внесения изменений. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
9. ГОСТ 19.604-78 Правила внесения изменений в программные документы, выполненные печатным способом. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
10. ГОСТ 15150-69 Машины, приборы и другие технические изделия. Исполнения для различных климатических районов. Категории, условия эксплуатации, хранения и транспортирования в части воздействия климатических факторов внешней среды. – М.: Изд-во стандартов, 1997.
11. ГОСТ 19.301-79 Программа и методика испытаний. Требования к содержанию и оформлению. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.
12. Серебряков, В. А. Основы конструирования компиляторов/ Серебряков, В. А., Галочкин М. П. – Россия: Едиториал УРСС, 1999 – стр. 36 - 38.
13. Hopcroft, J. Introduction to Automata Theory, Languages, and Computation / John E. Hopcroft, Jeffrey D. Ullman – 3rd edition – USA: Addison-Wesley, 1979 – 358 p.
14. Neumann, C. Converting Deterministic Finite Automata to Regular Expressions [Электронный ресурс] – Режим доступа: http://liacs.leidenuniv.nl/~bonsanguemm/FI2/DFA_to_RE.pdf, свободный. (дата обращения: 24.02.18)
15. van der Aalst, W.M.P. Process Mining: Data Science in Action / Wil van der Aalst- 2nd edition – Heidelberg, Germany: Springer, 2016.
16. Leemans, S. Discovering Block-Structured Process Models from Event Logs - A Constructive Approach/ Leemans, S., Fahland, D., van der Aalst, W// Application and Theory of Petri Nets and Concurrency/ Colom, J., Desel, J., eds.// 34th International Conference, PETRI NETS 2013 - Milan, Italy// Volume 7927 of Lecture Notes in Computer Science - Berlin, Heidelberg, Germany: Springer, 2013 – p. 311–329.

17. Leemans, S. Discovering Block-Structured Process Models from Incomplete Event Logs/ Leemans, S., Fahland, D., van der Aalst, W// Application and Theory of Petri Nets and Concurrency/ Ciardo, G., Kindler, E., eds.// 35th International Conference, PETRI NETS 2014 - Tunis, Tunisia// Volume 8489 of Lecture Notes in Computer Science. Springer International Publishing – Cham, Germany: 2014- p. 91–110.
18. van der Aalst, W.M.P. Process mining: a two-step approach to balance between underfitting and overfitting/ van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.// Software & Systems Modeling 9(1) – 2008 – p. 87.
19. Kleene, S.C. Representation of events in nerve nets and finite automata/ Kleene, S.C// Automata Studies/ Shannon, C., McCarthy, J., eds - Princeton, NJ, USA: Princeton University Press, 1956 – p. 3–4.1
20. Brzozowski, J.A. Derivatives of regular expressions/ Brzozowski, J.A.// J. ACM 11(4) - October 1964 – p. 481–494.
21. Linz, P. An Introduction to Formal Language and Automata/Peter Linz – USA: Jones and Bartlett Publishers, Inc., 2006.
22. Delgado, M. Approximation to the smallest regular expression for a given regular language// Delgado, M., Morais, J.// Implementation and Application of Automata/ Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S., eds. - Berlin, Heidelberg: Springer Berlin Heidelberg, 2005 -p. 312–314.
23. van Dongen, B.F. The prom framework: A new era in process mining tool support/ van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.// Applications and Theory of Petri Nets 2005/ Ciardo, G., Darondeau, P., eds. - Berlin, Heidelberg: Springer Berlin Heidelberg, 2005 – p. 444–454.

ПРИЛОЖЕНИЕ 1

ТЕРМИНОЛОГИЯ

Таблица 2.1

Термин (рус)	Термин (англ)	Определение
Глубинный анализ процессов	**Process mining**	Область исследований, занимающаяся построением моделей процессов по логам событий.
Способ представления	**Representational bias**	Набор ограничений, налагаемых на модели целевых процессов, играет важную роль глубинном анализе процессов.
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.
Язык	**Language**	Множество последовательностей действий.
Регулярный язык	**Regular language**	Регулярный язык – это язык на множестве активностей $\mathcal{A}$, если он может быть построен индуктивно, опираясь на следующие правила: 1. $\emptyset$ - это регулярный язык; 2. $\{\langle\,\rangle\}$ – это регулярный язык; 3. $\{\langle a_i\rangle\}$, где $a_i \in \mathcal{A}$ – это регулярный язык, обозначаемый a_i; 4. Пусть $P, Q \subseteq \mathcal{A}^*$ - регулярные выражения, тогда: 4.1. $P.Q = \{\langle p_1, \dots, p_n, q_1, \dots, q_m\rangle \mid \langle p_1, \dots, p_n\rangle \in P \mid \langle q_1, \dots, q_m\rangle \in Q\}$ – также является регулярным выражением; 4.2. $P\mid Q = \{\sigma \mid \sigma \in P \cup Q\}$ - также является регулярным выражением; 4.3. $P^* = U_{i=0}^{\infty} P^n$, где $P^0 = \{\langle\,\rangle\}, P^1 = P$, и $P^n = P^{n-1}.P$ при $n > 0$ - также является регулярным выражением.

ТЕРМИНОЛОГИЯ

Таблица 2.2

Термин (рус)	Термин (англ)	Определение
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.
Длина регулярного выражений	**Length of the regular expression**	Количество символов без учета ключевых знаков.
Максимальная вложенность	**Maximum nesting**	Максимальное количество замен символов переходов, с помощью которых были получены символы из алфавита.
Подпроцесс	**Subprocess**	Регулярное выражение, которое является частью регулярного выражения, задающего автомат, и может быть заменено одним символом.
Длина подпроцесса	**Length of the subprocess**	Длина регулярного выражения, задающего подпроцесс.
Алгоритм минимизации конечного автомата	**Minimization algorithm**	Алгоритм, позволяющий получить автомат, эквивалентный исходному, с наименьшим возможным числом состояний.
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.
Префикс	**Prefix**	Подстрока, с которой начинается строка символов.

ПРИЛОЖЕНИЕ 2

КЛЮЧЕВЫЕ СИМВОЛЫ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Таблица 3

Обозначение	Название	Определение	Пример
.	Конкатенация	Последовательно извлеченные события.	$a.b$
+	Объединение	Возможность извлечения одного из предложенных событий.	$a+b$
*	Итерация (звезда Клини)	Возможность извлечь событие неограниченное количество раз.	$(a)^*$
()	Скобки	Подпроцесс.	(a)
&	Амперсанд	Переход в единое конечное состояние.	&
ε	Пустой переход	Переход без извлечения события.	ε

ПРИЛОЖЕНИЕ 3

ОПИСАНИЕ И ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ КЛАССОВ

Описание и функциональное назначение классов.

Таблица 4

Класс	Назначение
StateRemoval	Класс, который оперирует удалением состояний.
OutEdgeList	Класс, представляющий пару состояние - переход, используется в списке исходящих дуг.
InEdgeList	Класс, представляющий пару состояние - переход, используется в списке входящих дуг.
DrawAssistant	Класс, оперирующий отрисовкой элементов создаваемого пользователем конечного автомата.
Vertex	Класс, задающий вершину графа, как имя и пару координат.
Edge	Класс, задающий переход, как символ перехода и два состояния, как вершины графа.
GraphBuilder	Класс, задающий конечный автомат.
FileDotEngine	Класс, участвующий в получении изображения из конечного автомата. Наследуется от IDotEngine
Minimizator	Класс, выполняющий минимизацию конечного автомата.
Splitter	Класс, представляющий сплиттер и хранящий символ разбиения и номер класса эквивалентности.
OutRange	Класс, хранящий символ перехода и состояние назначения.
TSMLtransformer	Класс, осуществляющий преобразование конечного автомата, заданного в формате .tsml в формат, используемый программой, и наоборот.
TSMLVertex	Класс, задающий вершину графа, как ее имя и идентификационное имя (необходимо для сохранения в формате .tsml).
UsersFiles	Класс, оперирующий работой с файлами.
ChoiceFSMForm	Класс, представляющий стартовое окно приложения с возможностью выбрать, каким образом начать работу с программой. Наследуется от Form.
Draw	Класс, представляющий окно с инструментами и полем для создания пользовательского конечного автомата. Наследуется от Form.
FormBefore	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. Наследуется от Form.
FormInfo	Класс, представляющий окно, содержащее информацию о разработчике. Наследуется от Form.
FormProcessDefinitions	Класс, хранящий определения процессов и подпроцессов. Наследуется от Form.
FormRegularExpression	Класс, представляющий окно, содержащее регулярное выражение, построенное по пользовательскому конечному автомату, и предоставляющее доступ к аналитическим данным. Наследуется от Form.
FormRemoveStatesWithAnimation	Класс, представляющий окно, являющееся графической реализацией одного шага анимации к методу последовательного удаления состояний. Наследуется от Form.
FormSavingRules	Класс, представляющий окно, позволяющее пользователю задать параметры сохранения. Наследуется от Form.
FormTransitionName	Класс, позволяющий ввести регулярное выражение, задающее переход. Наследуется от Form.
FormUsersFSM	Класс, представляющий окно с возможностью начать построение регулярного выражения с анимацией и без нее. Наследуется от Form.

ПРИЛОЖЕНИЕ 4

ОПИСАНИЕ И ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ ПОЛЕЙ, МЕТОДОВ И СВОЙСТВ

Описание полей методов и свойств класса `StateRemoval`

Таблица 5.1

Поля				
Имя	Модификатор доступа	Тип	Назначение	
exitingEdges	private	List<OutEdgeList>	Поле, задающее список исходящих из состояния дуг.	
enteringEdges	private	List<InEdgeList>	Поле, задающее список входящих в состояние дуг.	
untouchedStatesList	private	List<string>	Поле, задающее список неудаляемых состояний.	
startingState	private	string	Поле, задающее начальное состояние.	
uniqueFinal	private	string	Поле, задающее единое конечное состояние.	
graph	private	GraphBuilder	Поле, задающее автомат, у которого удаляются состояния.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
StateRemoval	public	конструктор	GraphBuilder changhedGraph,List<string> list, string headState, string unqueFinal	Конструктор шага удаления состояния.
isUntouched	public	bool	string vertexName	Метод, который проверяет, является ли состояние конечным или начальным.
TransformGraph	public	GraphBuilder	out string uniqueFinal	Метод, добавляющий в автомат единое конечное состояние.
NonFinalLeft	public	bool	out string name	Метод, проверяющий, остались ли в автомате неконечные состояния, при положительном результате возвращает имя состояния, которое необходимо удалить.
RemoveNonFinalState	public	GraphBuilder	string nameOfVertex	Метод, реализующий алгоритм удаления недопускающего состояния.
DefinePrefixes	private	void	Dictionary<string, int> prefixes	Метод извлечения подпроцессов по префиксам.

DefineSubprocesses	private	void	-	Метод извлечения подпроцессов в составе переходов на дугах конечного автомата.
CountOccurance	private	int	string str, string substr	Метод, подсчитывающий количество повторений подстроки в строке.
SortByLength	private	List<string>	List<string> list	Метод, сортирующий список строк по убыванию их длины.
RemoveIdentic	private	string	string str, string substr, string change	Метод, заменяющий подпроцессы в регулярном выражении на их ключевые символы.
RandomVertexName	public	string	-	Поиск свободного имени для состояния.
DeleteBrackets	public	List<string>	List<string> expressions	Метод, удаляющий лишние скобки в регулярных выражениях.
FinalRemoval	public	List<string>	GraphBuilder graph	Метод, осуществляющий финальное преобразование конечного автомата в регулярное выражение.
Change	public	string	string expression	Метод, удаляющий пустые переходы.

Описание полей методов и свойств класса `OutEdgeList`

Таблица 5.2

	Свойства			
Имя	Модификатор доступа	Тип	Доступ	Назначение
EdgeLabel	public	string	get, set	Регулярное выражение, задающее переход.
TargetVertex	public	string	get, set	Состояние входа.

Описание полей методов и свойств класса `InEdgeList`

Таблица 5.3

Свойства				
Имя	Модификатор доступа	Тип	Доступ	Назначение
EdgeLabel	public	string		Регулярное выражение, задающее переход.
SourceVertex	public	string		Состояние исхода.

Описание полей, методов и свойств класса `DrawAssistant`

Таблица 5.4

Поля				
Имя	Модификатор доступа	Тип	Назначение	
edgeList	private	List<Edge>	Список ребер графа.	
vertexList	private	List<Vertex>	Список вершин графа.	
finalsList	private	List<string>	Список конечных состояний графа.	
startingState	private	string	Начальное состояние графа.	
Rad	private	int	Радиус вершины графа.	
Свойства				
Имя	Модификатор доступа	Тип	Назначение	
StartingState	public	string	Свойство, возвращающее начальное состояние автомата.	
Radius	public	int	Свойство, возвращающее радиус состояния автомата.	
Image	public	Bitmap	Свойство, хранящее изображение построенного автомата.	
Finals	public	List<string>	Свойство, хранящее список конечных состояний автомата.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
DrawAssistant	public	конструктор	int width, int height, List<Edge> edgeList, List<Vertex> vertexList, List<string> finals	Конструктор класса DrawAssistant.
ChangeFinalsList	public	void	string name	Метод, изменяющий список конечных состояний автомата.
DrawSelectedVertex	public	void	int x, int y,string name	Метод, выделяющий вершину, выбранную как исходную для ребра.
DrawVertexToDelete	public	void	int x, int y, string name	Метод, выделяющий вершину, выбранную для удаления.
DrawVertex	public	void	int x, int y, string name	Метод, выполняющий отрисовку вершины.

DrawEdge	public	void	int x1, int y1, int x2, int y2, string name, Edge edge	Метод, выполняющий отрисовку ребра.
Draw	public	void	-	Метод, выполняющий отрисовку графа по списку ребер и вершин.
ReverseEdge	private	bool	Edge edge	Метод, определяющий, есть ли обратное к данному ребро.
ClearSheet	public	void	-	Метод, очищающий лист для рисования.
DeleteVertex	public	void	Vertex vertex	Метод, удаляющий вершину из списка вершин.
DeleteEdge	public	void	Edge edge	Метод, удаляющий ребро из списка ребер.
EdgeHitten	public	bool	ref Edge edge, int x, int y	Метод, определяющий, нажал ли пользователь на ребро.
CheckEdgeHitten	public	bool	Edge edge, int x, int y	Метод, определяющий, принадлежит ли данная точка ребру.
VertexHitten	public	bool	ref Vertex vertex, int x, int y, List<Vertex> vertices	Метод, определяющий, нажал ли пользователь на вершину.

Описание полей методов и свойств класса Vertex

Таблица 5.5

Свойства			
Имя	Модификатор доступа	Тип	Назначение
Name	public	string	Имя состояния.
X	public	int	Координата вершины графа по оси X.
Y	public	int	Координата вершины графа по оси Y.
Методы			
Имя	Модификатор доступа	Тип	Аргументы
Vertex	public	конструктор	string name, int x, int y

Описание полей методов и свойств класса Edge

Таблица 5.6

Свойства			
Имя	Модификатор доступа	Тип	Назначение

Transition	public	string	Символ перехода.
Source	public	Vertex	Вершина исхода.
Target	public	Vertex	Вершина входа.
Методы			
Имя	Модификатор доступа	Тип	Аргументы
Edge	public	конструктор	Vertex source, Vertex target, string tag

Описание полей методов и свойств класса GraphBuilder

Таблица 5.7

Поля				
Имя	Модификатор доступа	Тип	Назначение	
finalStates	private	List<string>	Поле, список конечных состояний.	
basicDefinitions	public	Dictionary<string, string>	Словарь процессов.	
finalDefinitions	public	Dictionary<string, string>	Словарь подпроцессов.	
myGraph	public	AdjacencyGraph<string, TaggedEdge<string, string>>	Пользовательский конечный автомат.	
Свойства				
Имя	Модификатор доступа	Тип	Назначение	
NumberOfGraphs	public	int	Количество совершенных перобразований автомата.	
FinalStates	public	List<string>	Свойство, список конечных состояний.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
InitializeFinalStates	public	void	List<string> finStates	Метод, инициализирующий список конечных состояний.
ConcatSelfLoops	public	void	-	Метод, объединяющий все петли состояния.
UserAddVertix	public	void	string nameVert	Метод, добавляющий в автомат состояние.
UserRemoveFinalVertix	public	void	string nameVert	Метод, удаляющий конечное состояние автомата из списка.
UserAddEdge	public	void	string startVert, string endVert, string edgeText	Метод, добавляющий в автомат переход.

UserAddFinalVertix	public	void	string nameVert	Метод, добавляющий в автомат конечное состояние.
MakeDot	public	string	int graphNum	Метод, преобразующий описание графа в картинку.
FormVertex	private	void	object sender, FormatVertexEventArgs<string> vert	Метод, задающий формат вершины графа.
FormEdge	private	void	object sender, FormatEdgeEventArgs<string, TaggedEdge<string, string>> edge	Метод, задающий формат ребра графа.

Описание полей методов и свойств класса FileDotEngine

Таблица 5.8

Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
Run	public	string	GraphvizImageType imageType, string dot, string outputFileName	Получение изображения автомата.

Описание полей методов и свойств класса ChoiceFSMForm

Таблица 5.9

Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
ChoiceFSMForm	public	конструктор	-	Конструктор окна ChoiceFSMForm.
buttonCreate_Click	private	void	object sender, EventArgs e	Метод, обрабатывающий желание пользователя создать конечный автомат и открывающий окно отрисовки автомата.
aboutToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, открывающий окно с информацией о разработчике.
buttonFromFile_Click	private	void	object sender, EventArgs e	Метод, обрабатывающий желание пользователя загрузить автомат из стороннего ресурса и открывающий окно открытия файла.
ChoiceFSMForm_FormClosed	private	void	object sender, FormClosedEventArgs e	Метод, завершающий работу приложения при закрытии окна.

Описание полей методов и свойств класса FormBefore

Таблица 5.10

Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
FormBefore	public	конструктор	List<string> regularExpressions, Dictionary<string,string> finals	Конструктор формы FormBefore.
Change	private	string	string expression	Изменение регулярного выражения. (Удаление символов, добавленных вместе с единым конечным состоянием).
FormBefore_FormClosing	private	void	object sender, FormClosingEventArgs e	Обработка закрытие формы. (Объект формы не должен удаляться).
Свойства				
Имя	Модификатор доступа	Тип	Назначение	
BeforeCount	public	string	Длина регулярного выражения до минимизации.	
MaxNesting	public	string	Максимальная вложенность.	

Описание полей методов и свойств класса FormProcessDefinitions

Таблица 5.11

Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
FormProcessDefinitions	public	конструктор	Dictionary<string, string> basics	Заполнение таблицы словаря определениями процессов.
FormProcessDefinitions	public	конструктор	Dictionary<string,string> basics, Dictionary<string,string> finals	Заполнение таблицы словаря определениями подпроцессов.
Change	private	string	string expression	Изменение регулярного выражения. (Удаление символов, добавленных вместе с единым конечным состоянием).

FormProcessDefinitions_FormClosing	private	void	object sender, FormClosingEventArgs e	Обработка закрытие формы. (Объект формы не должен быть удален).
FormProcessDefinitions_SizeChanged	private	void	object sender, EventArgs e	Обработка изменения размеров формы.
Свойства				
Имя	Модификатор доступа	Тип	Назначение	
MidLength	public	string	Средняя длина подпроцессов.	
tableContents	public	List<string>	Содержимое таблицы - словаря определений.	

Описание полей методов и свойств класса FormInfo

Таблица 5.12

Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
FormInfo	public	конструктор	-	Конструктор окна FormInfo.

Описание полей методов и свойств класса Draw

Таблица 5.13

Поля			
Имя	Модификатор доступа	Тип	Назначение
drawer	private	DrawAssistant	Инструмент, оперирующий операциями отрисовки.
graph	private	GraphBuilder	Построенный граф, представляющий конечный автомат.
addingVertices	private	bool	Истинно, если происходит добавление состояний.
addingEdges	private	bool	Истинно, если происходит добавление переходов.
movingVertex	private	bool	Истинно, если происходит передвижение вершины.
selectingStart	private	bool	Истинно, если происходит выбор начального состояния.
correctingFinalsList	private	bool	Истинно, если происходит изменение списка конечных вершин.
moovingName	private	string	Имя текущего передвигаемого состояния.
sourceVertex	private	Vertex	Вершина, из которой исходит ребро.
targetVertex	private	Vertex	Вершина, в которую входит ребро.
movedVertex	private	Vertex	Перемещаемая вершина.
vertexList	private	List<Vertex>	Список вершин.
edgeList	private	List<Edge>	Список ребер.
movingEdgeList	private	List<Edge>	Список ребер, которые нужно переместить.
finalsList	private	List<string>	Список конечных состояний.

startingState	private	string	Начальное состояние.	
newProjectForm	private	ChoiceFSMForm	Стартовое окно.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
Draw	public	конструктор	ChoiceFSMForm form	Конструктор окна Draw.
buttonVertex_Click	private	void	object sender, EventArgs e	Обработка желания пользователя добавить вершину.
buttonEdge_Click	private	void	object sender, EventArgs e	Обработка желания пользователя добавить ребро.
pictureBoxAutomata_MouseClick	private	void	object sender, MouseEventArgs e	Обработка нажатия на поле для рисования. Выполняет операции рисования в зависимости от того, какое действие выполняет пользователь в данный момент.
CompleteAddingEdge	public	void	string name	Завершение добавление ребра после введения текста перехода.
SetXY	private	void	out int x, out int y, MouseEventArgs e	Метод выполняющий проверку, не пытается ли пользователь переместить объект за поле для рисования.
pictureBoxAutomata_MouseUp	private	void	object sender, System.Windows.Forms.MouseEventArgs e	Метод, обрабатывающий разжатие кнопки мыши. Выполняет операции рисования в зависимости от того, какое действие выполняет пользователь в данный момент.
pictureBoxAutomata_MouseMove_2	private	void	object sender, MouseEventArgs e	Метод, обрабатывающий движение мыши. Выполняет операции рисования в зависимости от того, какое действие выполняет пользователь в данный момент.
pictureBoxAutomata_MouseDown	private	void	object sender, MouseEventArgs e	Метод, обрабатывающий зажатие кнопки мыши. Выполняет операции рисования в зависимости от того, какое действие выполняет пользователь в данный момент.
buttonStartingVertex_Click	private	void	object sender, EventArgs e	Метод, обрабатывающий желание пользователя назначить начальное состояние.
buttonAddFinal_Click	private	void	object sender, EventArgs e	Метод, обрабатывающий желание пользователя отредактировать список конечных состояний.
buttonFinish_Click	private	void	object sender, EventArgs e	Метод, обрабатывающий желание пользователя закончить построение автомата.
aboutToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, открывающий окно с информацией о разработчике.

newProjectToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, открывающий стартовое окно приложения.
saveToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, сохраняющий построенный автомат в формате .tsml.
Draw_FormClosing	private	void	object sender, FormClosingEventArgs e	Метод, обрабатывающий закрытие формы. Предупреждает о возможной потери данных.

Описание полей методов и свойств класса FormRegularExpression

Таблица 5.14

Поля				
Имя	Модификатор доступа	Тип	Назначение	
graph	private	GraphBuilder	Конечный автомат, по которому было построено регулярное выражение.	
formBasicsDictionary	private	FormProcessDefinitions	Окно, содержащее определение процессов.	
formFinalsDictionary	private	FormProcessDefinitions	Окно, содержащее определение подпроцессов.	
formBefore	private	FormBefore	Окно, содержащее регулярное выражение, полученное по конечному автомату без использования алгоритма минимизации регулярного выражения.	
formSavingRules	private	FormSavingRules	Окно, позволяющее пользователю задать параметры сохранения.	
regularExpression	private	List<string>	Строки, хранящие регулярное выражение.	
newProjectForm	private	ChoiceFSMForm	Стартовое окно приложения.	
Свойства				
Имя	Модификатор доступа	Тип	Назначение	
textBoxContents	private	List<string>	Свойство, считывающее содержимое richTextBoxRegularExpression. (Регулярное выражение).	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
FormRegularExpression	public	конструктор	List<string> regularExpression, GraphBuilder graph, ChoiceFSMForm form	Конструктор окна FormRegularExpression.
buttonDefinitions_Click	private	void	object sender, EventArgs e	Метод, открывающий окно с определениями подпроцессов.

buttonBasics_Click	private	void	object sender, EventArgs e	Метод, открывающий окно с определениями процессов.
buttonBefore_Click	private	void	object sender, EventArgs e	Метод, открывающий окно с регуляным выражением, полученным по конечному автомату без использования алгоритма минимизации регулярного выражения.
FormRegularExpression_FormClosing	private	void	object sender, FormClosingEventArgs e	Метод, обрабатывающий закрытие формы. Предупреждает о возможной потери данных.
aboutToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, открывающий окно с информацией о разработчике.
newProjectToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, открывающий стартовое окно приложения.
saveToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, обрабатывающий желание пользователя сохранить данные, полученные в результате работы алгоритма, и запускающий выбор параметров сохранения.
CompleteSaving	public	void	-	Метод, завершающий сохранение данных в зависимости от выбранных пользователем параметров сохранения.

Описание полей методов и свойств класса FormRemoveStatesWithAnimation

Таблица 5.15

Поля			
Имя	Модификатор доступа	Тип	Назначение
transformed	private	bool	Поле - индикатор, было ли добавлено единое конечное состояние.
picShown	private	int	Поле, хранящее номер открытого шага анимации.
picsMade	private	int	Поле, хранящее номер последнего проделанного шага анимации.
graph	private	GraphBuilder	Поле, хранящее текущее состояние автомата.
headState	private	string	Поле, хранящее начальное состояние автомата.
imagePath	private	string	Поле, хранящее путь к изображению с иллюстрацией шага анимации.
unqueFinal	private	string	Поле, хранящее имя единого конечного состояния.
count	private	int	Поле, хранящее номер шага анимации, который будет присвоен.
state	private	StateRemoval	Поле, представляющее класс с реализацией алгоритма удаления состояний.

newProjectForm	private	ChoiceFSMForm	Стартовое окно приложения.	
untouchedStatesList	private	List<string>	Поле, хранящее список состояний, которые нельзя удалять (начальное и конечные).	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
FormRemoveStatesWithAnimation	public	Конструктор	GraphBuilder graph, List<string> list, string headState, string imagePath, ChoiceFSMForm form	Конструктор окна FormRemoveStatesWithAnimation.
buttonRemoveNext_Click	private	void	object sender, EventArgs e	Метод, открывающий следующий шаг анимации.
buttonPreviousGraph_Click	private	void	object sender, EventArgs e	Метод, открывающий предыдущий шаг анимации.
newProjectToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, открывающий стартовое окно приложения.
buttonSkip_Click	private	void	object sender, EventArgs e	Метод, пропускающий все шаги анимации, начиная с текущего, и открывающий окно с регулярным выражением.
FormRemoveStatesWithAnimation_FormClosing	private	void	object sender, FormClosingEventArgs e	Метод, обрабатывающий закрытие формы. Предупреждает о возможной потери данных.
saveToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, сохраняющий картинку с иллюстрацией к текущему шагу анимации.
aboutToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, открывающий окно с информацией о разработчике.

Описание полей методов и свойств класса FormSavingRules

Таблица 5.16

Поля			
Имя	Модификатор доступа	Тип	Назначение
afterAlgo	private	bool	Поле - индикатор, желает ли пользователь сохранить регулярное выражение, полученное с применением алгоритма минимизации.
beforeAlgo	private	bool	Поле - индикатор, желает ли пользователь сохранить регулярное выражение, полученное без применения алгоритма минимизации.
basicDefinitions	private	bool	Поле - индикатор, желает ли пользователь сохранить определения процессов.

processDefinitions	private	bool	Поле - индикатор, желает ли пользователь сохранить определения подпроцессов.	
analytics	private	bool	Поле - индикатор, желает ли пользователь сохранить аналитические данные.	
savingCompleted	private	bool	Поле - индикатор, было ли завершено сохранение.	
formRE	private	FormRegularExpression	Окно, содержащее регулярное выражение, полученное из конечного автомата.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
FormSavingRules	public	Конструктор	FormRegularExpression form	Конструктор окна FormSavingRules.
buttonSaveChosen_Click	private	void	object sender, EventArgs e	Метод, вызываемый при желании пользователя сохранить некоторые из доступных аналитических данных.
buttonAddAll_Click	private	void	object sender, EventArgs e	Метод, вызываемый при желании пользователя сохранить абсолютно все аналитические данные.
MarkChecked	private	void	-	Метод, задающий параметры сохранения в зависимости от выбранных пользователем характеристик, и запускающий процесс сохранения.
Свойства				
Имя	Модификатор доступа	Тип	Доступ	Назначение
AfterAlgo	public	bool	get	Свойство, делающее поле afterAlgo доступным для чтения.
BeforeAlgo	public	bool	get	Свойство, делающее поле beforeAlgo доступным для чтения.
BasicDefinitions	public	bool	get	Свойство, делающее поле basicDefinitions доступным для чтения.
ProcessDefinitions	public	bool	get	Свойство, делающее поле processDefinitions доступным для чтения.
Analytics	public	bool	get	Свойство, делающее поле analytics доступным для чтения.
SavingCompleted	public	bool	get	Свойство, делающее поле savingCompleted доступным для чтения.

Описание полей методов и свойств класса FormUsersFSM

Таблица 5.17

Поля				
Имя	Модификатор доступа	Тип	Назначение	
graph	private	GraphBuilder	Конечный автомат, с которым необходимо работать.	
untouchedStatesList	private	List<string>	Список из состояний, которые нельзя удалять на начальном этапе.	
headState	private	string	Начальное состояние автомата.	
success	private	bool	Индикатор, была ли успешно начата работа с графом.	
formChoice	private	ChoiceFSMForm	Стартовое окно приложения.	
imagePath	private	string	Путь к изображению графа.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
FormUsersFSM	public	Конструктор	GraphBuilder graph, List<string> finalStates, string headState, bool skip, ChoiceFSMForm form	Конструктор вызывается из формы с анимацией, чтобы пропустить дальнейшие шаги анимации.
FormUsersFSM	public	Конструктор	GraphBuilder graph, List<string> finalStates, string headState,ChoiceFSMForm form	Конструктор вызывается, когда пользователь только начал работу с автоматом.
MinimizeGraph	private	void	GraphBuilder graph, List<string> finalStates, string headState	Метод, запускающий минимизацию автомата.
buttonConstructWithAnimation_Click	private	void	object sender, EventArgs e	Метод, запускающий построение регулярного выражения с анимацией при нажатии соответствующей кнопки.
buttonConstructExpression_Click	private	void	object sender, EventArgs e	Метод, запускающий построение регулярного выражения без анимации при нажатии соответствующей кнопки.
newProjectToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, открывающий стартовое окно приложения.
ConstructRE	private	List<string>	-	Метод, производящий построение регулярного выражения по

				конечному автомату без предоставления анимации.
FormUsersFSM_FormClosing	private	void	object sender, FormClosingEventArgs e	Метод, обрабатывающий закрытие формы. Предупреждает о возможной потери данных.
aboutToolStripMenuItem_Click	private	void	object sender, EventArgs e	Метод, открывающий окно с информацией о разработчике.
Свойства				
Имя	Модификатор доступа	Тип	Доступ	Назначение
Success	private	bool	get	Свойство, предоставляющее доступ к полю success.

Описание полей методов и свойств класса FormTransitionName

Таблица 5.18

Поля				
Имя	Модификатор доступа	Тип	Назначение	
form	private	Draw	Окно отрисовки.	
transition	private	string	Введенное регулярное выражение, задающее переход.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
FormTransitionName	public	Конструктор	Draw form	Конструктор окна FormTransitionName.
FormTransitionName_FormClosed	private	void	object sender, FormClosedEventArgs e	Добавляет переход перед закрытием формы.
CheckInput	private	bool	string input	Проверяет корректность введенного перехода.
buttonAdd_Click	private	void	object sender, EventArgs e	Обрабатывает запрос пользователя на добавление перехода.

Описание полей методов и свойств класса Minimizator

Таблица 5.19

Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
Minimizator	public	Конструктор	List<string> finalStates, string startingState, GraphBuilder graph	Конструктор, задающий минимизируемый автомат.

FindFreeClassName	private	int	-	Метод, который находит минимальное свободное число для номера класса разбиения.
Split	private	bool	int key, string symbol, int equivalenceClass	Метод, разбивающий один класс эквивалентности.
Divide	private	void	-	Метод, реализующий алгоритм разбиения множества состояний на минимальные подмножества.
Minimize	public	GraphBuilder	-	Метод, реализующий построение минимизированного автомата.
Свойства				
Имя	Модификатор доступа	Тип	Доступ	Назначение
StartingState	public	string	get, set	Свойство доступа к полю startingState.
Поля				
Имя	Модификатор доступа	Тип	Назначение	
StateInfoDictionary	private	Dictionary<string, List<OutRange>>	Поле, хранящее словарь исходящих дуг, ключом является состояние, значением - список дуг.	
classMembers	private	Dictionary<int, List<string>>	Поле, хранящее словарь классов эквивалентности, ключом является класс, значением - список состояний.	
possibleTransitions	private	List<string>	Поле, хранящее список всех возможных переходов (алфавит).	
queue	private	Queue<Splitter>	Поле, представляющее очередь из пар типа (символ перехода) - (класс).	
acceptStates	private	List<string>	Поле, хранящее первоначальный список допускающих состояний.	
startingState	private	string	Поле, хранящее начальное состояние.	

Описание полей методов и свойств класса `Splitter`

Таблица 5.20

Поля				
Имя	Модификатор доступа	Тип	Назначение	
symb	private	string	Поле, хранящее символ, по которому идет разбиение.	
equivalenceClass	private	int	Поле, хранящее номер класса эквивалентности.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение

Splitter	public	Конструктор	string symbol, int classNum	Конструктор, задающий сплиттер.
Свойства				
Имя	Модификатор доступа	Тип	Доступ	Назначение
EquivalenceClass	public	int	get	Свойство, делающее поле symb доступным для чтения.

Описание полей методов и свойств класса OutRange

Таблица 5.21

Поля				
Имя	Модификатор доступа	Тип	Назначение	
transition	private	string	Поле, хранящее символ перехода.	
distinationState	private	string	Поле, хранящее состояние в которое осуществляется переход.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
OutRange	public	Конструктор	string symb, string state	Конструктор, задающий исходящую из состояния дугу.
Свойства				
Имя	Модификатор доступа	Тип	Доступ	Назначение
Transition	public	string	get	Свойство, делающее поле transition доступным для чтения.
DictinationState	public	string	get	Свойство, делающее поле distinationState доступным для чтения.

Описание полей методов и свойств класса UsersFiles

Таблица 5.22

Поля				
Имя	Модификатор доступа	Тип	Назначение	
myStream	private	Stream	Файловый поток.	
filePath	private	string	Поле, хранящее путь к открытому пользователем файлу.	
myOpenFileDialog	private	OpenFileDialog	Диалоговое окно открытия файла.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
DeleteFiles	public	void	-	Метод, удаляющий папку, созданную во время работы программы.

OpenFile	public	bool	-	Метод, получающий путь к открытому пользователем файлу.
ChooseSavingPath	public	string	string format	Метод, получающий путь к файлу, в который нужно сохранить информацию.
SaveImage	public	void	string path, Bitmap bmp	Метод, осуществляющий сохранение изображения.
SaveMinedExpression	public	void	List<string> afterAlgo, List<string> beforeAlgo, List<string> basicDefinitions, List<string> processDefinitions, List<string> analytics	Метод, осуществляющий сохранение аналитических данных, полученных в результате работы алгоритма.
Свойства				
Имя	Модификатор доступа	Тип	Доступ	Назначение
FilePath	public	string	get	Свойство доступа к полю filePath.

Описание полей методов и свойств класса TSMLtransformer

Таблица 5.23

Поля				
Имя	Модификатор доступа	Тип	Назначение	
stateID	private	Dictionary<string, string>	Словарь состояний. Ключом является имя состояния, значением - ее идентификационное имя.	
startingState	private	string	Начальное состояние автомата.	
finalStates	private	List<string>	Список конечных состояний автомата.	
doc	private	XmlDocument	Документ в формате .tsml.	
graph	private	GraphBuilder	Конечный автомат.	
textWritter	private	XmlTextWriter	Поле для записи информации в формате .xml.	
Методы				
Имя	Модификатор доступа	Тип	Аргументы	Назначение
TSMLtransformer	public	Конструктор	string filePath,ChoiceFSMForm formChoice, bool opened	Конструктор, вызываемый при необходимости преобразовать конечный автомат, заданный в формате .tsml в формат, используемый программой.
TSMLtransformer	public	Конструктор	string path, List<Vertex> stateList,List<string>	Конструктор, вызываемый при необходимости преобразовать конечный автомат, заданный в

			finalStateList, string startingState, List<Edge> edgeList	формате формат, используемым программой в формат .tsml.
CreateTSML	private	void	string path, List<Vertex> stateList, List<string> finalStateList, string startingState, List<Edge> edgeList	Метод, выполняющий сохранение конечного автомата в формате .tsml.
GetNodeList	public	List<TSMlVertex>	XmlElement root	Метод, получающий список состояний автомата из корневого элемента tsml файла.
GetEdgeList	public	void	XmlElement root, List<TSMlVertex> nodes	Метод, получающий список переходов автомата из корневого элемента tsml файла.
AppendState	public	void	string name, int num, bool isFinal, bool isStarting	Метод, записывающий в файл формата .tsml информацию о состоянии автомата.
AppendEdge	public	void	string name, int num, string source, string target	Метод, записывающий в файл формата .tsml информацию о переходе автомата.

Описание полей методов и свойств класса TSMlVertex

Таблица 5.24

Свойства				
Имя	Модификатор доступа	Тип	Доступ	Назначение
name	public	string	get, set	Имя вершины.
id	public	string	get, set	Идентификационное имя вершины.

ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

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

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.502610-01 81				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата