

# Технически университет – Варна

Факултет: ФИТА

Катедра: Софтуерни и интернет технологии (СИТ)

Специалност: СИТ

Тема на проекта:

Недетерминиран краен автомат

Изготвил:

Име: Теодор Николаев Николов

Факултетен номер: 20621504

## Структура на документацията

1. Увод
  - a. Задание
2. Преглед на предметната област
  - a. Основни дефиниции, концепции и алгоритми
  - b. Дефиниране на проблем и сложност на поставената задача
  - c. Подходи, методи за решаване на поставените проблеми
3. Проектиране
  - a. Обща структура на проекта
  - b. Блок схема
4. Реализация и тестване
  - a. Реализация на класове
  - b. Алгоритми и оптимизация
  - c. Планиране, описание на тестови сценарии
5. Заключение
  - a. Обобщение на изпълнението на началните цели
  - b. Насоки за бъдещо развитие и усъвършенстване

Линк към github: [https://github.com/tedonikolov/OOP1\\_project](https://github.com/tedonikolov/OOP1_project)

Използвана литература

1. [Материали по учебна дисциплина Дискретни структури.](#)
2. [Записки по „Езици, автомати, изчислимост” - Стефан Вълчев, ФМИ-СУ](#)
3. [Youtube клип за направата на конзола като Command Prompt](#)
4. [Youtube клип за записване на XML файлове чрез Javabeans](#)
5. [JavaBeans Wikipedia](#)
6. [Youtube клип за работа с JavaDoc](#)

## 1. Увод

### а. Задание:

## Недетерминиран краен автомат

Да се реализира програма, която поддържа операции с недетерминиран краен автомат с  $\epsilon$ -преходи. над азбука, състояща се от цифрите и малките латински букви.

Автоматите да се сериализират по разработен от Вас формат. Всеки прочетен автомат да получава уникален идентификатор.

След като приложението отвори даден файл, то трябва да може да извършва посочените по-долу операции, в допълнение на общите операции (open, close, save, save as, help и exit):

list	Списък с идентификаторите на всички прочетени автомати
print <id>	Извежда информация за всички преходи в автомата
save <id> <filename>	Записва автомат във файл
empty <id>	Проверява дали езикът на автомата е празен
deterministic <id>	Проверява дали автомат е детерминиран
recognize <id> <word>	Проверява дали дадена дума е в езика на автомата
union <id1> <id2>	Намира обединението на два автомата и създава нов автомат. Отпечатва идентификатора на новия автомат
concat <id1> <id2>	Намира конкатенацията на два автомата и създава нов автомат. Отпечатва идентификатора на новия автомат
un <id>	Намира позитивна обвивка на автомат и създава нов автомат. Отпечатва идентификатора на новия автомат
reg <regex>	Създава нов автомат по даден регулярен израз (теорема на Клини). Отпечатва идентификатора на новия автомат

- да се реализира мутатор, който детерминира даден автомат
- да се реализира операция, която проверяват дали езикът на даден автомат е краен

## Работа с командния ред

Вашата програма трябва да позволява на потребителя да отваря файлове (open), да извършва върху тях някакви операции, след което да записва промените обратно в същия файл (save) или в друг, който потребителят посочи (save as). Трябва да има и опция за затваряне на файла, без записване на промените (close). За целта, когато програмата ви се стартира, тя трябва да позволява на потребителя да въвежда команди и след това да ги изпълнява.

Когато отворите даден файл, неговото съдържание трябва да се зареди в паметта, след което файлът се затваря. Всички промени, които потребителят направи след това трябва да се пазят в паметта, но не трябва да се записват обратно, освен ако потребителят изрично не укаже това.

### 2. Преглед на предметната област

#### а. Основни дефиниции, концепции и алгоритми

Краен автомат – най-простият вид абстрактен автомат. Описва се чрез ориентиран граф. Математически се описват със следната петорка  $M = \{S, E, f, S_0, F\}$ .

$S$  – крайно множество от състояния

$E$  – крайна входна азбука (език)

$S_0$  – начално състояние

$F$  – множество от крайните състояния

$f$  – функция на прехода:  $S \times E \rightarrow S$

Ориентиран граф – граф, на който всичките му ребра имат посока.

Детерминиран краен автомат – най-много един преход от дадено състояние за всяка буква от азбуката на автомата.

Азбука (език) – непразно крайно множество от символи.

Регулярен израз – правило за точно дефиниране на множества от низове, които са валидни в даден формален език.

Формален език – множество от низове с крайна дължина на азбуката.

Низ – дума образувана от крайна последователност от букви част от азбуката.

Теорема на Клини – за всеки регулярен език, съществува ДКА, който разпознава същото множество от низове.

б. Дефиниране на проблем и сложност на поставената задача

Проблем 1: Да се реализира програма, която поддържа операции с недетерминиран краен автомат с  $\epsilon$ -преходи. над азбука, състояща се от цифрите и малките латински букви.

Проблем 2: Автоматите да се сериализират по разработен от Вас формат. Всеки прочетен автомат да получава уникален идентификатор.

Сложност на поставената задача:

Високо ниво на абстрактно и логическо мислене за реализиране на програмата.

с. Подходи, методи за решаване на поставените проблеми

Решаване на проблем 1:

Вникване в същността на крайния автомат. Използване на следната математическа петорка  $M = \{S, E, f, S_0, F\}$ . За тази цел създаваме класове: *State* – представляващ едно състояние; *Symbol* – представляващ един символ; *Function* – представляващ една функция на прехода. Класът *Automation* представлява крайния автомат. Той съдържа следните атрибути: *states* – списък (множество) от състоянията на автомата; *alphabet* – списък (множество) от символите на автомата; *functions* – списък (множество) от функции на прехода; *startState* – началното състояние на автомата; *finaleStates* – списък (множество) от крайни състояния на автомата.

Решаване на проблем 2:

Създаване на клас *Machines*, който съдържа следните атрибути: *id* – текущия брой автомати; *Automations* – съдържа автомата с неговия уникален идентификатор.

### 3. Проектиране

а. Обща структура на проекта

Клас *Console* създава конзола на програмата с метод *print()* за принтиране върху конзолата.

Клас *Commands* съдържа главните команди, които програмата изпълнява като: *open*, *close*, *save*, *help*, *exit*, *submenu*.

Клас *Menu* представлява менюто на програмата, като менюто е скрито за потребителя.

Клас *State* представлява едно състояние на автомата.

Клас *Symbol* представлява един символ от автомата.

Клас *Function* представлява една функция на прехода.

Клас *Automation* представлява крайния автомат.

Клас *Machines* представлява списък от автомати с техните идентификатори.

Клас *SaveAutomation* записва автомата в два вида файл: xml и txt.

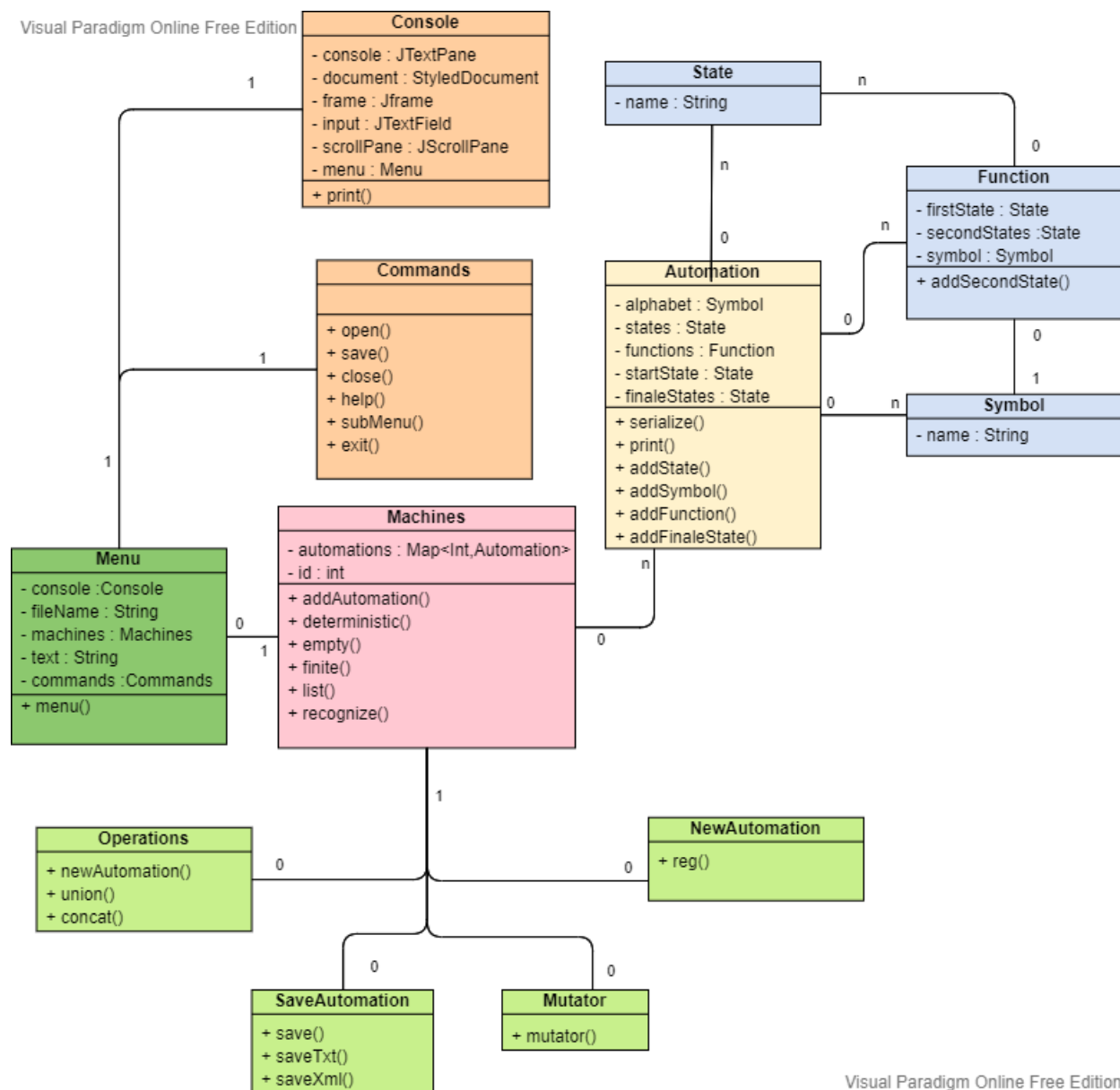
Клас *Operations* намира обединението и конкатенацията на два автомата и създава нов краен автомат.

Клас *NewAutomation* създава нов автомат по даден регулярен израз (теорема на Клини).

Клас *Mutator* превръща недетерминиран краен автомат в детерминиран.

Клас *Main* вика конзолата на програмата.

## б. Блок схема



## 4. Реализация и тестване

### а. Реализация на класове

Сереализирането на автоматите става чрез таблица на преходите. За тази цел клас *Automation* притежава метод *serialize()* *фигура 1*, който обхожда състоянията на автомата и за всяко състояние вика подметод *print()* *фигура 2*. Там от своя страна се обхождат функциите за съответното състояние и се събират на един ред от таблицата чрез *StringBuilder*.

```
public StringBuilder serialize() {
    StringBuilder string= new StringBuilder("\t\t\t");
    for(Symbol symbol:alphabet){
        string.append(symbol).append("\t");
    }
    string.append("\nSTART->\t{").append(print(startState)).append("\n");

    for(State state:states) {
        boolean flag=state != startState;
        for (State finaleState : finaleStates) {
            flag= state != startState && state != finaleState;
            if(!flag){
                break;
            }
        }
        if(flag){
            string.append("\t\t{").append(print(state)).append("\n");
        }
    }

    for (State state:finaleStates){
        string.append("    END->\t{").append(print(state)).append("\n");
    }
    return string;
}
```

Фигура 1

Фигура 2

```

public StringBuilder print(State state) {
    StringBuilder stringBuilder=new StringBuilder();
    stringBuilder.append(state).append("}\t");
    for (Symbol symbol : alphabet) {
        boolean flag = false;
        for (Function function : functions) {
            if (function.getFirstState() == state) {
                if (symbol == function.getSymbol()) {
                    int i=1;
                    stringBuilder.append("{");
                    for (State state1:function.getSecondStates()) {
                        if(i>1)
                            stringBuilder.append(",");
                        stringBuilder.append(state1);
                        i++;
                    }
                    stringBuilder.append("}\t");
                    flag = true;
                }
            }
        }
        if (!flag) {
            stringBuilder.append("-").append("\t");
        }
    }
    return stringBuilder;
}

```



Класът `Commands` реализира операциите за работа с командния ред, като това са методите `open` (фигура 3), `close`, `save` (фигура 4), `help`, `exit`, `submenu`.

За отварянето на `xml` файлове в програмата се използва `decoder`, който се извиква въз основа на `JavaBeans`.

За запаметяването на `xml` файл се използва `Encoder`, който се извиква въз основа на `JavaBeans`.

```
public Machines open(Console console, String fileName) throws IOException, BadLocationException {
    Machines machines = new Machines();
    File file = new File(fileName);
    if(file.exists()) {
        FileInputStream fileOpen = new FileInputStream(fileName);
        if(fileOpen.available() != 0){
            XMLDecoder decoder = new XMLDecoder(fileOpen);
            machines = (Machines) decoder.readObject();
            decoder.close();
            fileOpen.close();
        }
        console.print("Successfully opened " + fileName);
    }
    else {
        boolean newFile = file.createNewFile();
        console.print("Successfully created "+fileName);
    }
    return machines;
}
```

Фигура 3

```
public void save(Console console, String fileName, Machines machines) throws IOException, BadLocationException {
    FileOutputStream file= new FileOutputStream(fileName);
    XMLEncoder encoder = new XMLEncoder(file);
    encoder.writeObject(machines);
    encoder.close();
    file.close();
    console.print("Successfully saved "+fileName);
}
```

Фигура 4

Класът *Machines* реализира по-кратките и лесни операции за работа с автомати, като това са методите *list*, *print*, *empty*, *deterministic*, *recognize*, *finite*.

В метод *recognize* първо проверяваме дали първият символ съответства на някоя функция на стартовото състояние, ако не връща, че думата не е от автомата (*фигура 5*). Ако обаче е, започва поетапна проверка на всички състояния и техните функции със съответния следващ символ от думата. Като има множество проверки за по-ранно прекъсване на метода, ако някое условие е грешно изпълнено.

```
if(flag) {
    for(int i=k; i<symbol.length; i++){
        for (Function function: automation.getFunctions()){
            flag=false;
            for(State state:states){
                if(state==function.getFirstState()){
                    if(Objects.equals(symbol[i], function.getSymbol().getName())) {
                        states=function.getSecondStates();
                        flag=true;
                        break;
                    }
                }
            }
            if(flag)
                break;
        }
        if(!flag)
            break;
    }
    if (flag) {
        for (State state : states) {
            for (State lastState : automation.getFinaleStates()) {
                if (state != lastState) {
                    flag = false;
                }else{
                    flag=true;
                    break;
                }
            }
        }
        if(flag)
            break;
    }
}
```

Фигура 5

Класът *SaveAutomation* (фигура 6) служи за запаметяване на съответен автомат по два начина или като xml (фигура 7), или като txt(фигура 8).

```
public void save(int id, String fileName ,Machines machines, Console console) throws BadLocationException, IOException {
    Automation automation=machines.getAutomation(id);
    if(automation==null)
        console.print("Automation with ID:"+id+" didn't exist");
    else {
        String[] type=fileName.split( regex: "\\.");
        if(Objects.equals(type[1], b: "txt")) {
            saveTxt(fileName, automation);
            console.print("Successfully saved " + fileName);
        }
        else if(Objects.equals(type[1], b: "xml")) {
            saveXml(fileName, automation);
            console.print("Successfully saved " + fileName);
        }
        else {
            console.print("The program didn't support ." + type[1]);
            console.print("File must be .txt or .xml");
        }
    }
}
```

Фигура 6

```
public void saveTxt(String fileName, Automation automation) throws IOException {
    FileWriter fileWriter = new FileWriter(fileName);
    fileWriter.write(String.valueOf(automation.serialize()));
    fileWriter.close();
}
```

Фигура 7

```
public void saveXml(String fileName, Automation automation) throws IOException {
    FileOutputStream file= new FileOutputStream(fileName);
    XMLEncoder encoder = new XMLEncoder(file);
    encoder.writeObject(automation);
    encoder.close();
    file.close();
}
```

Фигура 8

Класът *Operation* изпълнява две от по-трудните операции: за намиране на обединението и за конкатенацията на два автомата.

В двата метода първоначално се създават копия на 2-та автомата чрез метода *newAutomation*, благодарение на което се избягва промяната на данни в оригиналните автомата със съответните им номера. Като за целта са използвани множество цикли, проверки и флагове.

Обединението се извършва като се добавя ново начало, от което по празна дума се тръгва по единия от 2-та автомата. Като за целта са използвани множество сложни алгоритми.

Конкатенацията се извършва като финалните състояния на 1-вия автомат се свържат с изходящите състояния на стартовото състояние на 2-рия автомат. Като за целта са използвани множество сложни алгоритми.

Класът *NewAutomation* се явява най-сложния в програмата. Той съдържа само един метод, като неговата цел е да създаде автомат въз основа на регулярен израз (теорема на Клини). Кода е много дълъг, като се състои от множество сложни алгоритми, които са изградени от множество цикли преплитащи се един в друг.

Първоначално регулярния израз се дели на символи и се записват съответните символи в азбуката на автомата. След това започва да се разглежда регулярния израз като се спазват зависимости скоби->повторение(звезда)->конкатенация(умножение)->алтернатива(събиране). Тъй като работим с обекти се правят множество проверки, за да се запамятат правилните обекти (с еднакъв хаш код), за да се избегнат грешки.

Класът *Mutator* изпълнява също сложна операция да направи недетерминиран краен автомат в детерминиран. Кода е сложен, тъй като трябва да създаден нов автомат, следователно цялата информация се презаписва от съответния автомат, като се правят леки промени. Създават се нови състояния, които се явяват съчетание от състоянията, при които входните състояния се явяват едно и също състояние по един и същ символ. Този процес се прави докато спрат да се появяват нови състояния.

#### б. Алгоритми и оптимизация

В клас *Operations* е създаден допълнителен метод *newAutomation* с цел избягване на повторение и оптимизиране на кода (*фигура 9*). Самото създаване на нов автомат става, като се създадат нови обекти, но със същите стойности на техните полета. За целта са използвани множество обхождания и проверки.

```
public Automation newAutomation(Automation automation){
    Automation automation1=new Automation(new State(automation.getStartState().getName()));
    automation1.setAlphabet(automation.getAlphabet());
    for(Function function:automation.getFunctions()){
        State state=new State(function.getFirstState().getName());
        boolean flag=true;
        for(State state1:automation1.getStates()){
            if (Objects.equals(state.getName(), state1.getName())) {
                state=state1;
                flag = false;
                break;
            }
        }
        if(flag){
            automation1.addState(state);
        }
        Function function1=new Function(state,function.getSymbol());
        for(State state1:function.getSecondStates()){
            State state2=new State(state1.getName());
            flag=true;
            for(State state3:automation1.getStates()){
                if(Objects.equals(state2.getName(), state3.getName())){
                    state2=state3;
                    flag=false;
                    break;
                }
            }
            if(flag){
                automation1.addState(state2);
            }
        }
    }
}
```

Фигура 9

Менюто на програмата остава скрито за потребителя, като за целта е използван switch (фигура 10), който въз основа на написаната команда от страна на потребителя, връща съответното действие. Ако се въведе команда, която не се поддържа от програмата или грешно написана такава, се извежда съобщение, че не съществува такава команда и дава подсказка да напише help (фигура 11). Програмата не дава достъп до останалите команди преди да се отвори файл (фигура 10 и фигура 12). При затварянето на файл данните в програмата се нулират (фигура 12).

```
String[] command=text.split( regex: " ");
switch (command[0]) {
    case "open":
        if (fileName == null) {
            if(command.length>2) {
                for (int i = 2; i < command.length; i++)
                    command[1] = command[1] + " " + command[i];
            }
            fileName = command[1];
            machines=commands.open(console,fileName);
        } else
            console.print("You have already opened file!");
        break;
```

Фигура 10

```
default:
    console.print("There is not such a command! \nPlease type: help");
```

```
case "close":
    if (fileName != null) {
        commands.close(console, fileName);
        fileName=null;
        machines=null;
    }
    else
        console.print("You first must open a file!");
    break;
```

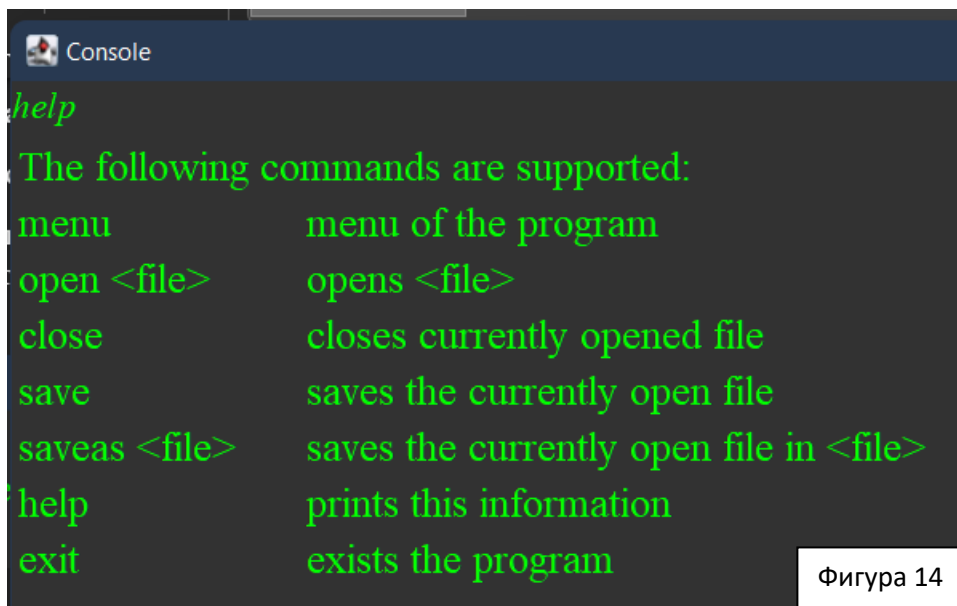
Фигура 12

Фигура 11

с. Планиране, описание на тестови сценарии

При стартиране на програмата, първоначално се изисква да се отвори xml файл. *Фигура 13*

Ако се напише друга команда се извежда съобщение, че трябва да се отвори файл. Ако се напише команда, която не се поддържа от програмата се извежда съобщение да се напише *help*. *Фигура 14*

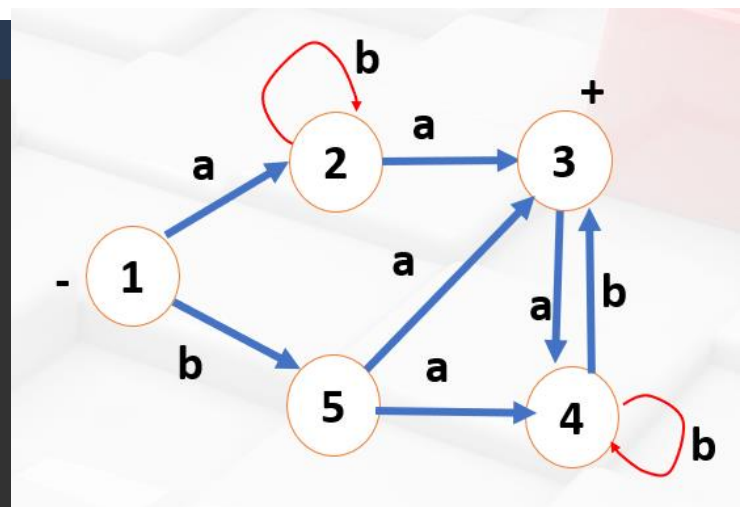


Файлът *proba.xml* съдържа предварително въведен недетерминиран краен автомат. За да се покажат ID-та на автоматите се извиква командата *list*. С команда *print 1* се показва автомата в табличен вид. *Фигура 15*

Console

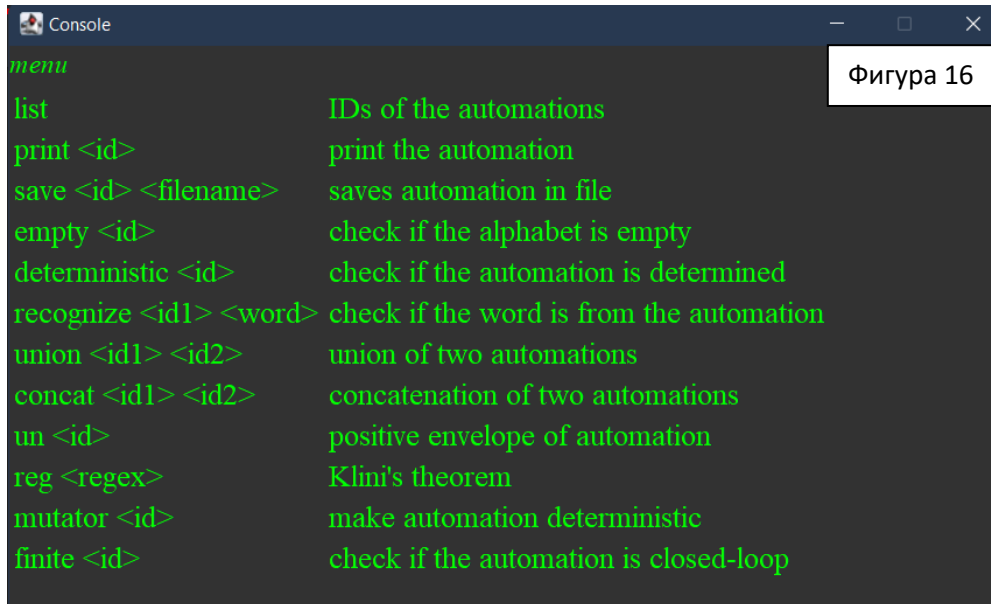
```
print 1
```

	a	b
START->	{1}	{2}
	{2}	{3}
	{4}	-
	{5}	{3,4}
END->	{3}	{4}



Фигура 15

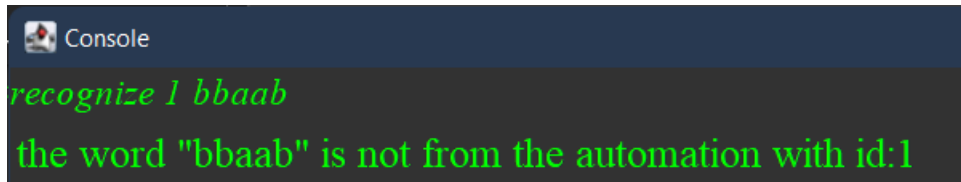
След като сме заредили съответния автомат може да тестваме останалите операции с атомати. Фигура 16



```
Console
menu
list          IDs of the automations
print <id>    print the automation
save <id> <filename> saves automation in file
empty <id>    check if the alphabet is empty
deterministic <id> check if the automation is determined
recognize <id1> <word> check if the word is from the automation
union <id1> <id2> union of two automations
concat <id1> <id2> concatenation of two automations
un <id>       positive envelope of automation
reg <regex>   Kline's theorem
mutator <id>  make automation deterministic
finite <id>   check if the automation is closed-loop
```


Фигура 16

За да проверим да ли даден низ е от автомата ползваме командата *recognize*. Фигура 17а 17б



```
Console
recognize 1 bbaab
the word "bbaab" is not from the automation with id:1
```

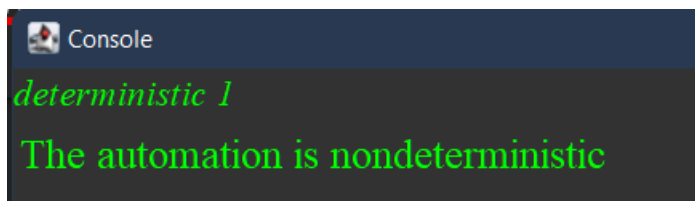
Фигура 17а



```
Console
recognize 1 baab
the word "baab" is from the automation with id:1
```

Фигура 17б

За да проверим дали автомата е детерминиран ползваме командата *deterministic*. Фигура 18



```
Console
deterministic 1
The automation is nondeterministic
```

Фигура 18



За да превърнем даден автомат от недетерминиран в детерминиран ползваме командата *mutator*. Фигура 19

```

Console
mutator 1
Automation with id:1 is now deterministic
    
```

Фигура 19

За да проверим дали е привърнат правилно в детерминиран ползваме пак командата *deterministic*. Фигура 20

```

Console
deterministic 1
The automation is deterministic
    
```

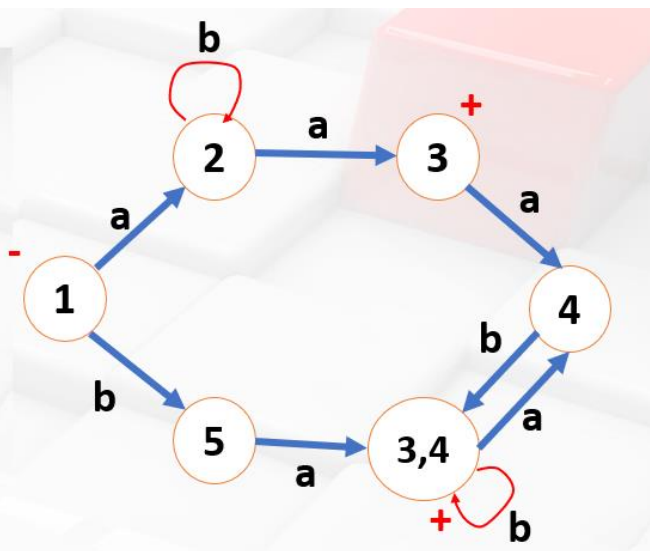
Фигура 20

Съответно да видим неговата сериализация ползваме командата *print*. Фигура 21

```

Console
print 1
START->
a      b
{1}    {2}    {5}
{2}    {3}    {2}
{4}    -      {3,4}
{5}    {3,4}  -
END->  {3,4}  {4}  {3,4}
END->  {3}    {4}  -
    
```

Фигура 21



За да създаден нов автомат ползваме командата *reg* като пише и регулярния израз, от който искаме да получим автомата. Фигура 22

```

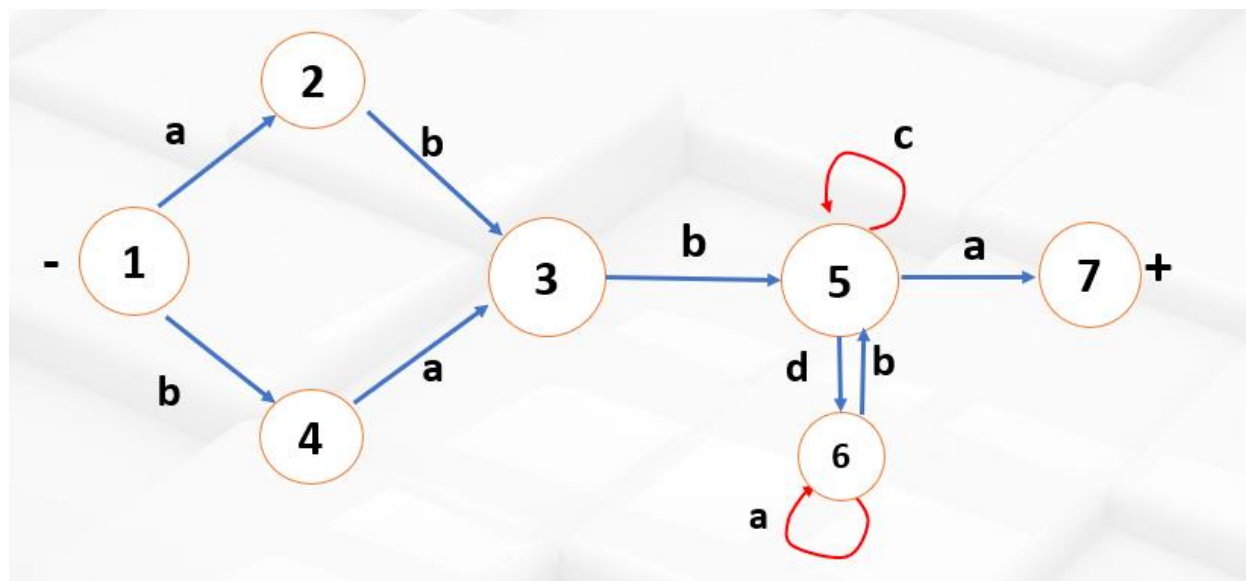
Console
reg (ab+ba)b(da*b+c)*a
The new automation id is:2
    
```

Фигура 22

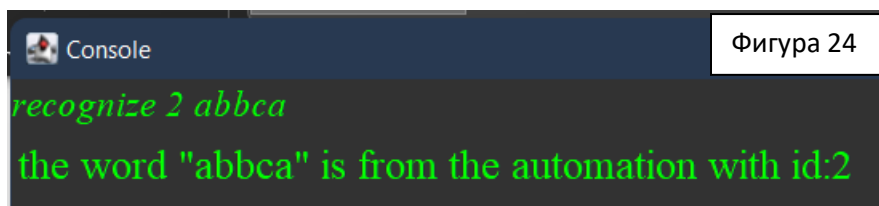
Получения автомат може да проверим чрез print. Фигура 23

		a	b	c	d
START->	{1}	{2}	{4}	-	-
	{2}	-	{3}	-	-
	{3}	-	{5}	-	-
	{4}	{3}	-	-	-
	{5}	{7}	-	{5}	{6}
	{6}	{6}	{5}	-	-
END->	{7}	-	-	-	-

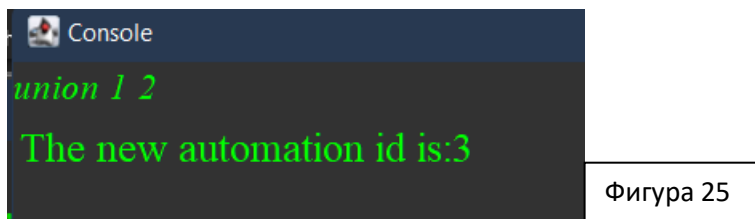
На съответния автомат отговаря съответния граф.



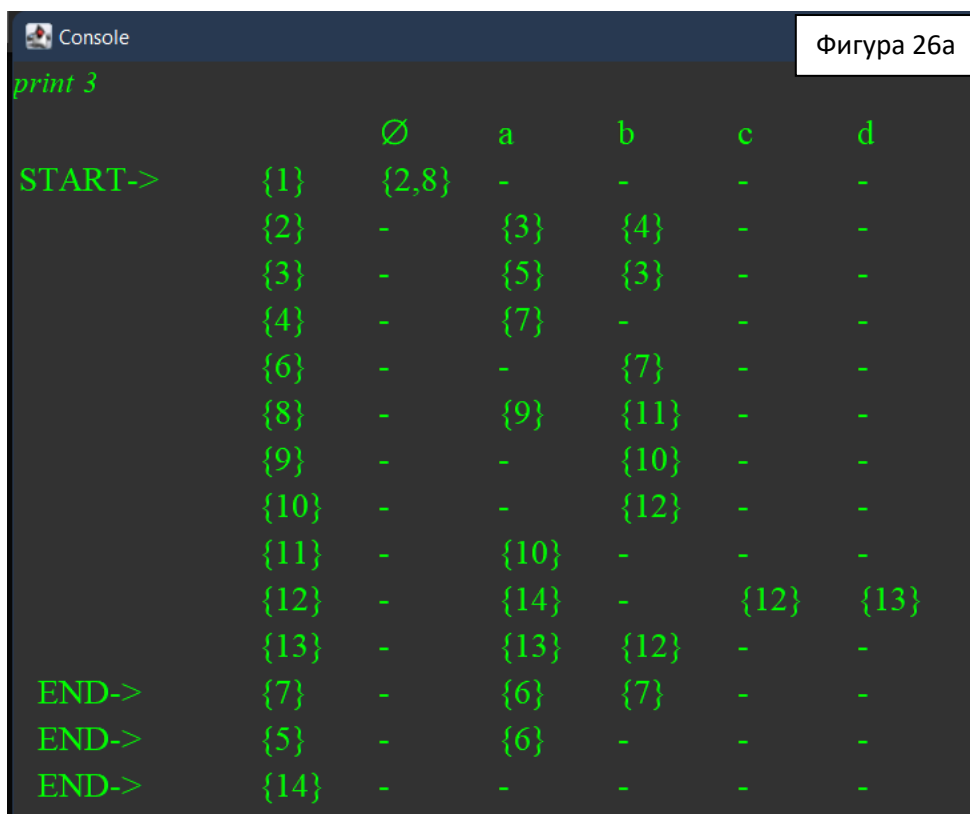
Проверка дали правилно се създават думите. Фигура 24



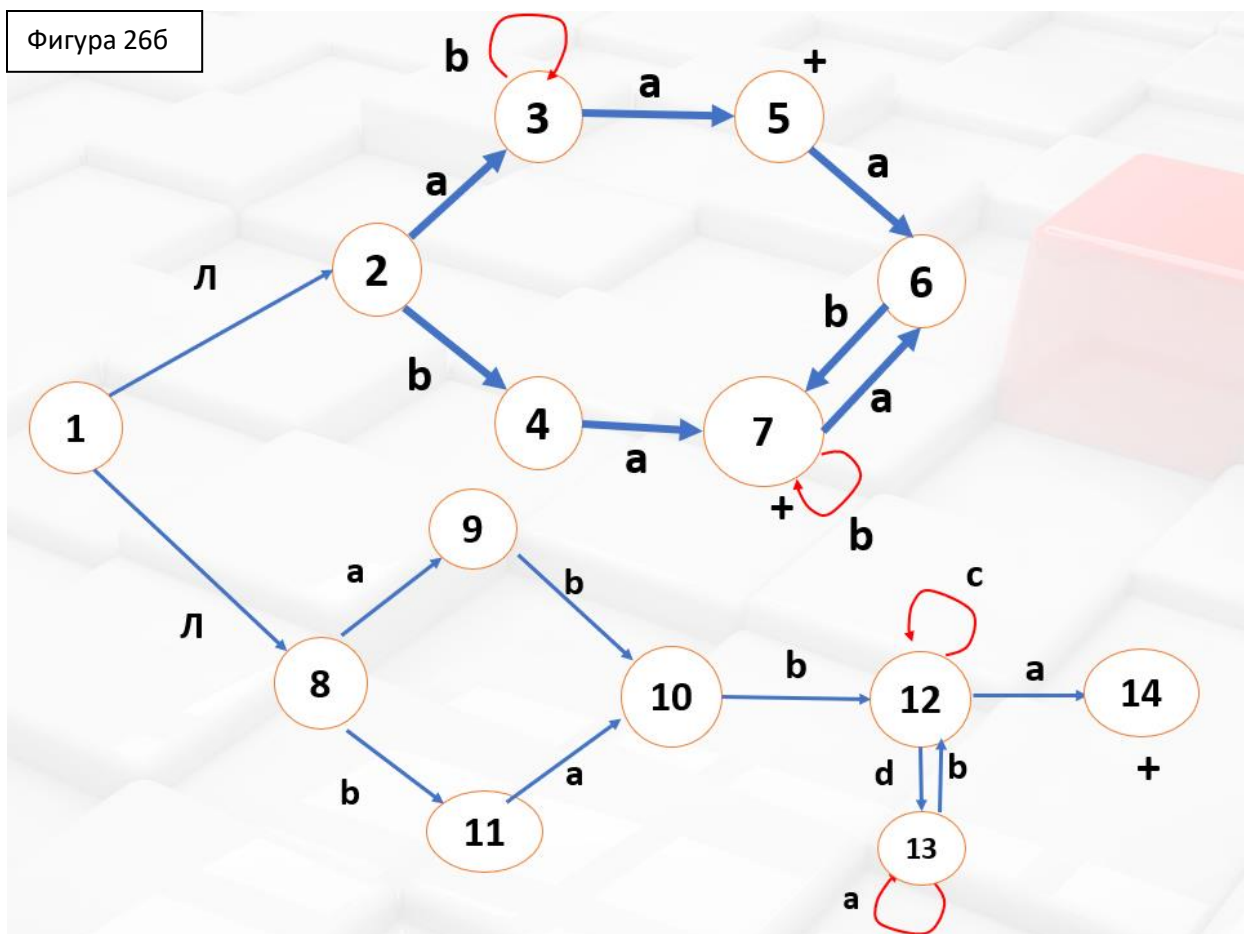
За намирането на обединението на два автомата се извиква командата union. В случая се прави обединение на текущите автомати 1 и 2. Фигура 25



Проверка на ново получения автомат. Фигура 26а и 26б



Фигура 26б



За намирането на конкатенацията на два автомата се извиква командата `concat`. В случая се прави конкатенация на текущите автомати 1 преди да се детерминира и 2. Фигура 27

```

Console
concat 1 2
The new automation id is:3
    
```

Фигура 27

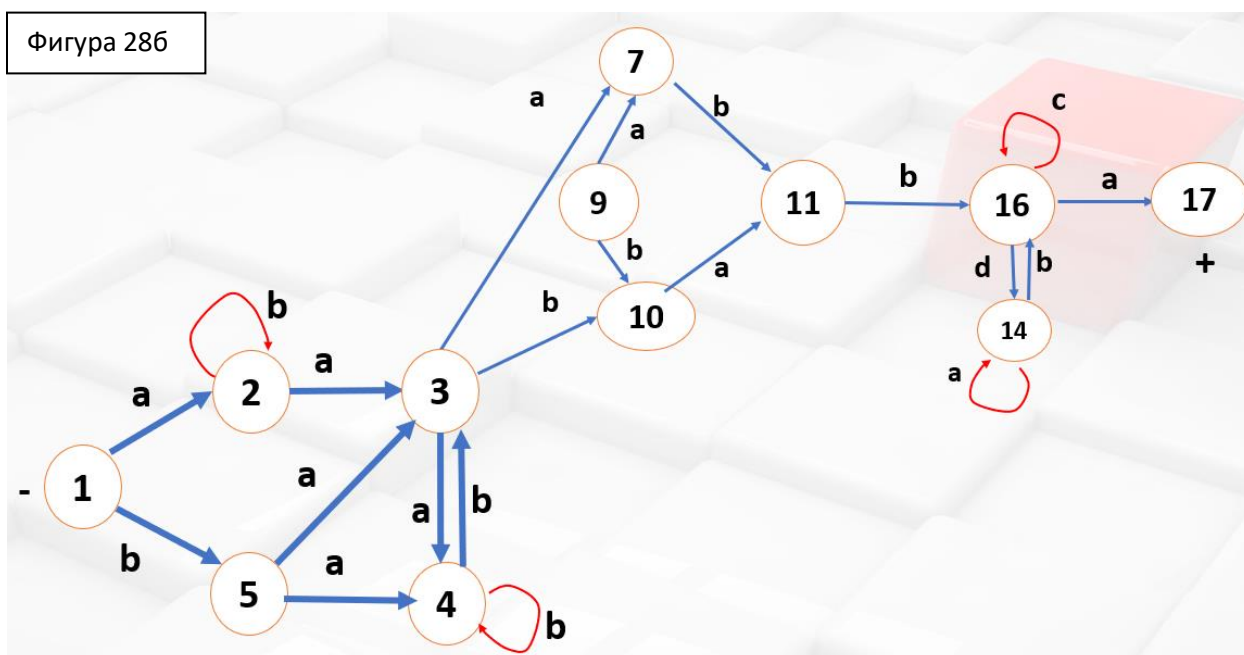
Проверка на ново получения автомат. Фигура 28а и 28б

Console

Фигура 28а

print 3

		a	b	c	d
START->	{1}	{2}	{5}	-	-
	{2}	{3}	{2}	-	-
	{3}	{4,7}	{10}	-	-
	{14}	{14}	{16}	-	-
	{4}	-	{3,4}	-	-
	{5}	{3,4}	-	-	-
	{16}	{17}	-	{16}	{14}
	{7}	-	{11}	-	-
	{9}	{7}	{10}	-	-
	{10}	{11}	-	-	-
	{11}	-	{16}	-	-
END->	{17}	-	-	-	-



## 5. Заключение

### а. Обобщение на изпълнението на началните цели

Изпълнени са почти всички операции дадени като задание, с изключение на намирането на позитивна обвивка на автомат. Причината: невъзможността да се намери отговор на въпроса що е то позитивна обвивка и съответно как да се изпълни операцията. Създадената програма с леки подобрения би била от полза за студентите по дисциплината Дискретни структури с цел решаване на задачи за крайни автомати.

### б. Насоки за бъдещо развитие и усъвършенстване

Програмата би могла да се до оптимизира при някои операции, с цел намаляване на кода или намиране на по-кратки решения на дадените проблеми. При по-задълбочено тестване на програмата биха могли да излязат някои неточности. Код за намиране на автомат по регулярен израз не е оптимизиран и най-вероятно не е универсален за всеки един регулярен израз. Нужно би било да се тестват множество входни данни и да се обхождат всички възможности. Код би станал обаче прекалено сложен и дълъг, тъй като нивото на абстракция е голямо и връзката между обектите в автомата е доста сложна.