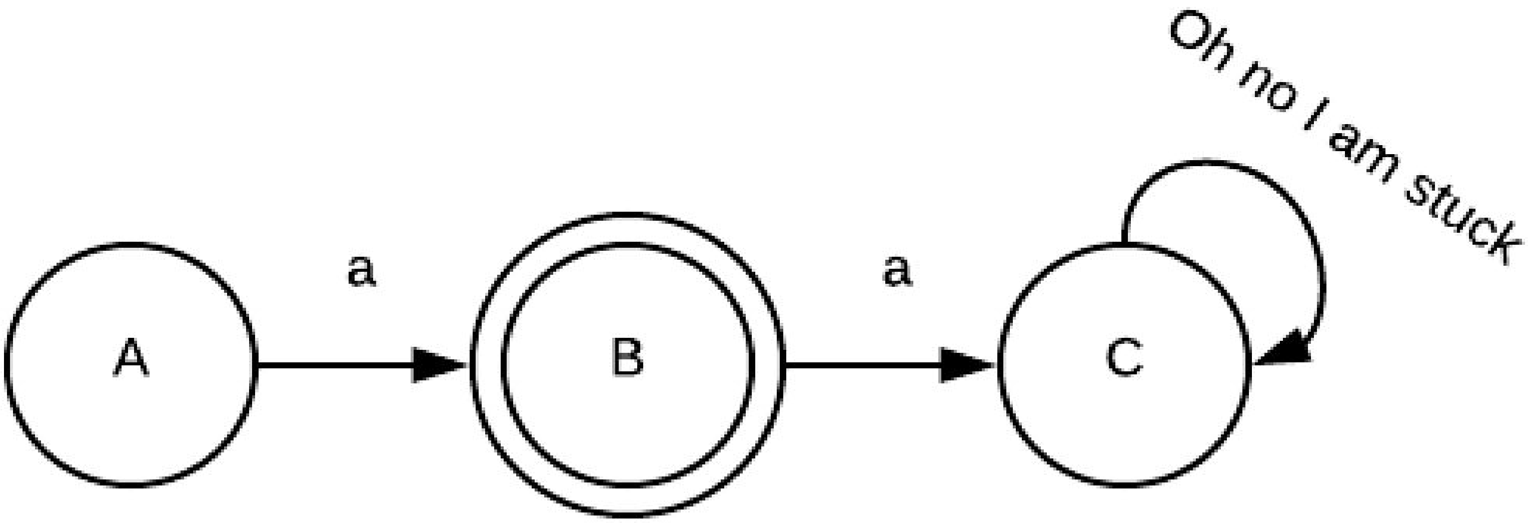


Automata Engineering

Boris Rokanov

Version 1.0

ALE2 Report

Table of contents

Chapter 1: Introduction

Doubtless, automata theory is one of the major understandings every software engineer should master. Automata exists everywhere in our software, a good example of an automaton are the compilers. They are responsible for converting our code into a binary format that the computers could understand.

The typical automata consist of a finite set of states(usually marked with capital letters), alphabet(usually marked with a small letter), which shows the “title” of our transitions, and transitions which are directed and have an initial state, destination state and a letter from the alphabet that relates them and shows the possible path. The connecting letter can also be an epsilon which represents the so-called empty transition - this connecting letter can be replaced by any other letter from the alphabet.

The transitions could be various types - those that point to a further state, those that point to the same state as their initial state, and back transitions that point to an already processed state by this path - they create cycles.

On the other hand, the states can be final and non-final. If a state is final it means that the automaton can finish at this state. This is especially important when it comes to defining whether the automaton contains a finite amount of words.

The alphabet represents a set of letters, where each letter should be unique.

Chapter 2: Parsing, Drawing, and DFA

Overview

The parsing of an automaton happens with the following procedure - initially, the user inputs an automaton in a string format, that needs to be converted into objects because this will help me to manipulate the objects and perform algorithms using them. The user specifies the alphabet, the states, the transitions, and if the automaton is expected to be a DFA. The automata is DFA when each state has one and only one outgoing transition for each letter from the alphabet. So, a state that has two outgoing transitions with the same letter will lead to a non-DFA. Another case is when the automaton contains the empty transition(the epsilon connecting letter), this immediately makes the automaton non-DFA.

Impediments

Initially, I was not sure what would be the best approach to identify whether the automaton is finite or not. I have tried to check the possible transitions from each state and then check if the connecting letter is repeated.

Solutions

For solving the impediment above I found a better way - namely for each state to keep track of the outgoing letters. Currently each state contains a list of outgoing letters, which makes it very efficient for the application to check if it is DFA. The time complexity for this algorithm would be $O(n.m^2)$ where “*n*” shows the number of states and “*m*” shows the number of outgoing letters in the given state.

Chapter 3: Accepting string

Overview

The automaton can accept a string and check whether it is possible. In order a word to be accepted it should satisfy the requirements, that namely are - starting from the initial state, each letter will move us with one state further by following the transition with corresponding connection letter. At the step where the word has been checked and if all letters are accepted so far we check if the current state is a final state and if it is final, then the string is accepted. In case a state has more than one possible transitions in which it can go, the algorithm checks for all of them and if one of the paths is possible, then the word is accepted. In case there is an epsilon appearing as a possible outgoing letter from the current state, we also follow and test this path.

Impediments

During this assignment, I did not face any significant impediments as the implementation was straight forward. I implemented it by the use of recursion, which makes it complicated for calculating the running time of the algorithm.

Chapter 4: Regular Expression

Overview

The application has the opportunity of building a regular expression based on Thompson's construction algorithm. Thompson's algorithm has 4 rules based on which it enhances the automaton. The automaton will be an NFA in every case. For the sake of this assignment, I have introduced my own data structure called “RegularExpression” which keeps track of all transitions, the initial state, and the last state. I have also introduced A class called “RegularExpressionController” which is responsible for defining the rules that should be applied and manipulating the “RegularExpression”.

Impediments

At the beginning, I tried to do the whole algorithm at once using a recursion. This took me not only to wrong software architecture but to an incorrectly working algorithm as well. I was also not keeping track of the initial and the last state of the given “RegularExpression” which made it harder for me to connect the parts.

Solutions

Although I spent much time figuring out what would be the best approach I decided to start from scratch and clearly define what are the requirements. I split every rule in a separate method, defining which rule to be executed in other as well. By performing this action I spent a bit more time than expected but the methods are working perfectly by following the Single Responsibility principle.

Chapter 5: Finite

Overview

The purpose of this assignment was to check whether the amount of words possible in the automaton is finite. There were several concerns for which I had to be aware of. The algorithm works on the principle that it finds all possible paths in the automaton and checks if they contain cycles. In case there is a cycle, the algorithm checks if the cycle consists only of epsilon transitions. In this case, the path is possible if it ends in a final state. If there is a path that contains a cycle, and this cycle is formulated of letters from the alphabet, and it ends in a final state then the path is infinite as it can generate an infinite amount of the given letter before it possibly ends in a final state. If the path contains cycles but does not result in a final state the algorithm will still accept it as a correct path, due to the fact that no matter how many letters it generates, then the word can never be finished. For this assignment, I created a class called “Trace” which contains all transitions in this path. Introducing this class made the execution of the algorithm quite handy because it also allowed me to keep track of whether the trace is finishable(it ends in a final state).

Impediments

Definitely this algorithm was challenging to implement as there are many edge cases. At the beginning I made the algorithms detect cycles and if there is one, then the algorithm returns false. This approach was incorrect, so I had to fix this case. The implementation of the algorithm took me more time than expected because I also had problems with the references.

Solutions

I researched a bit more about the algorithm and how it should function. I introduced new properties to the class “Trace” such as a list of visited states which helped me to detect cycles. I also made a method for creating deep copies to resolve the problems with the references.

Chapter 6: NFA to DFA Conversion

Overview

NFA or nondeterministic finite automaton is such a type of finite-state machine where each state does not contain one only one outgoing letter for each letter from the alphabet. Another way to define if the automaton is NFA is by the epsilon transitions. If there are any epsilons in our automaton, then the automaton is an NFA. The opposite rules define what a DFA is. In order to convert an NFA to DFA, we have to first remove all epsilon transitions and second, each state should contain one and only one outgoing letter for each letter from the alphabet. The algorithm has such a working flow, where we convert our automaton with transitions into a table format. The table itself shows for each state with every letter from the alphabet to which other states it could reach. The table contains a column for e-closure(epsilon closure) transitions that shows from the given state to what other state we can reach with an epsilon transition including the given state. Afterward based on the table generated we get the first e-closure values and set this as our initial state. From the initial state, we start checking for each letter from the alphabet to which states we can reach. If the set of reachable states is unique

we repeat the process for it as well and we continue so until there is no unique list of states to be processed. Finally, this results in a table as well which we use to convert into a format with transitions.
For this algorithm, I introduced several data structures - “AutomataTable”, “AutomataRow” and “AutomataCell” where the table has a list of rows and each row has a list of cells. This helped me to easily transform the automaton into different forms and manipulate the data.

Impediments

At the beginning, I was not aware of the e-closure transitions and how exactly to use them. Also, I was not checking whether a list of states already exists, which resulted in a StackOverflow because the list of states was being added infinitely many times.

Solutions

I did extra research on the topic and found a clear explanation about the conversion from NFA to DFA and how exactly to deal with e-closures.
I started checking if a list of states already exists in the table and not add it as a separate state twice.

Chapter 7: PDA

Overview

PDA or Push Down Automaton is a finite state machine that supports a stack. This assignment has the purpose to check whether a given word is accepted in the automaton. It functionates on a similar principle as assignment 2, but with the difference that here we need to add several rules related to the stack. The data structure “Transition” is updated to support a so-called “StackElement” which has an in and out element. By going through a transition we first pop from the stack the out element(if possible, otherwise we skip this transition and do not process it), and then we push to the stack the in-element of the given transition. In the end, the word can be finished if and only if there is a path for it, the last state is final and the stack is empty.

Impediments

I misunderstood the epsilon transitions, which caused me troubles with the algorithm as it was not working properly. I was treating the epsilon transition as a regular transition which consumes a letter from the word. For example, if the word is “aa” and the algorithm goes through an epsilon transition the rest of the word is “a”.

Solutions

I fixed the algorithm to check if the processed transition is an epsilon transition, and if this is the case, the word remains the same, but we skip the transition.

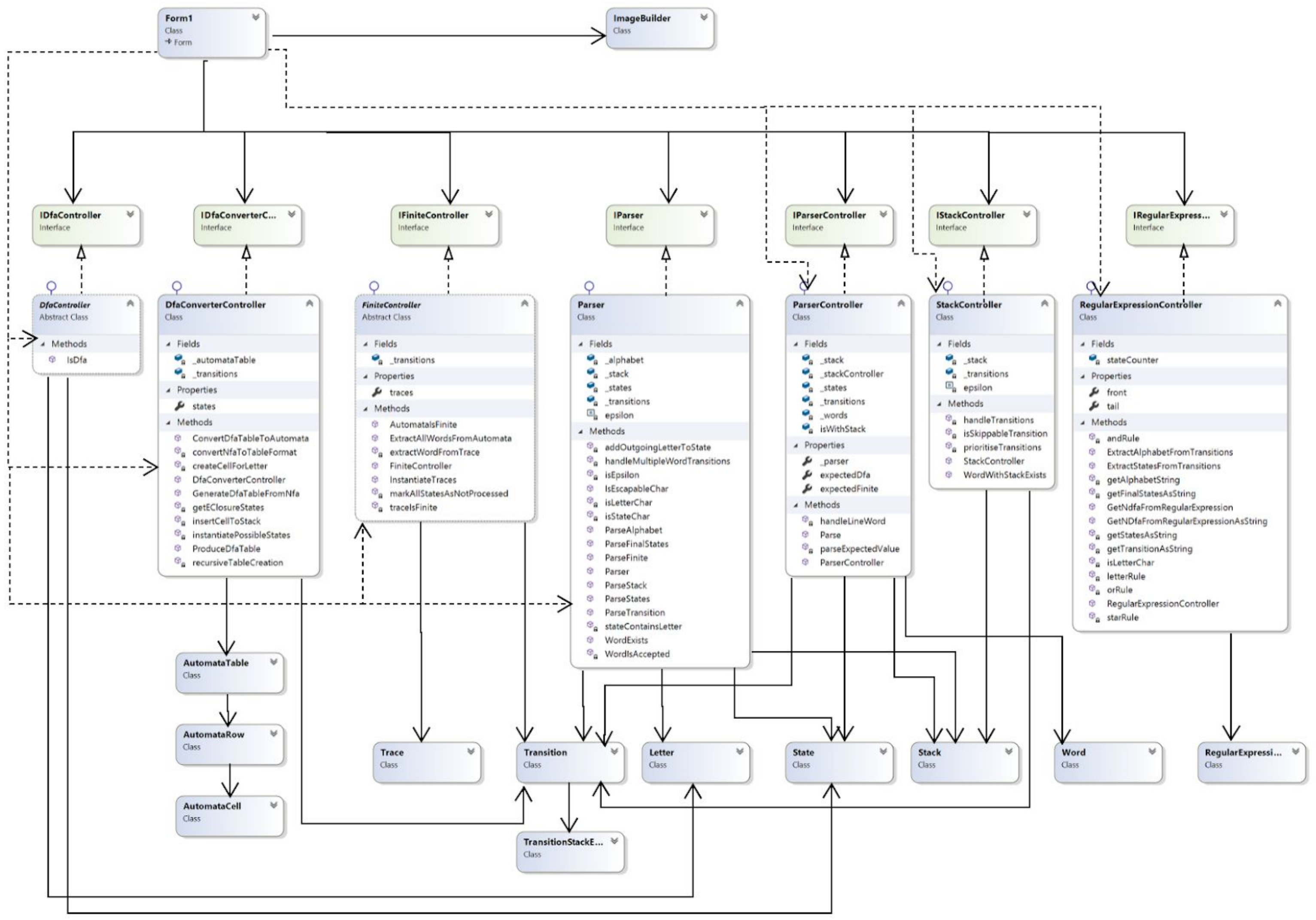
Chapter 8: Software Design

Chapter 8.1: System Overview

The application uses 3 main components. Controllers, models and a view. The models are the data structures. They represent the data which is used in the application. The controllers are the ones controlling or manipulating the models. They are responsible for the communication between the view and the models. The view represents the user interface, in this case the Form. I have tried to apply the SOLID and DRY principles. For every controller there states an interface as this refers to the Dependency Inversion principle which states “The components should depend on abstractions rather than concrete classes”. The methods and classes apply the Single Responsibility principle which states that “The class/method should have one and only one reason to change” or in other words it should have one and only one responsibility. By introducing an interface for each controller I have used the Open-Close principle which states that “Each component should be open for extensions but closed for modifications”. This interfaces apply the Interface Segregation principle, which means that the responsibilities of the interfaces are segregated as much as possible.

DRY or Don’t Repeat Yourself is one of the major concepts in software development. The name is descriptive and it means that code should not be repeated within the components. I have tried to use this concept as much as possible but I figured out that in several components I have methods with a similar functionality.

Chapter 8.2: System Architecture

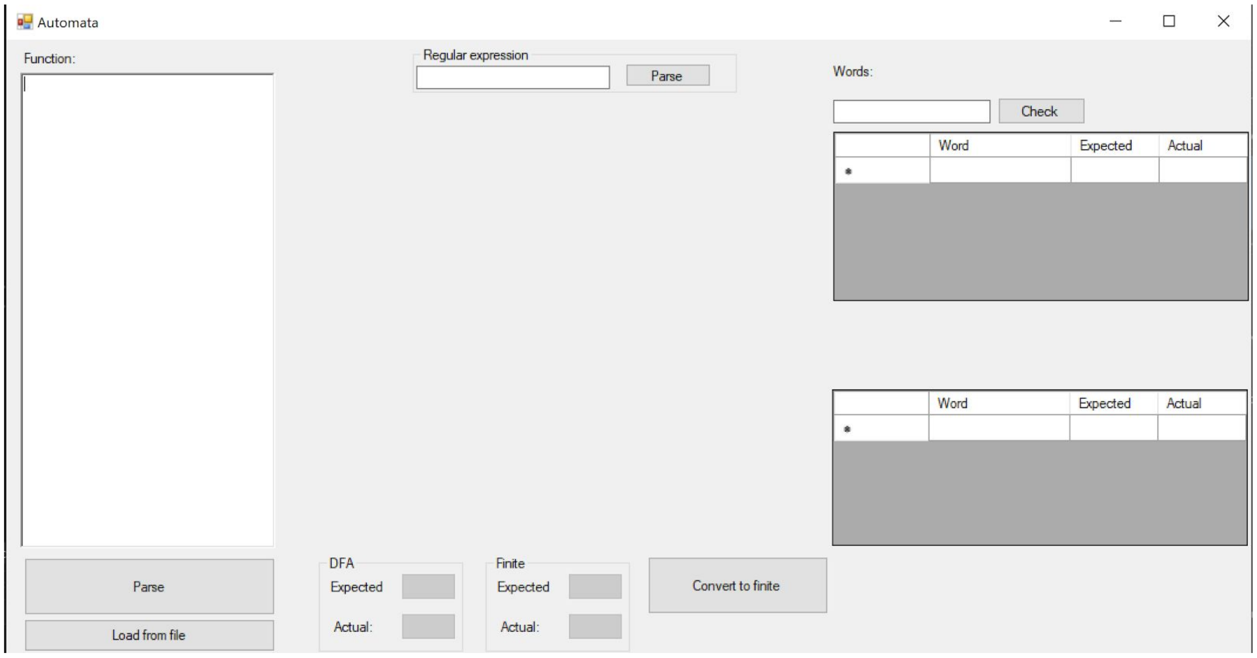


Chapter 8.3: Data Design

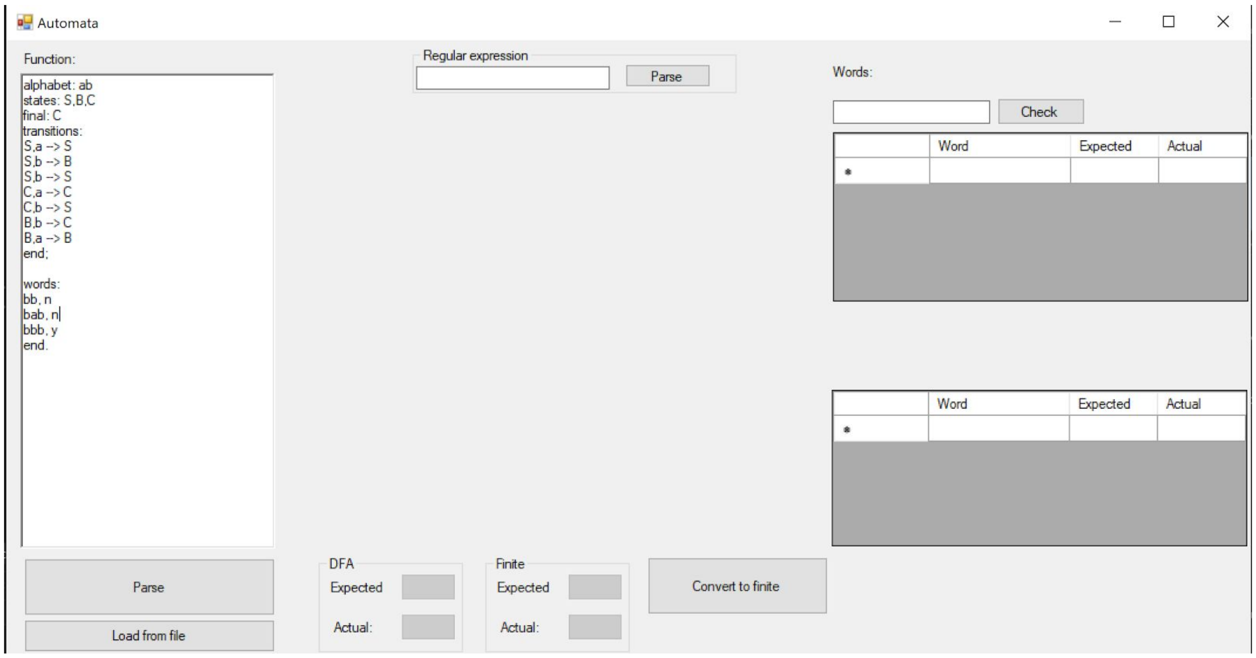
Initially the user inputs the data in a raw text, that the software should be able to convert into objects because the objects could be easily manipulated. The user is supposed to enter an alphabet which is being converted to a “List<Letter>”, input states that are converted into “List<State>”, transitions that are converted to “List<Transition>”, and words that are converted to “List<Word>”. The application is also able to read whether the automaton is expected to be finite and DFA. The transformation from text into objects is done in the "Parser" and "ParserController" classes.

Chapter 9: GUI

1) Initially when the user opens the application the following will appear:



2) In the text area called Function the user can input the desired function to test in the following way:

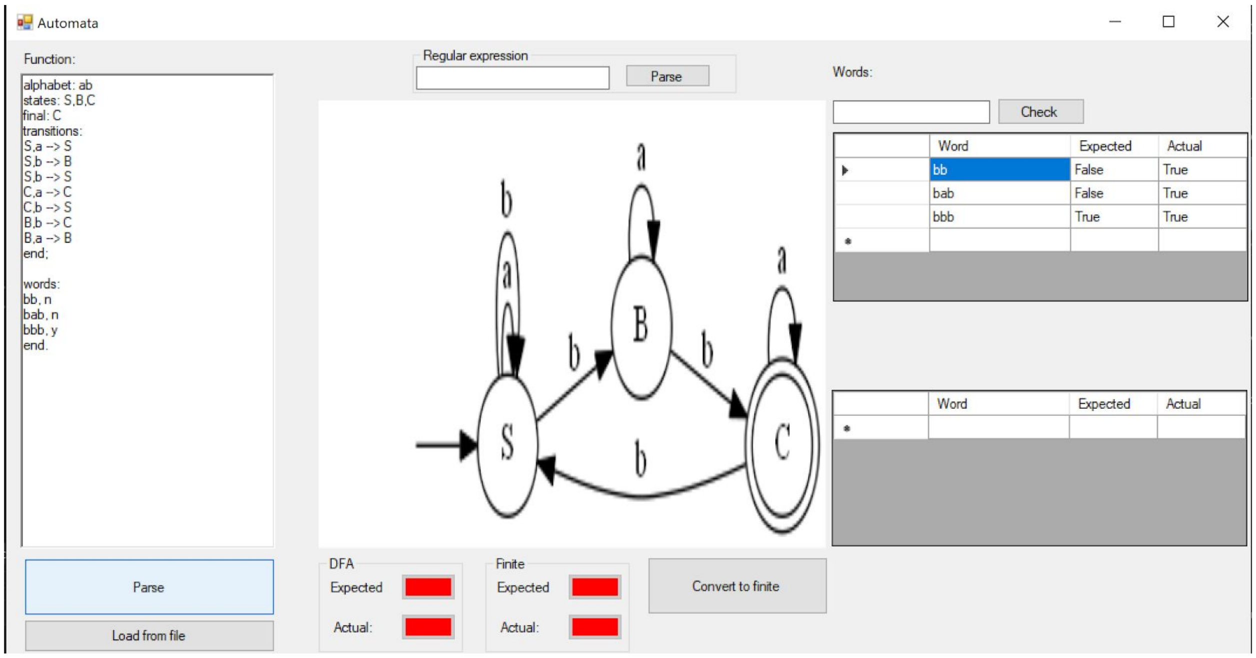


```
alphabet: ab
states: S,B,C
final: C
transitions:
S,a --> S
S,b --> B
S,b --> S
C,a --> C
C,b --> S
B,b --> C
B,a --> B
end;

words:
bb, n
bab, n
bbb, y
end.
```

	Word	Expected	Actual
*			

	Word	Expected	Actual
*			

3) After the function is placed in the field the user should press the button Parse. From this he/she will be able to see the automaton as an image, the expected DFA value and the actual, the expected finite and the actual, and all words which are checked in the automaton.



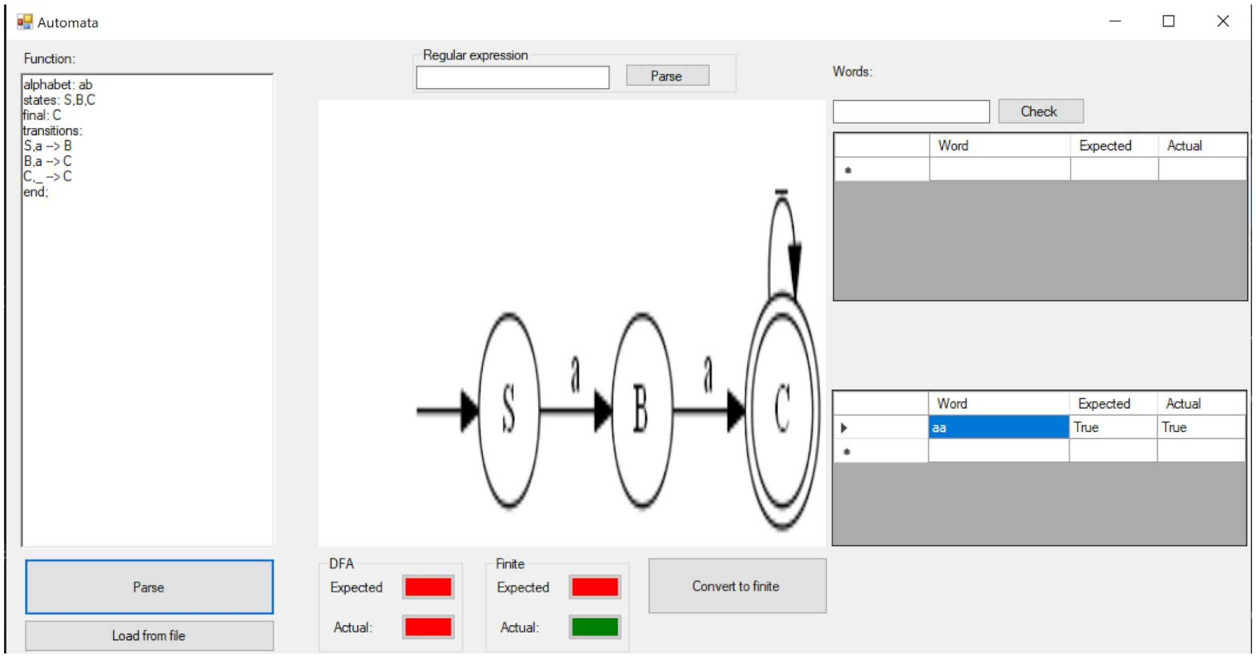
```
alphabet: ab
states: S,B,C
final: C
transitions:
S,a --> S
S,b --> B
S,b --> S
C,a --> C
C,b --> S
B,b --> C
B,a --> B
end;

words:
bb, n
bab, n
bbb, y
end.
```

	Word	Expected	Actual
▶	bb	False	True
	bab	False	True
	bbb	True	True
*			

	Word	Expected	Actual
*			

DFA		Finite	
Expected	🟥	Expected	🟥
Actual:	🟥	Actual:	🟥

4) In case the automaton has a finite amount of words, they will appear in the table below like this:



```
alphabet: ab
states: S,B,C
final: C
transitions:
S,a --> B
B,a --> C
C,_ --> C
end;
```

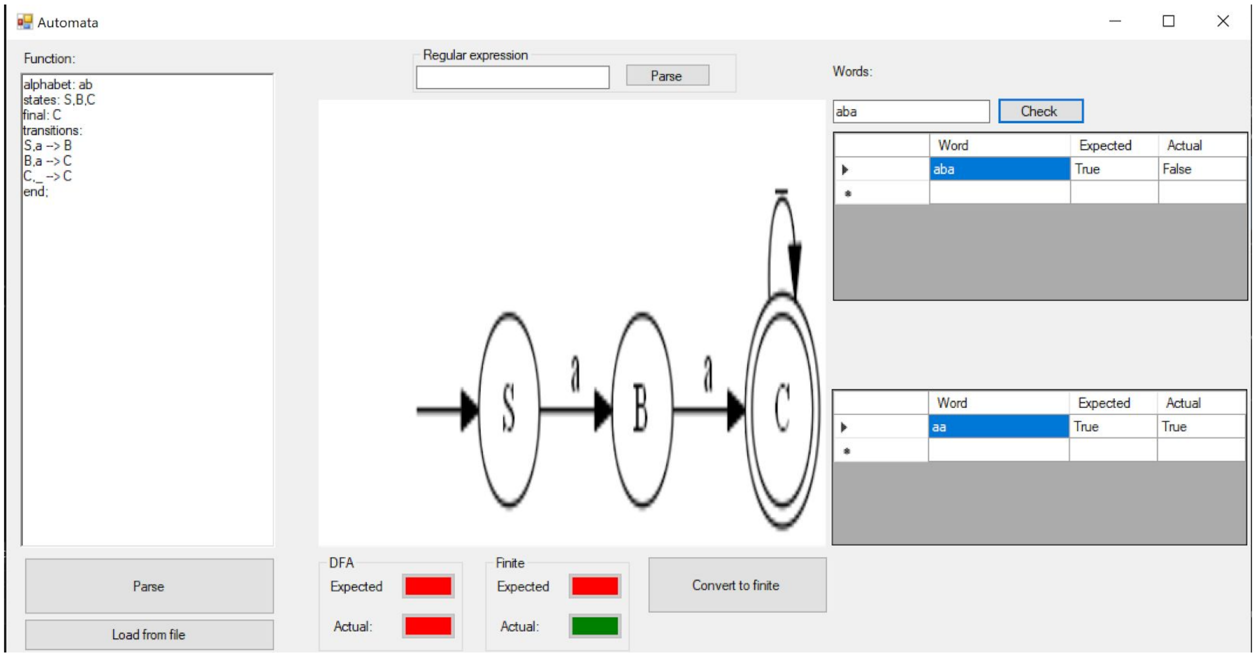
Word	Expected	Actual

Word	Expected	Actual
aa	True	True

DFA	
Expected	🟥
Actual:	🟥

Finite	
Expected	🟥
Actual:	🟩

5) The user has the option to search for a word dynamically by inputting the result in the field words at the top-right and then clicking on the button Check. He will get the following:



```
alphabet: ab
states: S,B,C
final: C
transitions:
S,a --> B
B,a --> C
C,_ --> C
end;
```

Words: aba

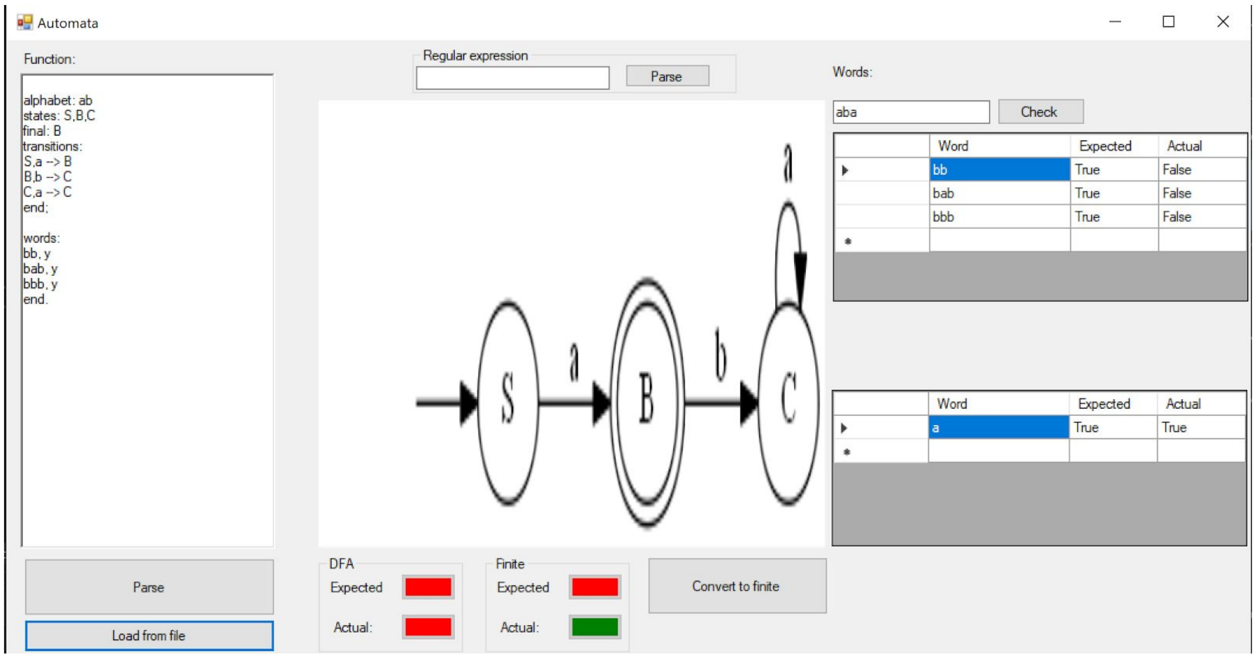
Word	Expected	Actual
aba	True	False

Word	Expected	Actual
aa	True	True

DFA	
Expected	🟥
Actual:	🟥

Finite	
Expected	🟥
Actual:	🟩

6) The user has the option to load a function from a file by pressing on the button at the bottom-left called Load from file and then selecting the text file containing the automaton function. This will read the function from the file and process all algorithms needed like that:



```
alphabet: ab
states: S,B,C
final: B
transitions:
S,a --> B
B,b --> C
C,a --> C
end;

words:
bb, y
bab, y
bbb, y
end.
```

aba

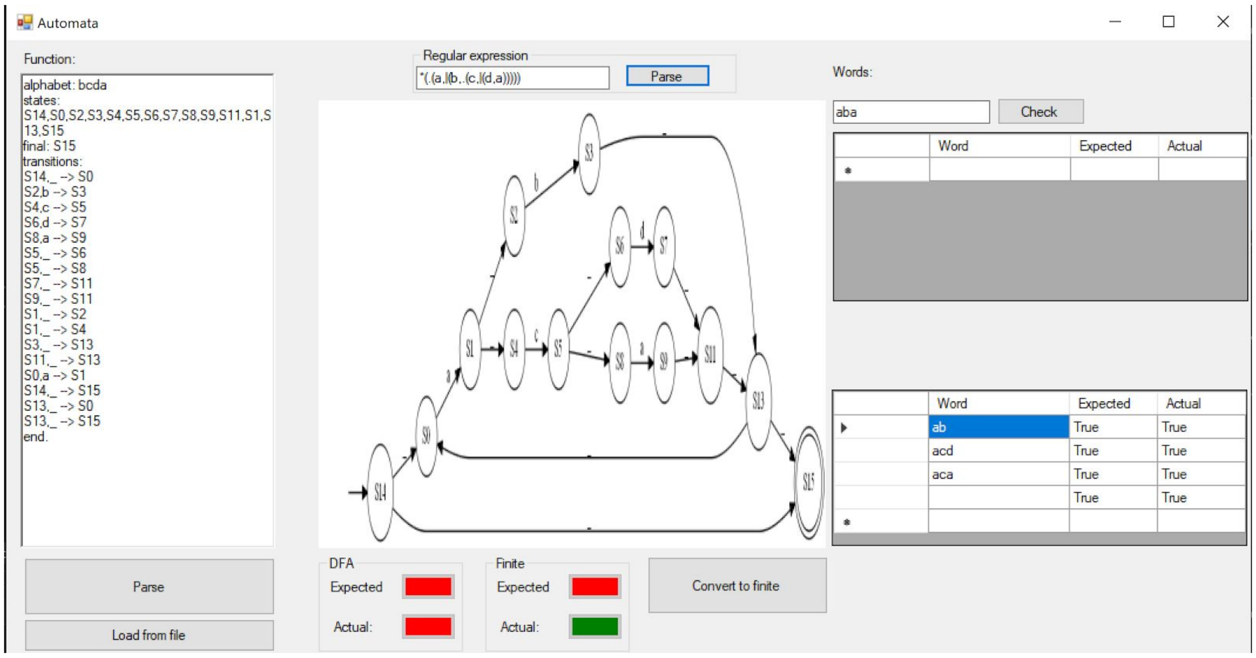
Word	Expected	Actual
bb	True	False
bab	True	False
bbb	True	False

Word	Expected	Actual
a	True	True

DFA	
Expected	🟥
Actual:	🟥

Finite	
Expected	🟥
Actual:	🟩

7) The user has the possibility to input a Regular expression function in the field at the middle-top field and then pressing the Parse button. This will read the infix format of the regular expression and process it. The following will appear:



*(.(a,|(b,.(c,|(d,a)))))

```
alphabet: bcda
states:
S14,S0,S2,S3,S4,S5,S6,S7,S8,S9,S11,S1,S
13,S15
final: S15
transitions:
S14,_ --> S0
S2,b --> S3
S4,c --> S5
S6,d --> S7
S8,a --> S9
S5,_ --> S6
S5,_ --> S8
S7,_ --> S11
S9,_ --> S11
S1,_ --> S2
S1,_ --> S4
S3,_ --> S13
S11,_ --> S13
S0,a --> S1
S14,_ --> S15
S13,_ --> S0
S13,_ --> S15
end.
```

aba

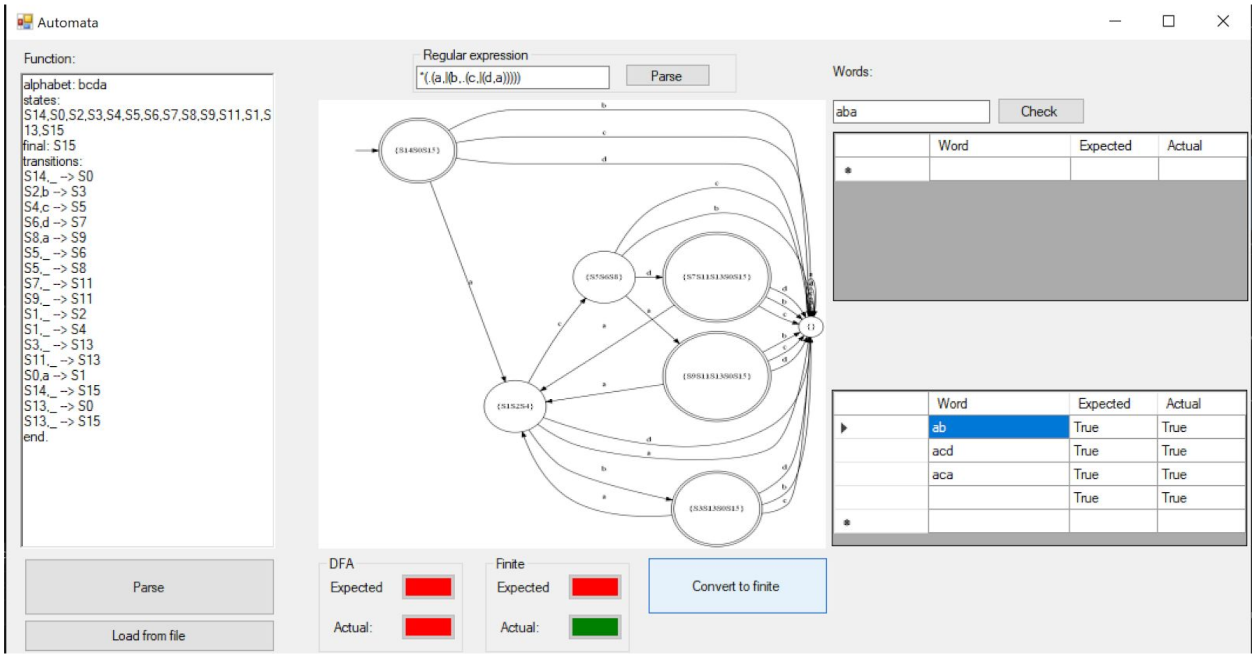
Word	Expected	Actual

Word	Expected	Actual
ab	True	True
acd	True	True
aca	True	True
	True	True

DFA	
Expected	🟥
Actual:	🟥

Finite	
Expected	🟥
Actual:	🟩

8) The user is able to Convert a NFA automaton into a DFA one by pressing on the Convert to finite button at bottom-middle. He/she will get a result like this:



```
alphabet: bcda
states:
S14,S0,S2,S3,S4,S5,S6,S7,S8,S9,S11,S1,S
13,S15
final: S15
transitions:
S14,_ --> S0
S2,b --> S3
S4,c --> S5
S6,d --> S7
S8,a --> S9
S5,_ --> S6
S5,_ --> S8
S7,_ --> S11
S9,_ --> S11
S1,_ --> S2
S1,_ --> S4
S3,_ --> S13
S11,_ --> S13
S0,a --> S1
S14,_ --> S15
S13,_ --> S0
S13,_ --> S15
end.
```

Regular expression: `*(.(a,|(b,.(c,|(d,a)))))`

Words: aba

	Word	Expected	Actual
*			

	Word	Expected	Actual
▶	ab	True	True
	acd	True	True
	aca	True	True
		True	True
*			

DFA — Expected: 🟥 Actual: 🟥

Finite — Expected: 🟥 Actual: 🟩

Chapter 10: Testing

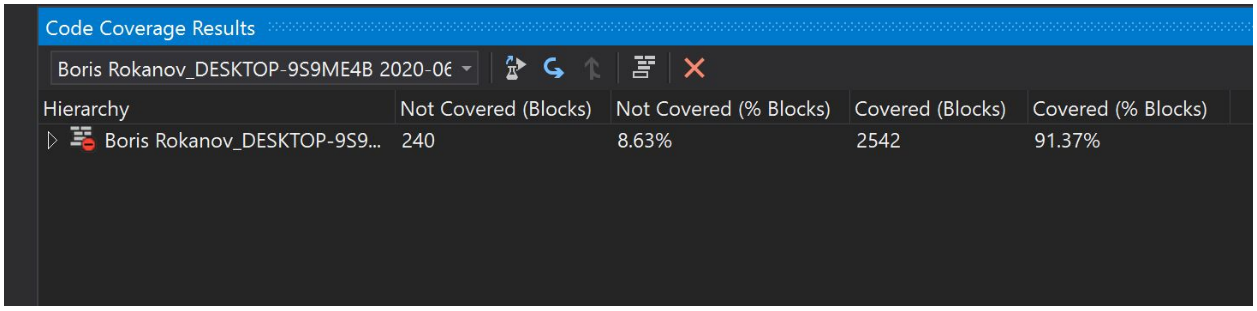
Unit testing has a huge impact on each project. Usually, when many algorithms are applied in an application, there is a high chance the application does not work as expected. On the other hand, unit testing still can not guarantee that the project works perfectly as there might be a small edge case for which the algorithm is not applied properly. In most cases though it is convenient to have a good testing set applied.

For each component of the application, there is a separate set of tests running and ensuring that the algorithm works properly. The tests of the components are segmented which also improves the readability of the given test. The AAA pattern is applied in the unit tests which states for:
Arrange - Declaration of the needed variables for this test (basically the configuration or preparation of the test is placed here).
Act - The component execution (usually calling the method from the domain logic or just calling the algorithm we want to test).

Assert - The verification of the test (here we verify whether the expected output we give matches the actual output received from the algorithm).

Some of the tests, that are not too coupled with other components or with itself allow the most proper unit testing - to test each unit separately. For these methods, I have applied mocks of the objects and of the methods.

In the end, the test coverage was around 90% and there were around 10% uncovered because they were too redundant or just simple getters and setters, which we can assume work properly.



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Boris Rokanov_DESKTOP-9S9...	240	8.63%	2542	91.37%

Chapter 11: Conclusion and recommendations

In conclusion, I want to mention that I have learned a lot from this project in terms of Software Architecture and Automata, which definitely is helpful. This project required much work and dedication on the subject, but in the end, it is all worth the time spent.
For the future, if I have to work on a similar project I would try to implement an initial design for the software. In the current case, I started coding and implementing without any design, which definitely slowed down the process. Although the good Software Architecture understanding requires much learning and coding, I am dedicated and ready to go through this path with high enthusiasm.