MSc Computer Science Project Report

Department of Computer Science and Information Systems

Birkbeck College, University of London 2020

Automata simulator

By Dimitar Yanev

Supervisor: prof. Michael Zakharyaschev

28th September 2020

Contents

1. Introduction. 4
1.1 Formal definition of finite automata 4
1.1.1 Deterministic finite automata DFA. 4
1.1.2 Non – deterministic finite automata NFA. 5
1.1.3 Epsilon closure. 5
1.2 Aims and objectives. 5
1.3 Project outcome and deliverables. 6
2. Background. 6
2.1 Introduction to simulation. 6
2.2 Applications of automata theory. 6
2.3 Existing technology. 6
2.3.1 Language based automata simulators. 7
2.3.2 Distinctive features of Visualization Centric simulators. 8
3. Main features and Project trailer. 9
3.1 Creating a new file. 10
3.2 Constructing features and tool panel. 10
3.3 Information feature and info panel 11
3.4 Open/Save features 12
3.5 Security constrains. 13
3.5.1 Constructing constrains. 13
3.5.2 Model integrity constrains. 14
4. System Design and Architecture 15
4.1 System Design. 15
4.1.1 Functional requirements. 15
4.1.2 Non – functional requirements. 16
4.1.1 User interface design. 17
4.1.1 System entities. 17
4.2 Software architecture 18
4.2.1 User interface layer 20
4.2.2 Application services layer 20
4.2.3 Domain model layer 20
4.2.4 Ports 20
4.3 Design patterns: 21
4.3.1 Mediator design pattern. 21
4.3.2 Command design pattern. 22
4.3.3 Model View Control design pattern (MVC). 22

4.3.4 Strategy design pattern. 23
4.3.5 Observer design pattern. 24
5. Development and implementation 24
5.1 Sprint 1: Domain Model. 24
5.1.1 Sub Sprint 1: Building DFA engine. 24
5.1.2 Sub Sprint 2: Building NFA engine. 25
5.1.3 Sub sprint 3: Building ε – NFA engine. 26
5.2 Sprint 2: User Interface. 28
5.2.2 Sub sprint 1: Developing basic interface components. 28
5.2.3 Sub sprint 2: Adding extra units. 28
5.3 Sprint 3: Application services. 29
5.3.1 Sub sprint 1: Adding action to sub menu items. 29
5.3.2 Sub sprint 2: Graphics 30
5.3.3 Sub sprint 3: Adding actions to tool panel buttons. 30
5.3.4 Sub sprint 4: Implementing the Mediator design pattern. 31
5.3.5 Sub sprint 5: Drawing graphics. 31
5.3.6 Sub sprint 6: Developing *DrawPanelController* adapter and Implementing the MVC design pattern. 33
5.3.7 Sub sprint 7: Implementing the Command design pattern. 33
5.3.8 Sub Sprint 8: Implementing the Observer design pattern. 34
5.3.9 Sub Sprint 9: Developing the *AutomatonController* adapter. 34
5.3.10 Sub Sprint 10: Developing *class Validator*. 34
5.4 Sprint 4: Final refinement. 35
6. Testing 35
6.1. Testing strategy. 35
6.2 Testing tools. 35
6.3 Unit testing. 35
6.3.1 Testing DFA engine 36
6.3.2 Testing NFA engine 36
6.3.2 Testing ε- NFA engine 36
6.4. User acceptance test. 36
7. Future recommendations 37
8. Summary and conclusions. 37
8.1 Lessons learned 38
9. References 39
Appendix A 40
Appendix B 44
Appendix C 49

Abstract.

The main concern of this project is to develop and implement an impressive, easy to use and navigate through application, allowing the user to construct, edit, modify and run a real time simulation on a graphically represented automaton models.

To aid the user when constructing the desirable models, the application is equipped with colourful self – explanatory buttons and menu item, two dimensional object representing the building components of any automaton model, section providing information in connection to the current model being constructing.

1. Introduction.

Automata theory is possibly one of the oldest and most researched areas in Computer Science. Over the past fifty years, several purposes of automata have been developed in an extensive spectrum of fields with a corresponding evolution of a variety of theoretical models. Early applications of automata theory included pattern matching, syntax analysis and software verification, where based theory was utilized to real world problems, resulting in the generation of beneficial software tools significantly in the domain of compilers [1].

The first description of finite automata was introduced by two neurobiologists Warren McCulloch and Walter Pitts in 1943, which intent was to illustrate the behaviour of the human brain through a model of neurons and synapses. The attempt contributed significantly in the domain of neuro network theory, theory of automata, the theory of computation and cybernetics [2].

Automata theory is an important branch of computer science which deals with the logical computation of a machine called **automata.**

Automata is a finite state machine model with only one purpose, performing computation based on sequence of symbols by moving through series of states. At each stage of the computation a transition function determines the next step based on the architecture of the machine.

1.1 Formal definition of finite automata

Finite Automata can be classified into two types:

- Deterministic finite automata DFA.
- Non – deterministic finite automata NFA.

1.1.1 Deterministic finite automata DFA.

A finite-state machine which accepts or rejects a predefined string of symbols, by running through a state sequence uniquely determined by the string. Deterministic refers to, for each input symbol, one can determine the state to which the machine will move.

DFA can be represented by a 5-tuple (Q, ∑, δ, q0, F).

- Q is a finite set of states.
- ∑ is a finite set of symbols called the alphabet.
- δ is the transition function where δ: Q × ∑ → Q
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- F is a set of final state/states of Q (F ⊆ Q).

1.1.2 Non – deterministic finite automata NFA.

For a specific input symbol, the machine is equipped for moving to any mix of the states in the machine. As such, the specific state to which the machine moves can't be determined.

NFA can be represented by a 5-tuple (Q, ∑, δ, q0, F).

- Q is a finite set of states.
- ∑ is a finite set of symbols called the alphabets.
- δ is the transition function where δ: Q × ∑ → 2^Q

  (Here the power set of Q (2^Q) has been taken because in case of NFA, from a state, transition can occur to any combination of Q states)
- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- F is a set of final state/states of Q (F ⊆ Q).

1.1.3 Epsilon closure.

The ε-closure of a particular state, is the set that contains the state itself, together with all states that can be reached starting at that state by following only ε-transitions.

Automata theory brings an immense importance to the domain of computation, thus it has become an irreplaceable part of every computer science curriculum. Even though its importance in the realm of computer science only limited automated digital tools are provided to the public, the majority of students still use the old fashion (pen and paper) when trying to construct automata models.

The limited numbers of provided simulation tools and the great importance of the subject to the field of computer science, were the main reason prompting the development of this project.

1.2 Aims and objectives.

The objectives of this project are to investigate the concept behind automated simulation and its visualisation.

Furthermore, the investigation process includes gaining sophisticated knowledge in the domain of automata theory and converting this knowledge into functioning simulating algorithms.

1.3 Project outcome and deliverables.

The primary aim of this project is to deliver a digitally educational tool which runs on a stand-alone machines providing the opportunity of constructing a multiple different graphically represented types of automata models and the options to edit, save, open and simulate these models.

2. Background.

2.1 Introduction to simulation.

A simulation is the imitation of the operation of a real – world process or system over time. The behaviour of a system as it evolves over time is studied by developing a simulation model, which usually takes the form of a set of assumptions concerning the operation of the system. These assumptions are expressed in mathematical, logical and symbolic relationship between the entities of the system. Once developed and validated, a model can be used to investigate a wide variety of "what if" questions about the real world system. Simulation can also be used to study systems in the design stage, before such a systems are built. Thus, simulation modelling can be used as an analytical tool for predicting changes to existing systems or design tool to predict performance of a new systems under the influence of set of circumstances [6].

2.2 Applications of automata theory.

Eventually in the improvement of PC equipment, engineers were over-whelmed by the errand of checking their equipment plans when highlight size decreased and the quantity of incorporated semiconductors surpassed many thousands.

Compiler scholars have perceived very early that different kinds of automata were unmistakably fit to comprehend and to actualize a few undertakings in accumulation finite state machines for lexical investigation, pushdown automata for punctuation examination and tree automata for code determination.

Current utilizations of automata hypothesis go a long way past compiler methods or equipment verification. Automata are generally utilized for displaying and verification of programming, dispersed frameworks, constant frameworks, or organized information. They have been furnished with highlights to display time and probabilities too.

2.3 Existing technology.

Due to the immense significance of automata theory in the field of computer science, different automata test systems have been created and utilized for instructive and scholastic purposes around the world over throughout the previous fifty years. Despite the fact that, the enormous number of previously existing re-enacting apparatuses for reproducing automata exists, established researchers is consistently open for new and better ones, which will proceed to contribute and improve to the educating cycle.

Automata simulators have been classified into two major groups based on their design paradigm. ***Language based automata simulators*** and ***Visualization Centric Automata simulators***.

2.3.1 Language based automata simulators.

This simulation approach is older amongst the two. There are two main steps to follow when using this type of simulation.

- First it has to be written as a program by using a symbolic language.
- Second to process the already written program. There are two ways to execute the simulation process.
    - First approach compiles and validates the program into intermediate language, an interpreter is used to create a simulation upon a string of input symbols.
    - Second approach directly uses interpreter, responsible for conducting lexical, syntaxes and error free analysis.

This group of simulator are divided into four sub groups.

2.3.1.1 Notational language based simulators.

The first existing tool to simulate automata type models using notational language was created by Coffin [3]. The main purpose of the simulator is to run a simulation on a Turing machine model. The process contains three steps.

- representing the model in the form of quintuples, each denoting transition.
- these quintuples are transformed into machine language by a program called *"The Builder"*.
- the actual simulation is executed by a third program called *"The Driver"*.

Pros and cons in Notational languages:

- Pros - easy to learn and define simple automata models.
- Cons – control and data abstraction are not supposed.

2.3.1.2 Assembly – like based simulators.

This kind of simulators includes a simple set of instructions.

- Return instructions.
- Conditional and unconditional jumps.
- Subroutine calls lead the executional process.

Assembly like simulators are capable of building more complex automata models in comparison with Notational language ones. Even though its superiority Assembly simulators share the same drawback e.g. inability to construct data abstraction.

2.3.1.3 Procedural language based simulators.

Providing the user with high level features for flow control, data abstraction and construction very large complex model, makes Procedural language simulators much more advanced than the Assembly based simulators.

2.3.1.4 Descriptive language based simulators.

Define by Chakraborty [4], Descriptive language simulators allow construction an automata model in the form of a written textbook – like way, without the knowledge of any programing language, unlike the other three which the requirement of knowing a certain language to some extend is essential.

The simulation undergoes through two phases.

- Compiling a written automata model. The compilation itself contains four parts.
  - lexical analyser
  - syntax analyser
  - sematic analyser
  - code generator
- Specially design interpreter simulates already compiled model.

2.3.2 Distinctive features of Visualization Centric simulators.

Main feature of this group simulators is the ability to demonstrate a working model usually accompanied with high quality graphics and animation.

Automata specifications are accepted as an input in predefined *structured* or *diagrammatic* forms. Simple tools like adjusting simulating speed and converting from one type to another are also available.

According to the nature of the input the Visualization centric automata simulators are clarified into two groups.

2.2.2.1 Visualization centric automata simulator accepting structured input.

Distinctive feature of this group of simulators is that it requires the user to fill in a form providing all the details of a desired automata model. This group is also famous with their easy to use feature due to quite a few of them have been developed over the years.

Figure 2.3.2.1.1 represents a **typical** simulator that accepts structured input is Automata en Java, a simple tool developed by Dominguez [5] to simulate a DFA.

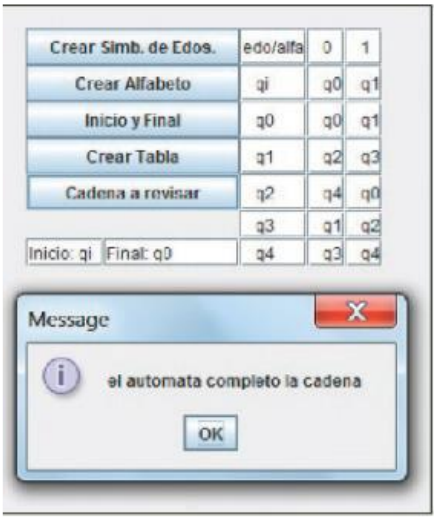

Figure 2.3.2.1.1. Simulation of deterministic finite automata using Automata en Java.

2.3.2.2 Visualization centric automata simulator accepting diagrammatic input.

One of the most successful and popular type of simulators ever developed. Normally, user is provided with a canvas where states and transitions are added and positioned by only clicking and dragging the mouse giving the user the experience of drawing automata on piece of paper. Due to its features Visualization centric automata simulators accepting diagrammatic input type has become the most favourite tool of simulating automata among students and teachers. As a result, several of simulators have been the developed.

Among all the most famous and sophisticated tool is Java Formal Language Automata Package (JFLAP).

JFLAP introduces to the user rich variety of options and friendly graphical interface, also it is the best documented tool for simulating automata. Here are some of the most important features that make this simulator so unique.

User is able to visually design a model e.g. draw, edit and simulate its work. The variety of models that can be design and simulate on JFLAP are: NFA, DFA, PDA, Turing machine, Mealy machine and Moore machine.

Another feature of the tool is, converting from NFA to DFA to minimal DFA.

Simulation can either be run in step – by – state mode, fast run mode or multiple run mode.

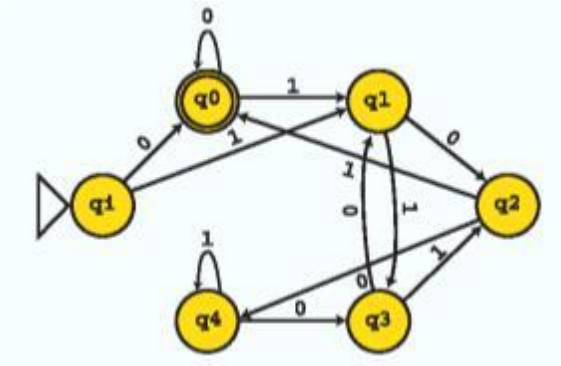

Figure 2.2.2.2.1. Simulation of deterministic finite automata using JFLAP.

This project focuses on building an application embracing the ***Visualization Centric*** paradigm accepting diagrammatic input, which can be used as a designing tool to predict performance of a systems under varying set of circumstances.

3. Main features and Project trailer.

The following sections provides a detailed overview of the final product and its main features.

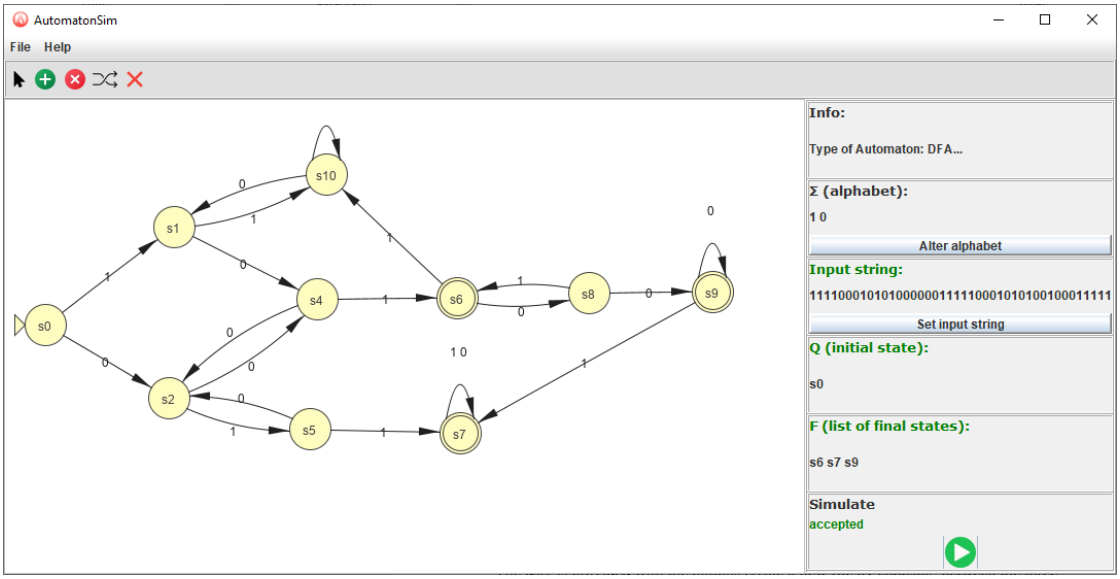

Figure 3.1 Full overview of the AutomatonSim application.

3.1 Creating a new file.

In order to create a new file, the user is provided with option, to choose between three different automaton models DFA, NFA, ε - NFA Figure 3.1.1.

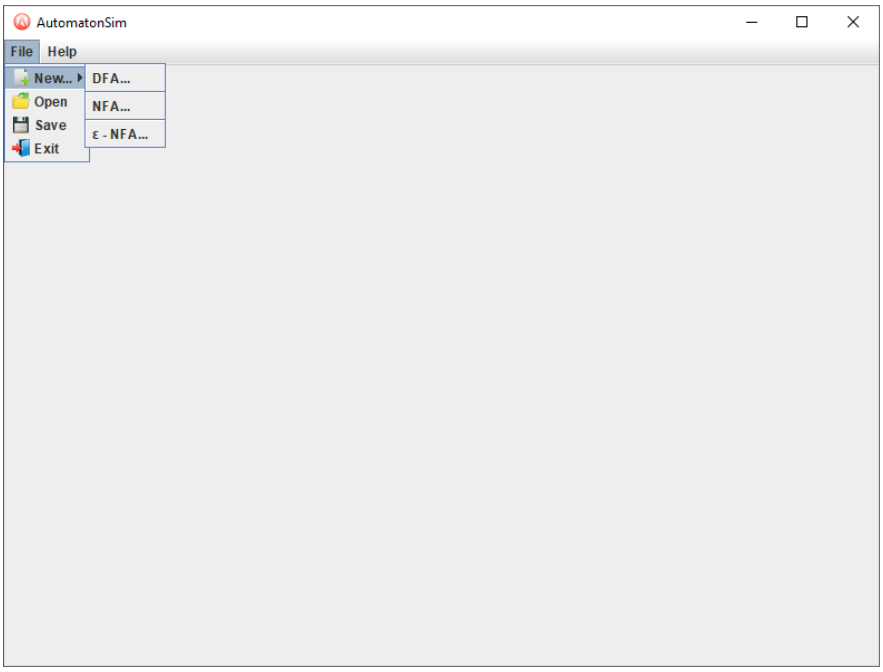

Figure 3.1.1 Showing the three optional automaton models available to the user.

3.2 Constructing features and tool panel.

The main idea which lays at the base of developing this product is to deliver a tool capable of allowing the final user to:

- Build graphically represented automata model and simulate it.
- Initialize alphabet despite its size and symbol content.
- To add/remove multiple states or transitions to a location solely chosen by the user.
- To change the type of the state dynamically as many times as the user desire.

In order to satisfy this requirement a tool panel is developed containing five buttons responsible for coping with user's preferences when constructing a given model. Figure 5.2.1 illustrates the functionality of the five tool buttons.

- Add new state button: It adds a new state object to a location chosen by the user by simply clicking over a certain location of the drawing area.
- Remove state button: It allows the user to remove every state (security restriction apply) from the current model by simply clicking over a state which has is desirable to be removed.
- Add new transition: It adds a new transition to the model with transition symbol chosen by the user (security restriction apply).
- Remove transition: It removes an existing transition at chosen symbol.
- Alter State: it allows the user to alter multiple times, the type (Initial, Final, Initial/Final or None) of a chosen state (security restriction apply).

Figure 3.2.1 illustrates the tool panel and the functionality of the five tool buttons.

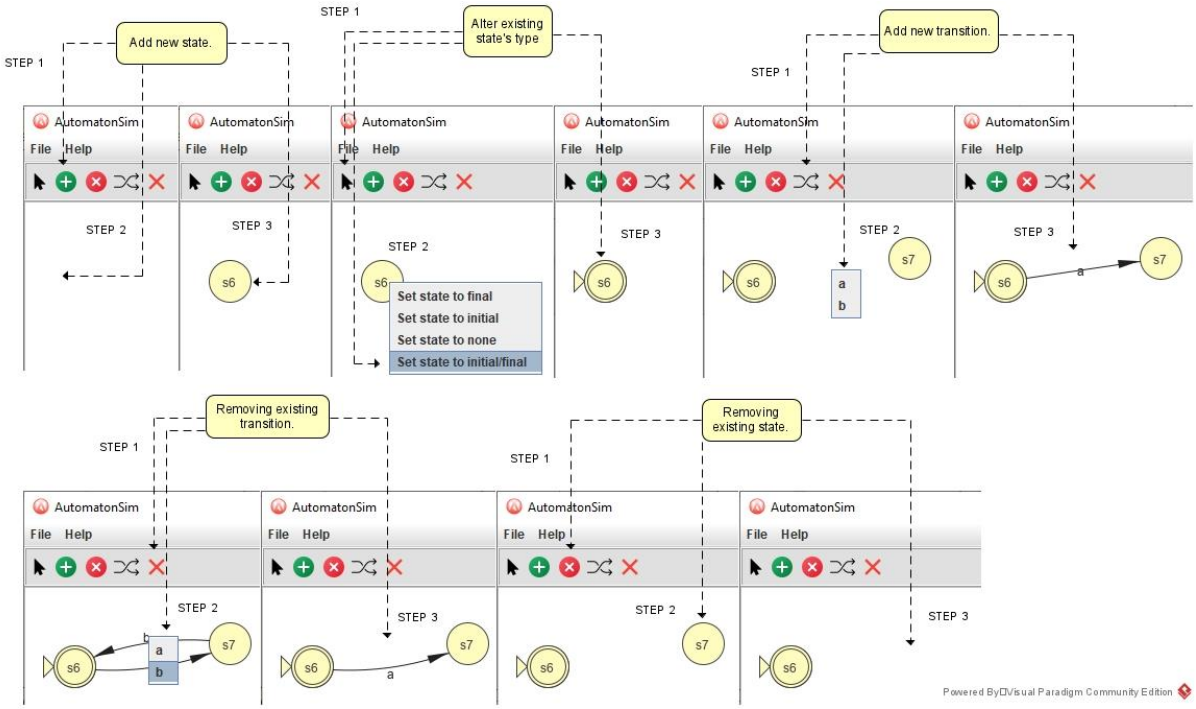

Figure 3.2.1 Tool panel containing five tool buttons.

3.3 Information feature and info panel

Along with constructing tools the user is also provided with information tools as well.

This feature is handled by the info panel which contains six sub panels responsible for displaying a vital, dynamically changing during the building process, information in connection with a currently automaton model.

Figure 3.3.1 illustrates the full list of info panel's features.

- Info sub panel displays general information of what type (DFA, NFA, ε - NFA) is the model upon which the user is currently working.
- Alphabet sub panel displays the current alphabet used. Also another interesting property provided to the user by this panel is the "*Alter alphabet*" button allowing dynamical altering of the existing alphabet (restrictions apply).
- Input string sub panel provides the user with the ability to dynamically initialize multiple times input string to be tested.
- Initial state sub panel displays information whether the initial state has been set or not.
  If an initial state has not been initialized the panel's label glows red and displays none, otherwise it glows green and displays the name of the state chosen to be initial.
- Final state sub panel holds similar features as the Initial state panel, but instead of displaying the initial state, it shows the list of final states.
- Simulate sub panel provides information about the outcome of the simulation (accepted or rejected), also holds the simulation button, which can be used multiple times simulating a variety of different input strings.

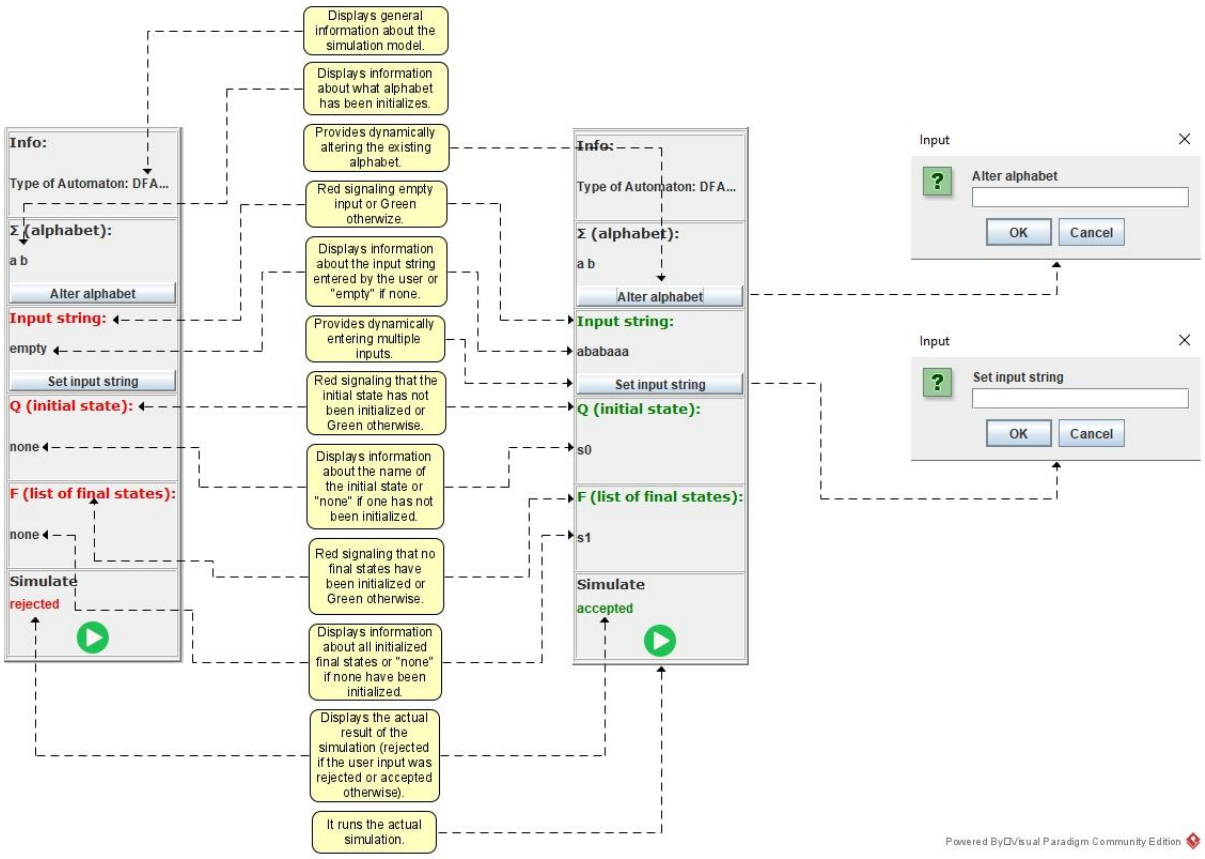

Figure 3.3.1 Features of the info panel.

3.4 Open/Save features

In addition to the ability of graphical modelling, the application also provides another two very useful properties, allowing the user to save (into any chosen directory and name) already constructed model or to open another already previously saved.

Figure 5.4.1 and Figure 5.4.2 depict these two features.

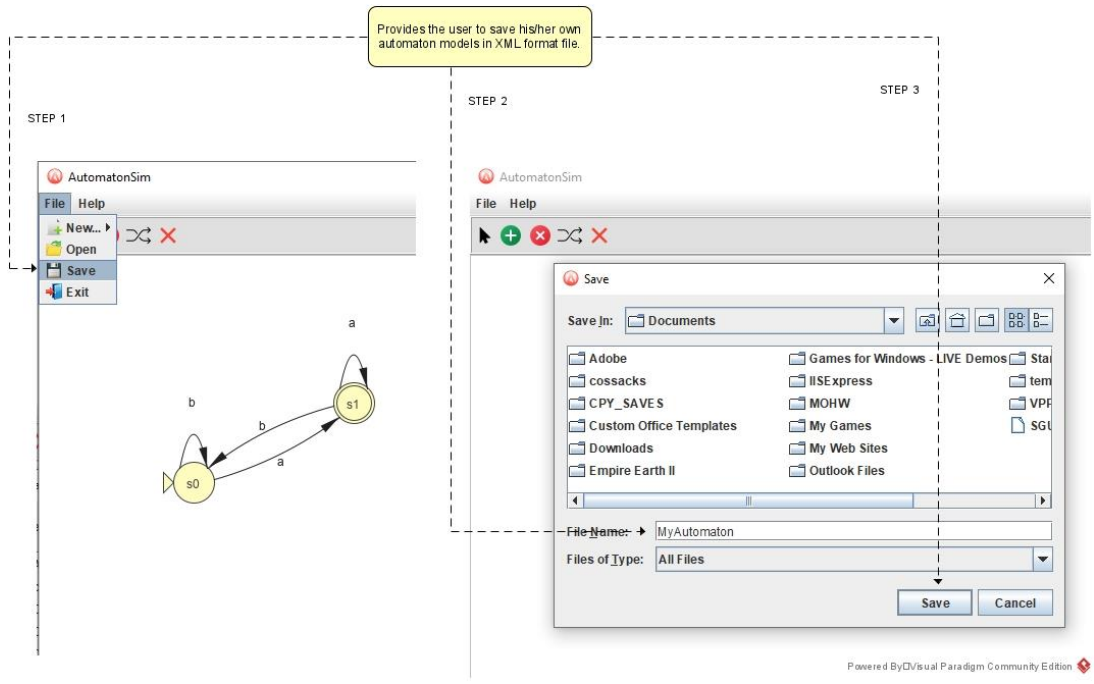

Figure 3.4.1 Illustrating the save menu item feature.

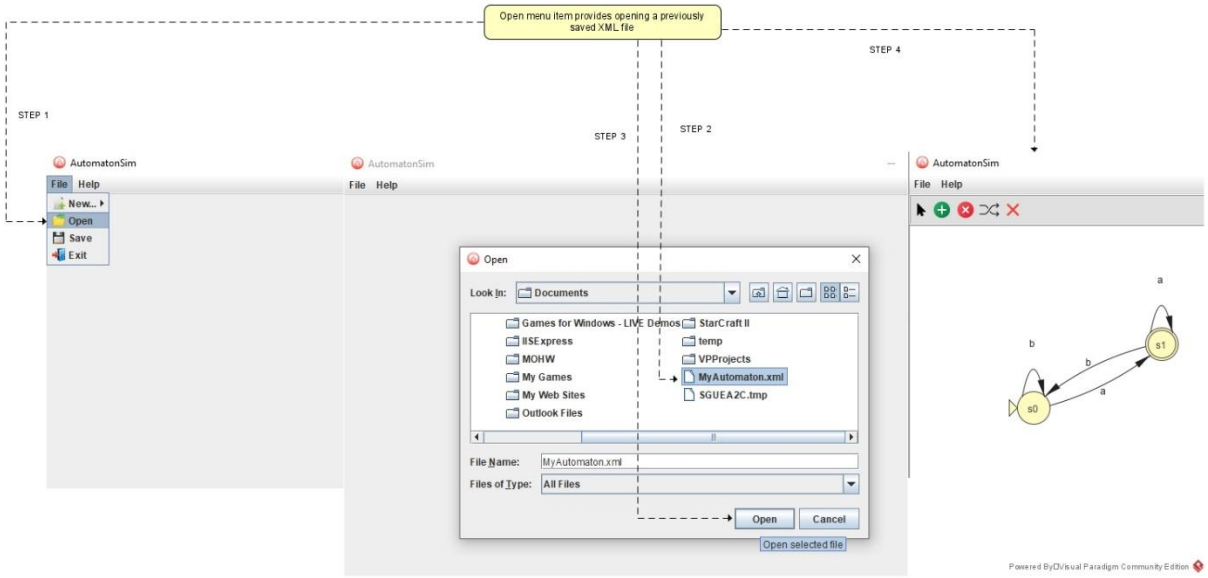

Figure 3.4.2 Illustrating the open menu item feature.

3.5 Security constrains.

Besides all options delivering freedom of choice, the application is also supplied with several security constrains adding extra highly beneficial assistance to the user. There are two main levels of constrains imbedded in the application.

3.5.1 Constructing constrains.

These constrains are applied during the development of any automaton model. The main purpose of the restrictions is to provide additional help to users who are new to the field of automata theory and may easily get confused during the process of building big models containing multiple states, transitions and long alphabets. Figure 3.5.1.1 and Figure 3.5.1.2 illustrate these restrictions.

These constrains are:

- User is restricted to initialize no more than one initial or initial/final state in one machine thus keeping the integrity of the model being under construction. If an initial state has been already established and the user wishes to initialise another one instead, then the already initial state has to be removed from the model or its type must be altered to NONE or FINAL.
- When adding transitions, the user is allowed to use only symbols from already initialised alphabet. If adding an additional transitions containing symbols not initialised in the alphabet is desirable, then "*Alter alphabet*" provides solution by adding or removing symbols to an existing alphabet or completely initializing a new one. If this is the case, a warning message is displayed informing the user that all previously created transition which do not contain symbols found in the newly established alphabet will be automatically removed.
- User is not allowed to remove a state which is a part of transition/s. In order to remove such a state, the user must remove all transitions in which the selected state participates, first and then remove the state itself.
- User is not allowed to save a non-existing automaton file.
- User is not allowed to open a file which does not belong to the application or any other file of type different than XML.

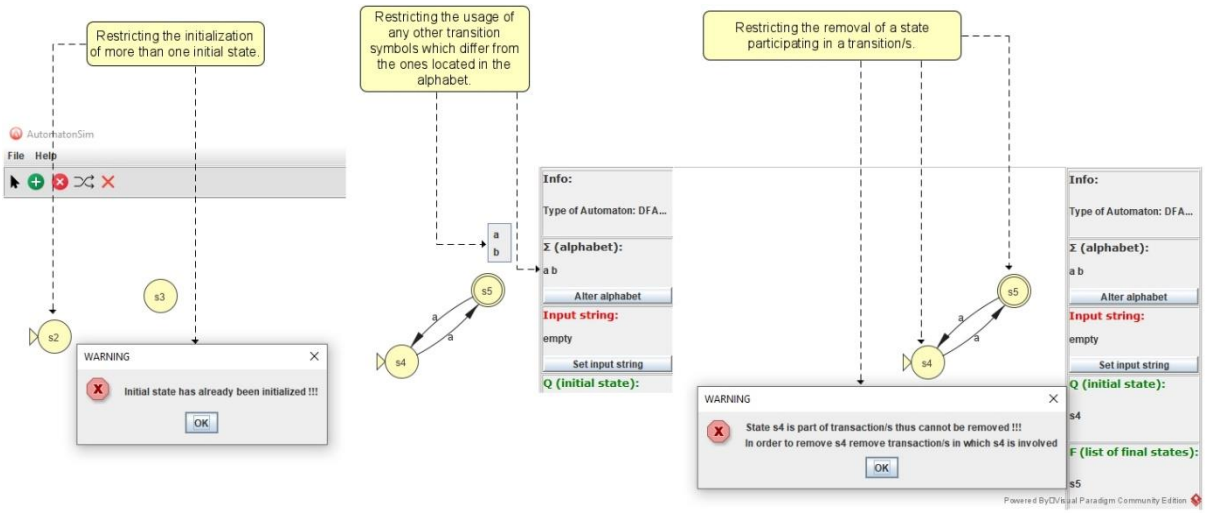

Figure 3.5.1.1. Security constrains applied during the building process of a sample model.

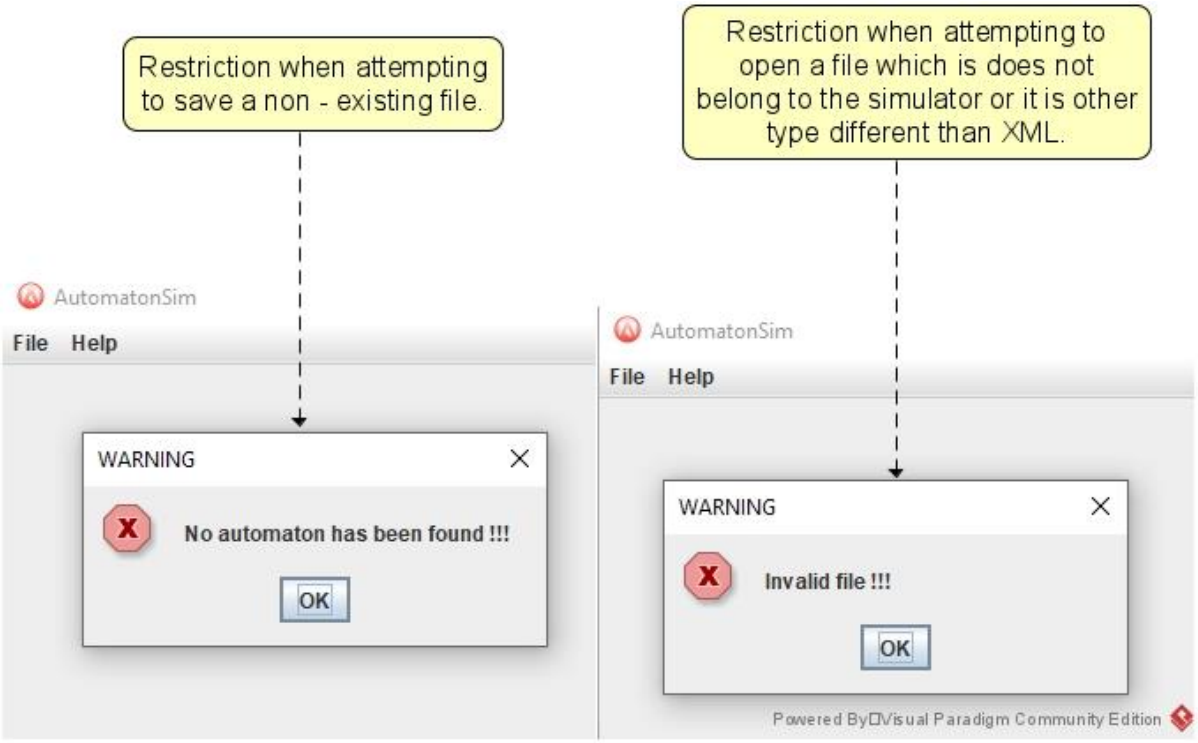

Figure 3.5.1.2. Security constrains applied during the building process of a sample model

3.5.2 Model integrity constrains.

These constrains are applied before executing the actual simulation. The main intention of the model integrity constrains is to provide the user with general information of *"what went wrong"* and why the simulation cannot be executed, and second, provides security assurance that the information, needed for executing the simulation process, is correct and will not lead to crashing the execution itself. Figure 3.5.2.1 demonstrates these constrains.

Automata model cannot be simulated when:

- There is no initial state.
- There is no final state.

- There is no input string to test against or contains symbols which are not presented in the alphabet.
- The structure of the model is incomplete (there is a state with no incoming or outgoing transitions).
- Its invalid DFA (it applies only to DFA machines).

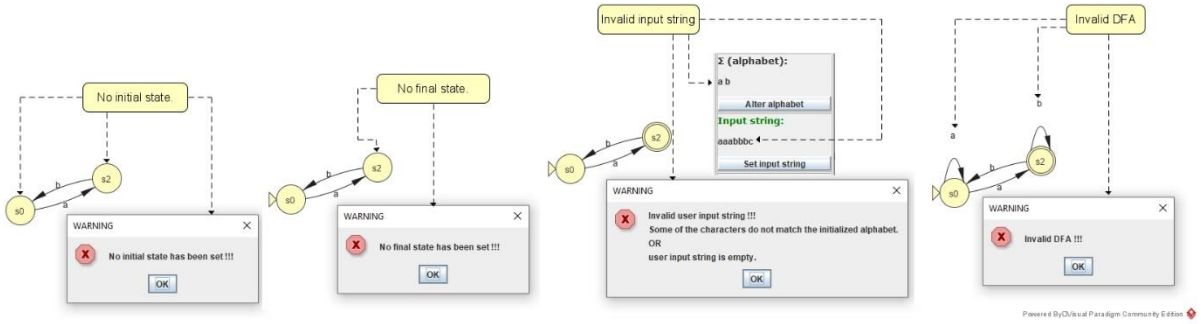

Figure 3.5.2.1 Model integrity constrains.

4. System Design and Architecture

4.1 System Design.

This application is design as an object oriented for a Desktop – based architecture using three-layer architecture, which could be used on standalone machines only.

4.1.1 Functional requirements.

This section outlines what the system should do when certain conditions are met.

The system should be capable of:

- Creating a working panel, equipped with all components needed to build and simulate a models, of type (DFA, NFA. ε -NFA) chosen only by the user.
- Taking and storing user's alphabet input regardless of size and symbol content and store it for further use.
- Verifying user's alphabet for any repeating symbols that may occur, and if it does find any, to removes all duplicates.
- Rendering all entities (states, transitions, transition symbols), created by the user, in the form of graphical objects (circle, arrow/curve or arc) at dynamically selected location.
- Providing the user with reliable security when:
    - adding a symbol to a particular transition
    - altering previously initialized alphabet with a new one
    - changing the type of a chosen state.
- Providing useful information in connections with the model on which the user is currently working on.
- Informing the user, in case of altering an existing alphabet, that all transition objects which do not contain symbols from the new alphabet will be automatically removed.
- Verifying the model's validity before executing any simulation process in order to:
    - protect the simulating engines from receiving an invalid data, hence crashing the simulation process.

    - deliver practical information about why the simulation cannot be proceeded.
- In case of:
    - "save" command being selected - creates data carrier object, to encapsulate all vital data (states, transitions, alphabet, user input string and model type), and convert it to a XML document.
    - "open" command being selected - ratifies the selected file, if the file is valid, converts it back to data carrier object and use the object to restore a previously saved model.
    - "exit" command being selected – checks if there is an existing model and if so provides the user with choice whether to save or not the existing model and exit the program.

4.1.2 Non – functional requirements.

This section describes how the system should behave and what limits are there on its functionality.

The system should provide the user with:

- Friendly and easy to use interface.
    - self-explanatory buttons and menus.
    - big enough drawing field space.
- Ability to select between three different models to build and simulate.
- Options to save already developed projects in a different directories chosen by the user and open already saved files back to the system.
- Capability to:
    - create unlimited numbers of graphical components needed to build a successful model.
    - remove these objects at any point of time.
    - create alphabet despite of its size or symbol content.
    - test variety of different input string regardless of their length.
    - perform simulation countless times over the same model at different input string.
    - Observe whether key elements needed for successful simulation have been created.
- Restrictions not to:
    - Set a transition symbol which do not exist in the current alphabet.
    - Create state of type INITIAN or INITIAL/FINAL at the presence of another state with the same type.
    - Create any graphical object outside the drawing panel
    - Remove state whit a transition/s already assign to it.
    - Initialise empty alphabet.
    - Perform simulation when:
        - There is no initial and final state.
        - There is no input string.
        - The input string contains symbols which do not exist in the alphabet.
        - The model is invalid (implies only to DFA type models).
    - Save a non-existing model.
    - Open a file of an incompatible type.

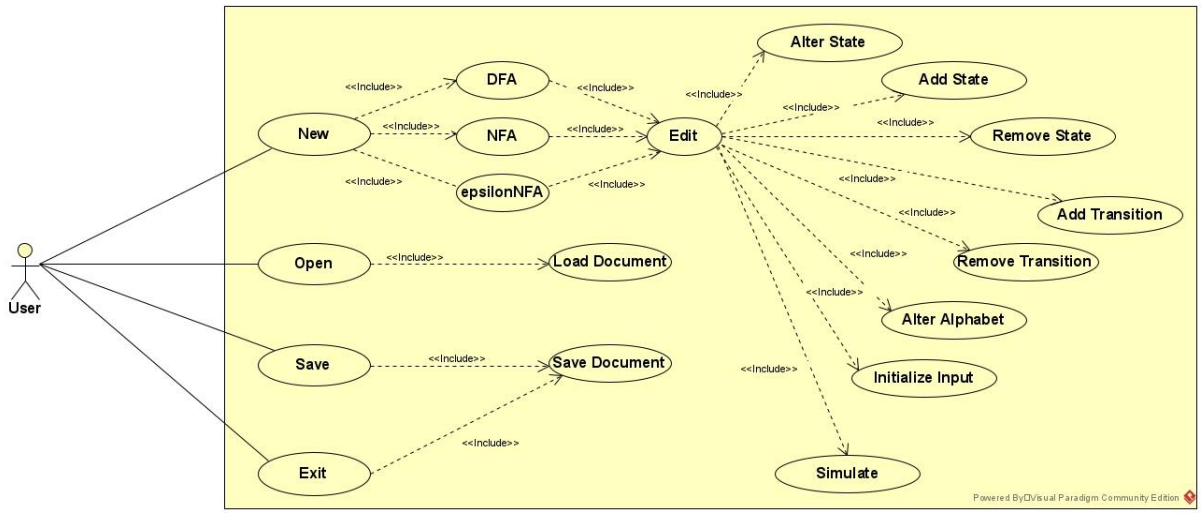

Figure 4.1.1 Use case illustrating the user's interaction with the system.

4.1.1 User interface design.

The main three goals of the user interface are to deliver an impressive, simple and easy to navigate through access point enabling users to interact with the rest of the system.

In order to achieve these goals, the user interface is equipped with the following features:

- **Single purpose** – To avoid ambiguity, every element serves only a single purpose.
- **Predictable performance and maintain high discoverability –** All action performing elements (buttons and menu items) are fitted with self-explanatory components e.g. icons, texts informing the purpose of the element and tool tip texts, informing the user what actions they can handle.
- **Proper alignment –** Edge alignment principle. This is very common technique which naturally positions elements against a margin that matches up with their outer edges.
- **Draw attention to key features –** sub panels, responsible for rendering key elements e.g. presence of initial state, final state and user input string, alter the colour of their labels according to the presence of a certain element (red when the element is not presented and green otherwise).
- **Place control near objects which user aims to control –** pop up windows providing list of selective symbols used for choosing what transition symbol to be inserted when creating a transition or what transition to be removed at specified symbol, list of options when user attempts to change the type of a particular state.
- **Keep user informed with respect to system responses –** pop up warning windows notifying the user of any changes or exceptions which may jeopardise the execution of the application.
- **Clear elements visibility –** all elements are fitted with highly distinctive attributes (borders, colours, icons)

4.1.1 System entities.

- ***Data objects*** – an objects with predefined structures capable of holding data in a structure which is quickly and easily accessible by other parts of the system.

    - *class Automaton2D:* Stores all graphical objects in a sequential list data structures.
    - *class DataCarrier:* Object of that class are used to stores vital data in a sequential list data structure. When save operation is executed the object is

converted into XML document and when open operation is executed the XML document is converted back to an object.

- ***Communication objects*** – a group of objects qualified to handle communication between other objects.

    - *class PanelMediator:* Qualified to handle communication between all components of the main panel (Tool panel, Info panel, Draw panel, Tool panel buttons, Info panel buttons).
    - *class FrameMediator:* Capable of dealing of communication between all components residing in the main frame (Menus, Menu items, Bar, Menu Bars).
    - *class MainMediator**:*** Unifies and provides communication among the two sub mediator classes.

- ***Command transmitting objects*** – objects which, first receive a command and second transmit it to the right execution object.

    - *class ButtonCommandInvoker*
    - *class ItemCommandInvoker*
    - *class SubPanelCommandInvoker*

- ***Rendering objects*** – responsible of displaying multiple graphical objects at the same time.
    - *class DrawingPanelImpl:*

- ***Execution objects*** – objects capable of executing actions.

All sub classes which extend the following abstract classes:

- *class ButtonExecutor*
- *class ItemExecutor*
- *class SubPanelButtonExecutor*

- ***Graphical objects*** – dynamically created objects with ability of graphically representing other objects (States, Transitions and Transition symbols).

All classes and interfaces implementing/extending the following interface:

- *interface Shape2D*

- ***Adapter (controller) objects*** – an objects serving as entry points between the three main layers.
    - *class AlphabetController*
    - *class AlterStateController*
    - *class DrawPanelController*
    - *class AutomatonController*

4.2 Software architecture.

At first the traditional layered architecture was chosen as a main architectural pattern to be implemented during the developing process, but after a thorough and further research the idea

of using layer architecture was replaced with the newly developed by ***Jeffery Palermo*** *onion software architecture.*

The main reason of choosing *onion software architecture* over the *layered software architecture* is:

In the traditional layer software architecture all the layers depend upon each other. The Domain services layer depends on the application services layer, user interface layer depends on the application service layer, virtually all sub layers depend on the top layer, which causes tight coupling between layers and leads to violating the *Dependency inversion principle*.

The main difference between the two architectures is the direction of the dependencies.

In the Onion architecture, there is a main rule: ***outer layer can depend on lower layers but not vice versa.***

The entire application structure is organised into three main layers. Each layer is responsible for delivering a different functionality to the system, as depicted in *Figure 6.2.1*. The three layers are:

- User interface layer.
- Application services layer.
- Domain model layer.

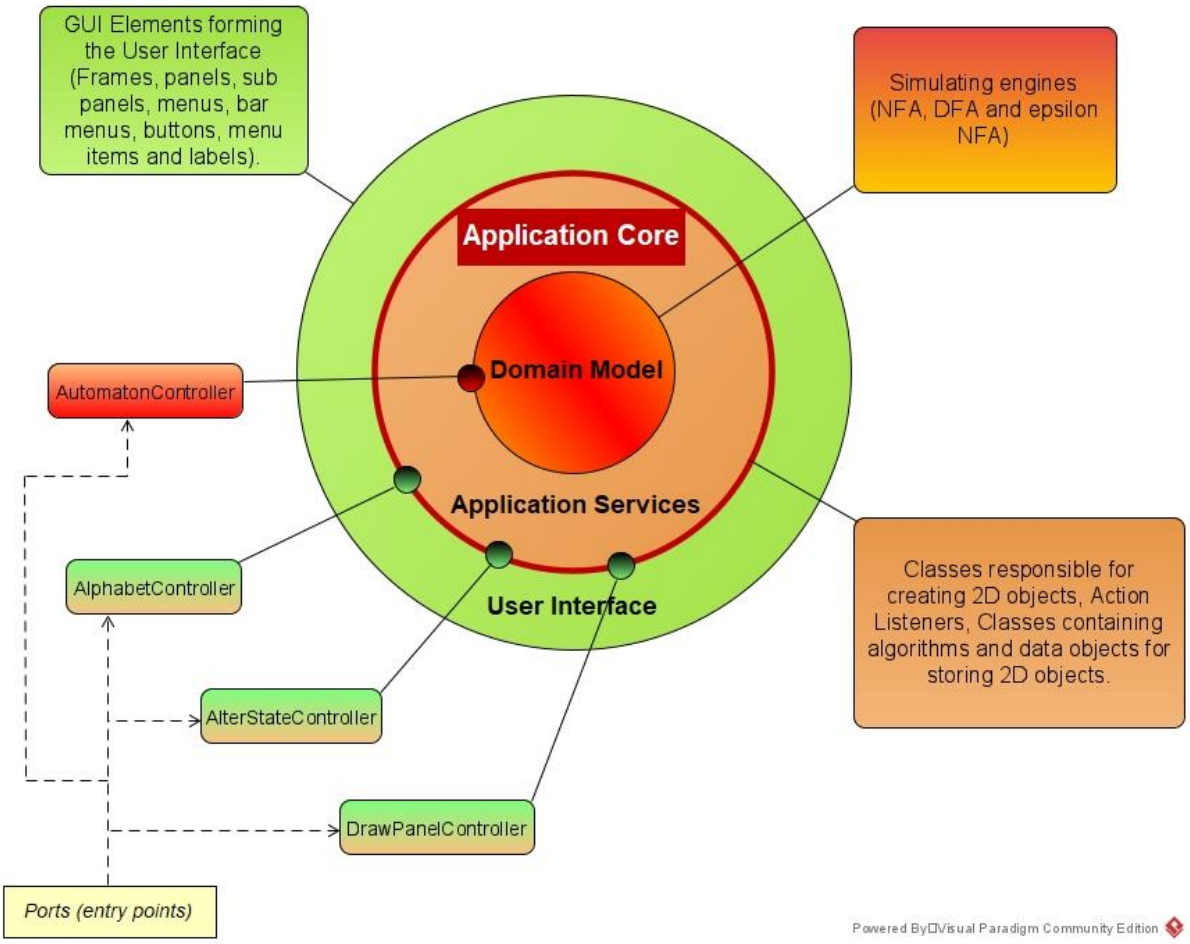

Figure 4.2.1: Graphical representation of the system's architecture.

4.2.1 User interface layer

This is the top layer of the application which has a two main purposes, first to render all the GUI components like panels subpanels buttons menus and also visual representation of the automaton and all its components (States, transition arrow and transition symbols), and second to take and forward any user input into the next layer to be processed. In other word the top layer is also used as a communication tool between the user and the *Application service layer*.

4.2.2 Application services layer

This is the second layer which is located between the User interface and the actual Domain model. The main responsibility of the Application service layer in general is to process input passed down from the User interface layer and return back a result which may be in the form of a pop up warning, conditional, or informing type of windows, altering the visual representation of a GUI object, depicting a 2D graphical object (State, transition arrow, transition symbol).

In more details the service layer contains classes responsible for creating and storing two dimensional graphic objects, classes accountable for executing complex algorithms, exception handling, making decisions based on the user's input, changing the shape of a state or transition arrow, replacing the value of a transition symbol for example, restricting the user of entering information that may damage the executional flow of the program.

4.2.3 Domain model layer

This is the bottom and most important layer of the application, it's the substantial functional core responsible for processing and running the actual simulation. It contains three different engines which implement all the rules and algorithms needed for the successful execution of the three different types of automaton (DAF, NFA, ε - NFA).

The Application service and the Domain model layers are very similar. They both contain algorithms and apply rules. The only difference between the two layers is that, the rules and algorithms in the service layer may frequently change over time, while rules and algorithms in the Domain model layer do not change, they remain the same.

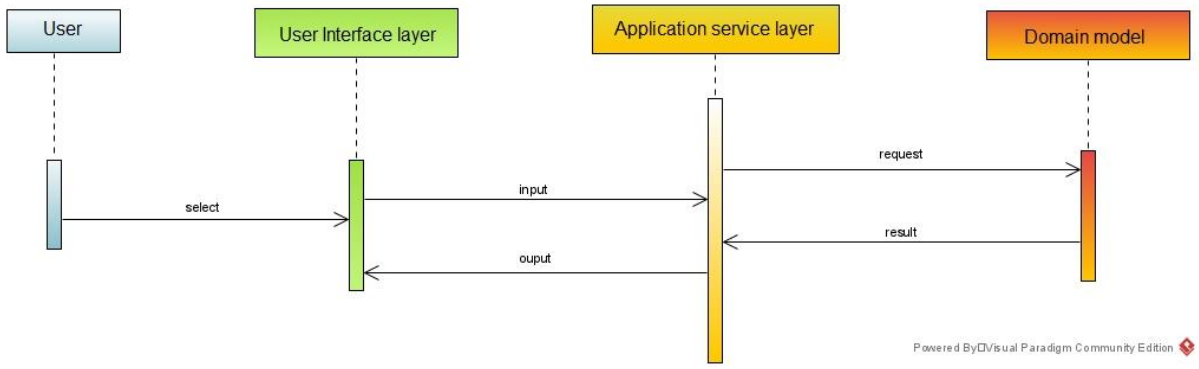

Figure 4.2.2: Interaction diagram between the user and the three main layers.

4.2.4 Ports

These ports (adapters) are not randomly created. They are created to fit a very specific purpose, to provide an entry points via which all three layers are connected with each other.

It is important to note that, these ports reside inside the application service layer.

Four adaptor classes have been used to serve the purpose of entry points. These are:

- *DrawPanelControler.class*
- *AlterStateControler.class*
- *AlphabetControler.class*
- *AutomatonControler.class*

The first three adapters care used to connect the user interface with the service layer. *AutomatonControler.class* is the only adapter connecting the Domain model with the Application service.

4.3 Design patterns:

There are five main software design patterns used in the development of the application.

- Mediator design pattern
- Observer design pattern
- Command design pattern
- Strategy design pattern(graphics)
- Model View Control design pattern (MVC).

4.3.1 Mediator design pattern.

Due to the growing number of component and the necessity of simple and reliable communication and collaboration, the mediator design pattern has been chosen to satisfy these requirements Figure 4.3.1.

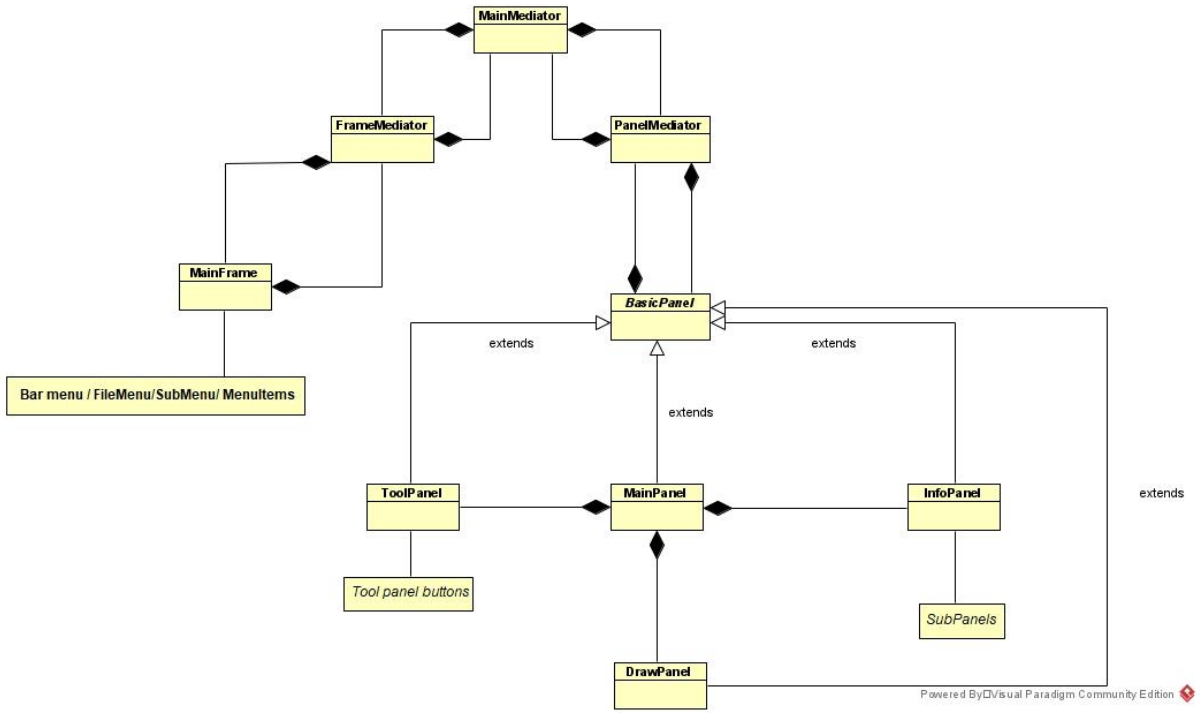

Figure 4.3.1. Simplified class diagram of the user interface layer showing the utilization of the mediator design pattern (attributes and methods omitted for clarity).

As presented in *Figure 4.3.1* there are two "sub" mediator classes connected with each other through the *MainMediator* class thus creating an elegant network and simplified communication (handled by each of the mediator classes) between all the elements in the application.

By applying the mediator pattern provided easily adding or removing new components to the system's communicational grid without braking any existing code.

Mediator pattern plays vital role especially when communication between action objects (buttons menu items) with other objects (panels, frames, subpanels) is crucial.

4.3.2 Command design pattern.

Following the nature of the application's architecture where all three layers should be separated and vary independently, adding functionality straight into the action components (menu items and buttons) wasn't the best option, another approach immerged after a few attempts to keep the user interface and the service layer decoupled, and also add functionality at the same time.

In this case the command design pattern provides the best solution to the problem. Instead of executing an algorithm directly inside the class where a certain component is located, the action is executed in a class situated inside the service layer specially design for the purpose. The component only gives a command which is wrapped by a wrapper class and passed down to the executor where the actual execution occurs.

Figure 4.3.2.1 demonstrates the application of the command design pattern to object of type *ToolPanelButton* where it adds functionality without coupling the two layers.

The button object issues a specific command which is received by the *ButtonCommandInvoker* class, which in turn recognises the command wraps it in the appropriate executor object which performs the actual execution.

There are three different types of objects using the command pattern in the application. These are objects of type *ToolPanelButton*, *SubPanelButton*, and *MenuItem*.

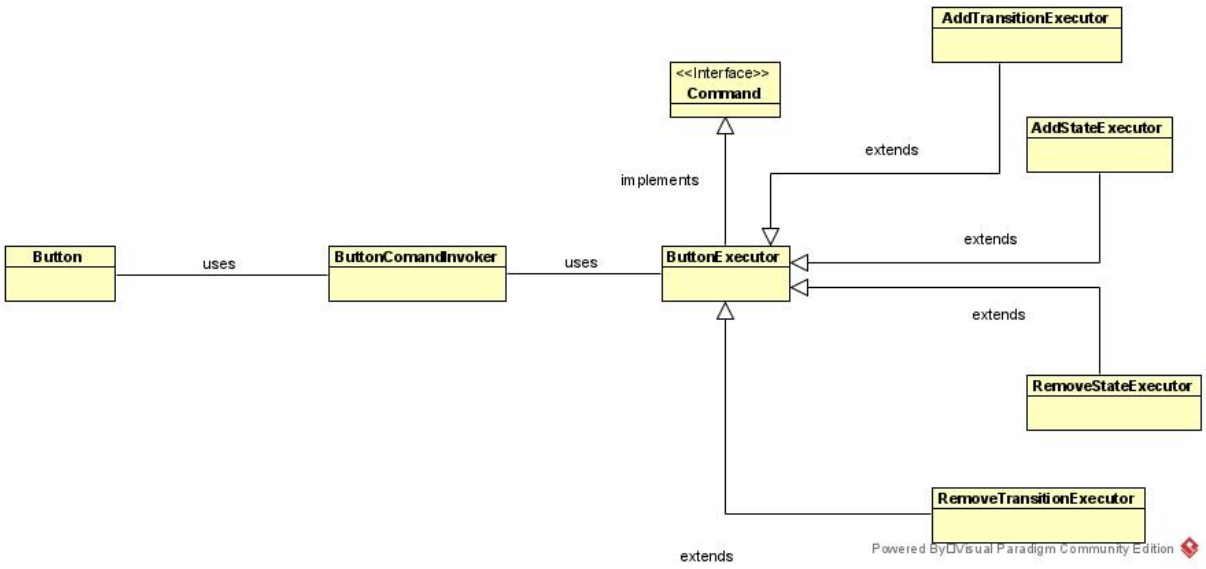

Figure 4.3.2.1 Example of Implementing the command design pattern in order to separate the user layer from the service layer (attributes and methods omitted for clarity).

4.3.3 Model View Control design pattern (MVC).

Even though all the layers are separated from each other, some sort of unification is required for the system to function as a whole. The necessity of decoupling view from function and providing some sort of bridge between gave birth to the usage of the MVC design pattern.

All adapter objects used for providing entry points between the three layers implement the MVC design pattern.

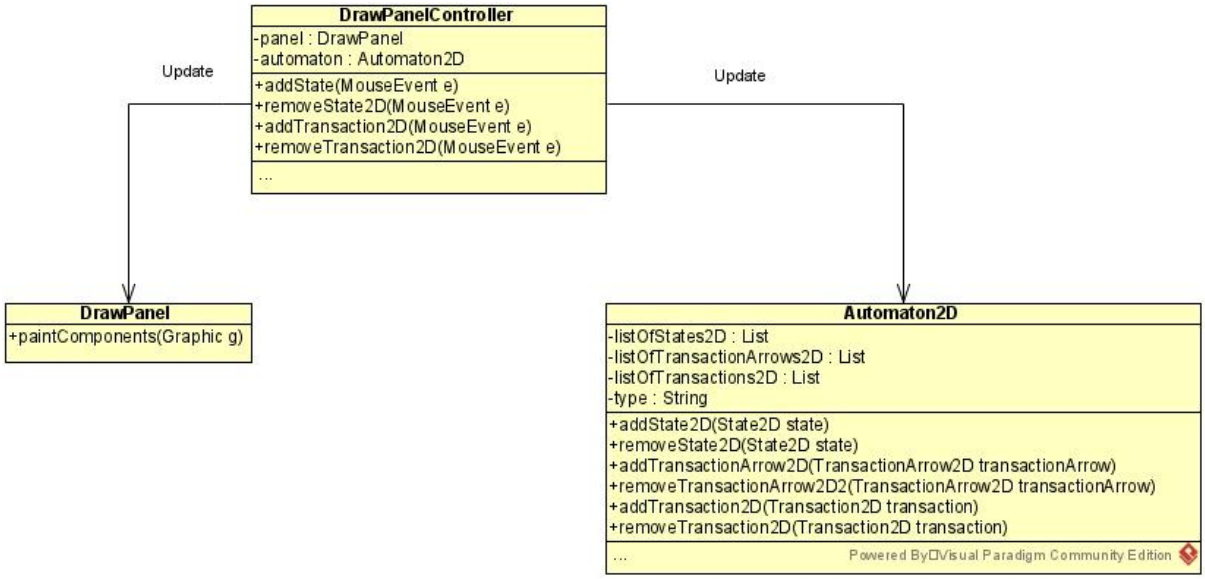

Figure 4.3.3.1. MVC pattern UML diagram.

4.3.4 Strategy design pattern.

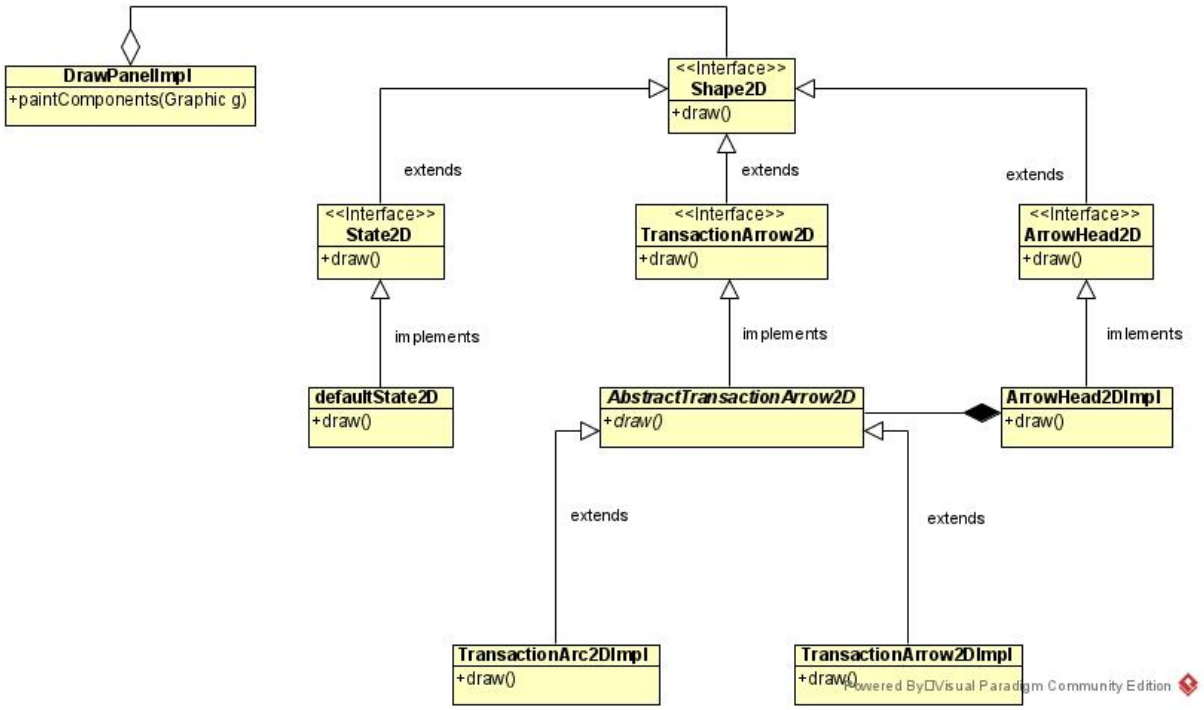

Figure 4.3.4.1. Strategy design pattern UML diagram (attributes and some of the methods omitted for clarity).

4.3.5 Observer design pattern.

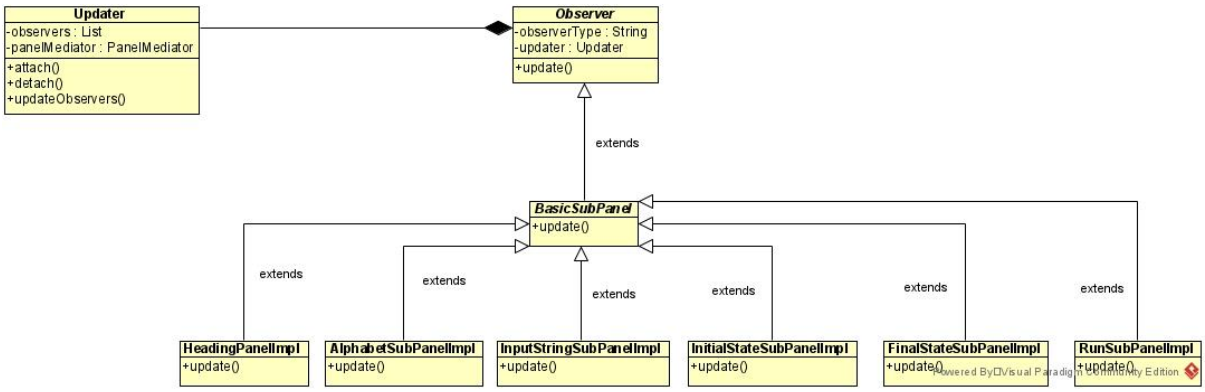

Figure 4.3.5.1. Observer design pattern UML diagram (attributes and some of the methods omitted for clarity).

5. Development and implementation

At first the agile development methodology was chosen but due to the unfamiliarity with the GUI library and Graphic2D objects, Prototype development model was employed during the software development.

The whole development process is divided into four main sprints in addition each of them was divided into multiple sub sprints. During each sub sprint every sub system (participating in the sub sprint) has undergone through multiple refinements and iterations.

- Sprint 1: Domain Model.
- Sprint 2: User interface.
- Sprint 3: Application services.
- Sprint 4: Final refinement.

5.1 Sprint 1: Domain Model.

The main objective of the initial sprint is to deliver a system capable of simulating three different types of Automaton models (NFA, DFA, epsilon NFA), which leads to splitting the whole process into three sub processes.

5.1.1 Sub Sprint 1: Building DFA engine.

During this stage, the simples and most important amongst the tree simulation engines was created, the DFA simulating engine.

The main idea behind of developing an effective simulation is to build a method which takes the four building components of a real Automaton model (list of states, list of transitions, alphabet and the user input) as an input, runs a simulating algorithm and returns back a result (accepted or rejected).

The workflow of method *runSim* is as follows:

- First step is to find the initial state by using the auxiliary method *getInitial*, and assign the returned value to a variable “tracker” (of type State) which is used as a starting point and also to keep track of the execution at later stage.
- Second it enters a loop which length is determined by the size of the user input being tested.

- Third at each iteration a single character of the user input is retrieved and used as a comparison tool against the transition symbol of each transition.
- Forth a stream of transition objects filters only those transitions which satisfy two conditions: startState of the transition equals the tracker State and second the transition symbol of the same transition equals the input symbol being tested.
  In the end if there is a transition that has passed the two conditional rules, the value of the tracker object is replaced with the value of the transition's endState value, thus in the next
  iteration the execution will not start from the initial state but from the state where previously was ended.

  Step 2, 3 and 4 are repeated at each iteration.

- Fifth and final step. At the end of the loop the "tracker" object goes into a validation stage, if the object's type is of type FINAL or INIFIN (initial/final) the method returns true otherwise returns false.

For the actual simulation code please refer to Appendix B.

Figure 5.1.1.1. shows the graphical representation of the simulation process applied to a sample automaton model.

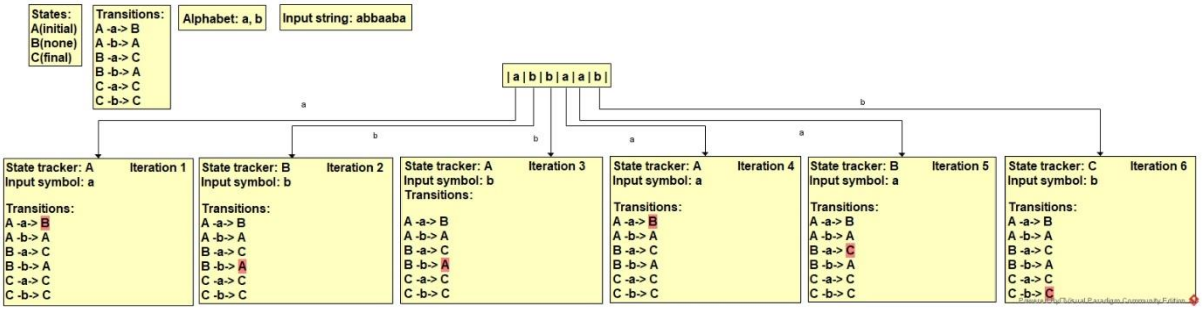

Figure 5.1.1.1 Graphical representation of simulating a sample DFA Automaton.

5.1.2 Sub Sprint 2: Building NFA engine.

During the second sub sprint the NFA engine was built.

The initial idea was to create an algorithm similar to the DFA simulating process e.g. to run NFA directly but after a few attempts it turned out it is impossible to simulate a non – deterministic model by using a deterministic approach.

The philosophy behind the second technique used for developing the software follows the statement "*every NFA is DFA*" scilicet every NFA can be DFA after transformation therefor instead of running directly NFA, the algorithm focuses on converting NFA into DFA and after being converted the transformed automata is passed down to the DFA simulating engine which performs the actual simulation.

Even thou the second idea was very elegant solution, the task of writing algorithm which converts NFA to DFA remained a challenge.

The algorithm which carries the task of conversion follows the same methodology as a human being converting NFA to DFA.

The whole process of transformation is divided into a four steps:

- Step 1: For each state find the possible set of states for each input symbol using transition function of NFA.
- Step 2: Combine these state/s into a single state, thus a new state is created and added to the list of states.
- Step 3: Create a new transition/s
- Step 4: Repeat step 1, 2 and 3 for every newly created state.

By following these for steps the process of developing a working transformation engine is broken down into five stages.

The first four stages involve building auxiliary methods used for dividing and simplifying the conversion process.

- *Stage 1:* During this stage a method ***subTransactionGenerator*** was developed that satisfies the requirements of "Step 1".
  The method takes a symbol, single state and list of NFA transitions as arguments, loops through the transition list and filters only those transitions where the input state is the first state at the given input symbol.
  After filtering it creates a new sequential list containing the filtered transitions and returns it, if the input state has no transition assign to it for the given input symbol an empty list is returned.
- *Stage 2:* The development of another helper method ***compositeStateGenerator,*** which covers "Step 2", took place, its main intent is to take a list of transitions convert it to a stream, maps the second state from each transition and finally creates a list of all states that have been mapped, thus finds all states at the given input symbol by using the presented list of transition functions.
- *Stage 3:* Establishing of method ***convertSubListOfStatesToState.*** The solely purpose of this method is to take a list of states as an argument and join them into a single state. Also verifies the type of each state and if there is a state with type different than NONE and INITIAL type FINAL is assign to the newly generated state.
- *Stage 4:* Creating methods ***addDeadState.*** This method finalizes the conversion process as it takes a list of DAF transitions, creates a stream, filters only those transition which contain an empty first, second or both states and replaces those states with a dead state.
- *Stage 5:* The fifth and final stage of developing the transformation engine includes building the final method ***convertNFAtoDFA*** where a given list of NFA transitions undergoes a full transformation and a converted list of DFA transitions is returned. In order to achieve transformation, the method uses all the auxiliary method previously developed.

For full textual and graphical description of how ***convertNFAtoDFA*** method works and the actual code please refer to Appendix A.1 and Appendix B.1.

5.1.3 Sub sprint 3: Building ε – NFA engine.

The third and longest sub sprint involves building the last and most complex amongst the three engines, the ε – NFA simulation engine.

The building process follows a similar approach applied when developing the NFA engine e.g. instead of running direct simulation, conversion of the model is performed firs. The first idea was to convert ε – NFA straight to DFA and run the simulation, but the process of

converting turned out to be too long and complex. After a substantial research a second approach was applied, instead of converting ε – NFA to DFA convert it to NFA and pass it down to already built conversion engine for further transformation into DFA and then run the actual simulation.

ε – NFA is NFA which contains epsilon moves. In order to convert ε – NFA to NFA all epsilon moves have to be removed. Removing epsilon transitions follows five main steps.

- Step 1: Find the epsilon closure for a given state. In other words, find every state which can be reached via epsilon symbol from a certain state.
- Step 2: For each extracted state/s from *Step 1* find which states a reachable at given symbol.
- Step 3: Repeat *Step 1* for every state that is delivered after executing *Step 2* e.g. find its epsilon closure.
- Step 4: After finishing *Step 3* the result is a single or more often group of states which can be reach from a given state at given symbol.
- Step 5: Repeat all the steps described above for every given state.

The development of the conversion engine pivots around these five steps and thus it is divided into five stages:

The same as the development of the NFA conversion engine, auxiliary method to aid and simplify the transformation process are build first.

- *Stage 1:* During the initial stage building method ***getEpsilonTransactions*** takes place. It is a simple method which takes the original list of transitions, turns it into a stream, filters all transitions that contain epsilon as transition symbol and creates a new list which containing only those transitions with epsilon transition symbol.
- *Stage 2:* During second stage method ***subEpsilonTransactionGenerator*** is developed, which finds the epsilon closures (if any) of a given state. It takes two parameters, a state and list of epsilon transitions (previously generated from method ***getEpsilonTransactons***). It creates a list of new transition/s containing the input state as a start state and all the second state/s are states which can be reached from the input state only via epsilon symbol.
- *Stage 3:* In the course of the converting process multiple duplicate transitions are created. In order to remove any remaining duplicates, ***removeDuplicates*** is created as a third auxiliary method. It is a simple method which main aim is to convert a given list of transitions into a stream and remove all duplicate transitions that might be.
- *Stage 4:* The actual ***convertToNFA*** method was generated in this stage. The method combines all three previously developed helper methods and also uses methods inherited from NFA conversion engine (***compositeStateGenerator*** and ***subTransactionGenerator***). It takes three parameters list of states, list of transitions, list of alphabet symbols and returns a converted list of transitions with all epsilon "jumps" removed.

Even thou the returned list from ***convertToNFA*** does not contain any epsilon transitions, the transforming process is still incomplete. Usually when converting ε – NFA to NFA the number of final states in the newly transformed model is increased. In order to fully complete the transformation process another two aider methods were built.

- *Stage 5:* ***assignFinal*** is one of the two additional methods. Its main purpose is to check if each state is able to reach any FINAL state only through epsilon symbol and

if so that state becomes FINAL as well (INITIAL/FINAL if the input state is INITIAL).

- *Stage 6:* The final stage of the development process focuses on building the last auxiliary method ***convertTransactionListStates***, which purpose is to loop through the transformed, by ***assignFinal*** method, list of states and the list of transitions delivered by method ***convertToNFA***. While looping it alters the type of every state, in the transition list, to the type of its corresponding state in the list of states.

For full textual and graphical description of how ε – NFA is converted to NFA method and the actual code please refer to Appendix A.2 and Appendix B.2.

5.2 Sprint 2: User Interface.

The foremost purpose of the second phase of development is to deliver an interaction tool which provides the user with the ability to interact with the rest of the system e.g. the user interface.

The second sprint aims to provide an impressive, simple, self-explanatory and easy to navigate user interface. In order to fulfil these requirements, the second phase is divided into three sub – phases.

Even though the mediator patter is used in this architectural layer, it wasn't implemented at the early stage of development, its implementation took place at the end of the third sprint.

5.2.2 Sub sprint 1: Developing basic interface components.

During this period the most basic building components, used in every standard user interface, are established. These are:

- Main frame – harbours the main panel, bar menu, menus and menu items.
- Main panel – contains all the other panels.
- Tool panel – holds the tool buttons.
- Drawing panel – the drawing canvas which is used by the user to explicitly draw the desirable automaton model.
- Info panel – accommodates all sub panels.
- Sub panels - accountable for informing the user with vital information in relation to the model during building and simulating.

The initial sub sprint is also, responsible to satisfy the requirement *"easy to navigate"* by positioning and sizing all the components in precise order and size, familiar to the user e.g. as any other drawing type application the tool panel can be found at the top of the main frame big enough to hold only the tool buttons, info panel to the utmost right hand side of the frame, drawing panel in the centre and the bar menu at the very top above the toll panel.

5.2.3 Sub sprint 2: Adding extra units.

The second and final phase of shaping the user interface involves generating and adding extra units to the already created frame of containers.

These units are:

- Tool panel buttons.

- Sub panel buttons.
- Bar menu.
- Menus.
- Sub menu.
- Menu items.
- Sub menu items.

It also involves searching and adding, to each action unit, a carefully selected, by shape and colour, icon images which main objectives are to fulfil the task of amplifying each entity with a visual explanatory feature thus meeting one of the main expectations provided by the user interface e.g. *self – explanatory.* In addition, toll tip texts are also added for achieving better clarity.

At the end of the second sprint all the components were forming a tree structure with the main frame taking the role of the root and all other elements formed subtrees or leafs.

5.3 Sprint 3: Application services.

The process of building the application services layer is the most complex and longest amongst all other layers due to the aim of delivering a fully functional and easy to use product.

The lack of knowledge and experience regarding the most important elements (Graphic objects, action listeners and mouse listeners) required for the correct functioning of this layer was the main factor contributing to the complexity and long period of development.

Another important characteristics is that the third sprint does not include only building the service layer but also involves a slight refactoring some of the user interface's components previously developed.

The list of multiple requirements and the increased complexity led to partitioning the entire sprint into multiple sub sprints.

5.3.1 Sub sprint 1: Adding action to sub menu items.

One of the prime features typical for most of the applications which allow the user to create and work on a project is to provide an option to constitute a new document.

This application does not make an exception in this case.

Adding the option of permitting the user to create a new document is the prime point of the initial sub sprint.

It's a simple process involving two steps.

- Step 1: It lets the user to initialize the alphabet.
- Step 2: It saves the alphabet and creates a new *MainPanel* container object, which is added to the main frame.

5.3.2 Sub sprint 2: Graphics

Developing classes with a single responsibility of creating graphical objects (Ovals, lines and polygons) which are essential for representing the graphical parts of any automaton model (states and transitions in the form of arrows) takes place in the second phase of building the Applications services layer.

Due to the fact that all graphical objects are different in shape and functionality but conventional by implementing the same method *draw*, used for rendering each graphical object, which is selected at run time led to implementing the Strategy design pattern.

5.3.3 Sub sprint 3: Adding actions to tool panel buttons.

The main challenge of the third sub sprint was to equip each tool button with the ability to provide a specific service.

"when some of the tool buttons is pressed the user should be able to do a specific operation on the drawing canvas".

The idea behind it, is when a button action is performed a mouse listener should be added to the drawing panel object to carry on the selected action.

The initial attempt to achieve this goal was, whether to implement interfaces *ActionListener* and *MouseListener* by each button class and override their methods or use anonymous classes and implement the executional code inside each class.

Using the two method achieves the desirable result, but it would lead to another two issues:

- First it breaks the first principle of SOLID e.g. a class must have a single responsibility.
- Second, the actual execution is performed inside a class, that belongs to the user interface layer, which by itself breaks one of the main rules of the software architecture implemented in the application e.g. all layers must be independent.

In order to solve the emerged issues, two interfaces and their implementing abstract classes were developed:

- *ButtonAction* interface extending *ActionListener*.
- *MouseAction* interface extending *MouseListener*.
- *AbstractButtonAction* class implementing *ButtonAction.*
- *AbstractMouseAction* class implementing *MouseAction.*

Classes extending *AbstractButtonAction* are only a wrapper classes, their main purpose is to wrap an object of type *MouseAction* and when a certain action is triggered to add this object to the draw panel.

On the other hand, classes extending *AbstractMouseAction* perform the required activity.

After successfully resolving the "*adding action"* problem, another issue emerged.

Even though all user interface components were connected with each other, there was no established connection between them.

5.3.4 Sub sprint 4: Implementing the Mediator design pattern.

The necessity of establishing communication between objects led to transforming the whole top layer.

During this phase the implementation of the Mediator design pattern eventuated.

Three classes were developed in order to introduce communication among all components.

- class *FrameMediatorImpl* responsible to carry communication between the frame and all of its components.
- class *PanelMediatorImpl* liable to support communication between the main panel and all its components.
- class *MainMediatorImpl* connecting the two sub mediator classes, thus forming an interconnected grid across the entire top layer.

After the transformation the tree structure formed previously took a totally different shape. The main mediator becomes the root element and each sub mediator its left and right sub trees.

5.3.5 Sub sprint 5: Drawing graphics.

After developing classes responsible for creating graphical objects, the next condition is to provide the user with option to dynamically draw graphical objects which number, type and location are set solely by him/her.

Developing method which carries the task of creating and drawing Spheres (states) at previously set two dimensional location *x* and *y* was the easy part. The work of *addState2D* method manifests in simply taking the two coordinates (x and y) of the mouse pointer at the time of "*clicking"* and "*releasing"* and create a spherical type of object, with statically set size and colour and assign the already taken coordinates to the object itself, thus delivering the option of adding numerous state object at different location is completed.

The second part of this phase consist in elaboration of a method responsible for drawing arrow between two previously selected states, thereby providing visual representation of a transition/s.

A successful operation must cover the following requirements:

1. allowing the user to the user to choose the two states between which the transition arrow should be drawn by simply *pressing, dragging and releasing* the mouse.
2. not allowing the user to draw a transition from existing state to non-existing and the opposite.
3. automatically calculating the shortest distance between the two spheres and draw a line starting from the first sphere's arc and finishing at the second sphere's arc, thereby establishing a perfectly straight line regardless the user's movements during dragging and the coordinates of the mouse pointer at pressing and releasing (as long as it's inside a spherical object).
4. In case of establishing two opposite transition arrows, one originating from *s1* and terminating to *s2* and the other one from *s2* to *s1*, the two arrows should slightly bend into opposite direction from one another in order to create two arcs, thus avoiding the two object to clash with each other.

A simple method fulfils the first two demands, which simply verifies whether given coordinates are encompassed within existing spherical object.

Sufficing the third stipulation provide a challenge. After a several unsuccessful attempts and detailed research, a solution to the problem was introduced.

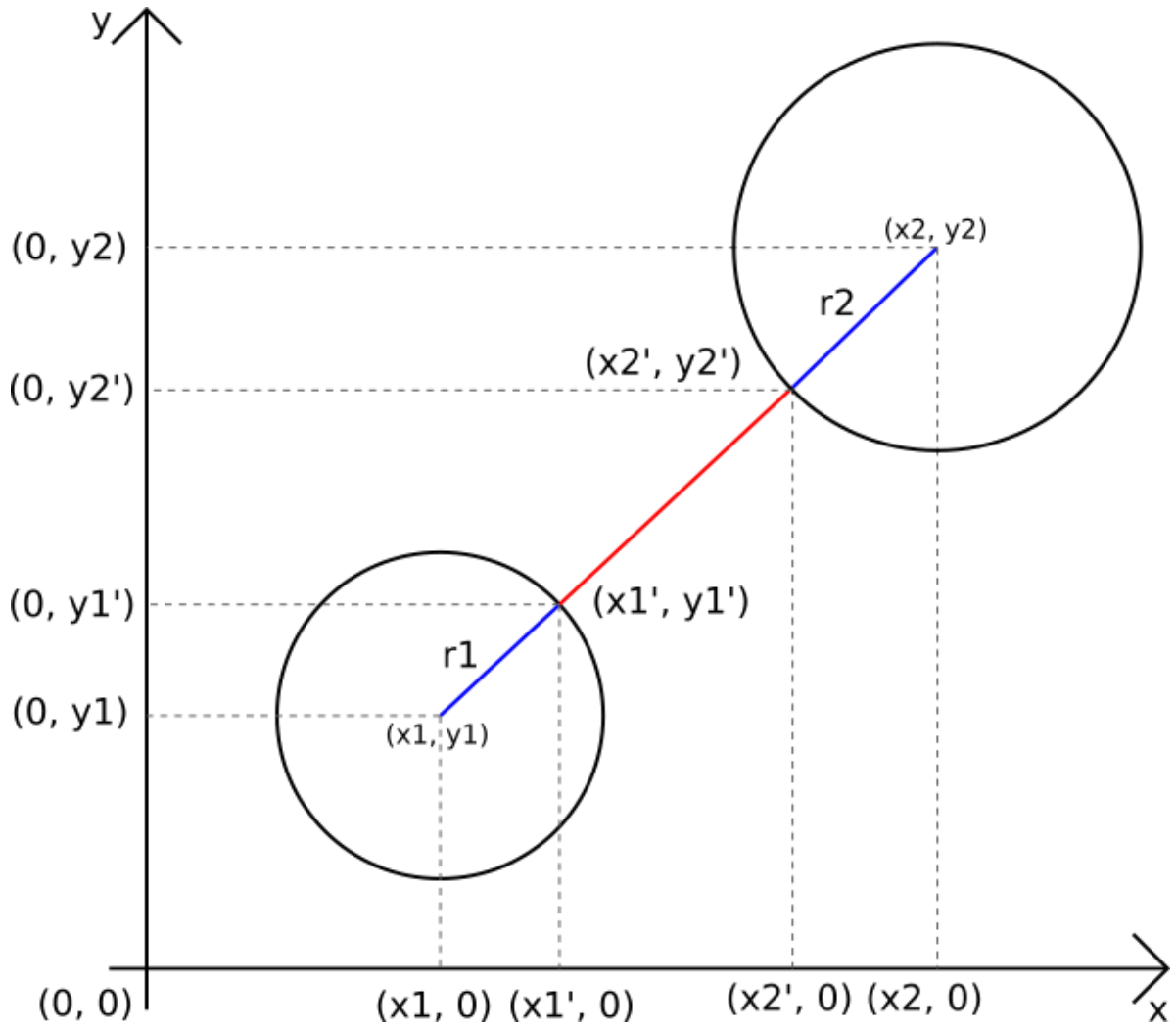

Figure 5.3.5.1 Illustrating the application of the method used to connect two spheres with a straight line.

Step 1: Having the coordinates of the two spherical object's centres (x1, y1) (x2, y2) ΔX and ΔY can be calculated.

$\Delta Y = y2 - y1$
$\Delta X = x2 - x1$

Step 2: Calculate the distance between the two spheres, as starting and ending point is used the centre of each sphere, using Pythagoras theorem.

$L = \sqrt{(\Delta X^2 + \Delta Y^2)}$

Step 3: Using each circle's radios Δy and Δx can be calculated from the centre to the arc of the circle.

$\Delta x = r/L \, \Delta X$
$\Delta y = r/L \, \Delta Y$

$\Delta x1 = r1/L \, \Delta X$

$\Delta y1 = r1/L \Delta Y$

As a result, the coordinates of the starting point of the line is ($x1+\Delta x, y1+\Delta y$) and the ending point is ($x2+\Delta x1, y2+\Delta y1$), thus the desirable outcome is achieved.

5.3.6 Sub sprint 6: Developing *DrawPanelController* adapter and Implementing the MVC design pattern.

The next step of the development of the final layer consists of providing the ability of rendering graphical objects. Java GUI provides such an option in the form of a method *painComponents* which takes an object of type *Graphics* and executes its *draw* method, and it is implemented by every class of type *Container*.

The aim of this stage is to develop a function which satisfies the following demands.

Every graphical object:

- created by the user must be displayed instantly, persist and remain intact.
- removed by the user must disappear instantly and permanently.
- modified by the user, its modification must be rendered instantly and persist.

The process responsible of accomplishing the goals described above consist three steps.

- Creation: using an appropriate action method.
- Storage: using object/s capable of storing and retrieving data.
- Visualization: using the *paintComponents* method implemented by the *DrawPanelImpl* class.

Due to the nature of the three steps the MVC design pattern provides an excellent solution, thus the first adapter (*DrawPanelController*) and data carrier objects (*Automaton2D*) originated.

Three classes participate in forming the MVC pattern:

- Class *Automaton2D* (model) contains three sequential lists. Two are used for storing graphical objects (States and arrows) and one transition objects.
- Class *DrawPanelController* (controller) responsible for executing all updates on the model (add/remove states and arrows).
- Class *DrawPanleImpl* (view) which carries the visualization of all graphical objects stored in the model.

All other adapters further built in the development process follow the same paradigm and implement the MVC pattern.

5.3.7 Sub sprint 7: Implementing the Command design pattern.

Introducing the MVC pattern led to moving the executional code from all *MouseAction* type classes to the controller type classes. This and the pursue of code separation and clarity resulted in implementing another design pattern, the Command pattern.

For the purpose several new classes were developed to improve clarity and code maintenance. Classes responsible for wrapping and forwarding a certain command and classes executing it.

After the implementation, all action components (buttons and menu items) were only responsible for issuing a certain command which would be wrapped and forward by another class to the correct object responsible for the execution. Moreover, objects of type *MouseAction* were striped from being responsible for storing the actual execution code to only executing the object containing the algorithm/s.

5.3.8 Sub Sprint 8: Implementing the Observer design pattern.

In order to justify the true purpose of the Info panel (to provide the user with vital information in connection with the model being built) component, the application should provide a method to dynamically keep all the information accurate and up to date at all times.

This requirement prompted the implementation of the Observer pattern.

5.3.9 Sub Sprint 9: Developing the *AutomatonController* adapter.

The ninth sub sprint involves constructing the only entry point that provides communication between the service layer and the core.

AutomatonController has a several purposes, to gather all information inserted by the user (list of states, list of transitions, alphabet and input string), convert it into a form, acceptable by the core and decide, based on the user's choice, which of the tree simulation engine to be executed.

After a multiple successful test of the three different automaton models a new issue aroused, the application was limited only in simulating errorless created models.

Even though there are previously developed restrictions guiding and aiding the user through the process of building a certain model in the correct way, there were some conditional restriction left unresolved:

What if the user:

- forgets to initialize initial or final state?
- forget to enter input string to be tested before the execution?
- enter input string containing symbols, which are missing in the alphabet?
- has created state, which do not participate in any transition e.g. doesn't have any in or out coming transition arrows?
- tries to execute invalid DFA model (Applies only to DFA type models)?

Even one of these condition passed down to the core would lead to miss functional or even break the simulation process.

The first thought was to add an additional functionality to the core, comprised in handling all harmful exceptions but that would fracture the true purpose of the simulation code.

In order to solve the arisen issue an additional class was created, having the purpose of validating any data that is passed down to the core.

5.3.10 Sub Sprint 10: Developing *class Validator*.

In the lass sub sprint the development of class *Validator* ("*Guardian of the core*") took place.

Its single purpose is to validate all the user input data by the usage of multiple methods each fitted with different functionality influenced by what part of the input data is being validated.

If some of the data is invalid the appropriate exception a thrown.

In parallel with developing class *Validator*, two additional functionalities were added to the core adapter class, exception handling (handling all exceptions thrown by class *Validator*) and informative (when an exception is caught an appropriate pop up warning message is displayed, to inform the user why a certain model cannot be proceeding for simulation).

5.4 Sprint 4: Final refinement.

During the final sprint the last two feature were added in order to complete the application, providing the options of Saving and Opening.

Various additional options (alter alphabet) were also constructed in the final stage of development.

6. Testing

6.1. Testing strategy.

There are two approaches taken in order to verify the behaviour of the system.

- Unit testing – used to test the behaviour of each implemented method. This approach is entirely used during testing the core functionality and all its methods implementation.
- User acceptance test – manually created models based on pre-defined different scenarios.

6.2 Testing tools.

Unit tests were written by using Junit 4 framework.

6.3 Unit testing.

Unit testing was mainly performed upon the three simulation engines. Every method is thoroughly tested at different scenarios.

An additional auxiliary class is developed and also tested in order to simplify the creation of mock objects needed through the testing process. The class contains two static methods one responsible of creating list of states at given input string and the other one for creating list of transitions.

All the testing code is available in Appendix C.

6.3.1 Testing DFA engine.

Due to its simplicity only one testing method is provided to test the validation of the DFA engine.

The method creates two mock Deterministic Automaton models, each with different set of states transitions and alphabets, and performs multiple assert operation at different input. [Appendix C.1]

6.3.2 Testing NFA engine.

In order to complete its task, NFA engine uses four different auxiliary methods, thus the testing procedures are far more complex than the once used in testing DFA engine. Two of them, the most important ones are thoroughly tested.

The testing process follows the execution process, thereby the first units to be tested are the first once called during execution.

The whole process can be described as visually represented pyramid, starts from the bottom of the pyramid and it works its way up to the top.

For the purpose at the beginning of the test three different mock Non deterministic automaton models using two symbolled alphabets (a, b) are created and used in testing each method's behaviour. [Appendix C.2]

- Testing *subTransactionGenerator* method – the most complex mock object is used during the test, containing four states and ten transitions. Each of the states is asserted at each symbol of the alphabet, whether it produces the correct sub transition. [Appendix C.3]
- Testing *compositeStateGenerator* method - already tested method *subTransactionGenerator* is used in this test. The method uses again one of the mock models as it returns a specific value, for any state and alphabet symbol, used to test *compositeStateGenerator*. [Appendix C.4]
- Testing the actual *convertNFAtoDFA* method. The method is tested by using all created mock models, versus predefined DFA mock objects. [Appendix C.5]
- Testing *runSim* method – the most thoroughly rested method amongst all. Each mock model is asserted multiple times at various input strings. [Appendix C.5]

6.3.2 Testing ε- NFA engine.

Testing ε- NFA engine uses absolutely the same approach and paradigm used in validating its predecessor the NFA engine. [Appendix C.6 – C.13]

6.4. User acceptance test.

The last testing phase, responsible for the overall system validation, is the user acceptance test.

Multiple automata models different in structure, size, type, alphabet content and user input string were simulated and each outcome was validated against the same models simulated on a different already approved simulator, JFLAP.

Apart from only simulating variety of different models, also, all other features (exception handling, save, open, alter alphabet) were tested during the user acceptance test.

7. Future recommendations

This section contains future recommendations for improving and enhancing the application.

7.1 Additional functionalities.

Even though the application is equipped with sufficient number of functionalities needed in the process of building a graphical Automaton model, adding more features to improve and simplify its usage are highly recommended.

Additional features enhancing user's experience:

- Adding two more buttons forward and backwords - allowing the user to go back or forward in the working process.
- Improved model's flexibility – providing the user with the option to change the location of already established graphical object by simply grab and drag to a new position.
- Zoom in and out option – an additional ability letting the user to enlarge or abbreviate the view size of the drawing area, thus improving the overall view of the model being build.
- Animated simulation – permitting the user to simulate a certain model in an animated mode e.g. to observe the simulation process at any stage, thus informing the user at what stage the input string was rejected or stuck.
- Graphically representing model conversion – Adding the option of manual converting NFA to DFA and graphically rendering the model on the drawing area.
- Adding new models – providing the user with the option to work and simulate Push down automaton models and Turing machines.

Features enhancing performance:

- Simplifying NFA and ε – NFA conversion engines – despite the fact that the two engines produce the desirable result, most of the operations are repeated in the conversion process. An alternative solution is to use *memoization,* thus memorizing operations that have been already executed and instead of performing the same operations again just use the already memorised result.

8. Summary and conclusions.

This report outlines the most important aspects of the entire projects development

In summary, undertaking a project responsible for developing a fully functional simulating application combining functionality and graphical user interface in one and keeping them separated at the same time was a difficult task to achieve.

The final date of the project's completion was September 2019. Due to personal reasons, lack of knowledge, experience and the aim of delivering a sufficiently completed and fully functional application, caused postponing the project to September 2020.

The extended time immensely contributes in improving, enriching and expending knowledge in variety arias involved in the process of software development and architecture. Areas like software design and architecture, Java GUI Library, Geometry, usage of design patterns and combining to achieve maintainable and easy to follow code.

8.1 Lessons learned

The most important lessons that were learned during the development process of this project are:

- Thorough planning and design in early stage of each project is crucial. The clear idea of what architecture, design patterns, communication and collaboration between components, will be used must be done before any coding takes place, in order to reduce time, unnecessary additional sprints and decision changes during implementing phase.
- Always follow the SOLID principles when using Object - oriented approach.

9. References

[1] Deepak D'Souza, Priti ShankarIndian, Institute of Science, India. Modern application of automata theory.

[2] Dominique Perrin, Universit´e de Marne-la-Vall´ee, "Automata and formal languages" July 15, 2003.

[3] Coffin, R. W., Goheen, H. E. and Stahl, W. R. 1963. Simulation of a Turing machine on a digital computer. Proceedings of the Fall Joint Computer Conference, pp. 35-43.

[4] Chakraborty, P. 2007. A language for easy and efficient modelling of Turing machines. Progress in Natural Science, 17(7): 867-871.

[5] Dominguez, A. E. O. 2009. Automata. http://torturo.com/wp content/uploads/Automata.jar.

[6] Jerry Banks, John S. Carson II, Barry L. Nelson, David M. Nicol. Discrete – Event System Simulation Fifth Edition.

Katya Lebedeva, Australian National University (Semester 2, 2016). Introduction to Theory of Computation.

Louis G. Birta and Gilbert Arbez. Modelling and Simulation Exploring Dynamic System Behaviour.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design patterns: Elements of reusable Object – Oriented Software.

John Zukowski (2005). The Definite Guide to Java Swing Third edition.

Brian Cole, Robert Eckstein, James Elliott, Marc Loy, David Wood (November 2002). O'Reilly, Java™ Swing, 2nd Edition

Jonathan Knudsen (1999). O'Reilly, Java 2D Graphics.

Java™ 2 SDK, Standard Edition, 1.3 Version November 19, 1999. Programmer's Guide to the Java 2D™ API, Enhanced Graphics and Imaging for Java.

https://stackoverflow.com/questions/34931902/how-to-draw-a-line-that-connects-two-circle-outlines-without-trig-functions

https://stackoverflow.com/questions/47369565/connect-two-circles-with-a-line?rq=1

https://math.stackexchange.com/questions/1250756/draw-the-line-segment-joining-the-centers-of-two-circles-where-does-it-meet-the

Appendix A

A.1 Full description of method convertNFAtoDFA method.

Work flow of method ***convertNFAtoDFA*** which takes List of states, list of transition functions and list of characters (alphabet) and returns a DFA list of transitions.

- First, at initiation point the method establishes two variables of type List, one to store the new DFA transitions and another one to store the newly created states throughout the transformation process (The second variable is of type list which stores other list. the purpose of it is that the newly created states might be a composition of states).

- Second adds the Initial state from the original list of states to the newly created one. In order to accomplish conversation method ***convertNFAtoDFA*** uses four auxiliary methods and three loops.

- Outer loop: it loops through the newly created list (which contains only the initial state at this point of time).

    - First inner loop: It goes through each symbol of the given alphabet.
        - Second inner loop: It loop trough the composition of states. For each state invokes method ***subTransactionGenerator*** which filters all transition functions in which the given state is the starting state at a given alphabet symbol and returns a new list of filtered transitions (if there are no transitions found an empty list is returned).

    After generating the filtered list of transition a second method is invoked ***compositeStateGenerator*** which takes the filtered list as an argument and returns a new list of state composition containing only the end states presented in the filtered list of transition (if the argument is empty list, an empty state is returned).

    The newly generated state/composition of states is verified if it already exists and if so it is added to the list of new states.

    After verification, method ***convertSubListOfStatesToState*** is used, which takes a composition of states as an argument and creates a new single state containing. Also checks if there are any states of FINAL or INITIAL/FINAL type and if so the new state is given a value of type FINAL otherwise its value remains of type NONE.

    The final step of the first inner loop is generating and adding to the list of DFA transitions a new DFA transition containing as a start state the state from the outer loop, the newly generated state as an end state and the symbol from the first inner loop as a transition symbol.

- After finishing looping through the List of new states and before returning the newly created DFA list of transitions, method ***addDeadState*** is invoked. The purpose of which is to go through each transition function and check for empty states that might be and if any found, are replaced with a dead state of type NONE.

Figure A.1.1 Illustrates the flow of converting a sample NFA model to DFA. Each colour represents a loop.

Green: outer loop, which loops through the new list of states.

Orange: first inner loop going through each symbol of the given alphabet.

Red: second inner, cycling trough each composition of states.

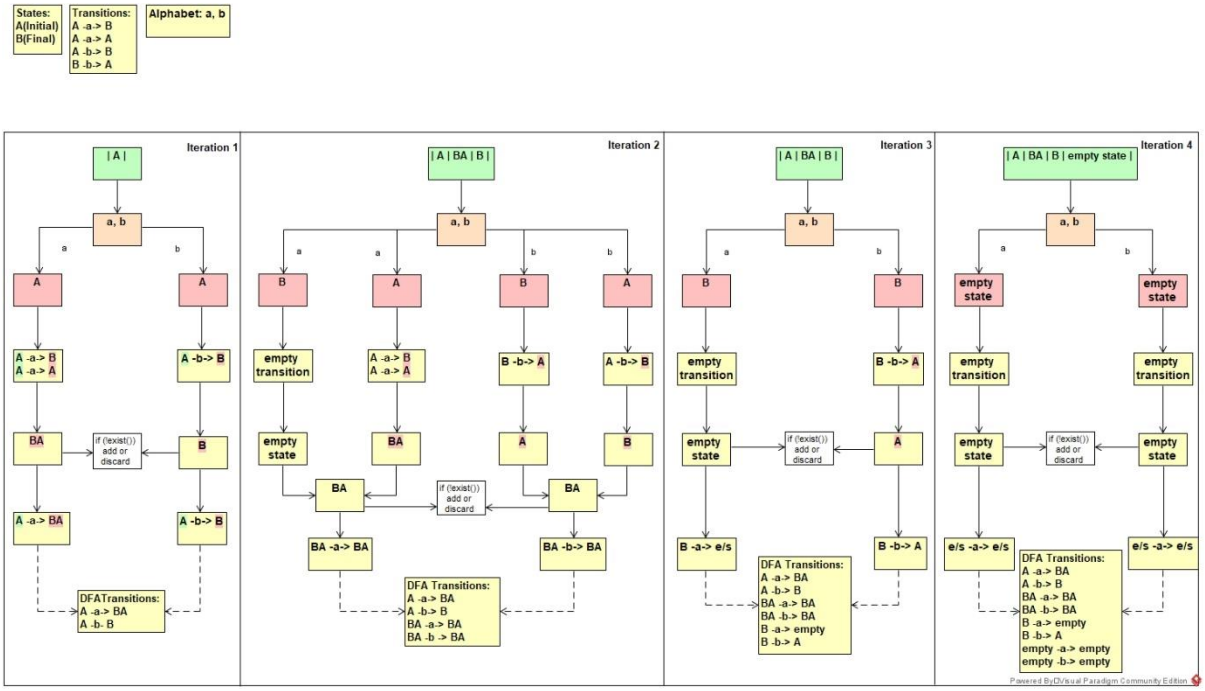

Figure A.1.1 Graphical representation of conversion of sample NFA to DFA.

A.2 Full description of method convertToNFA method.

Work flow of method ***convertToNFA*** which takes a sequential list of states, transition functions and characters (alphabet), and returns a list of NFA transitions. In order to convert ε – NFA to NFA the method uses five loops and five auxiliary methods.

- At the beginning of the execution a variable of type sequential List is created in order to store the newly generated NFA transition functions.
- Auxiliary method ***getEpsilonTransaction*** is used to filter all transition function that contain epsilon symbol out of a given list of transitions.
- Outer loop: it loops through the given sequential list of states.
    - First inner loop: it goes through each symbol in the given alphabet.
      It creates a temporary variable of type sequential list, which is used to store the outcome of the invoked method ***subEpsilonTransactionGenerator,*** that takes the state object from the outer loop and already generated list containing all epsilon transition functions as arguments and returns all epsilon closures (if any) in which the given state is involved.
      After storing all epsilon closures for a given state, another variable of type list is created to store the generated list of states done by invoking method ***compositeStateGenerator*** (inherited from the NFA engine class), which takes

the newly created list of epsilon closures as an input and return a list of every second state of each transition function.

- Second inner loop: it loops through the created list of states. For each state invokes method ***subTransactionGenerator*** (also inherited from the NFA engine class), which receives a symbol from the first inner loop, state from the current loop and the original list of transition functions, as input and produces a new list of transitions in which the given state participates as a first state at given symbol.
  Method ***compositeStateGenerator*** is invoked again but this time uses the newly created list of transition functions to generate list of states, which is used in the third inner loop.
  - Third inner loop: it circles through the list of states created from the second inner loop and repeats the steps of the first inner loop by finding all epsilon closures of a given state by using method ***subEpsilonTransactionGenerator***. The generated list of transitions is used in the final inner loop to produce actual NFA transition functions.
    - ✓ Forth inner loop: the last loop of the transforming process consists of, iterating trough the previously generated list of transitions, in each iteration produces a new transition object containing the current state of the outer loop as a start state, the second state of the current transition function object as end state and the current symbol from the first inner loop as a transition symbol.

After looping through all the states of a given list of states, as a result the newly created list of NFA transitions contains multiple duplicate objects, thus the final step of method ***convertToNFA*** is to remove all duplicates which might be. In order do to so method ***removeDuplicates*** is invoked.

To summarize, in order to generate a converted list of transition functions, method ***convertToNFA*** applies three main steps on each state.

- First step – it finds all epsilon closures (if any) of a particular state.
- Second step – it finds all possible states that can be reached at particular alphabet symbol for every state generated from step one.
- Third step – it repeats step one for every state generated from step two and creates new transition function/s.

Despite the fact that the list of transition function derived from ***convertToNFA*** method is NFA, the conversion process is not fully completed.

There is a rule when converting epsilon NFA to NFA which states that, ever state that can reach another state which is final only via epsilon jumps becomes final itself. This rule is applied in method ***runSim;*** it involves two steps and another two additional auxiliary methods.

- Step 1: List of states with altered type is generated by using method ***assignFinal***, which takes the initial list of states and list containing only epsilon transitions.
- Step 2: Method ***convertTransactionListStates*** is used for the final conversion to NFA. The method itself takes as arguments the previously established list

with altered type states in step one and generated list of transition functions by ***convertToNFA method*** and returns a fully converted list containing NFA transition functions.

Figure A.2.2. Illustrates the full transformation of a sample ε – NFA to NFA. Each colour represents a loop.

Green: outer loop – it loops through each state.

Light orange: first inner loop – iterates each alphabet symbol.

Dark orange: second inner loop – cycles previously created, by the first inner loop, list of states.

Light red: third inner loop – goes through the list of states produced by the second inner loop and repeats the first inner loop functionality e.g. finds all epsilon closures of a given state.

Dark red: forth inner loop – loops through the list of epsilon closures generated from the previous loop, creates and adds new transition functions to the previously established at the beginning of the execution list of NFA transitions.

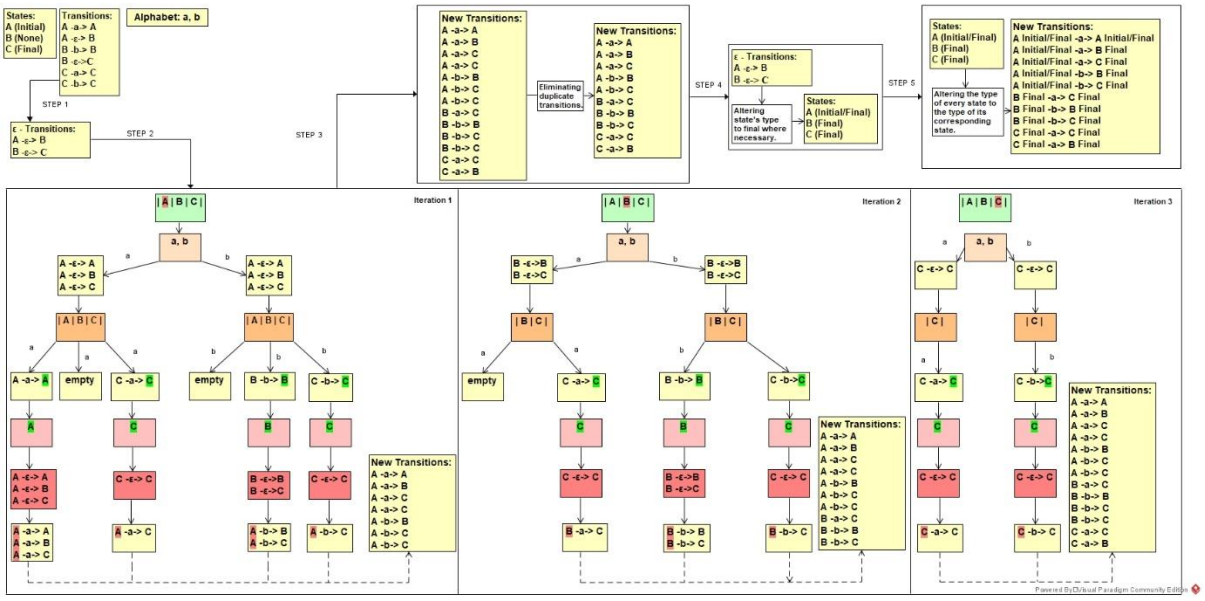

Figure A.2.2 Graphical representation of a complete transformation of a sample ε – NFA to NFA.

Appendix B

B.1 Algorithm used to convert NFA to DFA

```
protected List<Transaction> convertNFAtoDFA(List<State> listOfStates, List<Transaction>
listOfTransactions, List<Character> alphabet){

    List<State> subState = Arrays.asList(getInitial(listOfStates));
    List<List<State>> newListOfStates = new LinkedList<>();
    List<Transaction> dfaListOfTransactions = new LinkedList<>();
    Transaction newTransaction;

    newTransaction =
AbstractFactories.createFactory("TransactionFactory").createObject("Transaction");

    newListOfStates.add(subState);

    for(int list = 0; list < newListOfStates.size(); list++){
        for(int symbol = 0; symbol < alphabet.size(); symbol++){

            List<Transaction> transactionAccumulator = new LinkedList<>();
            List<State> stateAccumulator;

            for(int state = 0; state < newListOfStates.get(list).size(); state++){

                List<Transaction> temp = subTransactionGenerator(alphabet.get(symbol),
newListOfStates.get(list).get(state), listOfTransactions);
                transactionAccumulator = Stream.concat(transactionAccumulator.stream(),
temp.stream())
                                                .collect(Collectors.toList());
            }
           stateAccumulator = compositeStateGenerator(transactionAccumulator);

            if(!exist(stateAccumulator, newListOfStates)){
                newListOfStates.add(stateAccumulator);
            }

            State start = convertSubListOfStatesToState(newListOfStates.get(list));
            State end = convertSubListOfStatesToState(stateAccumulator);
            Transaction transaction = newTransaction.copy();
            char transactionSymbol = alphabet.get(symbol);

            transaction.setFirstState(start);
            transaction.setSecondState(end);
            transaction.setTransactionSymbol(transactionSymbol);

            dfaListOfTransactions.add(transaction);
        }
    }
    return addDeadState(dfaListOfTransactions);
}
```

B.1.1 Auxiliary method **subTransactionGenerator** which returns a list of transactions in which a given state is the start state at given transition symbol.

```
protected List<Transaction> subTransactionGenerator(char symbol, State state,
List<Transaction>listOfTransactions){

        List<Transaction> subListOfTransaction;
        subListOfTransaction = listOfTransactions.stream()
                                                .filter(x -> x.getFirstState().equals(state)
&& x.getTransactionSymbol() == symbol)
                                                .collect(Collectors.toList());
        return subListOfTransaction;
}
```

B.1.2 Auxiliary method **compositeStateGenerator** which extracts every end state from a given list of transition and returns a list made up from all extracted states.

```
protected List<State> compositeStateGenerator(List<Transaction> listOfTransactions){

    List<State> subList = listOfTransactions.stream()
                                .map(x -> x.getSecondState())
                                .sorted(Comparator.comparing(State::getName))
                                .distinct()
                                .collect(Collectors.toList());
    return subList;
}
```

B.1.3 Auxiliary method **addDeadState** which stream through a list of transitions and renames every state **that has empty string as a name.**

```
private List<Transaction> addDeadState(List<Transaction> listOfTransactions){

    return listOfTransactions.stream()
                             .map(transaction ->
{if(transaction.getFirstState().getName().equals("") &&
transaction.getSecondState().getName().equals("")){

transaction.getFirstState().setName("DS");

transaction.getSecondState().setName("DS");
                                                    }
                                                    else
if(transaction.getFirstState().getName().equals("")){

transaction.getFirstState().setName("DS");
                                                    }
                                                    else
if(transaction.getSecondState().getName().equals("")){

transaction.getSecondState().setName("DS");
                                                    }
                                                    return transaction;
                             })
                             .collect(Collectors.toList());

}
```

B.1.4 Auxiliary method **convertSubListOfStatesToState** which takes a list of states as an argument and returns an object of type State as a result of combining all states from the list.

```
private State convertSubListOfStatesToState(List<State> states){
    State converted = null;
    Type type = Type.NONE;
```

```
    //It joins all the states names in the list into a single string.
    String name = states.stream()
            .map(x -> x.getName())
            .collect(Collectors.joining());


    for(State state: states){
        if(states.size() == 1){
            type = state.getType();
        }
        else if(state.getType() == Type.INIFIN || state.getType() == Type.FINAL){
            type = state.getType();
        }
    }
    converted = AbstractFactories.createFactory("StateFactory").createObject("NState");
    converted.setName(name);
    converted.setType(type);

    return converted;
}
```

B.2 Algorithm used to convert ε - NFA to NFA

```
protected List<Transaction> convertToNFA(List<State> listOfStates, List<Transaction>
listOfTransactions, List<Character> alphabet){
    List<Transaction> convertedListOfTransactions = new LinkedList<>();
    List<Transaction> epsilonTransactionsList = getEpsilonTransactions(listOfTransactions);

    Transaction newTransaction =
AbstractFactories.createFactory("TransactionFactory").createObject("Transaction");
    for(int state = 0; state < listOfStates.size(); state++){
        for(int symbol = 0; symbol < alphabet.size(); symbol++){
            List<Transaction> subEpsilonTransactionList =
subEpsilonTransactionGenerator(listOfStates.get(state),epsilonTransactionsList);
            List<State> subEpsilonListOfStates =
compositeStateGenerator(subEpsilonTransactionList);
            for(int subState = 0; subState < subEpsilonListOfStates.size(); subState++){
                List<Transaction> listOfSubTransactions =
subTransactionGenerator(alphabet.get(symbol), subEpsilonListOfStates.get(subState),
listOfTransactions);
                List<State> subListOfStates = compositeStateGenerator(listOfSubTransactions);
                for(int subEpsilonState = 0; subEpsilonState < subListOfStates.size();
subEpsilonState++){
                    List<Transaction> listOfSubTransaction =
subEpsilonTransactionGenerator(subListOfStates.get(subEpsilonState), epsilonTransactionsList);
                    for(int transaction = 0; transaction < listOfSubTransaction.size();
transaction++){
                        Transaction clone = newTransaction.copy();
                        clone.setFirstState(listOfStates.get(state));

clone.setSecondState(listOfSubTransaction.get(transaction).getSecondState());
                        clone.setTransactionSymbol(alphabet.get(symbol));
                        convertedListOfTransactions.add(clone);
                    }
                }
            }
        }
    }

    convertedListOfTransactions = removeDuplicates(convertedListOfTransactions);

    return convertedListOfTransactions;
}
```

B.2.1 Auxiliary method **getEpsilonTransactions** which returns a list containing only transition functions using epsilon as a transition symbol.

```
protected List<Transaction> getEpsilonTransactions(List<Transaction> listOfTransaction){
    List<Transaction> epsilonTransactions;
    epsilonTransactions = listOfTransaction.stream()
                                           .filter(transaction ->
transaction.getTransactionSymbol() == epsilon)
                                           .collect(Collectors.toList());
    return epsilonTransactions;
}
```

B.2.2 Auxiliary method **subEpsilonTransactionGenerator**

```
protected List<Transaction> subEpsilonTransactionGenerator(State state, List<Transaction>
listOfEpsilonTransactions){

    List<Transaction> subEpsilonListOfTransactions = new LinkedList<>();

    Transaction subEpsilonTransaction =
AbstractFactories.createFactory("TransactionFactory").createObject("Transaction");
    subEpsilonTransaction.setFirstState(state);
    subEpsilonTransaction.setSecondState(state);

subEpsilonTransaction.setTransactionSymbol(listOfEpsilonTransactions.get(0).getTransactionSymb
ol());
    subEpsilonListOfTransactions.add(subEpsilonTransaction);

    for(int epsilonTransaction = 0; epsilonTransaction < subEpsilonListOfTransactions.size();
epsilonTransaction++){
        for(int transaction = 0; transaction < listOfEpsilonTransactions.size();
transaction++){

if(subEpsilonListOfTransactions.get(epsilonTransaction).getSecondState().equals(listOfEpsilonT
ransactions.get(transaction).getFirstState())){
                Transaction clonedSubEpsilonTransaction = subEpsilonTransaction.copy();

clonedSubEpsilonTransaction.setSecondState(listOfEpsilonTransactions.get(transaction).getSecon
dState());
                subEpsilonListOfTransactions.add(clonedSubEpsilonTransaction);
            }
        }
    }
   return subEpsilonListOfTransactions;
}
```

B.2.3 Auxiliary method **removeDuplicates**

```
protected List<Transaction> removeDuplicates(List<Transaction> convertedListOfTransactions){
    return convertedListOfTransactions.stream()
                              .distinct()
                              .collect(Collectors.toList());
}
```

B.2.4 Auxiliary method **assignFinal**

```
protected List<State> assignFinal(List<State> listOfStates, List<Transaction>
listOfEpsilonTransactions){
    List<State> alteredListOfStates = cloneListOfStates(listOfStates);

    for(int state = 0; state < alteredListOfStates.size(); state++){
        State start = alteredListOfStates.get(state);
        for(int transaction = 0; transaction < listOfEpsilonTransactions.size();
transaction++){
            if(listOfEpsilonTransactions.get(transaction).getFirstState().equals(start)){
                start = listOfEpsilonTransactions.get(transaction).getSecondState();
            }
```

```
        }
        if(alteredListOfStates.get(state).getType() == Type.INITIAL && (start.getType() ==
Type.FINAL || start.getType() == Type.INIFIN)){
            alteredListOfStates.get(state).setType(Type.INIFIN);
        }
        else if(start.getType() == Type.FINAL || start.getType() == Type.INIFIN){
            alteredListOfStates.get(state).setType(start.getType());
        }
    }
    return alteredListOfStates;
}
```

B.2.5 Auxiliary method **converTransactionListStates**

```
protected List<Transaction> convertTransactionListStates(List<State> convertedListOfStates,
List<Transaction> convertedListOfTransactions){
    List<Transaction> alteredListOfTransactions =
cloneListOfTransactions(convertedListOfTransactions);

    for(int state = 0; state < convertedListOfStates.size(); state++) {
        for(int transaction = 0; transaction < alteredListOfTransactions.size();
transaction++){

if(alteredListOfTransactions.get(transaction).getFirstState().getName().equals(convertedListOf
States.get(state).getName())){

alteredListOfTransactions.get(transaction).getFirstState().setType(convertedListOfStates.get(s
tate).getType());
            }

if(alteredListOfTransactions.get(transaction).getSecondState().getName().equals(convertedListO
fStates.get(state).getName())){

alteredListOfTransactions.get(transaction).getSecondState().setType(convertedListOfStates.get(
state).getType());
            }
        }
    }
    return alteredListOfTransactions;
}
```

B.3 Algorithm used to run the actual simulation.

```
public boolean runSim(List<State> listOfStates, List<Transaction> listOfTransactions,
List<Character> alphabet, List<Character> userInput){
        State toStart = getInitial(listOfStates);
        for(int symbol = 0; symbol < userInput.size(); symbol++){

            State start = toStart;
            char userSymbol = userInput.get(symbol);

            State end = listOfTransactions.stream()
                    .filter(t -> t.getFirstState().equals(start) && t.getTransactionSymbol()
== userSymbol)
                    .map(s -> s.getSecondState())
                    .findFirst().get();
            toStart = end;
        }
        if(toStart.getType() == Type.FINAL || toStart.getType() == Type.INIFIN){
            return true;
        }
        else{
            return false;
        }
    }
```

Appendix C

C.1 DFA testing method testRunSim.

```
@Test
public void testRunSim(){
    List<State> states =
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("S1/INITIAL",
"S2/NONE", "S3/FINAL"));


    List<Transaction> transactions =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("S1/INITIAL/S2/
NONE/a", "S1/INITIAL/S1/INITIAL/b",
            "S2/NONE/S1/INITIAL/a", "S2/NONE/S3/FINAL/b", "S3/FINAL/S3/FINAL/a",
"S3/FINAL/S2/NONE/b"));


    List<Character> input = Arrays.asList('a','b','a','a');


    List<Character> alphabet = Arrays.asList('a','b');


    assertTrue(automaton.runSim(states, transactions, alphabet, input));

    input = Arrays.asList('a','b','a','a','b');
    assertFalse(automaton.runSim(states, transactions, alphabet, input));

    input = Arrays.asList('a','b','a','a','b','a','b','b','a','b');
    assertTrue(automaton.runSim(states, transactions, alphabet, input));

    states =
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("A/INIFIN", "B/NONE",
"C/NONE"));


    alphabet = Arrays.asList('a','b','c');


    transactions =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INIFIN/B/NON
E/a", "B/NONE/A/INIFIN/a", "C/NONE/C/NONE/a",
            "A/INIFIN/A/INIFIN/b", "B/NONE/C/NONE/b", "C/NONE/B/NONE/b", "A/INIFIN/C/NONE/c",
"B/NONE/B/NONE/c", "C/NONE/A/INIFIN/c"));


    input = Arrays.asList('c','c');

    assertTrue(automaton.runSim(states, transactions, alphabet, input));

    input = Arrays.asList('a','b','c');

    assertTrue(automaton.runSim(states, transactions, alphabet, input));

    input = Arrays.asList('b','b','c','a','a','b','c','b','c','b','b');

    assertTrue(automaton.runSim(states, transactions, alphabet, input));

    input = Arrays.asList('b','b','b','b','b');
    assertTrue(automaton.runSim(states, transactions, alphabet, input));

    input = Arrays.asList('b','b','c','a','a','b','c','b','c','b','b','a');
    assertFalse(automaton.runSim(states, transactions, alphabet, input));
}
```

C.2 NFA setUp

```
@Before
public void setUp(){
    automaton = AbstractFactories.createFactory("Automaton").createObject("NFA");

    listOfStates_AB =
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("B/FINAL",
"A/INITIAL"));


    listOfStates_ABC =
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("A/INITIAL",
"B/NONE", "C/FINAL"));


    listOfStates_ABCD =
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("A/INITIAL",
"B/NONE", "C/NONE", "D/FINAL"));

    listOfTransaction_AB =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/FI
NAL/a"));


    listOfTransaction_ABC =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/A/IN
ITIAL/a",
            "A/INITIAL/B/NONE/a", "A/INITIAL/A/INITIAL/b", "B/NONE/C/FINAL/a",
"B/NONE/C/FINAL/b"));


    listOfTransaction_ABCD =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/A/IN
ITIAL/a", "A/INITIAL/B/NONE/a",
            "A/INITIAL/A/INITIAL/b", "B/NONE/C/NONE/a", "B/NONE/C/NONE/b", "C/NONE/D/FINAL/a",
"C/NONE/D/FINAL/b"));

    transactions =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/NO
NE/a", "A/INITIAL/A/INITIAL/b", "A/INITIAL/B/NONE/b",
            "A/INITIAL/D/FINAL/b", "B/NONE/C/NONE/a", "B/NONE/B/NONE/b", "B/NONE/A/INITIAL/b",
"C/NONE/C/NONE/a", "C/NONE/B/NONE/b", "C/NONE/D/FINAL/b"));
}
```

C.3 NFA testSubTransactionGenerator

```
@Test
public void testSubTransactionGenerator(){

expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/NO
NE/a"));
actual = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('a',
listOfStates_ABCD.get(0), transactions);
assertEquals(expected, actual);

expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/A/IN
ITIAL/b", "A/INITIAL/B/NONE/b", "A/INITIAL/D/FINAL/b"));
actual = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('b',
listOfStates_ABCD.get(0), transactions);
assertEquals(expected, actual);

expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("B/NONE/C/NONE/
```

```
a"));
actual = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('a',
listOfStates_ABCD.get(1), transactions);
assertEquals(expected, actual);

expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("B/NONE/B/NONE/
b", "B/NONE/A/INITIAL/b"));
actual = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('b',
listOfStates_ABCD.get(1), transactions);
assertEquals(expected, actual);

expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("C/NONE/C/NONE/
a"));
actual = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('a',
listOfStates_ABCD.get(2), transactions);
assertEquals(expected, actual);

expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("C/NONE/B/NONE/
b", "C/NONE/D/FINAL/b"));
actual = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('b',
listOfStates_ABCD.get(2), transactions);
assertEquals(expected, actual);

}
```

C.4 NFA testCompositeStateGenerator

```
@Test
public void testCompositeStateGenerator(){

State state = AbstractFactories.createFactory("StateFactory").createObject("NState");

State a = state.copy();
a.setName("A");
a.setType(Type.INITIAL);

State b = state.copy();
b.setName("B");
b.setType(Type.NONE);

State c = state.copy();
c.setName("C");
c.setType(Type.NONE);

State d = state.copy();
d.setName("D");
d.setType(Type.FINAL);

List<State> expected = Arrays.asList(b);
List<Transaction> subTransactions = ((NonDeterministicAutomaton)
automaton).subTransactionGenerator('a', listOfStates_ABCD.get(0), transactions);
List<State> actual = ((NonDeterministicAutomaton)
automaton).compositeStateGenerator(subTransactions);
assertEquals(expected, actual);

expected = Arrays.asList(a,b,d);
subTransactions = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('b',
listOfStates_ABCD.get(0), transactions);
actual = ((NonDeterministicAutomaton) automaton).compositeStateGenerator(subTransactions);
assertEquals(expected, actual);

expected = Arrays.asList(c);
subTransactions = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('a',
listOfStates_ABCD.get(1), transactions);
actual = ((NonDeterministicAutomaton) automaton).compositeStateGenerator(subTransactions);
assertEquals(expected, actual);

expected = Arrays.asList(a,b);
subTransactions = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('b',
```

```
listOfStates_ABCD.get(1), transactions);
actual = ((NonDeterministicAutomaton) automaton).compositeStateGenerator(subTransactions);
assertEquals(expected, actual);

expected = Arrays.asList(c);
subTransactions = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('a',
listOfStates_ABCD.get(2), transactions);
actual = ((NonDeterministicAutomaton) automaton).compositeStateGenerator(subTransactions);
assertEquals(expected, actual);

expected = Arrays.asList(b,d);
subTransactions = ((NonDeterministicAutomaton) automaton).subTransactionGenerator('b',
listOfStates_ABCD.get(2), transactions);
actual = ((NonDeterministicAutomaton) automaton).compositeStateGenerator(subTransactions);
assertEquals(expected, actual);

}
```

C.5 NFA testConvertNFAtoDFA

```
@Test
public void testConvertNFAtoDFA(){
    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/AB/N
ONE/a", "A/INITIAL/A/INITIAL/b",
           "AB/NONE/ABC/FINAL/a", "AB/NONE/AC/FINAL/b", "ABC/FINAL/ABC/FINAL/a",
"ABC/FINAL/AC/FINAL/b", "AC/FINAL/AB/NONE/a", "AC/FINAL/A/INITIAL/b"));
    actual = ((NonDeterministicAutomaton)automaton).convertNFAtoDFA(listOfStates_ABC,
listOfTransaction_ABC, Arrays.asList('a','b'));
    assertEquals(expected, actual);

    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/AB/N
ONE/a", "A/INITIAL/A/INITIAL/b", "AB/NONE/ABC/NONE/a",
            "AB/NONE/AC/NONE/b", "ABC/NONE/ABCD/FINAL/a", "ABC/NONE/ACD/FINAL/b",
"AC/NONE/ABD/FINAL/a", "AC/NONE/AD/FINAL/b", "ABCD/FINAL/ABCD/FINAL/a",
"ABCD/FINAL/ACD/FINAL/b",
            "ACD/FINAL/ABD/FINAL/a", "ACD/FINAL/AD/FINAL/b", "ABD/FINAL/ABC/NONE/a",
"ABD/FINAL/AC/NONE/b","AD/FINAL/AB/NONE/a", "AD/FINAL/A/INITIAL/b"));
    actual = ((NonDeterministicAutomaton)automaton).convertNFAtoDFA(listOfStates_ABCD,
listOfTransaction_ABCD, Arrays.asList('a','b'));
    assertEquals(expected, actual);

    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/FI
NAL/a", "A/INITIAL/DS/NONE/b", "B/FINAL/DS/NONE/a",
            "B/FINAL/DS/NONE/b", "DS/NONE/DS/NONE/a", "DS/NONE/DS/NONE/b"));
    actual = ((NonDeterministicAutomaton)automaton).convertNFAtoDFA(listOfStates_AB,
listOfTransaction_AB, Arrays.asList('a','b'));
    assertEquals(expected, actual);

    listOfTransaction_AB =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/A/IN
ITIAL/a", "A/INITIAL/B/FINAL/a",
            "A/INITIAL/B/FINAL/b", "B/FINAL/A/INITIAL/b"));

    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/AB/F
INAL/a", "A/INITIAL/B/FINAL/b", "AB/FINAL/AB/FINAL/a",
            "AB/FINAL/AB/FINAL/b", "B/FINAL/DS/NONE/a", "B/FINAL/A/INITIAL/b",
"DS/NONE/DS/NONE/a", "DS/NONE/DS/NONE/b"));
    actual = ((NonDeterministicAutomaton)automaton).convertNFAtoDFA(listOfStates_AB,
listOfTransaction_AB, Arrays.asList('a','b'));

    assertEquals(expected, actual);
}
```

C.6 NFA testRunSim

```
@Test
public void testRunSim(){
    assertTrue(automaton.runSim(listOfStates_AB, listOfTransaction_AB, Arrays.asList('a','b'),
Arrays.asList('a')));
    assertFalse(automaton.runSim(listOfStates_AB, listOfTransaction_AB,
Arrays.asList('a','b'), Arrays.asList('a','a')));
    assertFalse(automaton.runSim(listOfStates_AB, listOfTransaction_AB,
Arrays.asList('a','b'), Arrays.asList('b')));
    assertFalse(automaton.runSim(listOfStates_AB, listOfTransaction_AB,
Arrays.asList('a','b'), Arrays.asList('a','b')));

    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('a','b'), Arrays.asList('a','a')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('a','b'), Arrays.asList('a','b')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('a','b'), Arrays.asList('a','a','a')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('a','b'), Arrays.asList('a','a','b')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('a','b'), Arrays.asList('b','a','b')));
    assertFalse(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('a','b'), Arrays.asList('b','a')));
    assertFalse(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('a','b'), Arrays.asList('b','a','b','a')));
    assertFalse(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('a','b'), Arrays.asList('b','a','b','b')));
    assertFalse(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('a','b'), Arrays.asList('a','a','b','b')));

    assertTrue(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('a','a','a')));
    assertTrue(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('a','a','b')));
    assertTrue(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('a','b','a')));
    assertTrue(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('a','b','b')));
    assertTrue(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('b','b','a','a','a')));
    assertTrue(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('a','a','a','a','a')));
    assertFalse(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('a','a')));
    assertFalse(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('a','a','a','b','b','b')));
    assertFalse(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('b','b','a')));
    assertFalse(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('b','a','a','b','b','b')));
    assertFalse(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('b','a','a','b','b','b','a')));
    assertFalse(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('a','b'), Arrays.asList('b','a','a')));
}
```

C.7 ε - NFA setUp

```
@Before
public void setUp(){
    automaton = AbstractFactories.createFactory("Automaton").createObject("eNFA");

    listOfStates_ABC =
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("A/INITIAL",
"B/NONE", "C/FINAL"));

    listOfStates_ABCD =
```

```
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("A/INITIAL",
"B/NONE", "C/NONE", "D/FINAL"));

    listOfStates_ABCDE =
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("A/INITIAL",
"B/NONE", "C/NONE", "D/NONE", "E/FINAL"));

    listOfTransaction_ABC =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/A/IN
ITIAL/0", "A/INITIAL/B/NONE/"+epsilon,
            "B/NONE/B/NONE/1", "B/NONE/C/FINAL/"+epsilon, "C/FINAL/C/FINAL/0",
"C/FINAL/C/FINAL/1"));
    listOfTransaction_ABC_2 =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/A/IN
ITIAL/1", "A/INITIAL/B/NONE/"+epsilon,
            "B/NONE/B/NONE/0", "B/NONE/A/INITIAL/1", "B/NONE/C/FINAL/1", "C/FINAL/B/NONE/0"));
    listOfTransaction_ABCD =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/A/IN
ITIAL/0", "A/INITIAL/B/NONE/"+epsilon,
            "B/NONE/C/NONE/0", "B/NONE/D/FINAL/"+epsilon, "C/NONE/B/NONE/1",
"D/FINAL/D/FINAL/0", "D/FINAL/D/FINAL/1"));
    listOfTransaction_ABCD_2 =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/NO
NE/0", "B/NONE/B/NONE/1",
            "B/NONE/C/NONE/"+epsilon, "C/NONE/C/NONE/0", "C/NONE/D/FINAL/1"));

    listOfTransaction_ABCDE =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/NO
NE/a", "B/NONE/B/NONE/a",
            "B/NONE/C/NONE/"+epsilon, "C/NONE/D/NONE/a", "C/NONE/C/NONE/b", "C/NONE/D/NONE/b",
"D/NONE/D/NONE/a", "D/NONE/B/NONE/b", "D/NONE/E/FINAL/"+epsilon));
}
```

C.8 ε - NFA testGetEpsilonTransactions

```
@Test
public void testGetEpsilonTransactions(){
    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/NO
NE/"+epsilon, "B/NONE/D/FINAL/"+epsilon));
    actual =
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABCD);
    assertEquals(expected, actual);
}
```

C.9 ε - NFA testSubEpsilonTransactionGenerator

```
@Test
public void testSubEpsilonTransactionGenerator(){
    State testState = listOfStates_ABCD.get(0);
    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/A/IN
ITIAL/"+epsilon, "A/INITIAL/B/NONE/"+epsilon, "A/INITIAL/D/FINAL/"+epsilon));
    actual =
((EpsilonNonDeterministicAutomaton)automaton).subEpsilonTransactionGenerator(testState,
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABCD));
    assertEquals(expected, actual);

    testState = listOfStates_ABCD.get(1);
    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("B/NONE/B/NONE/
"+epsilon, "B/NONE/D/FINAL/"+epsilon));
    actual =
((EpsilonNonDeterministicAutomaton)automaton).subEpsilonTransactionGenerator(testState,
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABCD));
    assertEquals(expected, actual);

    testState = listOfStates_ABCD.get(3);
```

```
    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("D/FINAL/D/FINA
L/"+epsilon));
    actual =
((EpsilonNonDeterministicAutomaton)automaton).subEpsilonTransactionGenerator(testState,
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABCD));
    assertEquals(expected, actual);
}
```

C.10 ε - NFA testRemoveDuplicates

```
@Test
public void testRemoveDuplicates(){
    List<Transaction> test =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/NO
NE/1","A/INITIAL/B/NONE/1"));
    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/NO
NE/1"));
    actual = ((EpsilonNonDeterministicAutomaton)automaton).removeDuplicates(test);
    assertEquals(expected, actual);
}
```

C.11 ε - NFA testConvertTransactionListStates

```
@Test
public void testConvertTransactionListStates(){
    List<Transaction> listOfConvertedNFA =
((EpsilonNonDeterministicAutomaton)automaton).convertToNFA(listOfStates_ABCD,
listOfTransaction_ABCD, Arrays.asList('0','1'));
    List<Transaction> listOfEpsilonTransactions =
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABCD);
    List<State> listOfConvertedStateTypes =
((EpsilonNonDeterministicAutomaton)automaton).assignFinal(listOfStates_ABCD,listOfEpsilonTrans
actions);

    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INIFIN/A/INI
FIN/0", "A/INIFIN/B/FINAL/0",
            "A/INIFIN/D/FINAL/0", "A/INIFIN/C/NONE/0", "A/INIFIN/D/FINAL/1",
"B/FINAL/C/NONE/0", "B/FINAL/D/FINAL/0", "B/FINAL/D/FINAL/1",
            "C/NONE/B/FINAL/1", "C/NONE/D/FINAL/1", "D/FINAL/D/FINAL/0",
"D/FINAL/D/FINAL/1"));
    actual =
((EpsilonNonDeterministicAutomaton)automaton).convertTransactionListStates(listOfConvertedStat
eTypes, listOfConvertedNFA);
    assertEquals(expected, actual);

    listOfConvertedNFA =
((EpsilonNonDeterministicAutomaton)automaton).convertToNFA(listOfStates_ABCD,
listOfTransaction_ABCD_2, Arrays.asList('0','1'));
    listOfEpsilonTransactions =
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABCD_2)
;
    listOfConvertedStateTypes =
((EpsilonNonDeterministicAutomaton)automaton).assignFinal(listOfStates_ABCD,listOfEpsilonTrans
actions);

    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/NO
NE/0", "A/INITIAL/C/NONE/0",
            "B/NONE/C/NONE/0", "B/NONE/B/NONE/1", "B/NONE/C/NONE/1", "B/NONE/D/FINAL/1",
"C/NONE/C/NONE/0", "C/NONE/D/FINAL/1"));
    actual =
((EpsilonNonDeterministicAutomaton)automaton).convertTransactionListStates(listOfConvertedStat
eTypes, listOfConvertedNFA);
    assertEquals(expected, actual);
```

```
    listOfConvertedNFA =
((EpsilonNonDeterministicAutomaton)automaton).convertToNFA(listOfStates_ABC,
listOfTransaction_ABC, Arrays.asList('0','1'));
    listOfEpsilonTransactions =
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABC);
    listOfConvertedStateTypes =
((EpsilonNonDeterministicAutomaton)automaton).assignFinal(listOfStates_ABC,listOfEpsilonTransa
ctions);
    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INIFIN/A/INI
FIN/0", "A/INIFIN/B/FINAL/0",
            "A/INIFIN/C/FINAL/0", "A/INIFIN/B/FINAL/1", "A/INIFIN/C/FINAL/1",
"B/FINAL/C/FINAL/0", "B/FINAL/B/FINAL/1", "B/FINAL/C/FINAL/1",
            "C/FINAL/C/FINAL/0", "C/FINAL/C/FINAL/1"));
    actual =
((EpsilonNonDeterministicAutomaton)automaton).convertTransactionListStates(listOfConvertedStat
eTypes, listOfConvertedNFA);
    assertEquals(expected, actual);

    listOfConvertedNFA =
((EpsilonNonDeterministicAutomaton)automaton).convertToNFA(listOfStates_ABC,
listOfTransaction_ABC_2, Arrays.asList('0','1'));
    listOfEpsilonTransactions =
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABC_2);
    listOfConvertedStateTypes =
((EpsilonNonDeterministicAutomaton)automaton).assignFinal(listOfStates_ABC,listOfEpsilonTransa
ctions);

    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfTransactions(Arrays.asList("A/INITIAL/B/NO
NE/0", "A/INITIAL/A/INITIAL/1",
            "A/INITIAL/B/NONE/1", "A/INITIAL/C/FINAL/1", "B/NONE/B/NONE/0",
"B/NONE/A/INITIAL/1", "B/NONE/B/NONE/1", "B/NONE/C/FINAL/1",
            "C/FINAL/B/NONE/0"));
    actual =
((EpsilonNonDeterministicAutomaton)automaton).convertTransactionListStates(listOfConvertedStat
eTypes, listOfConvertedNFA);
    assertEquals(expected, actual);
}
```

C.12 ε - NFA testAssignFinal

```
@Test
public void testAssignFinal(){

    List<Transaction> epsilonTransactions =
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABCD);
    List<State> expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("A/INIFIN",
"B/FINAL", "C/NONE", "D/FINAL"));
    List<State> actual =
((EpsilonNonDeterministicAutomaton)automaton).assignFinal(listOfStates_ABCD,epsilonTransaction
s);
    assertEquals(expected, actual);

    epsilonTransactions =
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABC);
    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("A/INIFIN",
"B/FINAL", "C/FINAL"));
    actual =
((EpsilonNonDeterministicAutomaton)automaton).assignFinal(listOfStates_ABC,epsilonTransactions
);
    assertEquals(expected, actual);

    epsilonTransactions =
((EpsilonNonDeterministicAutomaton)automaton).getEpsilonTransactions(listOfTransaction_ABC_2);
    expected =
AuxiliaryTestingClass.convertListOfStringsIntoListOfStates(Arrays.asList("A/INITIAL",
"B/NONE", "C/FINAL"));
    actual =
((EpsilonNonDeterministicAutomaton)automaton).assignFinal(listOfStates_ABC,epsilonTransactions
```

```
);
    assertEquals(expected, actual);
}
```

C.13 ε - NFA testRunSim

```
@Test
public void testRunSim(){
    assertTrue(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD,
Arrays.asList('0','1'), Arrays.asList('0','0','1','0','1','0','0')));

    assertTrue(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD_2,
Arrays.asList('0','1'), Arrays.asList('0','1')));
    assertTrue(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD_2,
Arrays.asList('0','1'), Arrays.asList('0','1','0','1')));
    assertTrue(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD_2,
Arrays.asList('0','1'), Arrays.asList('0','0','0','1')));
    assertFalse(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD_2,
Arrays.asList('0','1'), Arrays.asList('0','1','0','1','0')));
    assertFalse(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD_2,
Arrays.asList('0','1'), Arrays.asList('0','0')));
    assertFalse(automaton.runSim(listOfStates_ABCD, listOfTransaction_ABCD_2,
Arrays.asList('0','1'), Arrays.asList('1','1','0','1','0')));

    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC_2,
Arrays.asList('0','1'), Arrays.asList('1')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC_2,
Arrays.asList('0','1'), Arrays.asList('1','1')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC_2,
Arrays.asList('0','1'), Arrays.asList('1','0','1')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC_2,
Arrays.asList('0','1'), Arrays.asList('0','0','1')));
    assertFalse(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC_2,
Arrays.asList('0','1'), Arrays.asList('1','0')));
    assertFalse(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC_2,
Arrays.asList('0','1'), Arrays.asList('0','0')));
    assertFalse(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC_2,
Arrays.asList('0','1'), Arrays.asList('1','0','0')));
    assertFalse(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC_2,
Arrays.asList('0','1'), Arrays.asList('1','1','1','0')));

    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('0','1'), Arrays.asList('1')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('0','1'), Arrays.asList('0')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('0','1'), Arrays.asList('1','1','0')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('0','1'), Arrays.asList('0','0','1')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('0','1'), Arrays.asList('1','0')));
    assertTrue(automaton.runSim(listOfStates_ABC, listOfTransaction_ABC,
Arrays.asList('0','1'), Arrays.asList('0','1','0')));

    assertTrue(automaton.runSim(listOfStates_ABCDE, listOfTransaction_ABCDE,
Arrays.asList('a','b'), Arrays.asList('a','b','a')));
    assertTrue(automaton.runSim(listOfStates_ABCDE, listOfTransaction_ABCDE,
Arrays.asList('a','b'), Arrays.asList('a','b','a','b','a')));
    assertTrue(automaton.runSim(listOfStates_ABCDE, listOfTransaction_ABCDE,
Arrays.asList('a','b'), Arrays.asList('a','a','a','b','b','b')));
    assertTrue(automaton.runSim(listOfStates_ABCDE, listOfTransaction_ABCDE,
Arrays.asList('a','b'), Arrays.asList('a','b','a','b','b')));
    assertFalse(automaton.runSim(listOfStates_ABCDE, listOfTransaction_ABCDE,
Arrays.asList('a','b'), Arrays.asList('a','b','a','b')));
    assertFalse(automaton.runSim(listOfStates_ABCDE, listOfTransaction_ABCDE,
Arrays.asList('a','b'), Arrays.asList('b')));
}
```