6CCS3PRJ

Implementing algorithms connecting finite automata and regular expressions

Final Project Report

Author: Kacper Dudzinski

Supervisor: Agi Kurucz

Student ID: 1805487

Programme of Study: MSci Computer Science

April 8, 2022

Abstract

Finite automata and regular expressions are important tools that have a wide variety of uses. Every finite automaton has a corresponding regular expression, and every regular expression has a corresponding finite automaton, such that their languages are equivalent. Various algorithms exist for converting between finite automata and regular expressions. Two such algorithms are considered for this project: the State Elimination with GNFAs algorithm for converting finite automata into regular expressions, and the McNaughton-Yamada-Thompson algorithm for converting regular expressions into finite automata. The aim of this project is to create a user-friendly educational tool that implements and visually demonstrates these algorithms. The target audience for this tool is teachers and students, who can use the tool to show and explain the algorithms, and to gain a better understanding of them.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Kacper Dudzinski

April 8, 2022

Acknowledgements

I would like to thank my supervisor, Dr. Agi Kurucz, for her invaluable help and support during this project.

Contents

Chapter 1

Introduction

This chapter will introduce the project by discussing what the project aims to achieve, what the motivations for completing the project are and what the scope of the project is.

1.1 Aim

The aim of this project is to create a piece of software that can be used by teachers (and students) to aid in the teaching (and learning) of the algorithms used to convert between regular expressions and finite automata. In particular, the software must have a visual representation of each step of the algorithms employed, allowing the teacher to explain the steps taken, and the student to follow along.

1.2 Motivation

Various studies have shown that visual aids are a critical component of learning. Not only do they facilitate the process of understanding new and unknown concepts, but they also help memory retention (Bigelow & Poremba, 2014). The brain can process images much faster than it can process text. In addition, information can be presented much more succinctly in an image (the age-old adage "a picture is worth a thousand word" comes to mind). As such, it is important that teachers have access to visual tools to assist in their teaching, and students in their learning.

Regular expressions and finite automata are both important tools that have a wide variety of

uses. Some examples include language recognition (for example to find out what language a text is written in), search (as used in text editors and search engines) and programming (for example to validate input). Hence it is important that students learn these tools.

While some programs already exist that can convert between regular expressions and finite automata (for example JFLAP, which is discussed in Chapter 2.7.1), their number is limited and they leave a lot to be desired when it comes to using the program for educational purposes. This project aims to remedy that by creating a new piece of software that has been specifically designed to be used as an educational tool.

1.3 Scope

The algorithm chosen to convert regular expressions into finite automata (more specifically into NFAs) is the McNaughton-Yamada-Thompson algorithm, also known as Thompson's Construction. This is a very popular algorithm and one that lends itself to visualisation, making it perfect for the purpose of the software. The algorithm chosen to convert finite automata into regular expressions is the State Elimination algorithm. While other popular algorithms exist, many of them are quite mathematical in nature and therefore not suitable for visualisation. The State Elimination algorithm works by repeatedly removing states from a so called GNFA and as such, it can be easily visualised.

Both DFAs and NFAs will be valid input to convert into regular expressions. The output will always be an NFA; converting NFAs into DFAs falls outside the scope of this project, as does NFA and DFA minimisation. Additionally, only finite automata that are acceptors (sometimes also called detectors or recognisers) will be considered. Other types of finite automata, for example Moore or Mealy machines, will not be considered.

Only regular expressions with the three base operators (concatenation, union and Kleene star) will be considered. More complex regular expressions, for example those using the 'one-or-more' or 'zero-or-one' operators, fall outside the scope of this project (note that in the case of these two additional operators, they can both be expressed using just the three base operators).

Chapter 2

Background

This chapter will start off by briefly introducing the idea of formal languages, before moving on to consider finite automata and regular expressions. It will then discuss their equivalence and the algorithms for converting between them, namely the McNaughton-Yamada-Thompson algorithm (also called Thompson's Construction) and the State Elimination algorithm. It concludes by reviewing existing software for converting between finite automata and regular expressions.

2.1 Formal Languages

Before discussing formal languages, it is important to understand the underlying concepts. Formal languages must be defined over an alphabet, which specifies what symbols may be used to create words of that language. The following definition of alphabets and words is taken from Rozenberg and Salomaa (1997).

> An *alphabet* is a finite nonempty set. The elements of an alphabet Σ are called *letters* or *symbols*. A *word* or *string* over an alphabet Σ is a finite sequence consisting of zero or more letters of Σ, whereby the same letter may occur several times. The sequence of zero letters is called the *empty word*, written λ. Thus, λ, 0, 1, 110, 00100 are words over the "binary" alphabet $\Sigma = \{0, 1\}$[1]. The set of all words (resp. of all nonempty words) over an alphabet Σ is denoted by Σ^* (resp. Σ^+). Observe that Σ^* and Σ^+ are always infinite. (Rozenberg & Salomaa, 1997, p. 10)

[1] In fact, λ is a word over any alphabet.

Note that Rozenberg and Salomaa use λ for the empty word, while ϵ is used throughout the rest of this report.

If a is a letter of some alphabet Σ, or a word over some alphabet Σ, then a^n will be used to denote a word consisting of n instances of a concatenated together. For example, $1^3 = 111$ and $(abc)^2 = abcabc$. Note that in the second example, the parenthesis are used to indicate the operation is applied to the whole word and not just to the preceding letter. For any letter or word a, $a^0 = \epsilon$.

Having defined alphabets and words, a language over an alphabet Σ can now be defined as as a subset of Σ^*, i.e. a set of words over the alphabet Σ. For example,

$$L_1 = \{\text{bananas, apples, strawberries, grapes, oranges}\}$$

is a language over the Latin alphabet and

$$L_2 = \{01^n0 \mid n \text{ is even}\}$$

is a language over the binary alphabet.

There are two 'special' languages to be aware of. The empty language $\emptyset$ is a language that doesn't contain any words. Meanwhile, $\{\epsilon\}$ is the empty string language; that is, a language that accepts the empty string and nothing else.

Formal languages are very closely linked to both finite automata and regular expressions. The discussion of formal languages will continue in the next two sections.

2.2 Finite Automata

Finite automata are abstract machines consisting of a finite set of states with directed labelled edges between them. A finite automaton can be used to recognise certain patterns of symbols (these types of finite automata are called acceptors, detectors or recognisers). Given a string of symbols as input, a finite automaton will perform a series of transitions from an initial state. If the automaton finishes in an accepting state, then the string is accepted; otherwise the string is rejected. The set of strings that are accepted by the automaton is called the language of the automaton. If two finite automata accept the same language, they are equivalent.

Finite automata can be further split into two types: **deterministic finite automata (DFA)** and **nondeterministic finite automata (NFA)**. These are defined as follows (Aho, Lam, Sethi, & Ullman, 2007, p. 147):

- Deterministic finite automata (DFA) have, for each state and for each symbol of its input alphabet, exactly one edge with that symbol leaving that state.
- Nondeterministic finite automata (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state and ϵ, the empty string, is a possible label.

A transition labelled with the empty string, from here onwards referred to as an ϵ-transition, can be taken by the finite automaton regardless of the current input symbol. Using an ϵ-transition does not consume the current input symbol.

A deterministic finite automaton is deterministic because the next state of the automaton can be determined from the current state and the current input symbol. As a result, there is always only one possible sequence of transitions. On the other hand, in a nondeterministic finite automaton there may be several possible sequences of transitions. Whenever the finite automaton can choose between several possible transitions, it will attempt all of them in parallel. Some of the possible transitions may end in an accepting state and some may not. For an NFA, the input string is accepted if at least one of the possible transitions terminates in an accepting state.

Furthermore, NFAs are not guaranteed to terminate. In an NFA, there is no requirement that there has to be one edge leaving the state for every possible symbol and every state. As a result, there is now the possibility that there are no edges out of a state for a particular symbol. The automaton cannot transition anywhere and so it becomes stuck. The sequence of transitions that lead to this state is discarded and is ignored when it comes to determining whether the input string is accepted or rejected.

2.2.1 Deterministic Finite Automata

A formal definition of deterministic finite automaton can be given as follows (Sipser, 2013, p. 35): A deterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,

2. Σ is a finite set called the alphabet,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the transition function,
4. $q_0 \in Q$ the start or initial state,
5. $F \subseteq Q$ is the set of accepting or final states.

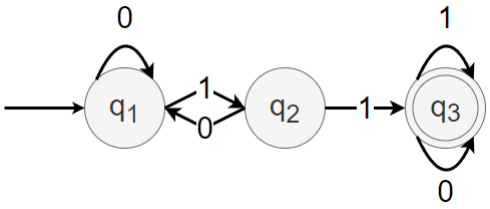


Figure 2.1: Example of a deterministic finite automaton M_1.

Consider Figure 2.1, showing an example of a deterministic finite automaton M_1. M_1 can be formally defined by writing $M_1 = (Q, \Sigma, \delta, q_0, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0, 1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_3	q_3

4. q_1 is the start or initial state,
5. $F = \{q_3\}$.

The language that M_1 recognises, $L(M_1)$ is defined as follows:

$$L(M_1) = \{x \mid x \text{ contains } 11\}$$

That is, M_1 recognises strings that contain the sequence '11' somewhere in them. What symbols appear before or after does not matter; there could be any number of any symbols before and after the '11' and the string would still be accepted.

2.2.2 Nondeterministic Finite Automata

A formal definition of nondeterministic finite automaton can be given as follows (Sipser, 2013, p. 53): A nondeterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite set called the alphabet,
3. $\delta : Q \times \Sigma_\epsilon \longrightarrow P(Q)$ is the transition function,
4. $q_0 \in Q$ is the start or initial state,
5. $F \subseteq Q$ is the set of accepting or final states.

Here Σ_ϵ is $\Sigma \cup \{\epsilon\}$ while $P(Q)$ is a collection of all possible subsets of Q, called the power set of Q.

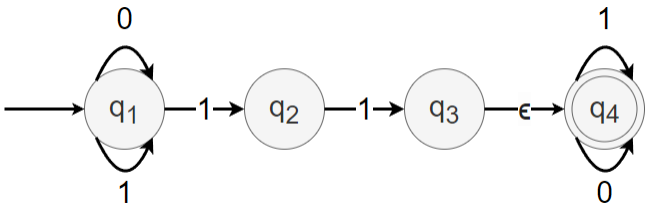


Figure 2.2: Example of a nondeterministic finite automaton M_2.

Consider Figure 2.2, showing an example of a nondeterministic finite automaton M_2. M_2 can be formally defined by writing $M_2 = (Q, \Sigma, \delta, q_0, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0, 1\}$,
3. δ is described as

	0	1	ϵ
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\emptyset$	$\{q_3\}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\{q_4\}$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

4. q_1 is the start or initial state,
5. $F = \{q_4\}$.

The language that M_2 recognises, $L(M_2)$ is defined as follows:

$$L(M_2) = \{x \mid x \text{ contains } 11\}$$

That is, M_2 recognises strings that contain the sequence '11' somewhere in them. What symbols appear before or after does not matter; there could be any number of any symbols before and after the '11' and the string would still be accepted. Note that this is the same as the language of M_1. M_1 and M_2 are equivalent finite automata. The expressive power of NFAs is equal to DFAs. For every NFA, there is a corresponding, equivalent DFA.

2.3 Regular Expressions

Regular expressions are a formal notation used to describe a certain pattern of symbols. This notation can be used to generate different sequences of symbols (i.e. words) that match the described pattern. The set of words that can be generated from a regular expression forms the language of that regular expression. Languages that can be represented using regular expressions are called regular languages.

There are three different operations that can be performed on languages (Sipser, 2013, p. 44):

- **Union:** $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
- **Star:** $A^* = \{x_1 x_2 ... x_k \mid k \geq 0 \text{ and } x_i \in A\}$

A formal, inductive definition of regular expressions can be given as follows (Sipser, 2013, pp. 64–65): Say that R is a regular expression if R is:

1. a for some a in the alphabet Σ,
2. ϵ,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions,
6. (R_1^*), where R_1 is a regular expression.

The regular expression a represents the language $\{a\}$ and the regular expression ϵ represents the language $\{\epsilon\}$, i.e. the language containing a single string - namely, the empty string. The

regular expression $\emptyset$ represents the language that doesn't contain any strings, i.e. an empty language. Finally, for items 4, 5 and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

Parenthesis in an expression may be omitted: in this case, evaluation is done in the order star, then concatenation, then union. The symbol for concatenation, $\circ$, is also often omitted under the assumption that if there is no operator between two regular expressions then they are concatenated, for example $R_1R_2 = R_1 \circ R_2$.

The notation $L(R)$ is used to describe the language of a regular expression R.

Finally, if R is a regular expression, the following identities hold (Sipser, 2013, pp. 65–66):

- $R \cup \emptyset = R$.

  Adding the empty language to any other language will not change it.

- $R \circ \epsilon = \epsilon \circ R = R$.

  Joining the empty string to any string will not change it.

- If $R = \emptyset^*$, then $R = \epsilon$.

  The star operator takes 0 or more strings from the language and concatenates them to get a string as the result. If the language is empty, the star operator can put together 0 strings, giving only the empty string. Hence the regular expression $\emptyset^*$ is equal to the regular expression ϵ.

- $R \circ \emptyset = \emptyset \circ R = \emptyset$.

  Concatenating the empty set with any other set results in the empty set.

2.4 Kleene's Theorem

Kleene's theorem states that a language is regular if and only if it can be accepted by some finite automaton. In other words, regular expressions and finite automata are two sides of the same coin; any regular expression can be turned into an equivalent finite automaton and vice versa (equivalent meaning that the language of the regular expression and the language of the finite automaton are the same). Various algorithms exist for converting between finite automata and regular expression. The following sections describe two such algorithms.

2.5 McNaughton-Yamada-Thompson Algorithm

The **McNaughton-Yamada-Thompson** algorithm (Aho et al., 2007, pp. 159–161; Sipser, 2013, p. 66), also called **Thompson's Construction**, is an algorithm for converting regular expressions into an equivalent NFA. The algorithm takes as input a regular expression r over some alphabet Σ and returns as output an NFA N accepting precisely $L(r)$ - nothing more, nothing less. The algorithm works recursively, splitting the regular expression into sub-expressions, constructing finite automata for them, and then joining the finite automata together.

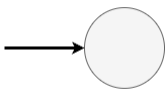
2.5.1 Theory

The finite automata created by the McNaughton-Yamada-Thompson algorithm have several important properties. Suppose that $N(r)$ is the NFA for the regular expression r.

- $N(r)$ accepts the language $L(r)$. That is, $L(N(r)) = L(r)$.
- $N(r)$ has exactly one initial state[2] and one accepting state. The initial state has no incoming transitions and the accepting state has no outgoing transitions. This property is true for the base cases and remains true for the inductive cases, hence is true for all $N(r)$ by induction.
- Each state of $N(r)$, other than the accepting state, has either one outgoing transition on a symbol in Σ or up to two outgoing transitions, both on ϵ. This property is true for the base cases. In the inductive cases, only ϵ-transitions are added, up to two per state, hence the property remains true for the inductive cases. As a result, the property is true for all $N(r)$ by induction.
- $N(r)$ has at most twice as many states as there are operators and operands in r. This bound follows from the fact that each step of the algorithm creates at most two new states.

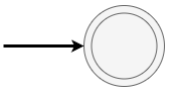
Base cases:

1. $r = \emptyset$. Then $L(r) = \emptyset$ and the following NFA recognises $L(r)$.

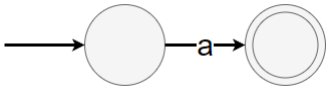


[2]This property holds true for all DFA/NFA.

2. $r = \epsilon$. Then $L(r) = \{\epsilon\}$ and the following NFA recognises $L(r)$.



3. $r = a$ for some $a \in \Sigma$. Then $L(r) = \{a\}$ and the following NFA recognises $L(r)$.



Inductive cases:

1. Suppose $N(s)$ and $N(t)$ are NFAs for regular expressions s and t. Now suppose that $r = s \cup t$. Then $N(r)$ is constructed as shown in Figure 2.3.

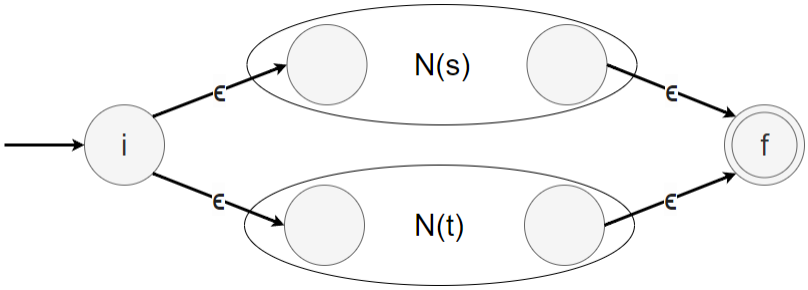


Figure 2.3: NFA for the union of two regular expressions.

Two new states, i and f, are created. The state i is the initial state of $N(r)$ and it is connected to the previous initial states - those of $N(s)$ and $N(t)$ - by ϵ-transitions. Meanwhile, f is the final or accepting state of $N(r)$. The previous accepting states - those of $N(s)$ and $N(t)$ - are connected to f by ϵ-transitions. Note that as a result, the previous accepting states of $N(s)$ and $N(t)$ are no longer accepting states in $N(r)$, which has one and only one accepting state: f.

2. Suppose $N(s)$ and $N(t)$ are NFAs for regular expressions s and t. Now suppose that $r = s \circ t$. Then $N(r)$ is constructed as shown in Figure 2.4.

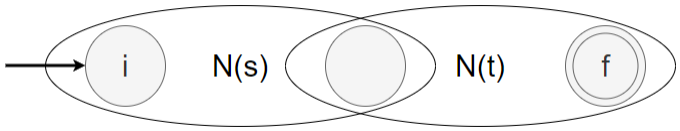


Figure 2.4: NFA for the concatenation of two regular expressions.

The initial state of $N(r)$, labelled in Figure 2.4 as i, is simply the initial state of $N(s)$. Meanwhile, the accepting state of $N(r)$, labelled in Figure 2.4 as f, is simply the accepting state of $N(t)$. Furthermore, the accepting state of $N(s)$ and the initial state of $N(t)$ are merged together into a single state, containing the same transitions as the previous two states combined.

3. Suppose $N(s)$ is an NFA for the regular expressions s. Now suppose that $r = s^*$. Then $N(r)$ is constructed as shown in Figure 2.5.

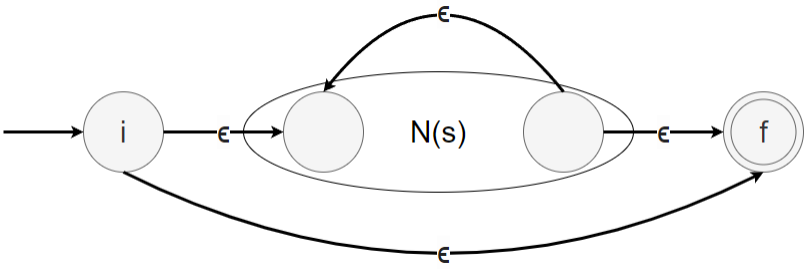


Figure 2.5: NFA for the closure of a regular expression.

Two new states, i and f, are created. The state i is the initial state of $N(r)$ while f is the accepting state of $N(r)$. In addition, four ϵ-transitions are added: one from i to f, one from i to the previous initial state of $N(s)$, one from the previous accepting state of $N(s)$ to the previous initial state of $N(s)$ and one from the previous accepting state of $N(s)$ to f.

As mentioned at the start, the McNaughton-Yamada-Thompson algorithm is recursive: it breaks the given regular expression into sub-expressions until it reaches the base cases. This breakdown can be represented by a **parse tree**. The details of how parse trees are created are discussed in Chapter 4.2 and Chapter 4.3.

2.5.2 Example

Consider the regular expression

$$r = (a \cup b)(aba)^*$$

and its corresponding parse tree, shown in Figure 2.6.

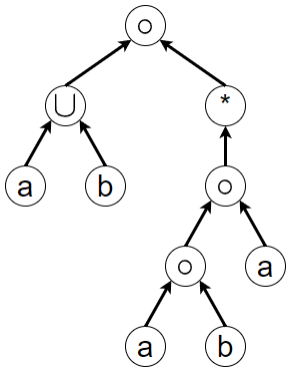


Figure 2.6: Parse tree for the regular expression $r = (a \cup b)(aba)^*$.

The algorithm will perform a post-order traversal of the tree, building the NFA as it does so. Starting with the leaf nodes on the left subtree, the following automata are created by applying the base rules.

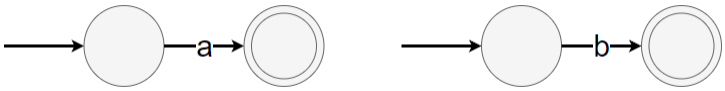


Figure 2.7: NFA for $r = a$ and $r = b$.

Next, the two NFA are combined by applying the inductive rule for union.

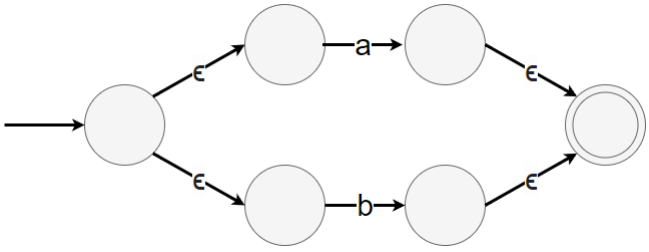


Figure 2.8: NFA for $r = a \cup b$.

Moving onto the right subtree, the automata for $r = a$ and $r = b$ are created just like in Figure 2.7. These are then concatenated by applying the inductive rule for concatenation.

Figure 2.9: NFA for $r = ab$.

Another NFA for $r = a$ is created and then joined onto the NFA from Figure 2.9.

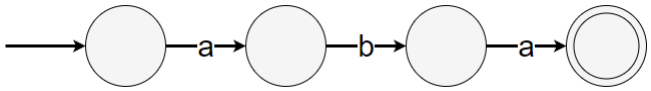


Figure 2.10: NFA for $r = aba$.

The NFA for $r = (aba)^*$ is created by applying the inductive rule for the star operator to the NFA in Figure 2.10.

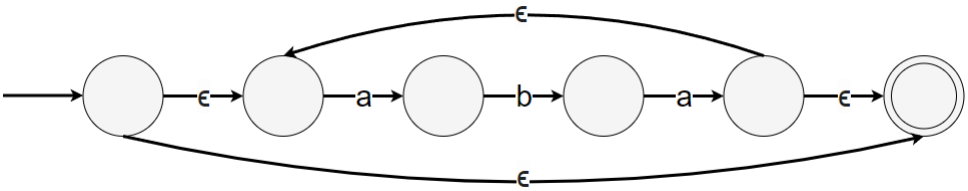


Figure 2.11: NFA for $r = (aba)^*$.

Finally, the NFA from Figure 2.8 and the NFA from Figure 2.11 are concatenated.

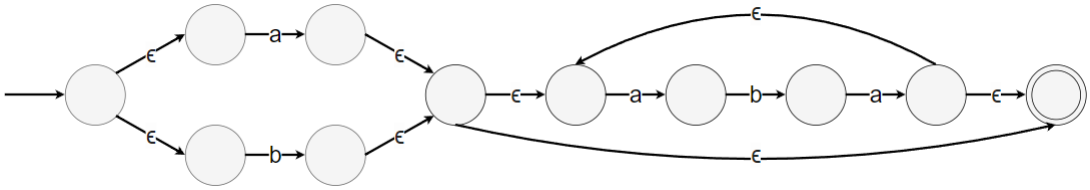


Figure 2.12: NFA for $r = (a \cup b)(aba)^*$.

The resulting NFA, shown in Figure 2.12, is the output of the algorithm; it is a finite automaton whose language is the same as the language of the regular expression $r = (a \cup b)(aba)^*$. Hence, the algorithm has transformed the regular expression into an equivalent finite automaton.

2.6 State Elimination using GNFAs

A finite automaton can be converted into a regular expression through the use of GNFAs and the State Elimination algorithm. The automaton is first transformed into a GNFA. States are then eliminated until only two remain: the initial state and the accepting state. The label on the edge between these two states gives the regular expression corresponding to the original finite automaton.

2.6.1 Theory

A **GNFA** or **generalised nondeterministic finite automaton** (Sipser, 2013, p. 70) is a nondeterministic finite automaton where the labels on the edges may be regular expressions instead of just members of the alphabet or ϵ. In addition, a GNFA reads input differently: instead of reading one symbol at a time, a GNFA can read whole blocks of symbols. Furthermore, the following conditions are imposed on a GNFA:

- The initial state has a transition to every other state, but there are no transitions to the initial state.
- There is only one accepting state. Every state has a transition to the accepting state, but there are no transitions from the accepting state.
- The initial and accepting states cannot be the same.
- Except for the initial and accepting states, all other states have one transition to every other state, as well as a transition to themselves.

A formal definition of a GNFA can be given as follows (Sipser, 2013, pp. 72–73): A generalised nondeterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$, where

1. Q is a finite set of states,
2. Σ is a finite set called the alphabet,
3. $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \longrightarrow R$ is the transition function,
4. $q_{start} \in Q$ is the start or initial state,
5. $q_{accept} \in Q$ is the final or accepting state.

Note the different form of the transition function. Here R is the collection of all regular expressions over the alphabet Σ. Unlike the transition function for DFAs and NFAs, which

took in a state and a symbol as input, and returned a state or a set of states as output, the transition function for GNFAs takes in a pair of states as input, and returns the regular expression found on the edge between the two states as output.

Converting a finite automaton into a GNFA is simple (Sipser, 2013, p. 71). Two new states are created: a new initial state and a new accepting state. The new initial state is connected to the previous initial state with an ϵ-transition, and the previous accepting states are connected to the new accepting state with ϵ-transitions as well. Next, if any two states have more than one transition between them in the same direction, these transitions are replaced by a single transition whose label is the union of the labels of the previous transitions. Finally, for any two states with no transitions between them (including a state to itself), a new transition is added with $\emptyset$ as the label. This step won't change the language of the automaton because a transition with the $\emptyset$ label can never be used. The finite automaton is now a GNFA and satisfies all the previously described conditions.

Suppose the GNFA has k states. Since the GNFA must have a unique initial and accepting state, $k \geq 2$. If $k > 2$, the State Elimination algorithm is used to construct an equivalent GNFA that has $k - 1$ states. This procedure is repeated until $k = 2$, at which point the GNFA will have only two states with a single transition between them. The label of this transition is the regular expression corresponding to the original finite automaton.

The first step in the State Elimination algorithm is to select which state to remove. This can be done arbitrarily, as long as the selected state is not the initial state or the accepting state. From here henceforth, the state selected to be removed will be called q_{rem}. After removing q_{rem} from the GNFA, the transitions have to be updated to reflect the absence of q_{rem} and to ensure that the language of the GNFA is not affected. The transitions that will need to be updated are those between the neighbours of q_{rem}. Consider the set of states $Q_s \subset Q$ that have transitions into q_{rem}, and the set of states $Q_d \subset Q$ that have transitions from q_{rem}. The transitions that will need to be updated are those between the elements of Q_s and Q_d. Note that q_{rem} itself will not be considered an element of these sets.

When updating a transition between two states, $q_i \in Q_s$ and $q_j \in Q_d$, there are two possible paths that need to be considered: the direct path from q_i to q_j or the indirect path that goes through q_{rem}. The first path is simple: it is just the transition from q_i to q_j. The label for this path is simply the label for that transition. The second path is slightly more complex. First,

the automata transitions from q_i to q_{rem}. It may then stay in q_{rem} by taking the transition from q_{rem} to q_{rem} an arbitrary amount of times. Finally, the finite automaton transitions from q_{rem} to q_j, completing the path. The label for this second path is therefore the concatenation of the label for the first transition from q_i to q_{rem}, the star of the label of the second transition from q_{rem} to q_{rem}, and the label of the third transition from q_{rem} to q_j. The new updated label between q_i and q_j is the union of the labels of the two paths between q_i and q_j.

A single round of the state elimination process is illustrated below. Figure 2.13 shows the GNFA before state elimination.

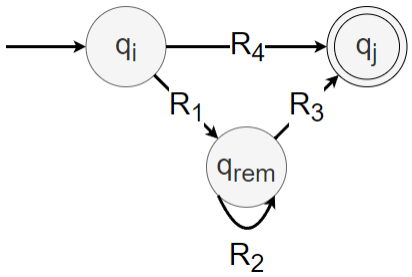


Figure 2.13: The GNFA before state elimination.

Here, $Q_s = \{q_i\}$ and $Q_d = \{q_j\}$. There is only one transition that needs to be updated: the transition from q_i to q_j. The label for the direct path from q_i to q_j is simply R_4. The label for the indirect path through q_{rem} is $R_1R_2^*R_3$. Therefore the updated label is the union of the two, namely $R_4 \cup R_1R_2^*R_3$. The new GNFA is shown in Figure 2.14.

Figure 2.14: The GNFA after state elimination.

2.6.2 Example

An example of the whole process of converting a finite automaton into a GNFA and then applying the State Elimination algorithm will now be given. To keep the diagrams simple, transitions with the $\emptyset$ label will not be included. Consider the finite automaton shown in Figure 2.15.

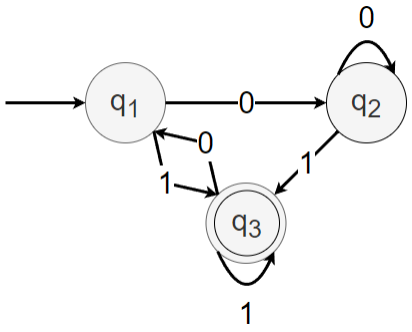


Figure 2.15: The initial finite automaton.

The finite automaton is first converted into a GNFA by introducing two new states: a new initial state q_{init} and a new accepting state q_{acpt}. In this example, there is no need to change any of the transitions as the conditions imposed on GNFA are already satisfied. The created GNFA is shown in Figure 2.16.

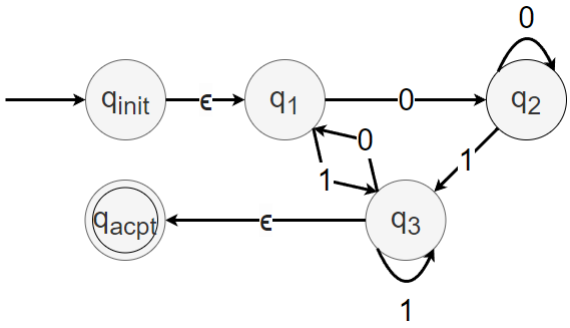


Figure 2.16: The GNFA created from the initial finite automaton.

Since the number of states is greater than 2, a state is selected and removed. In this example, the state q_2 is chosen, but q_1 and q_3 are also valid choices. After removing the state, the transitions need to be updated. Only one state - q_1 - has a transition to q_2 and only one state - q_3 - has a transition from q_2. Hence there is only one transition that needs to be updated: the one between q_1 and q_3.

There are two paths between q_1 and q_3 that need to be considered. The first path is to go directly from q_1 to q_3. This can be done using a single transition with the label 1. The indirect path is to go from q_1 to q_3 via q_2. This path is made up of three transitions: q_1 to q_2, q_2 to q_2 and q_2 to q_3. The corresponding labels are 0, 0 and 1. The middle transition can be taken an arbitrary number of times, so the star operator is applied to its label. The labels are then concatenated together, producing the regular expression 00^*1. The updated label of the

transition between q_1 and q_3 is the union of the labels of the two paths: $1 \cup 00^*1$. The GNFA produced is shown in Figure 2.17.

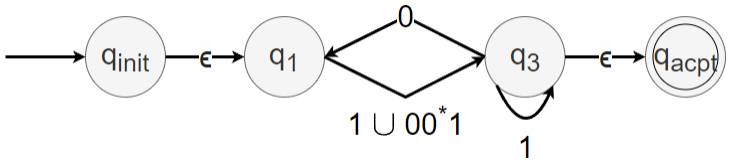


Figure 2.17: The GNFA after the first round of state elimination.

The number of states has been decreased but is still greater than two, so the state elimination process is repeated. This time, the state q_1 is chosen to be eliminated. There are two transitions into q_1: one from q_{init} and one from q_3. There is only one transition from q_1: the one going to q_3. Therefore there are two transitions that need to be updated: the transition from q_{init} to q_3 and the transition from q_3 to q_3.

Consider the transition from q_{init} to q_3. The direct path is simply the transition between these states. This transition is not shown in the diagram because the label of the transition is $\emptyset$. The indirect path is to go from q_{init} to q_1, from q_1 to q_1 and then from q_1 to q_3. The corresponding labels are ϵ, $\emptyset$ and $1 \cup 00^*1$. These labels are concatenated, with the star operator being applied to the middle label, resulting in the regular expression $\epsilon\emptyset^*(1 \cup 00^*1)$. However, using the identities for regular expressions defined in Chapter 2.3, this is reduced to $1 \cup 00^*1$. The label for the updated transition from q_{init} to q_3 is therefore $\emptyset \cup (1 \cup 00^*1)$, which reduces to $1 \cup 00^*1$.

Now consider the transition from q_3 to q_3. The direct path is simply the transition between these states, with the label 1. The indirect path is to go from q_3 to q_1, from q_1 to q_1 and from q_1 to q_3. The corresponding labels are 0, $\emptyset$ and $1 \cup 00^*1$. These labels are concatenated, with the star operator being applied to the middle label, resulting in the regular expression $0\emptyset^*(1 \cup 00^*1)$. However, using the identities for regular expressions defined in Chapter 2.3, this is reduced to $0(1 \cup 00^*1)$. The label for the updated transition from q_3 to q_3 is therefore $1 \cup 0(1 \cup 00^*1)$.

The GNFA produced from the second round of state elimination is shown in Figure 2.18.

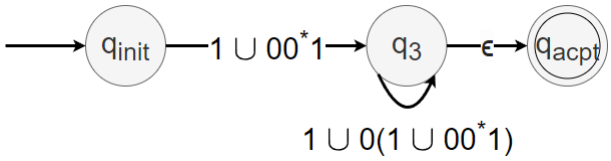


Figure 2.18: The GNFA after the second round of state elimination.

The number of states is still greater than two, so the state elimination process is repeated. This time, the state selected for elimination is q_3; there are no other states that could be selected since the the initial state and the accepting state cannot be eliminated. There is only one transition into q_3: the one from q_{init}. There is also only one transition from q_3: the one into q_{acpt}. Therefore the only transition that needs to be updated is the one from q_{init} to q_{acpt}.

The direct path from q_{init} to q_{acpt} is simply the transition between these two states, which has the label $\emptyset$. The indirect path is to take the transition from q_{init} to q_3, from q_3 to q_3 and then from q_3 to q_{acpt}. The corresponding labels are $1 \cup 00^*1$, $1 \cup 0(1 \cup 00^*1)$ and ϵ. The star operator is applied to the middle label and the labels are concatenated to produce the regular expression $(1 \cup 00^*1)(1 \cup 0(1 \cup 00^*1))^*\epsilon$. This can be reduced to $(1 \cup 00^*1)(1 \cup 0(1 \cup 00^*1))^*$. Hence, the updated transition from q_{init} to q_{acpt} is the union of the labels of the two paths: $\emptyset \cup (1 \cup 00^*1)(1 \cup 0(1 \cup 00^*1))^*$. This is reduced to $(1 \cup 00^*1)(1 \cup 0(1 \cup 00^*1))^*$.

The GNFA produced from the third round of state elimination is shown in Figure 2.19.

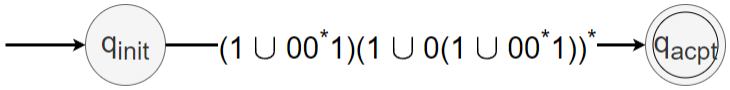
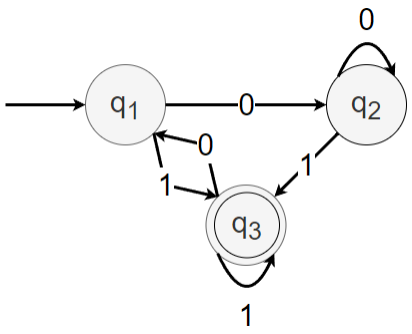


Figure 2.19: The GNFA after the third round of state elimination.

The number of states is now equal to two and hence the State Elimination algorithm terminates. The output is the regular expression on the label between the two states. The language of this regular expression is equivalent to the language of the original finite automaton that was provided as input. Hence, for the finite automaton shown in Figure 2.15 (and included once again below), the equivalent regular expression is

$$(1 \cup 00^*1)(1 \cup 0(1 \cup 00^*1))^*$$



2.7 Review of existing software

This section aims to review existing software (including web-based software) for converting between finite automata and regular expressions. As it turns out, very few programs offer this functionality. Out of all the programs found (considering only those that were functioning, as several appeared to be broken or bugged), only one was capable of converting between finite automata and regular expressions in both directions - JFLAP. Several programs were found for turning regular expressions into finite automata (but not the other way around) and not a single program was found for turning finite automata into regular expressions (but not the other way around). Apart from JFLAP and a program called Seshat, none of the programs found showed the steps involved in the conversion process and so are not suitable as an educational tool. It is clear that there is a distinct lack of software for this purpose. In this section, JFLAP and Seshat will both be reviewed.

2.7.1 JFLAP

> "JFLAP is software for experimenting with formal languages topics including non-deterministic finite automata, nondeterministic pushdown automata, multi-tape Turing machines, several types of grammars, parsing, and L-systems. In addition to constructing and testing examples for these, JFLAP allows one to experiment with construction proofs from one form to another, such as converting an NFA to a DFA to a minimal state DFA to a regular expression or regular grammar" (`https://www.jflap.org/`).

JFLAP is the only program found that was capable of converting between finite automata and regular expressions in both directions. JFLAP was created to be a learning aid, allowing users to experiment with its various features. It is very versatile and offers a wide variety of functions.

One such function is to convert regular expressions into finite automata. First, the user is asked to enter a regular expression. JFLAP will then create a GNFA with two states, with the regular expression as the label of the edge between them. What follows next is an interactive process in which the user can repeatedly break down the regular expressions on transitions into sub-expressions. This results in a GNFA with additional states. The user must then connect these new states together in the correct order by using ϵ-transitions (JFLAP actually uses λ instead of ϵ for the empty string, but to avoid confusion ϵ will continue to be used in this section). JFLAP can aid in this process by keeping track of the number of transitions that need to be broken down or added, by informing the user if a transition is valid and in the correct order, or by filling in the transitions automatically. It is even possible to skip the whole process and go straight to the end result. Eventually, all the regular expressions will be the base cases described in Chapter 2.3 and the process is finished.

However, the method of constructing GNFAs for sub-expressions that JFLAP uses results in a larger than necessary number of ϵ-transitions and states. This makes the diagrams unnecessarily large and more complicated. For example, compare the finite automaton shown in Figure 2.12 from Chapter 2.5.2, (included once again below) to the finite automaton created by JFLAP, shown in Figure 2.20. Both automata were created from the regular expression $(a \cup b)(aba)^*$, but JFLAP's finite automaton has an extra 7 states, an increase of 64%! This difference in the number of states and edges will only get larger as the complexity of the regular expression increases.

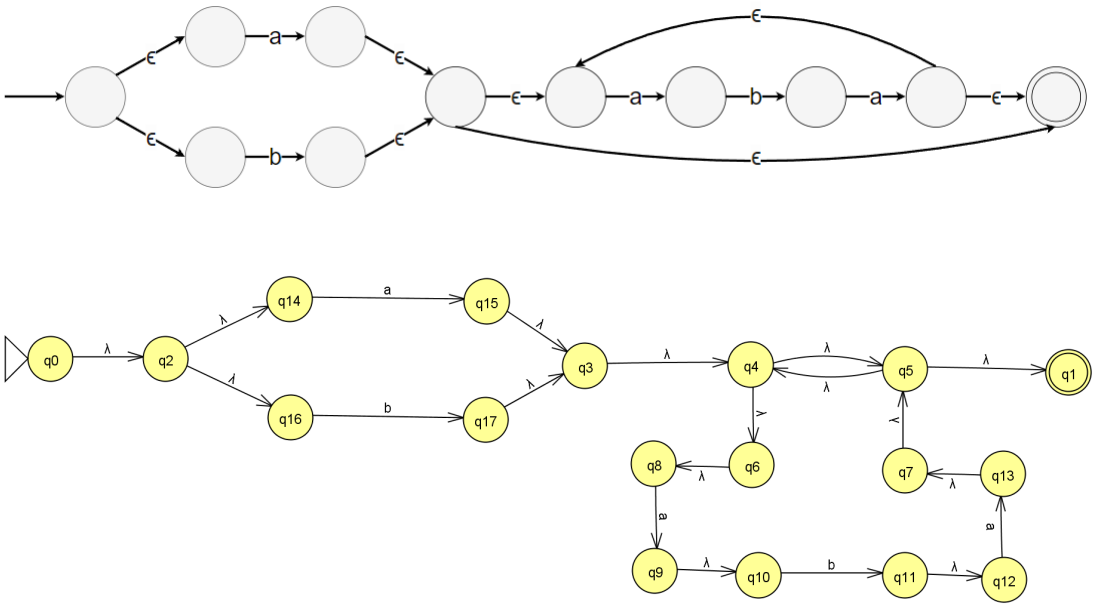


Figure 2.20: JFLAP's finite automaton for $r = (a \cup b)(aba)^*$.

The process is also missing an undo button. While this is not strictly necessary, it would be helpful to go back and see what a transition looked like before it was split into sub-expressions, or to re-visit a particular stage of the process. Unfortunately, the only way to do this in JFLAP is to start the process from scratch.

Moving on, JFLAP also has a function for converting finite automata into regular expressions. First, the user creates the desired finite automaton. This is done through a graphical editor, where the user can create states and then add transitions between them. Once the finite automaton has been created, the user can start the conversion process. This too is an interactive process. The first step is to change the finite automaton into a GNFA, so that the conditions imposed on GNFAs from Chapter 2.6.1 are satisfied (with the exception of the condition that the initial state cannot have any incoming transitions and the accepting state cannot have any outgoing transitions). This may involve adding and connecting a new accepting state, combining multiple transitions between states that have more than one transition and adding extra $\emptyset$-transitions between states that don't have any transitions. Once that is done, the user can begin the state elimination process by selecting a state to eliminate. This opens a new window showing all the transitions that will be present in the finite automaton once the state is removed. The user can click on a transition to see which transitions in the 'old' finite automaton are used to create the new transition. Once the transitions have been confirmed, the state is removed and the transitions are updated by JFLAP. This process repeats until there are only two states. By considering the transitions present in the final GNFA, JFLAP comes up with a regular expression corresponding to the initial finite automaton.

However, JFLAP does not explain how it works out the regular expression from the final GNFA. While this last step is not particularly hard, it would still be nice to have this shown.

Another problem arises from the requirement to have transitions between every pair of states. This can result in a very large number of $\emptyset$-transitions, which cover the GNFA and make it hard to read. An option to hide $\emptyset$-transitions would be a welcome addition.

Finally, this process is also missing an undo button, which would be useful for comparing the GNFA before and after a state has been eliminated.

Two helpful features that JFLAP has are the ability to auto-zoom, making sure the finite automaton not only fits on the screen at all times but is also as large as possible, and the ability to layout the finite automaton, which moves the states and edges around in an attempt

to place them so that they are not on top or too close to each other. JFLAP also allows the user to save and load inputs and outputs, which can be very handy.

In general, the UI of JFLAP is quite plain and uninspired. While it does get the job done, it could be improved to make it more intuitive to use and more user-friendly. A greater focus could also be placed on explaining what is happening in each step of the processes involved.

2.7.2 Seshat

"Seshat Tool contains some of the most common algorithms of lexical analysis taught in language processing courses. It is mainly designed for helping to understand the algorithms by executing them iteratively, step by step." (`http://cgosorio.es/Seshat/`) (Arnaiz-González, Díez-Pastor, Ramos-Pérez, & García-Osorio, 2018).

Seshat is capable of converting regular expressions into both DFAs and NFAs. It is the only tool - other than JFLAP - which shows the steps involved in the process. Like JFLAP, Seshat was created to be a learning aid. However, unlike JFLAP, Seshat cannot convert finite automata into regular expressions. On the other hand, Seshat is a web-based tool which means it is much more accessible. In addition, the UI of Seshat is more user-friendly and intuitive to use compared to JFLAP, although there is still room for improvement.

To convert a regular expression into an NFA, Seshat uses the McNaughton-Yamada-Thompson algorithm. As such, it does not face the same problems as JFLAP. At each step of the algorithm, Seshat shows which part of the regular expression is currently being worked on, the finite automaton for that part of the regular expression and all the previously created finite automata. It also shows which rule is being applied at each step. While it doesn't allow the user to rename the states or to move them about, the fixed-structure results in a more organised finite automaton being created. Seshat also allows the user to move back and forth in the conversion process, a feature missing from JFLAP.

Seshat can also convert a regular expression into a DFA. While this is possible by first using the McNaughton-Yamada-Thompson algorithm to convert the regular expression into an NFA and then using another algorithm (such as Subset Construction) to convert the NFA to a DFA, Seshat is capable of creating the DFA directly. However, the algorithm for this will not be discussed in this project and this function of Seshat will not be discussed further.

Chapter 3

Specification & Design

3.1 Specification

This chapter will discuss what the purpose of the software is and who the intended users are. It will then demonstrate some use cases. The chapter will conclude with a list of requirements that the software must satisfy. Requirements will be split into functional (what the software should do) and non-functional (how the software should do it), and grouped according to their MoSCoW prioritisation.

3.1.1 Purpose & Users

The purpose of this project is to create a piece of software that can be used to aid in the teaching and learning of the algorithms used to convert between regular expressions and finite automata. It does this by allowing the user to create an input (a finite automaton or a regular expression) and then displaying each step of the process of converting the input into an output (a regular expression or a finite automaton). This allows the user to see the steps that are involved in the process.

The software has two main intended users:

- **Teachers** will be able to use the software to assist in teaching the algorithms required to convert between finite automata and regular expressions. They can use the software to demonstrate examples and talk through each step of the algorithms employed.

- **Students** will be able to use the software to get a better understanding of the algorithms used to convert between finite automata and regular expressions. They can use the software to experiment and try out different examples, seeing how the algorithms are applied to specific cases. They could also use the software as a means of checking their understanding and making sure that they themselves are able to carry out each step correctly.

3.1.2 Use Cases

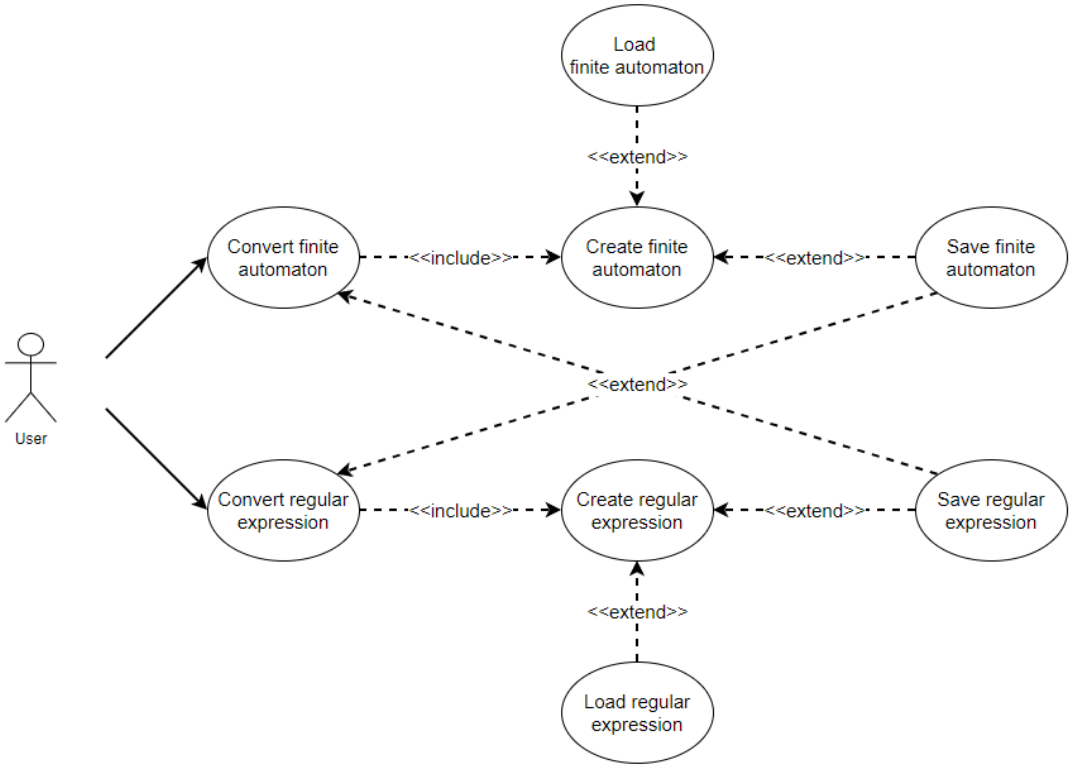


Figure 3.1: Use Case Diagram.

Use Case Name:	Create Finite Automaton
Summary:	A user creates a finite automaton by creating several states and adding transitions between them.
Standard Flow:	**1.** The use case starts when a user indicates that they want to convert a finite automaton into a regular expression. **2.** The software will open up the finite automaton editor. **3.** The user creates several states and adds transitions between them. The user may also save the finite automaton. **4.** The use case ends when the user indicates that they have finished.
Alternative Flow:	**Step 3:** The user may choose to load a previously created finite automaton. They can then continue with step 3 or move on to step 4. **Step 4:** If the finite automaton is not valid, the use case goes back to step 3.
Preconditions:	None
Postconditions:	The user can now proceed and convert the created finite automaton into a regular expression.

Table 3.1: Create Finite Automaton Use Case

Use Case Name:	Convert Finite Automaton
Summary:	A user converts a previously created finite automaton into a regular expression.
Standard Flow:	**1.** The use case starts after a user has created a valid finite automaton. **2.** The software will present the first step in the process of converting the finite automaton into a regular expression. **3.** The user can move to the next step at will. **4.** The use case ends when the finite automaton has been converted into a regular expression. The regular expression may be saved.
Alternative Flow:	**Step 3:** The user might choose to move to the previous step instead.
Preconditions:	The user must have previously created a valid finite automaton.
Postconditions:	The finite automaton has been converted into a regular expression.

Table 3.2: Convert Finite Automaton Use Case

Use Case Name:	Create Regular Expression
Summary:	A user creates a regular expression.
Standard Flow:	**1.** The use case starts when a user indicates that they want to convert a regular expression into a finite automaton. **2.** The software will open up the regular expression editor. **3.** The user types in a regular expression. **4.** The user indicates that they have created a regular expression. **5.** The software will display the parse tree for the regular expression. The user may also save the regular expression. **6.** The use case ends when the user indicates that they have finished.
Alternative Flow:	**Step 3:** The user may choose to load a previously created regular expression. They can then continue with step 3 or move on to step 4. **Step 5:** If the regular expression is not valid, the use case goes back to step 3. The user may also choose to return to step 3 voluntarily.
Preconditions:	None
Postconditions:	The user can now proceed and convert the created regular expression into a finite automaton.

Table 3.3: Create Regular Expression Use Case

Use Case Name:	Convert Regular Expression
Summary:	A user converts a previously created regular expression into a finite automaton.
Standard Flow:	**1.** The use case starts after a user has created a valid regular expression. **2.** The software will present the first step in the process of converting the regular expression into a finite automaton. **3.** The user can move to the next step at will. **4.** The use case ends when the regular expression has been converted into a finite automaton. The finite automaton may be saved.
Alternative Flow:	**Step 3:** The user might choose to move to the previous step instead.
Preconditions:	The user must have previously created a valid regular expression.
Postconditions:	The regular expression has been converted into a finite automaton.

Table 3.4: Convert Regular Expression Use Case

3.1.3 Functional Requirements

Must Have:

- Users must be able to create finite automata, including creating new states and adding labelled transitions between them.
- Users must be able to transform a created finite automaton into an equivalent regular expression.
- Users must be able to see each step of the process of transforming a finite automaton into an equivalent regular expression.
- Users must be able to create regular expressions with the union, concatenation and Kleene star operators.
- Users must be able to transform a created regular expression into an equivalent finite automaton.
- Users must be able to see each step of the process of transforming a regular expression into a finite automaton.

Should Have:

- Users should be able to save and load previously created finite automata.
- Users should be able to save and load previously created regular expressions.
- Users should be able to save and load finite automata created by the software.
- Users should be able to save and load regular expressions created by the software.
- Users should be able to select which state to remove during the state elimination process.
- Users should be able to toggle whether $\emptyset$-transitions are shown in GNFAs.
- Users should be able to move back and forth in the process of transforming finite automata into regular expressions.
- Users should be able to move back and forth in the process of transforming regular expressions into finite automata.
- Users should be able to zoom in and out when viewing finite automata.
- Users should be able to see the parse tree for the regular expressions they create.
- Users should be able to zoom in and out when viewing the parse tree.

Could Have:

- Users could be able to create regular expressions with the 'one-or-more' operator.

- Users could be able to create regular expressions with the 'zero-or-one' operator.
- Users could be able to rename states in the finite automata.
- Users could be able to see a brief explanation of each step of the process of transforming finite automata into regular expressions.
- Users could be able to see a brief explanation of each step of the process of transforming regular expressions into finite automata.

Won't Have:

- Users will not be able to convert NFAs into DFAs.
- Users will not be able to minimise DFAs.
- Users will not be able to minimise NFAs.
- Users will not be able to simulate DFAs.
- Users will not be able to simulate NFAs.

3.1.4 Non-Functional Requirements

Must Have:

- The software must be able to run on Microsoft Windows.
- The software must always return a correct (equivalent) regular expression for a valid given finite automaton.
- The software must always return a correct (equivalent) finite automaton for a valid given regular expression.
- The software must respond to user input within three seconds.

Should Have:

- The software should be able to run on Linux.
- The software should be able to run on macOS.
- The software should be intuitive and easy to use.
- The software should respond to user input within two seconds.

3.2 Design

This section will discuss design of the software. Where the specification looked at *what* will be implemented, the design looks at *how* it will be implemented. The focus will be on the GUI,

with the intention to make it as simple and intuitive as possible, to facilitate ease of use. Many of the design features have been inspired by JFLAP (`https://www.jflap.org/`), but aim to improve and extend what JFLAP offers.

3.2.1 Menu Bar

The menu bar is found at the top of the program window. It contains four different menus:

1. **Convert** - This menu contains options for converting finite automata and regular expressions.
    - **Convert finite automaton** - This option is the start point for the process of converting a finite automaton into a regular expression. It will swap the window to the finite automaton editor, where the user can create a finite automaton.
    - **Convert regular expression** - This option is the start point for the process of converting a regular expression into a finite automaton. It will swap the window to the regular expression editor, where the user can create a regular expression.
2. **Save** - This menu contains options for saving finite automata and regular expressions (depending on what is currently on the screen).
    - **Save finite automaton** - This option will allow the user to save the finite automaton currently on the screen to a location on their device.
    - **Save regular expression** - This option will allow the user to save the regular expression currently on the screen to a location on their device.
3. **Load** - This menu contains options for loading previously created finite automata and regular expressions (depending on what editor is currently opened, if any).
    - **Load finite automaton** - This option will allow the user to load a previously created finite automaton.
    - **Load regular expression** - This option will allow the user to load a previously created regular expression.
4. **Settings** - This menu contains various options.
    - **Show $\emptyset$-transitions** - This option will toggle whether $\emptyset$-transitions are shown or not.

- **Settings** - This option will open the settings window where the user can make various changes.

3.2.2 Finite Automaton Editor

The finite automaton editor is the window used to create finite automata.

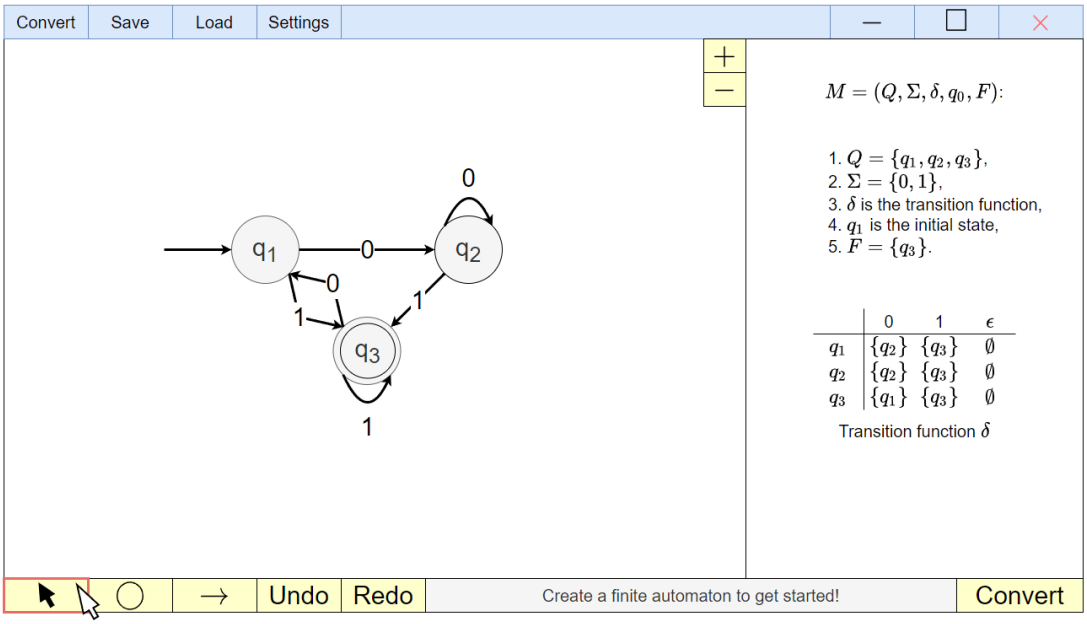


Figure 3.2: Finite Automaton Editor.

The finite automaton editor window consists of a workspace, a details tab and a toolbar. The workspace area is where the user creates the finite automaton by placing states and connecting them with transitions. It contains buttons for zooming in and out. The details tab is automatically updated as the finite automaton is created. It shows the formal definition of the finite automaton (see Chapter 2.2). Finally, the toolbar at the bottom of the screen contains the tools needed to create finite automata:

1. The first button enables cursor mode. In cursor mode, the user can move the states of the automaton around by clicking and dragging. This mode also allows the user to select initial and final states, and rename states and transitions.
2. The second button enables the user to create new states by clicking anywhere in the workspace area.
3. The third button allows the user to add transitions between two states.
4. 5. The fourth and fifth buttons are undo and redo buttons for any unfortunate mishaps.

6. To the right of those is a small text area that displays instructions and error messages.
7. The last button in the toolbar is a convert button that will begin the process of converting the created finite automaton into a regular expression.

3.2.3 Finite Automaton Conversion

This window is displayed when converting a finite automaton into a regular expression.

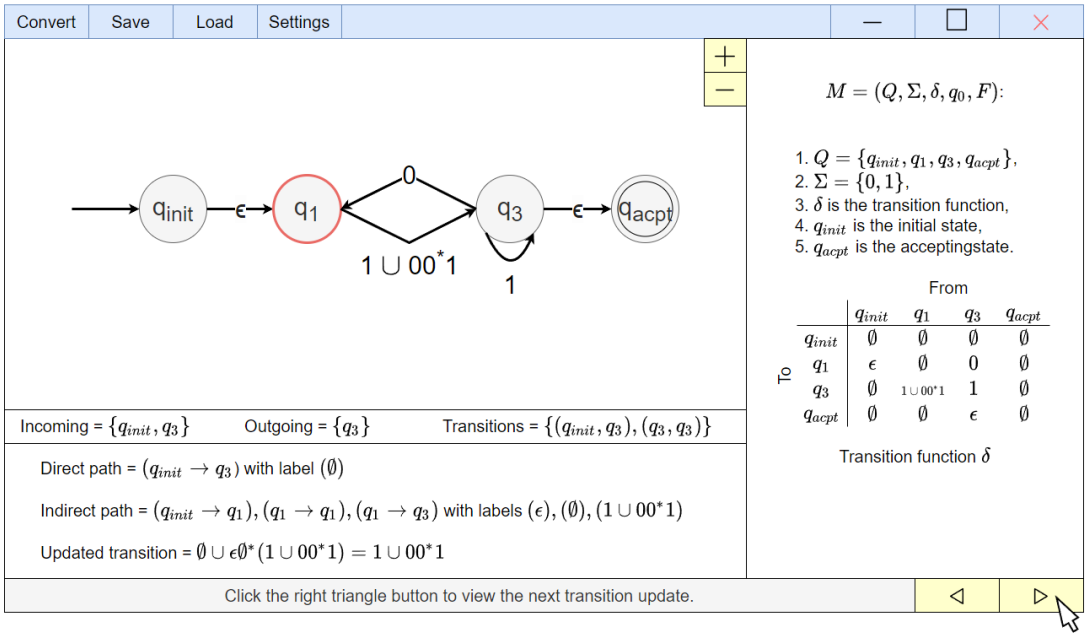


Figure 3.3: Converting a finite automaton into a regular expression.

The software will first convert the user's finite automaton into a GNFA. It will then ask the user to select a state to remove. Once the user has selected a valid state, the software will show how each transition is updated. It will then display the updated GNFA and the process repeats.

The window consists of a workspace, a details tab, a conversion tab and a toolbar. The workspace area displays the current iteration of the GNFA as well as buttons for zooming in and out. The details tab displays the formal definition of the GNFA. The conversion tab shows a step in the process of converting the finite automaton into a regular expression. Displayed in the conversion tab is a set of states with transitions into the state being removed, a set of states with transitions out of the state being removed, and a set of transitions that need to be updated. Below that, the steps involved in updating a single transition are shown in the form of the the direct and indirect paths as well as the updated label. The toolbar at the bottom

of the screen contains a short text area with instructions and error messages, and buttons for moving to the next or previous step.

3.2.4 Regular Expression Editor

The regular expression editor is the window used to create regular expressions.

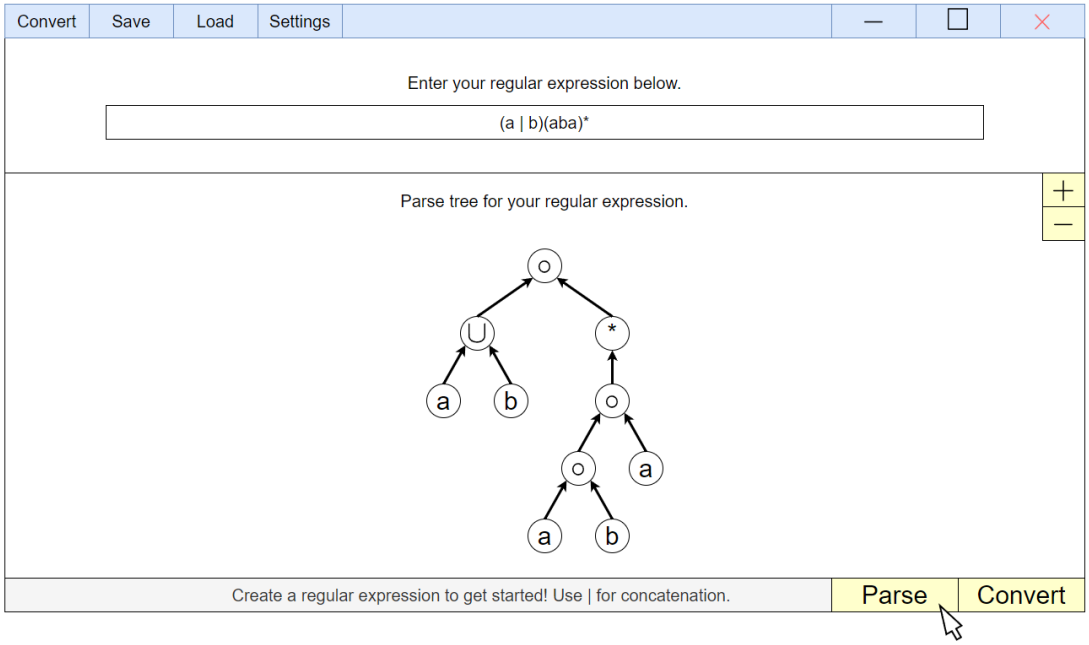


Figure 3.4: Regular Expression Editor.

The regular expression editor consists of a workspace, a details tab and a toolbar. The workspace area is where the user can input a regular expression. Once the user has entered a regular expression and pressed the parse button in the toolbar, the details tab will show the parse tree for the regular expression (assuming the regular expression entered was valid). This allows the user to check that the expression they have created is correct. The parse tree will also be used in the conversion process. The details tab also contains buttons for zooming in and out. Finally, in addition to the parse button already mentioned, the toolbar at the bottom of the screen also contains a short text area with instructions and error messages, and a convert button that begins the process of converting the regular expression into a finite automaton.

3.2.5 Regular Expression Conversion

This window is displayed when converting a regular expression into a finite automaton.

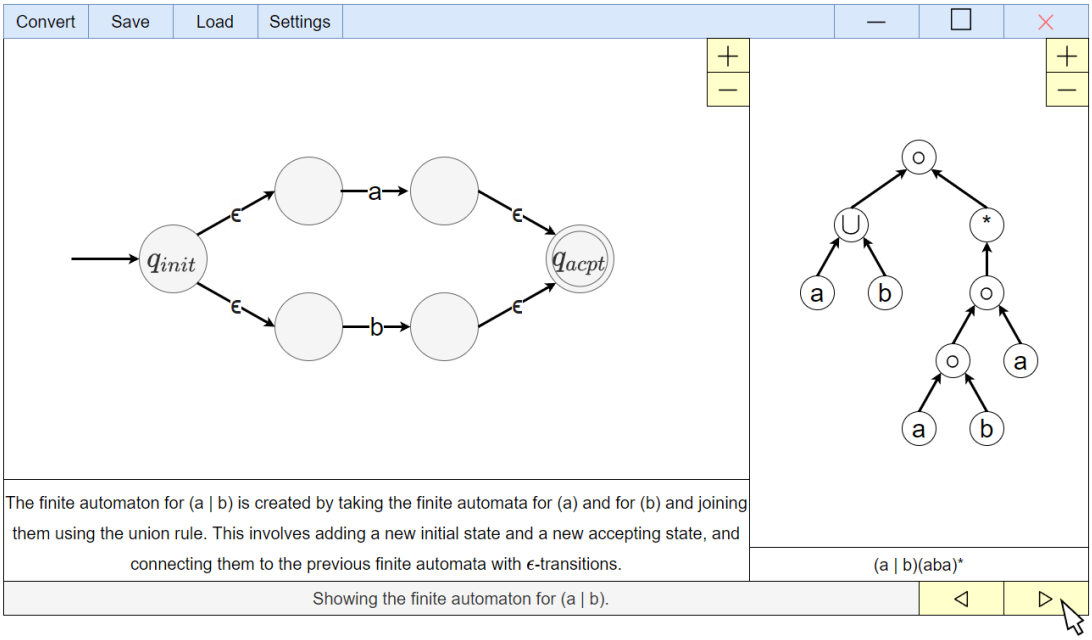


Figure 3.5: Converting a regular expression into a finite automaton.

The software will perform a postorder traversal of the parse tree, showing the finite automata created at each node.

The window consists of a workspace, a details tab, a conversion tab and a toolbar. The workspace area displays the current iteration of the finite automaton. The details tab displays the original regular expression and its parse tree. Both contain buttons for zooming in and out. The conversion tab explains how the current iteration of the finite automaton was created. The toolbar at the bottom of the screen contains a short text area with instructions and error messages, and buttons for moving to the next or previous step.

Chapter 4

Implementation

This chapter will discuss the implementation of the software. It will take a closer look at some of the key classes and methods, justify the decisions made and explain the reasoning behind them, and consider some of challenges faced and how they were overcome.

The software was written in the Java[1] programming language, using the JavaFX[2] platform for the GUI and the JUnit[3] framework for testing. This decision was made for two reasons. Firstly, I am familiar with Java, JavaFX and JUnit, having used them before in the past. This meant that I didn't have to spend time learning new libraries or frameworks. It also decreased the risk associated with the project. Secondly, I wanted to try and make the application compatible with multiple operating systems. Java is well known for it's "write once, run anywhere" ideology and so I hoped that by using Java, I could easily get the software to run on other operating systems.

4.1 Regular Expressions

Regular expressions are represented by the abstract `RegularExpression` class and its two subclasses. The `SimpleRegularExpression` subclass represents the base case in the inductive definition of regular expressions. A simple regular expression is a regular expression consisting of a single symbol from the alphabet Σ. For the purpose of the software, the alphabet Σ is defined as all Latin alphanumeric characters. The `ComplexRegularExpression` subclass

[1] `https://www.java.com/en/`
[2] `https://openjfx.io/`
[3] `https://junit.org/junit5/`

represents the inductive cases in the inductive definition of regular expressions. It consists of a regular expression operator (a regex operator) applied to one or two `RegularExpression` objects (depending on the type of operator). The `RegularExpression` objects, referred to as the left operand and right operand respectively, may themselves also be complex regular expressions. This allows for a tree-like structure where regular expressions are composed of other regular expressions. This makes dealing with regular expressions significantly easier. The tree-like structure of regular expressions is made explicit by the `ParseTree` class.

4.2 Parse Trees

The `ParseTree` class represents a regular expression parse tree, which is a tree where internal nodes are regex operators and leaf nodes are symbols from the alphabet Σ. The parse tree is created in two stages. First, the `buildParseTree` method creates all the nodes of the tree and places them in the correct positions. The `connectNodes` method then connects the nodes with edges. The nodes of the parse tree are represented by the `ParseTreeNode` class, which stores the visual representation of the node as well as references to the children of the node.

4.2.1 The buildParseTree method

The `buildParseTree` method is a recursive method that builds the parse tree for a given regular expression. It also takes in two additional variables, `currentX` and `currentY`, which are used to keep track of the position of the root node of the current parse tree. If the given regular expression is a simple regular expression, a leaf node representing the regex symbol is created and moved into position. If the regular expression is a complex regular expression, an internal node representing the regex operator is created instead. The method is then recursively called with the left and right operands of the complex regular expression to build the left and right subtrees. Before this can happen however, the program must first calculate the position of the root node of the subtrees. The vertical offset from the current position is always the same, but the horizontal offset varies and needs to be calculated. If the regex operator is the star operator, this task is trivial: since the star operator is unary, there is no need for a horizontal offset and the root node of the subtree can be placed directly below the current position. However, if the regex operator is the concatenation or union operator, the horizontal offset needs to be calculated such that the subtree of the left operand and the subtree of the right operand do not overlap. This will depend on the number of leaf nodes in the subtrees.

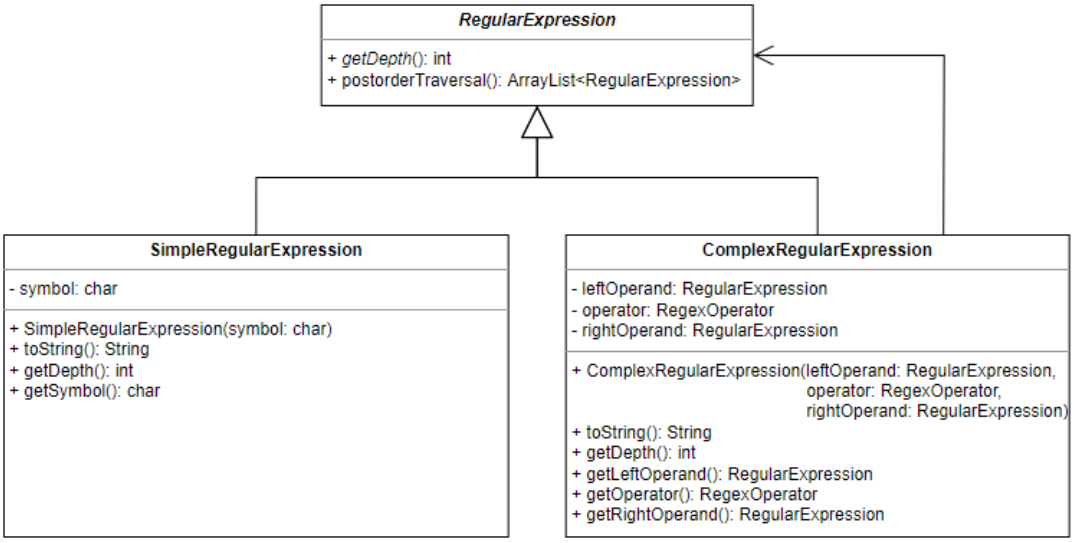


Figure 4.1: The classes representing regular expressions.

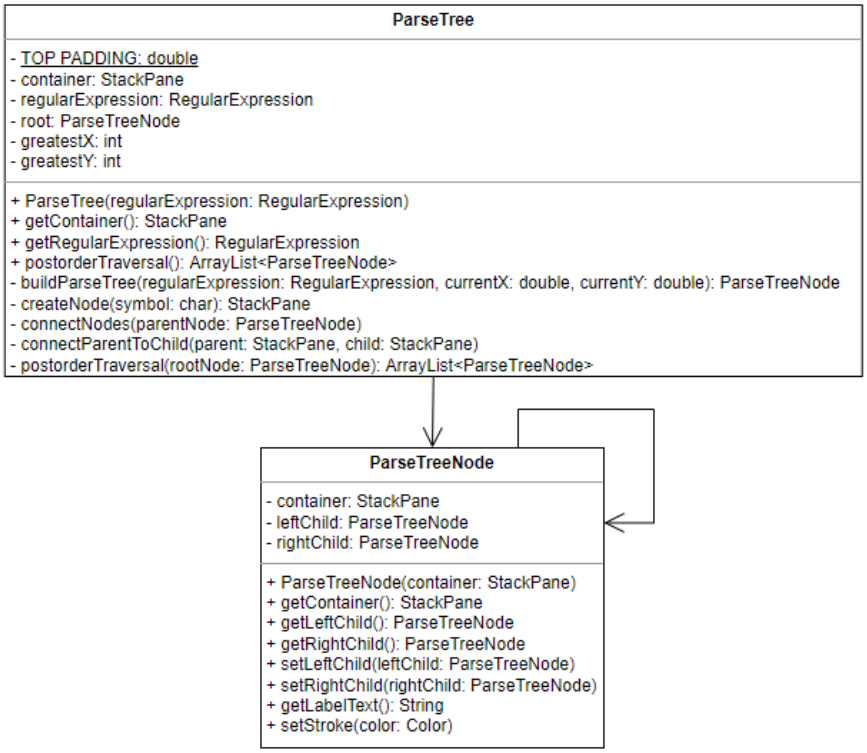


Figure 4.2: The `ParseTree` class is a graphical representation of a regular expression.

The parse tree is built top-down, which means the method does not know the number or structure of leaf nodes in the lower levels of the tree. As such, it has to assume the worst case scenario: that the parse tree will be a perfect tree (meaning that all interior nodes have two children and all leaf nodes are at the same level). As the number of leaf nodes doubles with depth, so will the horizontal offset. If k is the horizontal offset needed for a subtree with a max depth of 1, the horizontal offset x needed for a subtree with max depth $d > 1$ can be calculated as follows: $x = 2^{d-1} \times k$.

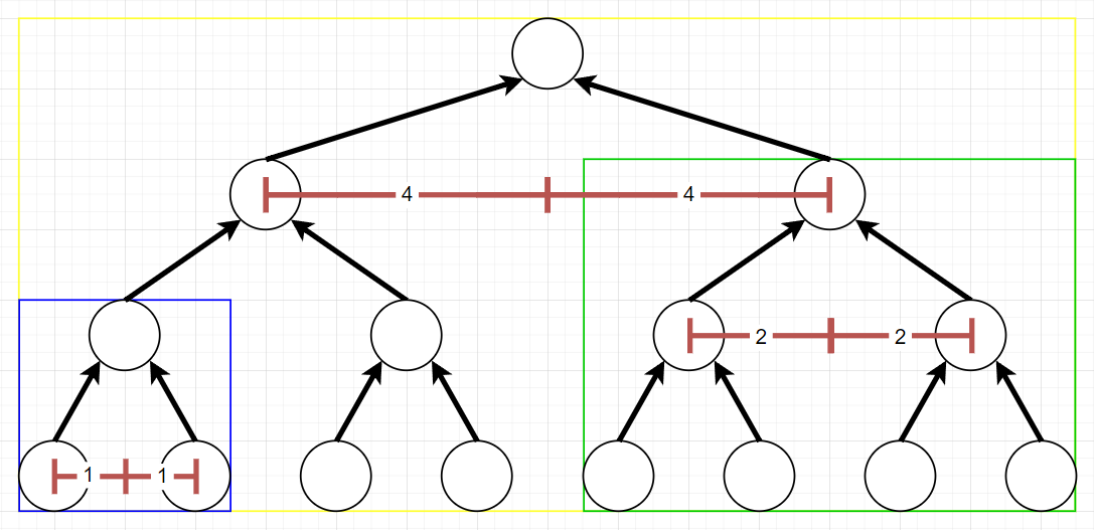


Figure 4.3: The horizontal offset needed at the root doubles with each increase in depth.

For example, consider the nodes in the blue rectangle in Figure 4.3. The horizontal offset needed for a tree with a depth of 1 is 1 square. If the max depth of the tree increases to 2 (green rectangle), the horizontal offset needed is now $2^{2-1} \times 1 = 2$ squares. Finally, if the max depth of the tree is 3 (yellow rectangle), the horizontal offset needed is $2^{3-1} \times 1 = 4$ squares.

Once the method has calculated the horizontal offset needed, it first subtracts it from the `currentX` when building the left operand subtree and then adds it to the `currentX` when building the right operand subtree. Having built the subtrees and set the children of the operator parse tree node accordingly, the method then returns the operator node of the tree being built. The last operator node returned will be the root node of the parse tree for the whole regular expression.

4.2.2 The connectNodes method

Although the `ParseTreeNode` class keeps references to its children, its `container` variable (which stores the graphical representation of the node) does not include edges between the nodes, just the nodes themselves. It is the responsibility of the `connectNodes` method to visually connect the nodes by adding edges to the parse tree. This method is also recursive. Given a `ParseTreeNode` as input, it first connects the node to its children using the `connectParentToChild` helper method, and then recursively calls itself on both children.

4.3 Parser

The `RegularExpression` class represents regular expressions and the `ParseTree` class displays them visually, but it is the `Parser` class that creates them in the first place. It contains several methods for dealing with strings representing regular expressions (regex strings).

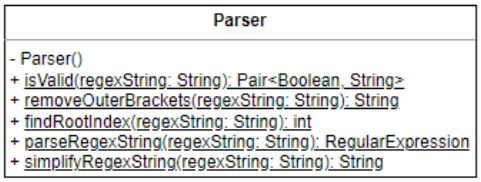


Figure 4.4: The `Parser` class is responsible for parsing strings representing regular expressions into `RegularExpression` objects.

4.3.1 The isValid method

The `isValid` method checks that the given regex string is valid, i.e. that it contains only symbols from the alphabet Σ, the empty string and empty set symbols, the available regex operators and brackets. The default symbols for the regex operators in the program are as follows:

- The Kleene star regex operator uses the asterisk symbol `*`.
- The concatenation regex operator uses the vertical pipe symbol `|`.
- The union regex operator uses the plus symbol `+`.

The `isValid` method also checks that the number and placement of brackets is valid. It returns a pair (K, V), where K is a boolean representing whether the regex string is valid or not, and V is an error message for the case that the regex string is invalid.

4.3.2 The removeOuterBrackets method

The `removeOuterBrackets` method removes the outer brackets from a string. Regular expressions with outer brackets such as $(a + b)$ or even $((a + b))$ are valid, but the outer brackets are superfluous and can be safely removed without changing the regular expression. The string remains unchanged if it does not contain outer brackets - for example, the regex string $(a+b)|(c+d)$ would not have its brackets removed, since the opening bracket at the start and the closing bracket at the end are not linked and therefore not considered outer brackets.

4.3.3 The findRootIndex method

The `findRootIndex` method takes in a regex string that has had its outer brackets removed and finds the index of the root regex operator of the regular expression. The root regex operator is the operator that will be the root node in the regular expression's parse tree. To find this root operator, the method looks for regex operators that are not contained within any brackets. If such an operator is found, its index is returned. For example, in the regular expression $(a+b)|d$, the root operator is the concatenation operator at index 5.

Problems arise when the regex string has more then one eligible root operator. For example, in the regular expression $a + b^*|c$, all three operators are not contained within any brackets and so appear to be eligible root operators. In practise, the union and concatenation operators could be the root operator, but taking the star operator as the root would result in an invalid regular expression being formed. One solution to this problem would be to enforce a restriction on the regex strings such that there is always only one regex operator that is not contained within brackets. The regular expression from the previous example could then be written as $a + ((b^*)|c)$. This solution is not desirable, as it places the burden on the user to add the correct brackets. Instead, the `findRootIndex` method considers the precedence of the regex operators when looking for the root operator. The star operator has the highest precedence and the union operator has the lowest precedence, with concatenation being in the middle. The root operator can now be defined as the lowest precedence operator that is not contained within brackets. Using this new definition, the root operator in the regular expression $a + b^*|c$ is the union operator. This makes the regular expression equivalent to $a + ((b^*)|c)$, but without forcing the user to explicitly add in those extra brackets.

The updated definition of the root operator creates a new issue: regular expressions such as $a|b|c$ contain several operators that are not contained within brackets and have equal precedence. A

tie-breaker must be added to resolve situations such as this. If there are multiple candidates for the root node operator, all with equal precedence, then the right-most operator is chosen as the root. This means that regular expressions such as $a|b|c$ are considered to be equivalent to $(a|b)|c$ and not $a|(b|c)$. Thinking of $a|b|c$ as appending c to $a|b$ seemed more natural to me than prepending a to $b|c$. This way of thinking is reflected in the finite automaton construction for $(a|b)|c$: the finite automaton for $a|b$ is created first, and then the finite automaton for c is added at the end.

4.3.4 The parseRegexString method

Having defined the previous methods, the `parseRegexString` method can now be discussed. This method takes in a regex string and returns its equivalent `RegularExpression` object, or throws an `IllegalArgumentException` if the regex string does not represent a well-formed regular expression. The `parseRegexString` method first checks that the regex string is valid using the `isValid` method. The `removeOuterBrackets` method is then used to strip the regex string of any outer brackets. At this point, the regex string is checked to see if it matches a simple regular expression. If this is the case, a `SimpleRegularExpression` object is created and returned, and the method terminates. Otherwise, the regex string must represent a complex regular expression. The root operator of the complex regular expression is found by the `findRootIndex` method. Everything to the left of the root operator is taken to be the left operand. The left operand is parsed into a `RegularExpression` object by recursively calling the `parseRegexString` method. The root operator of the regex string is then checked. If the operator is concatenation or union, the complex regular expression should have a right operand. Everything to the right of the root operator is taken to be the right operand and parsed into a `RegularExpression` object. Finally, the method creates and returns a new `ComplexRegularExpression` object using the parsed left and right operands as well as the root operator.

Throughout the execution of the `parseRegexString` method, the regex string is checked to ensure that it represents a well-formed regular expression. One example of such a check is that, in a well-formed regular expression, if the root operator of the regex string is the star operator, the regex string shouldn't have a right operand. If a right operand is found, the method will throw an exception. The error message of the exception is displayed to the user and describes the cause of the problem, highlighting which part of the regex string is to blame. This allows the user to easily identify mistakes in the regex string.

4.3.5 The simplifyRegexString method

The `Parser` class has one more method: the `simplifyRegexString` method. This method takes as input a regex string representing a regular expression, and returns as output a regex string representing the same regular expression, but with all unnecessary brackets removed. For example, given the regex string $(((a^*)|b) + (c|(d^*)))$, the `simplifyRegexString` method returns the equivalent regex string $a^*|b + c|d^*$. The method searches the regex string for a pair of brackets to remove. If a pair of brackets can be found, they are removed the from regex string, which is then parsed by the `parseRegexString` method. The resulting regular expression is compared to the original regular expression, represented by the original regex string. If the two expressions are equivalent, the new regex string is kept: otherwise, it is discarded and the method searches for another pair of brackets to remove. If such a pair cannot be found, the regex string cannot be simplified further and the method terminates.

4.4 Graphical Finite Automata

Graphical finite automata are used when converting regular expressions into finite automata. The construction of a finite automaton from a regular expression follows defined rules and the finite automaton produced has a fixed structure. Emphasis is placed on the presentation of the finite automaton, for example ensuring that it is always centred on the screen. The components of a graphical finite automaton can be manipulated during its construction to position them correctly. Because they are used solely for presentation, graphical finite automata concern themselves with just the graphical aspect of a finite automaton. They do not keep track of their components: a graphical finite automaton does not know what states or edges belong to it, with the exception of the initial and final state. Each component of a graphical finite automaton is isolated from the other components and does not know about their existence.

4.4.1 The `GraphicalFiniteAutomaton` class

Graphical finite automata are represented by the abstract `GraphicalFiniteAutomaton` class, which is extended by two subclasses. The `SimpleGraphicalFiniteAutomaton` subclass is used for simple regular expressions and consists of only two states with a single edge between them. The `ComplexGraphicalFiniteAutomaton` subclass is used for complex regular expressions and consists of multiple states and edges. The states and edges of a graphical finite automaton are stored in its `container` variable. The `GraphicalFiniteAutomaton` class also contains methods

for enabling and disabling the initial and final states of the automaton: this is necessary when the automaton is used in the construction of other automata.

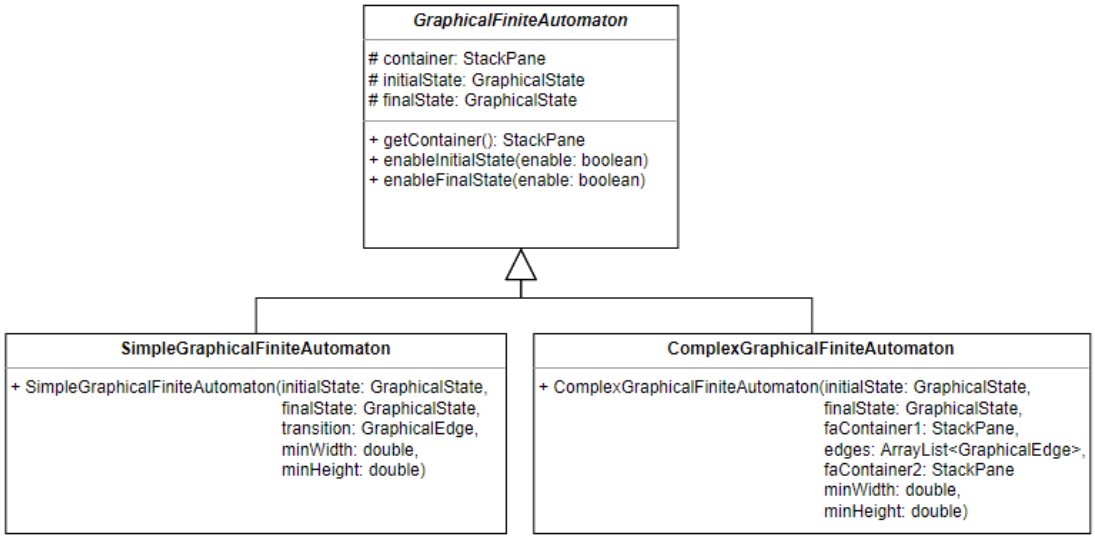


Figure 4.5: The classes representing graphical finite automata.

4.4.2 Graphical States and Graphical Edges

Graphical finite automata are composed of graphical states and graphical edges, represented by the `GraphicalState` class and the `GraphicalEdge` class respectively. Both have a `container` variable that holds the components that make up the state or edge. The `GraphicalState` class also contains methods for changing the state to an initial or final state. These methods pass on the request to the `GraphicalFiniteAutomatonBuilder` class.

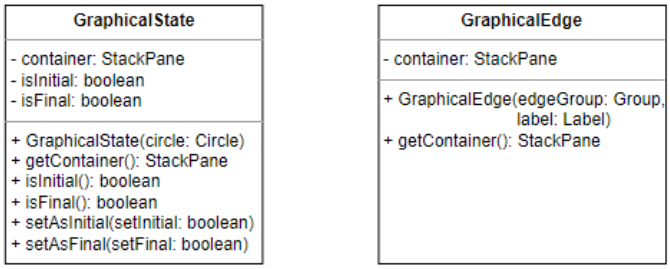


Figure 4.6: The classes representing graphical states and edges.

4.4.3 Graphical Finite Automaton Builder

The `GraphicalFiniteAutomatonBuilder` class is responsible for building and manipulating graphical finite automata and their components. The most important method in this class is the `buildFiniteAutomaton` method, which takes as input a `RegularExpression` object and

returns as output a corresponding `GraphicalFiniteAutomaton` object. This is a recursive method, which first builds the finite automata/finite automaton for the operands/operand of the regular expression and then combines them/extends it according to the construction rule of the regex operator used in the regular expression (see Chapter 2.5). Each regex operator has its own construction rule defined in its own helper method. These methods manipulate the initial states, final states and `container` variables of the given finite automata to create a new complex finite automaton. The base case of the recursion is when the regular expression is a simple regular expression, in which case a `SimpleGraphicalFiniteAutomaton` is constructed.

The `GraphicalFiniteAutomatonBuilder` class also contains methods for creating graphical states and edges, for adding and removing the marking that sets a state apart as an initial or final state, and for generating text explaining how a finite automaton was constructed.

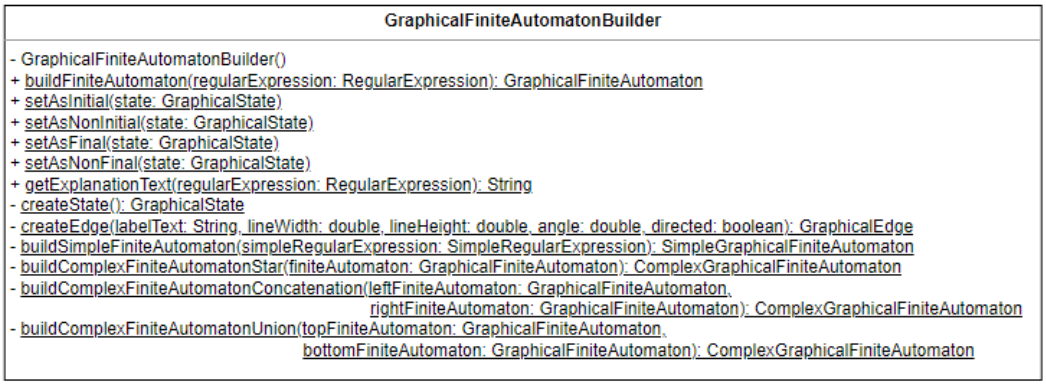


Figure 4.7: The `GraphicalFiniteAutomatonBuilder` class is responsible for building and manipulating graphical finite automata and their components.

4.5 Smart Finite Automata

Smart finite automata are used when converting finite automata into regular expression. Unlike graphical finite automata, smart finite automata do not have a fixed structure. They are created and manipulated by the user rather than the program. As such, there is no need to centre them or position their components in precise locations: the user is free to move the components around as they see fit. This requires the finite automata to be 'smarter' than their graphical counterparts; smart finite automata keep track of their states and edges. The components themselves are also smarter; smart states know what edges are connected to them and smart edges know what states they connect.

4.5.1 The `SmartFiniteAutomaton` class

Smart finite automata are represented by the `SmartFiniteAutomaton` class. Just like graphical finite automata, smart finite automata keep their components in their `container` variable. They also keep a list of states and edges belonging to the finite automaton, in addition to the initial state and final state. Smart finite automata are also linked to the `CreateFAScreenController` responsible for creating them and the `ConvertFAScreenController` responsible for converting them into regular expressions (see Chapters 4.6.1 and 4.6.2). Whenever a state or edge is added to the finite automaton, the automaton will ask the appropriate controller to set the user interaction behaviour for that state or edge. The user interaction behaviour defines how the user can interact with the state or edge through mouse clicks and mouse drags, as well as what kind of actions the user can perform.

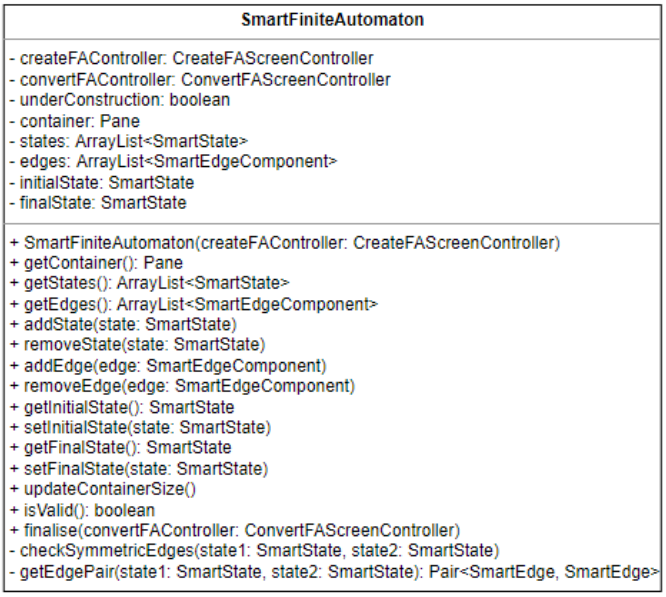


Figure 4.8: The `SmartFiniteAutomaton` class.

The `setInitialState` and `setFinalState` methods are used to set the initial state and the final state of the automaton. Smart finite automata are more restrictive then the finite automata described in Chapter 2.2. Smart finite automata can only have one initial state and one final state. The initial state and final state cannot be the same. Note that this does not restrict the user in the types of finite automata that can be created: finite automata with multiple final states or where the initial state and final state are the same can be converted into equivalent finite automata that follow the smart finite automata restrictions, by introducing new states and connecting them with ϵ-transitions.

The `SmartFiniteAutomaton` class has methods for adding and removing states and edges. When a state is removed from a smart finite automaton, all edges connected to that state are also removed. Smart finite automata are limited to having at most one edge in each direction between any two states. Displaying multiple edges between two states in the same direction would quickly become problematic: to prevent the edges from overlapping each other, each subsequent edge would need to become more and more curved. Instead, a common notation is adopted where an edge may have multiple symbols on its label separated by commas. For example, an edge with the label a, b, c is equivalent to three edges with one symbol each.

Two states may still have two edges between them, as long as the edges are going in opposite directions. These edges are referred to as symmetric edges. Whenever an edge is added to or removed from a smart finite automaton, the automaton will check for symmetric edges. This is done using the `checkSymmetricEdges` method. If the method finds symmetric edges, it will replace the straight edges with curved ones. This makes it easier to see which label corresponds to which edge. If there is only one edge between the two states, the `checkSymmetricEdges` method ensures that the edge is straight.

One final method of interest is the `isValid` method, which checks that the finite automaton created is valid. A valid finite automaton must have an initial state and a final state. In addition, every state in the finite automaton must be reachable from the initial state. This is checked by performing a breadth-first search from the initial state and making sure that every state in the finite automaton is encountered.

4.5.2 Smart Components

The abstract class `SmartComponent` represents a component in a smart finite automaton. It contains methods for that are common to all finite automata components. A smart component is either a `SmartState` or a `SmartEdgeComponent`. The `SmartState` class represents a state in a smart finite automaton. A smart state keeps track of its incoming and outgoing edges. The `SmartEdgeComponent` class is an abstract class representing an edge component in a smart finite automaton. Smart edge components keep track of the states they are connected to. There are two types of smart edge components: a `SmartEdge` and a `SmartLoopEdge`. A smart edge is an edge that connects two different states. In addition to the variables and methods it inherits from the `SmartEdgeComponent` class, it also has an extra method called `setCurved` that allows the edge to switch between a straight line and a curve. On the other hand, a smart loop

edge connects a state to itself. In addition to the variables and methods it inherits from the `SmartEdgeComponent` class, it also has an extra method called `flip`, which changes the position of the loop edge. The loop edge may be either above the state or below it.

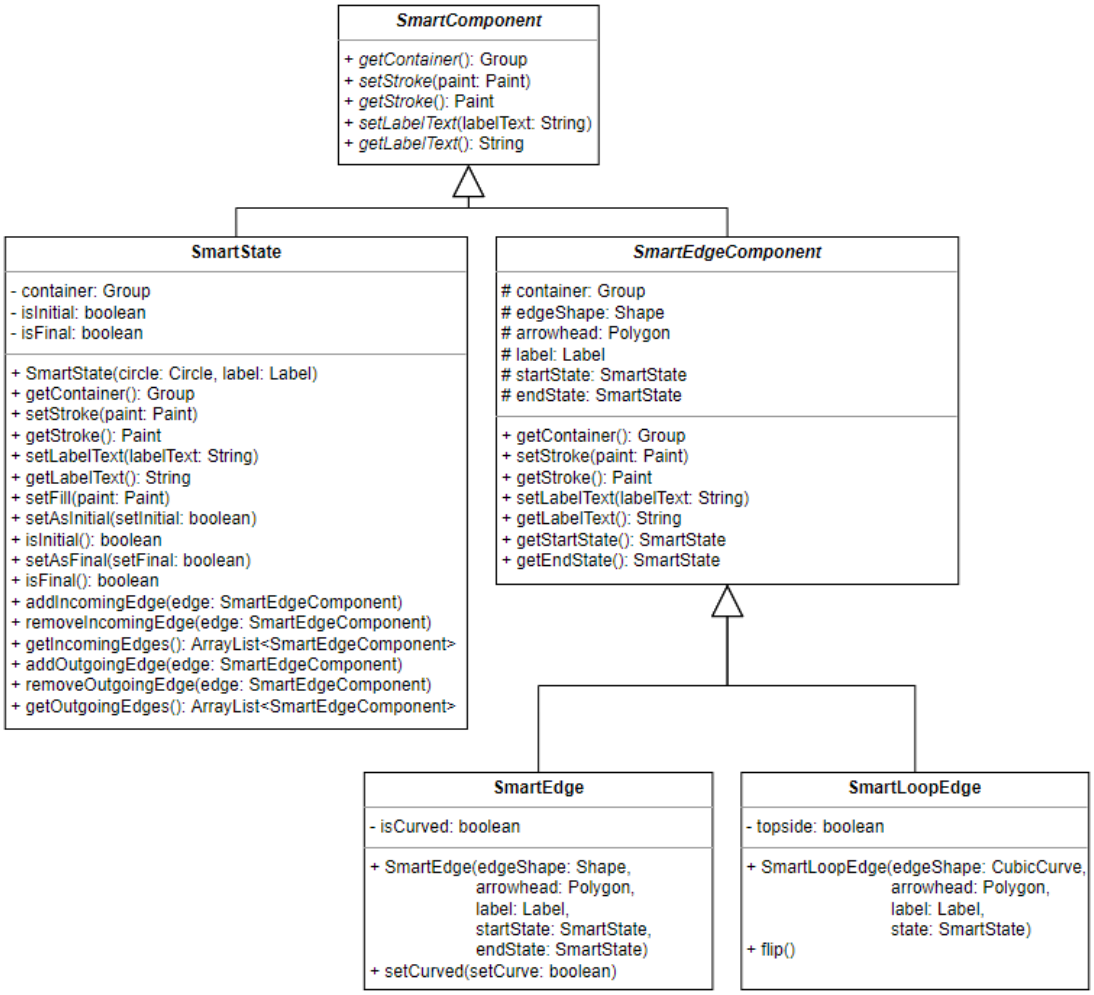


Figure 4.9: Smart components make up smart finite automata.

4.5.3 Smart Finite Automaton Builder

The `SmartFiniteAutomatonBuilder` class is responsible for building and manipulating smart finite automata and their components. It can create smart states, straight and curved smart edges, and smart loop edges. It contains methods for adding and removing the marking that sets a state apart as an initial or final state. It also has a method for setting the bindings for a smart loop edge (smart loop edges need to have their bindings set when they are created or when they have their position flipped).

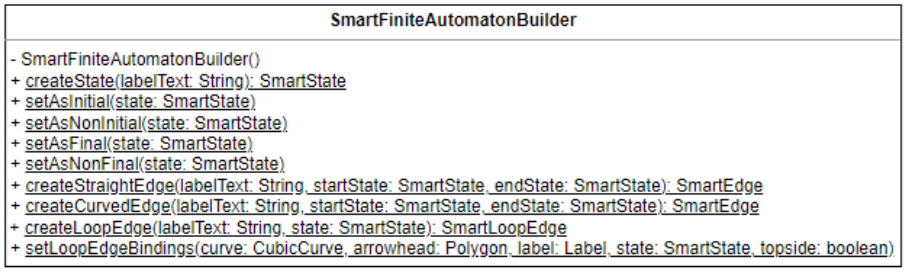
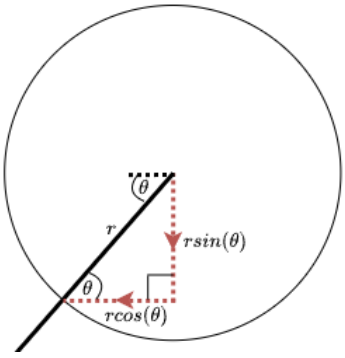


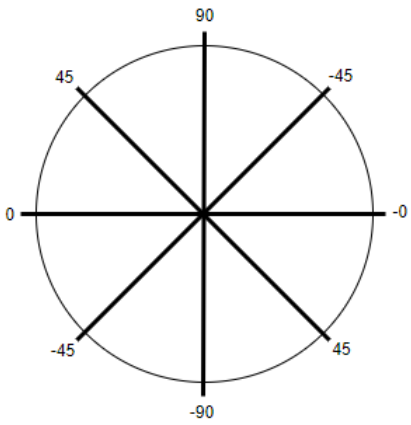
Figure 4.10: The `SmartFiniteAutomatonBuilder` class is responsible for building and manipulating smart finite automata and their components.

The methods for creating straight, curved and loop edges were particularly difficult to implement. The problem was that, due to the dynamic nature of smart finite automata, the edges had to move along with the states they were connected to.

For straight edges, the first step in achieving this behaviour was to bind the start position and end position of the line to the centre of the start state and end state respectively. This ensured that the length and gradient of the line would change appropriately when either one of the states was moved. The next step was to move the arrowhead into position, such that it sits on the line with its point touching the circumference of the end state. The arrowhead was bound to the centre of the end state, but with an x-offset and y-offset added to move it from the centre of the state to the circumference. These offsets were calculated by working out the horizontal and vertical components of the radius of the circle at the point where the line crosses the circumference. These offsets are shown in Figure 4.11(a) as dotted red lines. To work out the offsets, the angle θ needed to be calculated.



(a) The offsets for the arrowhead in a straight `SmartEdge`.



(b) The range of angles returned by `Math.atan(sToEY / sToEX)`.

Figure 4.11

In order to calculate θ, the translation needed to get from the start state to the end state had to be determined first. This was calculated from the position of the two states and stored in the `sToEX` (for the x-translate) and `sToEY` (for the y-translate) variables. It's important to note that both `sToEX` and `sToEY` may be negative, depending on the positions of the start and end states. Once these translates have been calculated, the angle θ could be worked out as $tan^{-1}(\texttt{sToEY} \div \texttt{sToEX})$. This could return a negative value - see Figure 4.11(b) - so the absolute value was used in the calculation of the offsets. The sign on `sToEX` and `sToEY` was then used to decide whether to add or subtract the respective offsets. For example, if `sToEX` was positive, that meant that the end state was to the right of the start state, and so the x-offset needed to be subtracted to move the arrowhead to the left from the centre of the end state.

The last step was to work out the angle needed to rotate the arrowhead so that it faced towards the centre of the end state. If the end state was on the right of the start state, the angle was just the true (non-absolute) value of θ; if the end state was on the left side, the angle was $180 + \theta$ (see Figure 4.11(b) for the different angles of θ).

Curved smart edges use a quadratic Bézier parametric curve segment as their edge and so have an additional control point that defines the shape and position of the curve (note that the control point does not actually lie on the curve itself). This control point also needs to be positioned appropriately (see Figure 4.12). The control point was bound to the centre of the start state, but with an x-offset and y-offset added. Each one of these offsets is made up of two parts:

1. The first part (shown as dotted blue lines in Figure 4.12) moves the control point to the centre of a line between the start state and the end state (this line does not actually exist, but it is shown in Figure 4.12 as a dotted green line for illustrational purposes). This primary offset was computed by calculating `sToEX` and `sToEY` and then multiplying them by 0.5.

2. The second part (shown as dotted red lines in Figure 4.12) moves the control point a constant distance k perpendicular to the line between the two states (in Figure 4.12, this distance is shown as a dotted yellow line). This secondary offset was calculated in a similar manner to the offset for arrowheads in straight smart edges, using the angle θ. The position of the end state relative to the start state was used to decide whether to add or subtract this part of the offset.

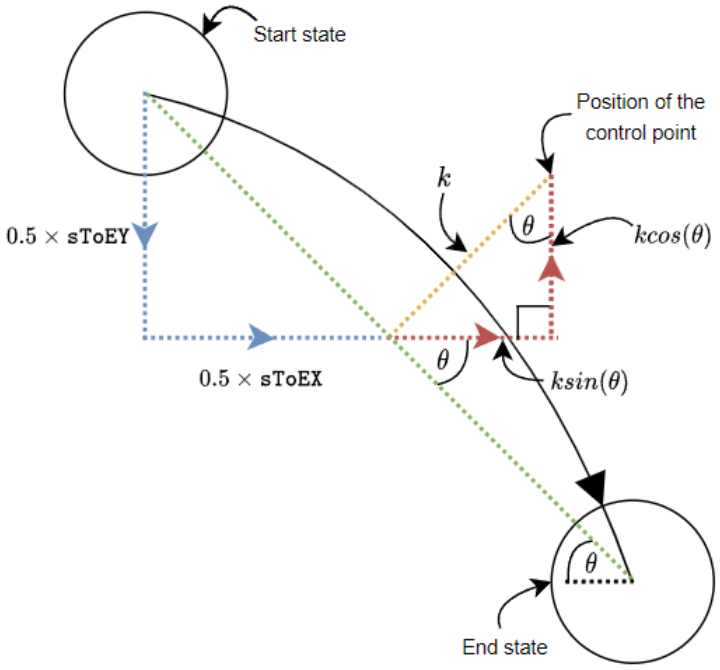


Figure 4.12: The offsets for the control point of a `QuadCurve` in a curved `SmartEdge`.

The arrowhead for a curved smart edge was positioned and angled in a very similar manner to the arrowhead for a straight smart edge. The only difference is that instead of using `sToEX` and `sToEY`, a different pair of translates were used: `cToEX` and `cToEY`, representing the translation needed to get from the control point to the end state. I had to resort to using this translation as I was unable to calculate the tangent to the curve at the point where it crosses the circumference of the end state. As long as the two states are not very close to each other, the translation does a fairly good job of approximating the tangent.

Smart loop edges use a cubic Bézier parametric curve segment as their edge and so have two control points instead of one. However, because loop edges connect a state to itself, calculating the bindings for the control points was not as difficult as calculating the bindings for the control point of a curved smart edge. First of all, the start and end points of the curve are bound to the centre of the state, but with a horizontal offset subtracted or added to move them to the far left and far right sides of the circumference (these horizontal offsets are shown in Figure 4.13 as dotted blue lines). The two controls points are also bound to the centre of the state, but their offsets move them directly above or directly below the start and end points (recall that a user can decide whether the loop edge is above or below the state). The offsets for the control points can be seen in Figure 4.13 by considering the dotted blue and red lines together.

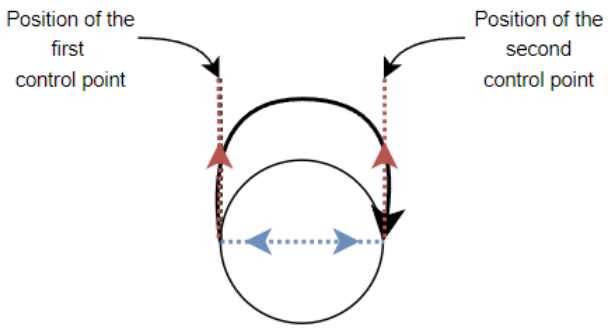


Figure 4.13: The positions of the control points of a `CubicCurve` in a `SmartLoopEdge`.

4.6 Controllers

This project uses the Model-View-Controller (MVC) design pattern to separate classes with different concerns.

- Models deal with storing and manipulating data. An example of a model is the `Parser` class, which manipulates regular expressions.
- Views are concerned with the presentation of the program. They are implemented through the use of JavaFX FXML files.
- Controllers are responsible for user interaction. Each view has its own controller.

4.6.1 CreateFAScreenController

The `CreateFAScreenController` class is the controller for the view that is used to create finite automata. The controller implements the logic that allows the user to create states and edges, move states around the workspace, rename labels etc. It does this by defining a `WorkMode` that changes how the user is able to interact with the workspace. The user can switch between different work modes by using the buttons in the toolbar at the bottom of the screen, or by using the number keys on the keyboard. There are three available work modes:

1. In the MOVE work mode, the user can move states around the workspace by dragging them while holding the left mouse button. The user can drag states outside the right and bottom edges of the window to increase the size of the workspace. States cannot be moved past the left and top edges of the window.
2. In the STATE work mode, clicking on the workspace with the left mouse button will create a new state at that location.

3. In the EDGE work mode, the user can create new edges between states by holding down the left mouse button while on top of on one state and releasing it while on top of another state (if the start state and the end state are the same, a loop edge is created instead).

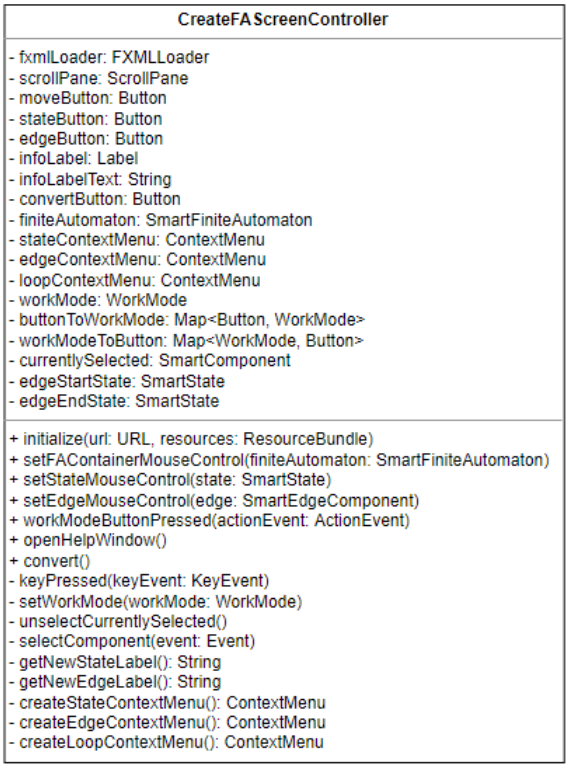


Figure 4.14: The `CreateFAScreenController` class is the controller for the view that is used to create finite automata.

Each finite automaton component also has its own context menu, which is opened when the user right-clicks on the component. This is available regardless of the current work mode. The context menus allow the user to perform different actions on the components:

- The context menu for states allows the user to rename the state, set the state as the initial state, set the state as the final state and to delete the state.
- The context menu for edges allows the user to rename and delete the edges.
- The context menu for loop edges allows the user to rename, flip and delete the loop edges.

Renaming a state or edge will open a small window where the user can type in the new label. The user will not be able to press the Submit button to confirm the new label unless it meets the requirements for that component. States and edges have different requirements for their labels.

Whenever the user interacts with a component, that component becomes selected. Selecting a component highlights it and sets it as the target of any actions. For example, opening the context menu for a state will automatically select it. If the user then chooses the Delete option, the program will know which state it needs to remove.

4.6.2 ConvertFAScreenController

The `ConvertFAScreenController` class is the controller for the view that is used to convert finite automata into regular expressions. Just like the `CreateFAScreenController` class, it defines the user interaction behaviour for the components of a finite automaton. In this view, the user is able to move states around the workspace, but they can no longer create new components or change existing ones (the only exception are loop edges, which can still be flipped).

Figure 4.15: The `ConvertFAScreenController` class is the controller for the view that is used to convert finite automata into regular expressions.

The `ConvertFAScreenController` class also contains methods that help in converting finite automata into regular expressions. The `getDirectPath` method returns the direct path (i.e. the edge) between two states. By calling this method repeatedly, the `getIndirectPath` method

is able to obtain the indirect path between two states via a third state. The labels of these paths are then combined and simplified using the `simplifyLabel` method. This method uses the identities outlined in Chapter 4.1 as well as the additional identity $R \cup R = R$ to simplify the combined label.

The conversion process itself was implemented through the use of the Command design pattern. This design pattern turns an action into its own class that contains all the information about the action. By storing this information, it is possible for the action to be undone. The abstract class `Command` represents all possible commands. The class consists of two abstract methods. The `execute` method performs the action represented by the command. It saves the values of any variables it changes to allow the action to be undone. When the `undo` method is called, these variables are returned to their original values. For the conversion of a finite automaton into a regular expression, three commands were created. The `SelectStateToRemoveCommand` represents the action of the user selecting which state of the finite automaton to remove in the next iteration of the State Elimination algorithm. The `UpdateEdgeLabelCommand` represents a single update step of the algorithm, where the label of one edge of the finite automaton is updated to reflect the chosen state being removed. Finally, the `RemoveStateCommand` removes the state from the finite automaton.

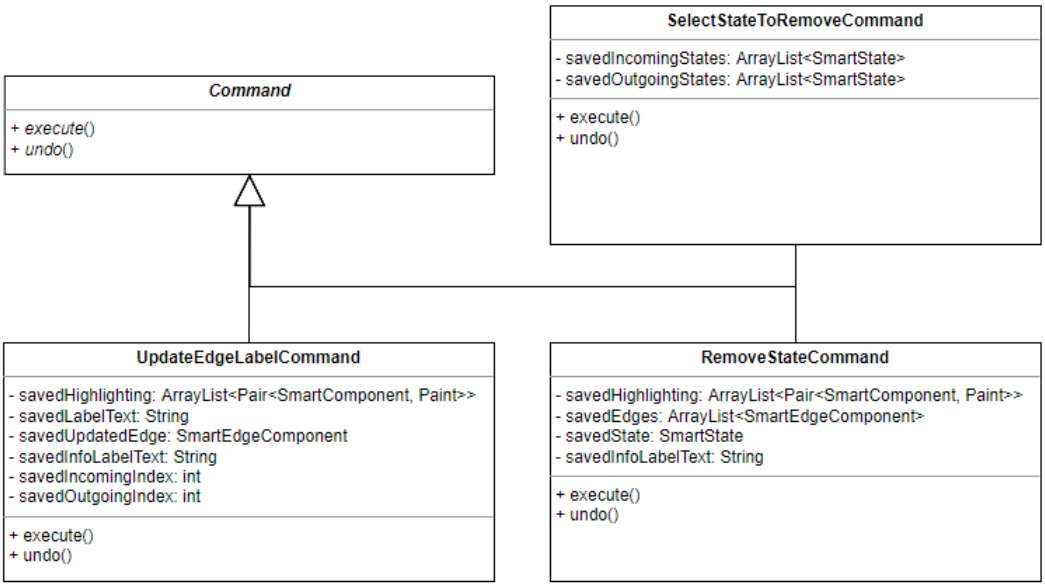


Figure 4.16: The Command design pattern turns an action into its own class that stores information about the action. This allows the action to be executed and undone.

The `ConvertFAScreenController` keeps track of which command to call next through the use of a `WorkMode` (note that this is not the same `WorkMode` as in `CreateFAScreenController`).

There are three available work modes, one for each of the three commands: SELECT, UPDATE and REMOVE. When the user presses the Next button, the `next` method is called. This in turn calls the `execute` method of the command corresponding to the current work mode. The command is also added to a stack representing the command history. When the user presses the Prev button and the `prev` method is called, the command at the top of the stack is popped and its `undo` method is called. This will undo the effects of the last performed action. This functionality allows the user to go back and revisit a previous step of the algorithm, for example so that they can try removing a different state.

4.6.3 CreateREScreenController

The `CreateREScreenController` is the controller for the view that is used to create regular expressions. Compared to the controllers for finite automata, this controller is very simple. The `parseRegexString` method is called when the user clicks the Parse button. The regex string entered by the user is parsed by the identically named method of the `Parser` class. If the regex string is invalid, the error message thrown by the method is displayed to the user. Otherwise, the `ParseTree` for the regular expression is built and displayed. This also activates the Convert button, which allows the user to switch the view to the screen for converting regular expressions into a finite automata.

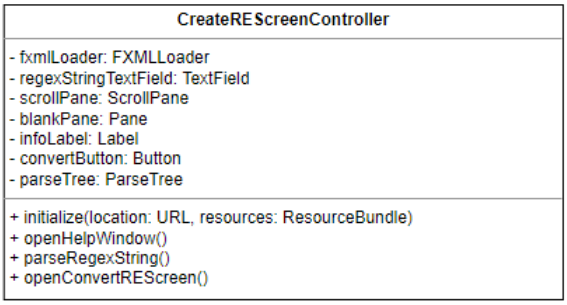


Figure 4.17: The `CreateREScreenController` is the controller for the view that is used to create regular expressions.

4.6.4 ConvertREScreenController

The `ConvertREScreenController` is the controller for the view that is used to convert regular expressions into finite automata. The process is set off by the `setParseTree` method, which is used to pass the parse tree (and by extension the regular expression) created in the previous screen to this controller. The `setParseTree` method gets the postorder traversal of the parse tree and of the regular expression by calling their respective `postorderTraversal` methods.

It then highlights the first parse tree node and displays the finite automaton for the first regular expression. The finite automaton is created by passing the regular expression to the `buildFiniteAutomaton` method of the `GraphicalFiniteAutomatonBuilder` class. The various labels on the screen are also updated.

The `prev` and `next` methods work in a similar manner: they highlight the previous/next parse tree node and display the finite automaton for the previous/next regular expression. This allows the user to step forwards and backwards through the different steps of the McNaughton-Yamada-Thompson algorithm.

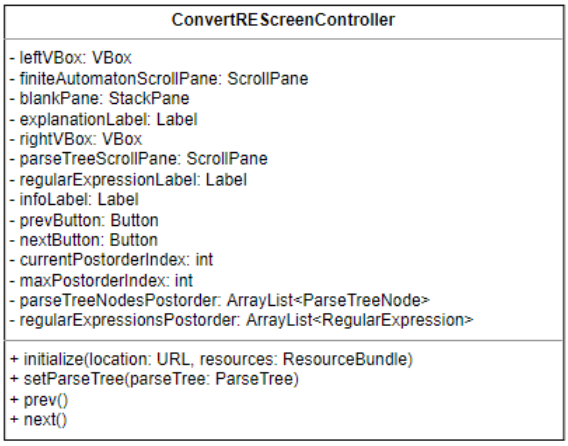


Figure 4.18: The `ConvertREScreenController` is the controller for the view that is used to convert regular expressions into finite automata.

4.7 Settings

During the implementation of the program, various decisions had to be made about its looks and to a lesser extent its functionality. Many of these decisions were not technical in nature but just personal preference; for example, the radius of a state in a finite automaton. To avoid the problem of magic numbers and to allow these decisions to be easily changed, each decision was stored in its own variable. However, it would be desirable to allow the user to change those decisions to suit their own preferences and needs. While this is not possible in the current iteration of the software, in order to future-proof the code the `Settings` class and its subclasses were created. These classes contain all the customisation variables. In the future, these variables could be modified from being `public static final` variables to being `private static` variables. Getter and setter methods could then be added to allow the user to modify them. These methods have already been created for the symbols used as regex operators.

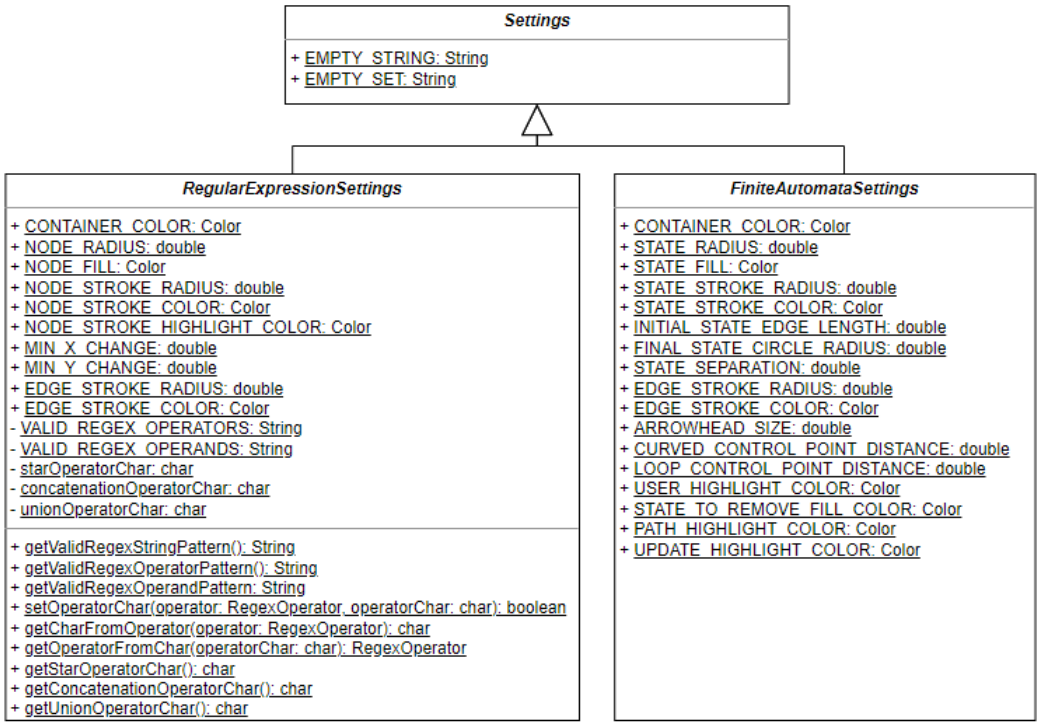


Figure 4.19: The `Settings` classes contain customisation variables.

4.8 Testing

To ensure that the program functions correctly, a range of automated tests were written by using the JUnit testing framework. These tests are comprised of both positive and negative unit tests and focus on the backend logic of the program as opposed to the GUI. This allowed for regression testing to be carried out every time the code was changed, to ensure that the functionality of the program was not affected. Over 100 test cases were written.

4.9 Documentation

To allow other developers to work on and make use of the code base, extensive documentation has been written. In addition to the usual code comments, Javadoc comments[4] were written for all classes and methods. These comments describe what the class/method represents and what it is used for, as well as other useful information. For methods, they also list and explain the parameters and return value of the method, as well as any exceptions thrown. Comments may also contain references to other related comments.

[4] `https://www.oracle.com/uk/technical-resources/articles/java/javadoc-tool.html`

Chapter 5

Evaluation

This chapter will evaluate the software produced by first considering the requirements set out in Chapters 3.1.3 and 3.1.4, and then by comparing it to other existing software from Chapter 2.7. Throughout the rest of this chapter, the software produced will be referred to as FAREC (Finite Automata and Regular Expression Converter).

5.1 Evaluation against requirements

FAREC satisfies all the 'must have' functional requirements that were laid out at the beginning of the project. Users are able to create finite automata and convert them into equivalent regular expressions, with each step of the process being shown. Users can create regular expressions with the union, concatenation and Kleene star operators, and then transform them into equivalent finite automata. For each step of this process, the finite automaton produced is displayed to the user. The 'must have' non-functional requirements have also been met, with FAREC being able to run on Windows, returning correct results and responding promptly to user input.

Many of the 'should have' and 'could have' requirements have also been implemented. During the State Elimination algorithm for converting finite automata into regular expressions, the user is able to select which state to remove. While creating regular expressions, the user can see the parse tree for the regular expression they have created. For both conversion processes, in addition to moving forwards through the process, FAREC also allows the user to move backwards.

Some additional 'quality-of-life' features have also been implemented:

- During the creation of a finite automaton, the user may rename some or all of the states. If the names of any states are left blank, the program will automatically rename them. The new names will include a number that indicates how close the state is to the initial state.

- The user is able to change the position of loop edges to be either above or below the state they are connected to.

- Regular expressions displayed to the user are simplified to remove unnecessary brackets. This improves their readability.

- The user can open a help windows with information on how to create finite automata or regular expressions.

Unfortunately, some of the features that were initially planned have not been implemented due to time-constraints:

- FAREC does not support saving and loading finite automata or regular expressions. For regular expressions, an easy workaround is available; since regular expressions are strings, the user can simply copy-and-paste them to and from a text file. No workaround is available for finite automata.

- A settings window was planned to allow the user to customise the software to match their preferences. One of the things that the user would have been able to change is the symbol associated with each regex operator. Setters and getters for these symbols were written but the setting window itself was never implemented.

- The option to show $\emptyset$-transitions during the State Elimination process was not implemented.

- Zoom functionality for finite automata and parse trees was not implemented.

- The production version of FAREC does not currently run on Linux or macOS. The production version of FAREC was created using a tool called jlink[1], which creates a custom Java runtime image that contains all the modules and dependencies needed to run the

[1] `https://docs.oracle.com/javase/9/tools/jlink.htm`

program, as well as the program itself. This allows the user to run the program without having to do any prior setup. Unfortunately, although it appears to be possible, I was unable to get the Java runtime image to run on any operating systems other than Windows. A different approach would have been to package the application into a .jar file. This approach might have resulted in a program that worked on different operating systems, but it would also require the user to download and setup both Java and JavaFX.

5.2 Evaluation against existing software

This section will compare and contrast FAREC to both JFLAP and Seshat. For more information about these two software, see Chapter 2.7.

5.2.1 Evaluation against JFLAP

FAREC improves on JFLAP in several ways. In JFLAP, there is no way for a user to go back to a previous step of the conversion process. The user would have to restart the whole process from scratch. Meanwhile FAREC allows the user to move backwards and forwards through the algorithms. The user is able to return to any previous step of the process, including starting the whole process from the beginning. Since FAREC is supposed to be an educational tool, this functionality is crucial; someone who is learning these algorithms for the first time will often want to see how a particular step is done several times, which is only possible if they can undo the step. When converting finite automata into regular expressions this functionality becomes particularly important, as it allows the user to go back and select different states to remove in the State Elimination algorithm.

Another advantage that FAREC has over JFLAP is that when converting regular expressions into finite automata, the parse tree for the user's regular expression is displayed. The finite automata are then built by following a postorder traversal of this parse tree. This structured approach makes it much easier to follow the steps of the McNaughton-Yamada-Thompson algorithm, and the parse tree makes it easy to see how the finite automata for the different regular expressions are combined. FAREC also provides a short text explaining each step of the process. The finite automata produced from regular expressions by FAREC are smaller than the ones produced by JFLAP. In addition, they are positioned in a more organised manner - see Figure 5.1.

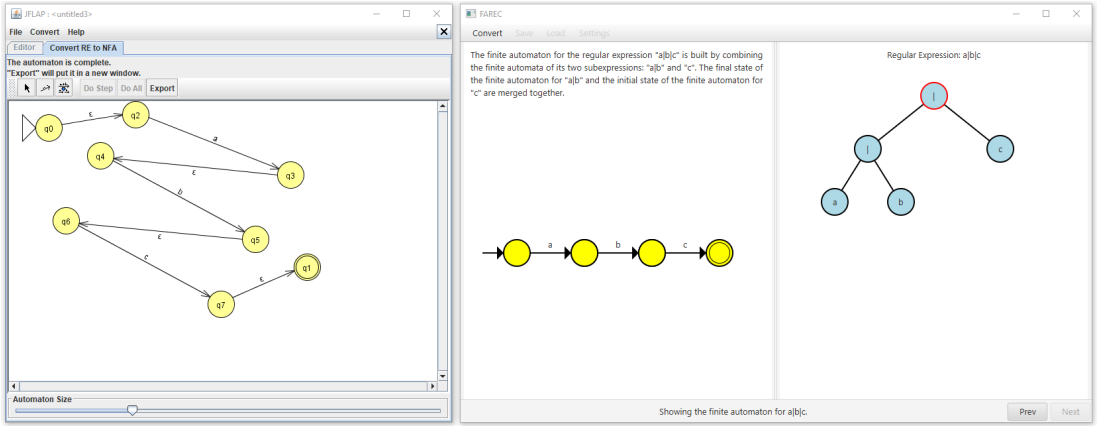


Figure 5.1: The finite automaton produced by JFLAP (left) and FAREC (right) for the regular expression a|b|c.

When converting finite automata into regular expressions, JFLAP shows every single transition in the GNFA, including $\emptyset$-transitions. There is no option to hide these transitions. FAREC on the other hand does not show $\emptyset$-transitions. This results in GNFA that are significantly less cluttered - see Figure 5.2. When updating the labels on edges, JFLAP considers all edges at once, including edges that will not be changed by the update. Meanwhile FAREC will update the edges one at a time and will only consider the edge being updated, as well as other edges involved in the update. FAREC highlights these edges and explains how the new label is constructed. This approach helps the user to understand what is going on.

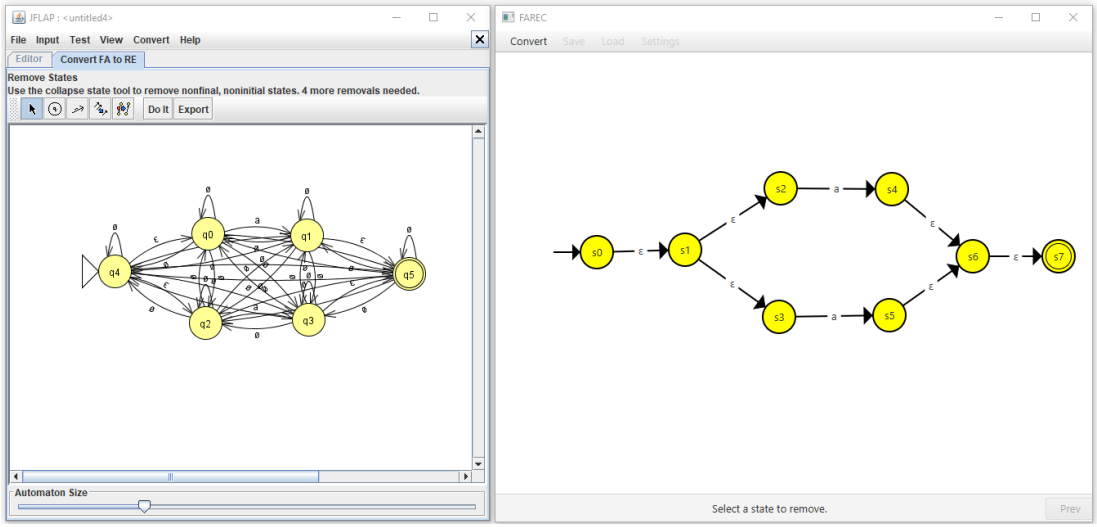


Figure 5.2: Equivalent GNFA in JFLAP (left) and FAREC (right).

One issue with JFLAP is its unintuitive UI. JFLAP uses an interactive process when converting between finite automata and regular expressions. However, it is not always clear what is expected of the user. The UI for FAREC is simpler and better at guiding the user. An

information label is displayed at the bottom of the screen. This label will show instructions and error messages to the user. These instructions and error messages are more informative than what JFLAP has to offer. For example, if the user creates an invalid regular expression, JFLAP will display an error window saying "Operators are poorly formatted"; FAREC on the other hand will identify what the problem is and display the incorrect part of the regular expression.

There are several areas where JFLAP does outperform FAREC:

- JFLAP has save and load functionality.
- JFLAP has undo and redo buttons during the finite automaton creation process.
- In the finite automaton creation process, JFLAP allows the user to drag edges to change their shape/curvature.
- JFLAP does not require an explicit concatenation symbol: if there is no regex operator between two letters from the alphabet Σ, JFLAP will recognise that there is an implicit concatenation between them.
- JFLAP offers some (rudimentary) customisation options, for example changing the empty string character from ϵ to λ or changing some of the colours of the application.
- JFLAP offers other functionality that FAREC does not, for example creating and running Turing machines.

5.2.2 Evaluation against Seshat

The biggest advantage that FAREC has over Seshat is that Seshat is not capable of converting finite automata into regular expressions. On the other hand, Seshat does have other functionality that FAREC does not, for example automaton minimization. In many other ways, Seshat is quite similar to FAREC. It uses the McNaughton-Yamada-Thompson algorithm to convert regular expressions into finite automata and so the finite automata it produces are structured the same way as the finite automata produced by FAREC. Although it doesn't display a parse tree for the given regular expression to the user, it still constructs the finite automata in postorder. Whereas FAREC has a small text explaining each step, Seshat has a brief explanation of the algorithm at the top of the page and then shows which inductive rule is applied to each

step. Just like FAREC, Seshat allows the user to move backwards and forwards through the algorithm (unlike FAREC, it also has buttons for skipping to the start and end).

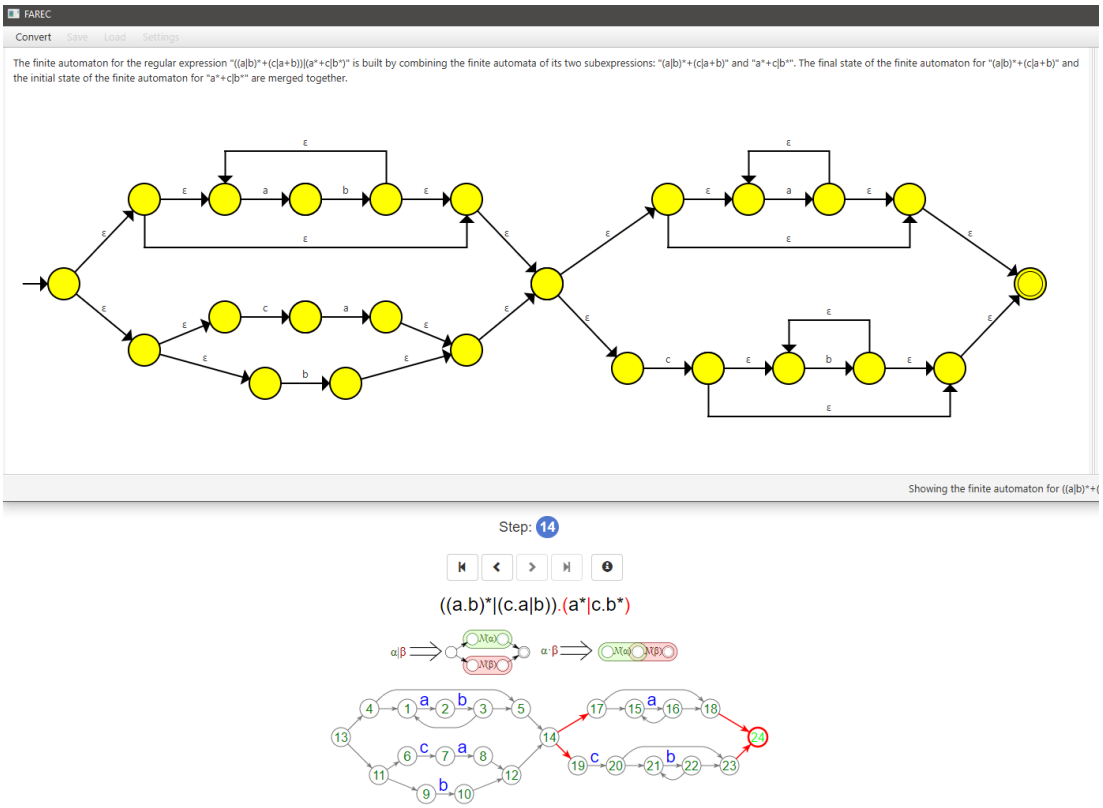


Figure 5.3: The finite automaton produced by FAREC (top) and Seshat (bottom) for the regular expression ((a|b)*+(c|a+b))|(a*+c|b*)

Despite not being able to convert finite automata into regular expressions, Seshat does offer some improvements over FAREC:

- Seshat shows all the parts of the finite automaton created so far and uses highlighting to show which parts are currently being constructed.
- When converting a regular expression into a finite automaton, Seshat numbers the states of the automaton in the order that they were created.
- Seshat is web application and so is much more accessible.

Chapter 6

Legal, Social, Ethical and Professional Issues

This chapter will discuss the legal, social, ethical and professional issues related to this project.

During the development of the software, special care was taken to follow best practises in the field and to adhere to the British Computer Society (BCS) Code of Conduct[1]. The BCS Code of Conduct defines rules and standards that all of its members must follow.

The purpose of this project was to create an educational tool to demonstrate the algorithms for converting between finite automata and regular expressions. It would be very difficult, if not impossible, for such a tool to be misused for malicious purposes, or to cause damage or harm to others. It is not a privacy or security risk to the user or anyone else, nor does it pose a risk to the environment. The work produced was entirely my own unless explicitly stated otherwise. Wherever the work of others was used, it was attributed and credited appropriately, and any software licensing was respected.

[1] https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/

Chapter 7

Conclusion and Future Work

This chapter will conclude the project by summarising what has been achieved and what may still be achieved in the future.

7.1 Conclusion

The aim of this project was to create an educational tool to demonstrate the algorithms for converting between finite automata and regular expressions. The target audience was teachers and students, who could use the tool to aid in the teaching and learning of these algorithms. As such, it was important that the tool shows each step of the algorithms used. To convert finite automata into regular expressions, the State Elimination method with GNFAs was chosen. To convert regular expressions into finite automata, the McNaughton-Yamada-Thompson algorithm was used. These are popular algorithms that can be easily displayed visually and so are well suited to the purpose of the tool.

The aim of this project was satisfied through the creation of FAREC: Finite Automata and Regular Expression Converter. FAREC is capable of converting between finite automata and regular expressions in both directions. Not only does it show each step of the process, it allows the user to move both backwards and forwards through the algorithms. A great deal of work has gone into making FAREC visually appealing and simple. This is particularly important as some of its users may not be familiar with regular expressions or finite automata, and so it is crucial that the program is intuitive and easy to use.

7.2 Future Work

Although FAREC has achieved the goal of this project, there are still many areas in which it could be improved (see Chapter 5 for details on how it compares to what was originally planned as well as other existing software). The most important functionality missing from FAREC is the ability to save and load finite automata and regular expressions. This functionality should be easy to implement for regular expressions: since they are just strings, they can simply be written to and read from a text file. An alternative approach would be to create a mapping between the identifier of a regular expression and the expression itself (for example through the use of a `HashMap`). This mapping could then be saved to a file through the use of serialization. Unfortunately, saving finite automata is not as easy, as JavaFX does not support serialization of components. In order to save finite automata, the program would need to save every detail about the finite automaton and its components. Then, when the finite automaton is loaded, the program would recreate the same finite automaton from scratch, using the information saved.

Another major area that needs work is compatibility with different operating systems. As discussed in Chapter 5.1, FAREC is packaged as a Java runtime image, which includes the program itself as well as all dependencies needed to run it. However, I was unable to get this runtime image to run on any operating systems other than Windows. A different approach would be to deploy the software as a .jar file, which should work on different operating systems but would also require dependencies such as Java and JavaFX to be installed and setup correctly. An ideal solution would combine the benefits of both of these approaches.

Other ways that FAREC can be improved:

- Smart finite automata are currently more restrictive than they need to be (see Chapter 4.5.1). Smart finite automata cannot have the initial state and the final state be the same. In addition, they cannot have multiple final states. These restrictions are necessary for GNFAs and so need to hold once the conversion process has started. However, they could be removed from the finite automaton creation process and then re-enforced later on, when the finite automaton is turned into a GNFA.

- Parse trees for regular expressions are currently built in a top-down manner. For most regular expressions, this results in a significant amount of empty space in the parse tree. This empty space increases exponentially with the depth of the tree. Parse trees should be built bottom-up instead. See Chapter 4.2.1 for more details.

- A settings window where the user can customise the program was planned but never implemented. In particular, it would be useful to allow the user to change the symbols associated with the different regex operators. See Chapter 5.1 for more information.

- Finite automata and regular expression parse trees can become quite big. Scroll bars will appear if the content cannot fit inside the available space, but it would be useful for the user to see everything at once. This could be achieved by implementing zoom functionality, so that the user can zoom in and out of finite automata and parse trees.

- During finite automaton creation, users may make mistakes such as deleting a state or renaming an edge. An undo button (and potentially a redo button) to revert changes would improve the user's experience.

- Double-clicking a state or edge should allow the user to rename it.

- During finite automata creation, dragging a state out of the bounds of the window should not only expand the size of the workspace (as it currently does), but should also move the scroll bars along with the state, so that it remains visible.

- Creating an edge between two states should have some visual indicator that edge creation is taking place. One possible indicator would be a faint/semi-transparent arrow from the start state to the cursor.

- An Edit button should be added to the screen for converting finite automata into regular expressions. Pressing this button would return the user to the finite automaton creation screen, but it would keep the constructed finite automaton.

- A short text could be added to the screen for converting finite automata into regular expressions explaining how the current edge label is updated (similar to the text found when converting regular expressions into finite automata that explains how the finite automata are built).

- When converting regular expressions into finite automata, highlighting could be added to the finite automaton to show which parts have been added in the current step. The previous finite automata used in the construction could also be highlighted.

- Non-empty-string labels should be more visible than empty-string labels. For example, non-empty-string labels could be bigger or bold, or empty-string labels could be grey

instead of black.

- Most text in the program is displayed through the use of `Label` objects. However, this text cannot be selected or copied. The program should be changed to allow text to be selected and copied.

- It might be possible to remove graphical finite automata and simply use smart finite automata for everything, without losing the benefits that graphical finite automata have.

- In the `SmartState` class, the methods for adding and removing incoming and outgoing edges could be merged together. The edge itself could be checked when being added or removed to see if it is an incoming or outgoing edge.

- Since all smart components have a container, shape and label, the variables for these could be moved into the `SmartComponent` class from its subclasses.

References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, & Tools* (2nd ed.). Pearson Education.

Arnaiz-González, Á., Díez-Pastor, J.-F., Ramos-Pérez, I., & García-Osorio, C. (2018). Seshat—a web-based educational resource for teaching the most common algorithms of lexical analysis". *Computer Applications in Engineering Education*. doi: 10.1002/cae.22036

Bigelow, J., & Poremba, A. (2014). Achilles' Ear? Inferior Human Short-Term and Recognition Memory in the Auditory Modality. *PLoS ONE*, *9*(2). doi: 10.1371/journal.pone.0089914

Rozenberg, G., & Salomaa, A. (Eds.). (1997). *Handbook of formal languages, vol. 1: Word, language, grammar*. Springer-Verlag.

Sipser, M. (2013). *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.