FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Finite automata editor on touch devices
Student:	Marek Fořt
Supervisor:	Ing. Jan Trávníček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2021/2022

Instructions

Study the web interface of the algorithms library[1] and the algorithms library itself[2] with focus on design and drawing of finite automata.Study the possibilities of strokes detection on touch devices. Study the approaches of shapes detection with focus on those used in the automata drawing on iOS platform.
Implement a prototype finite automata editor capable of finite automata recognition from strokes. Perform usability tests of the prototype.

–

[1] Michael, Vrána. Knihovna algoritmů ALT-webové rozhraní. BS thesis. České vysoké učení technické v Praze. Vypočetní a informační centrum., 2020.[2] Algorithms Library Toolkit: https://alt.fit.cvut.cz.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 26 January 2021 in Prague.

Bachelor's Thesis

FINITE AUTOMATA EDITOR ON TOUCH DEVICES

Marek Fořt

Czech Technical University, Faculty of Information Technology
Department of Theoretical Informatics
Supervisor: Ing. Jiří Trávníček, Ph.D.
May 11, 2021

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Marek Fořt. *Finite Automata Editor on Touch Devices*. Bachelor's Thesis. Czech Technical University, Faculty of Information Technology, 2021.

Contents

List of Figures

List of code snippets

Acknowledgment

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act.

In Prague on 13th May 2021 .....

Abstract

This thesis is concerned with designing and implementing a prototype of a finite automata editor for iPad and a subsequent usability testing of the prototype. It also contains analysis of the Algorithms Library Toolkit library's web interface, of the library itself, and of approaches to recognizing strokes on touch devices.

Keywords iPad application, finite automata, interactive editor, Algorithms Library Toolkit, Composable Architecture, Swift

Abstrakt

Tato bakalářská práce se zaobírá návrhem a implementací prototypu editoru konečných automatů na iPad zařízení a následném uživatelském testováním na vytvořeném prototypu. Také obsahuje analýzu webového rozhraní knihovny Algorithms Library Toolkit, této knihovny samotné a přístupů rozeznávání tahů na dotykových zařízení.

Klíčová slova iPad aplikace, konečné automaty, interaktivní editor, Algorithms Library Toolkit, Composable Architecture, Swift

Chapter 1
Introduction

1.1 Motivation, Focus of Thesis

The theory of finite automata is an important part of the computer science curriculum at FIT CTU in Prague and other universities around the world. And although there is a lot of resources one can learn from, there is a lack of those that utilize modern tools. One of such modern tools is the iPad (and touch devices in general). This thesis will fill in this gap as the result will be a finite automata editor application for iPad.

Furthermore, I will expand on the recent work done at FIT CTU in Prague concerning the development of algorithms library and, more importantly for this thesis, finite automata algorithms including simulating input. This library is named Algorithms Library Toolkit (ALT) [1] and it has been open sourced.

The main motivation of this thesis is to improve how students learn finite automata and more specifically, enhance the current course BI-AAG that is taught at FIT CTU in Prague. It is also an opportunity to use the ALT in practice and create a concrete example of how it can be leveraged.

1.2 Thesis Goals

The main goal of this thesis is to implement a prototype of an automata editor for iPad. This application should enable users to create and edit finite automata with emphasis on touch-based input. I will also study the ALT's web interface [2] [3], ALT itself focusing on design and drawing of finite automata, the possibilities of strokes detection on touch devices, and the approaches of shape detection, especially those used in automata drawing on iOS platform.

After the initial study of current approaches and theory, I will implement a prototype of a finite automata editor iPad app that will be capable of recognizing finite automaton elements from strokes and simulating input.

I will then conduct a usability testing to assess the usability and shortcomings of the prototype.

1.3 Thesis Structure

Let me now introduce you to the structure of the rest of the thesis:

- In **Chapter 2** I will go over the necessary theoretical background to properly explain terms and concepts on which it will be built upon later.

- **Chapter 3** is concerned with the analysis of already existing solutions of creating an automata editor, the existing ALT web interface and ALT itself.
- **Chapter 4** is about the design of the editor itself.
- In **Chapter 5** I will write about the implementation.
- **Chapter 6** will go into the specifics of usability testing and its outcomes.
- **Conclusion** is the last chapter of this thesis where I will assess the success of fulfilling the aforementioned goals and lay out possible future development.

Chapter 2
Theory

Firstly, I will need to define terms and formal definitions concerning mainly finite automata theory, as that is the main subject of this thesis, and then machine learning as some of its concepts were important during the implementation.

2.1 Formal Languages and Grammars

The following definitions are taken from Automata and Grammars by Eliška Šestáková [4], Introduction to Automata Theory, Languages, and Computation [5], and materials from BIE-AAG course [6].

2.1.1 Formal Languages

▶ **Definition 2.1.** *Alphabet (conventionally denoted by Σ) is a finite set whose elements are called symbols.*

Alphabets therefore can be:

- $\Sigma = \{0, 1\}$
- $\Sigma = \{a, b, c, d, e\}$
- $\Sigma = \{one, two\}$

▶ **Definition 2.2.** *String (word) over an alphabet is a finite sequence of symbols from that alphabet.*

- ϵ - *empty string* (string with zero occurences of symbols)
- Σ^* - set of all strings over Σ
- Σ^+ - set of all nonempty strings over Σ

For a binary alphabet $\Sigma = \{0, 1\}$ ϵ, 1001, 100, 1, 001 are all strings over the alphabet Σ.

▶ **Definition 2.3.** *Formal language L over an alphabet Σ is any subset of all the strings over Σ - i.e., $L \subseteq \Sigma$*

For a binary alphabet $\Sigma = \{0, 1\}$ a formal language over Σ is then subsets of *all* binary strings. We can denote the language either by:

- enumeration notation where all possible strings in the language are listed, e.g.: $L_1 = \{\epsilon\}, L_2 = \{1\}, L_3 = \{0, 00, 000, 01\}$.
- set-builder notation where the languages are described in the following way: { w | something about w }. Examples are: $L_4 = \{w \mid w \in 0, 1^* \wedge |w| \mod 2 = 0\}$, $L_5 = \{0^n 1^n : n \in \mathbb{N}_0\}$.

2.1.2 Grammar

Grammars are used to generate languages. You can find how they are defined below:

▶ **Definition 2.4.** *Grammar is a quadruple of $G = (N, \Sigma, P, S)$ where:*

- N is a finite non-empty set of nonterminal symbols.
- Σ is a finite set of terminal symbols ($\Sigma \cap N = \emptyset$). Note that $N \cap \Sigma = \emptyset$.
- P is a finite set of *production rules*, assuming the following form:

$$\alpha A \beta \to \gamma \ (\alpha, \beta, \gamma \in (N \cup \Sigma)^*)$$

The following is an example of a grammar that describes the language $L = \{01^n 0 : n \in \mathbb{N}_0\}$: Grammar $G = (\{A, S\}, \{0, 1\}, P, S)$ where P:

- $S \to 0A$
- $A \to 1A$
- $A \to 0$

2.1.3 Chomsky Classification of Grammars

Grammars are divided into four classes where they differ in their production rules.

▶ **Definition 2.5.** *Let $G = (N, \Sigma, P, S)$. We say that G is:*

1. *Unrestricted grammar* (type 0), if every rule is in the form of:
$$\alpha A \beta \to \gamma \ (\alpha, \beta, \gamma \in (N \cup \Sigma)^*, A \in N)$$
2. *Context-sensitive* (type 1), if every rule is in the form of:
$$\gamma A \delta \to \gamma \alpha \delta \ (\gamma, \delta \in (N \cup \Sigma)^*, A \in N, \alpha \in (N \cup \Sigma)^+)$$
or in the form of $S \to \epsilon$ if S is not present on the right hand side of any rule of a given grammar.
3. *Context-free grammar* (type 2) if every rule is in the form of:
$$A \to \alpha \ (A \in N, \alpha \in (N \cup \Sigma)^*)$$
4. *Regular grammar* (type 3), if every rule is in the form of:
$$A \to a \text{ or } A \to aB \ (a \in \Sigma, A, B \in N)$$
or in the form of $S \to \epsilon$ if S is not present on the right hand side of any rule of a given grammar.

2.1.4 Classification of Languages

Classification of languages, also known as the Chomsky hierarchy, has the following definition:

▶ **Definition 2.6.** *We say that language is:*

1. *formal* if it is a formal language but is neither regular, context-free, context-sensitive, nor recursive enumerable. These languages are not accepted by a Turing machine.
2. *recursively enumerable* if and only if ∃ unrestricted grammar which generates it
   - accepted by a Turing machine
3. *context-sensitive* if and only if ∃ context-sensitive grammar which generates it
   - recognized by a linear bounded Turing machine
4. *context-free* if and only if ∃ context-free grammar which generates it
   - recognized by a nondeterministic pushdown automaton
5. *regular* if and only if ∃ regular grammar that generates it
   - recognized by a finite automaton

For most of the thesis, only regular languages will be needed since those are recognized by finite automata. Finite automata will be defined in the following section.

2.2 Finite Automata

The final editor app will be for finite automata, therefore they are very important for this thesis. Informally, a finite automaton is a model for simple computation. States, that serve as memory, and transitions together form a *control unit*. Along with a control unit, the finite automaton has a *read-only input tape*, which is divided into individual cells, and the *head* that scans the input tape as the automaton continuously reads it, cell by cell. Automaton starts in its initial state and with a head pointing at the first cell. As the input is read, the head moves until it has read all of the input tape. If there is a missing transition for an input, the automaton does not accept the input. Otherwise, it accepts the input if it is in a final state at the end of the input.

Let's define a finite automaton formally:

▶ **Definition 2.7.** *Finite automaton is a quintuple* $M = (Q, \Sigma, \delta, q_0, F)$ *where:*

- Q *is a finite non-empty set of states*
- Σ *is a finite input alphabet*
- δ *is the transition function (the exact definition is determined by which type of a finite automaton it is - see below)*
- $q_0 \in Q$ *is the initial state*
- $F \subseteq Q$ *is the set of initial states*

Finite automaton can also be either a *deterministic* finite automaton or a *nondeterministic* finite automaton. This dictates the exact definition of δ - transition function. For a deterministic finite automaton (DFA) the definition of δ is:

$$\delta \text{ is a mapping from } Q \times \Sigma \text{ to } Q$$

δ for a nondeterministic finite automaton (NFA) is defined as:

$$\delta \text{ is a mappping from } Q \times \Sigma \text{ into the set of all subsets } Q \text{ (denoted by } 2^Q)$$

Expanding upon the difference between the definitions of the transition function:

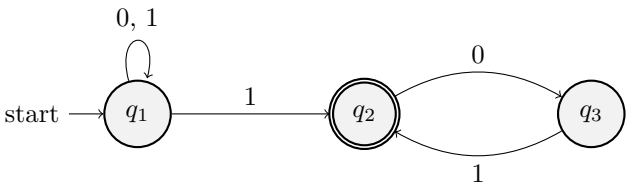


Figure 2.1 FA graph representation

δ_{NFA}	0	1
$\rightarrow q_1$	q_1	q_1, q_2
$\leftarrow q_2$	q_3	
q_3		q_2

Figure 2.2 FA table representation

- DFA can only transition from one state to another, e.g. from q_2 to q_3 (q_2, $q_3 \in Q$)
- NFA can transition to a set of states, e.g. from q_1 to q_1, q_2 (q_1, $q_2 \in Q$)

If we change the definition of NFA's δ to a mapping from $Q \times (\Sigma \cup \{\epsilon\})$ we allow, what are called, ϵ-transitions that allow us to move to a different state while not reading any input from the tape. This finite automaton is then called a *nondeterministic finite automaton with ϵ-transitions*.

2.2.1 Representation of Finite Automata

Finite automata's transition functions δ are generally represented in the form of:

- *Formal notation*
  (NFA) $\delta(S, 0) = \{S, A\}$ (transition from the state S and symbol 0 to the states S and A)
  (DFA) $\delta(A, 0) = B$ (transition from the state A and symbol 0 to a single state, not a set of states, B)
- *Weighted directed graph* (state diagram)
  Automata can be represented graphically as directed weighted graphs. Each state is represented as a vertice in the graph and final states are recognized by being a double circle, instead of a single one. Initial state is the one with an arrow that points to the state but does not start anywhere - it is often additionaly denoted with *start*. The transitions are then directed edges between states. You can see a FA represented as a weighted directed graph in 2.1.
- *Table*
  Table representation has in the first column all states where the initial state is marked with $\rightarrow$ while final states are marked with $\leftarrow$. In the first row, excluding the first column, there are symbols of the alphabet, Σ. In the rest of the rows are states (or a set of states) that will be transitioned to on a given input (defined in the first row). You can see an example of it in 2.2.

In this thesis we will mostly be working with the representation in the form of a weighted directed graph as that is what will the users edit in the app. This also concludes the theory about finite automata and formal languages.

2.3 Machine Learning

Machine learning does not have an exact definition but e.g. in the book Foundations of Machine Learning it's loosely defined as "computational methods using experience to improve performance or to make accurate predictions" [7]. *Experience* means something we know from the past that we can leverage for making predictions in the future. Usually, this experience comes in the form of data. The book Foundations of Machine Learning [7] and materials from BIE-VZD from FIT CTU in Prague [8] will be used further in this section to define terms and concepts necessary for this thesis.

2.3.1 Classification

Machine learning, in order to cluster problems that can be solved in a similar way, can be categorized into a few learning scenarios, most notably supervised and unsupervised learning. The learning scenario is a basic description of what type of data we have, how we receive the data and the test data that we use to evaluate the learning algorithm.

- *supervised learning*: Our goal is to explain *variable* Y given *independent variables* X_0, X_1, ..., X_{p-1}. We do this by finding a "function" for which most of its examples the following holds:
$$Y \approx f(X_0, X_1, ..., X_{p-1})$$
- *unsupervised learning*: Our goal is to find structures of "similar" data. We do not predict any class and there is no clear way to assess the quality of an unsupervised learning algorithm since it is not clearly defined what the end result should be.

In this thesis we will be only interested in supervised learning. We can also divide common problems that machine learning is trying to solve by learning tasks - that includes classification, regression, ranking, clustering, etc. Let's look more closely at classification which will be later used in the implementation:

- Classification is a problem of assigning a category to each item.

It is also a problem solved via supervised learning. To expand on the definition of supervised learning from above, classification is a special case where Y has only a few (countable amount) of values. The simplest example of classification is *binary classification*. E.g. we want to predict whether a patient has flu and our data - gender of a patient, whether a person can leave the bed, etc. - can be represented in a binary format (yes/no).

Chapter 3
Analysis

In the analysis I will study the following:

- existing applications that enable users to edit finite automata, including ALT web interface
- ALT itself, focusing on design and drawing of finite automata
- possibilities of detection of strokes on touch devices.

3.1 Existing Applications

This section will be concerned with the study of existing applications - be it applications for mobile or web.

3.1.1 ALT Web Interface

ALT web interface has been built as a part of a bachelor's thesis made by Michael Vrána [3] leveraging work already done in ALT itself. ALT web interface uses Pipe-and-Filter [9] architecture to easily combine input and outputs of the individual algorithms that ALT offers which can be seen in figure 3.1. Apart from ALT algorithms it also includes a finite automata editor done by Petr Svoboda [2].

This finite automata editor is called Statemaker and you can see a screenshot of how it looks in figure 3.2. To summarize its capabilities, users can:

- Add states.
- Add transitions between states.
- Edit transition string.
- Mark states as initial or final.
- Remove states and transitions.
- Import and export automaton in supported formats.
- Automatic positioning of transitions and states.

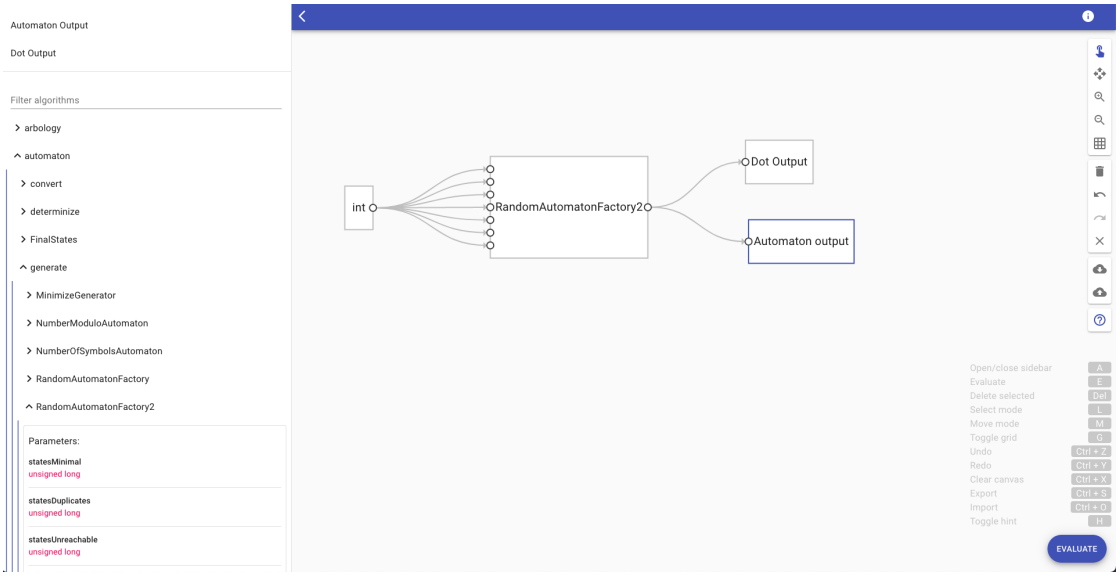


Figure 3.1 ALT web interface screenshot

All of the above features work reliably and are done in an intuitive manner - a user can quickly understand how to work with all the components. The sidebar includes all buttons to interact with the interface and icons used make clear the intent of the buttons. The dragging of FA elements works well with a mouse or a trackpad but it is not possible to use it on a touch device.

The most notable missing feature is the easy simulation of input - this can be done via the ALT web interface but if someone is looking for only editing FAs and simulating whether the input string is accepted, they have to transition between two interfaces. The benefit is that they can then tap into all the other functionality that ALT offers. The author of Statemaker has chosen React and Typescript as underlying technologies [2].

3.1.2 Other Existing Applications

As the main goal of this thesis is to write a finite automata editor for iPad, in this part I will study existing applications mainly for touch devices.

One of such applications is TuringSim [10]. Although it is not for FAs but for a Turing machine, it also consists of an editor where the user can add and edit states and transitions, thus making it similar to a FA editor. You can see its interface in figure 3.3. This editor lets users add and edit automaton's states and transitions. Users can also simulate input on the Turing machine's read-and-write tape. Editing of the automaton is done only via tap gestures which is similar to Statemaker with the difference that there are no distinct buttons for those actions. Therefore, it does not fully utilize the potential of touch devices as the UX is very similar to what would one experience on the web. The app on the iPad is broken at the time of writing as it is missing a bottom toolbar for simulating input but it is functional on the iPhone.

Finite Automata [11] is another iPad app. In this app the user cannot edit automata in their weighted graph representation but instead has to use a command line that takes individual commands which are described in the app. This app does not utilize touch device features at all.

There are also apps available as desktop applications. One is the Finite Automaton Editor by Jaime Rangel-Mondragon that is available as an interactive Wolfram notebook [12]. This app allows you to edit the automaton via a transition table and does not allow to simulate any input. Automata Editor by Max Shawabkeh [13] is another desktop application. In it users can create and edit their automaton either via a table representation or a regular expression. There are

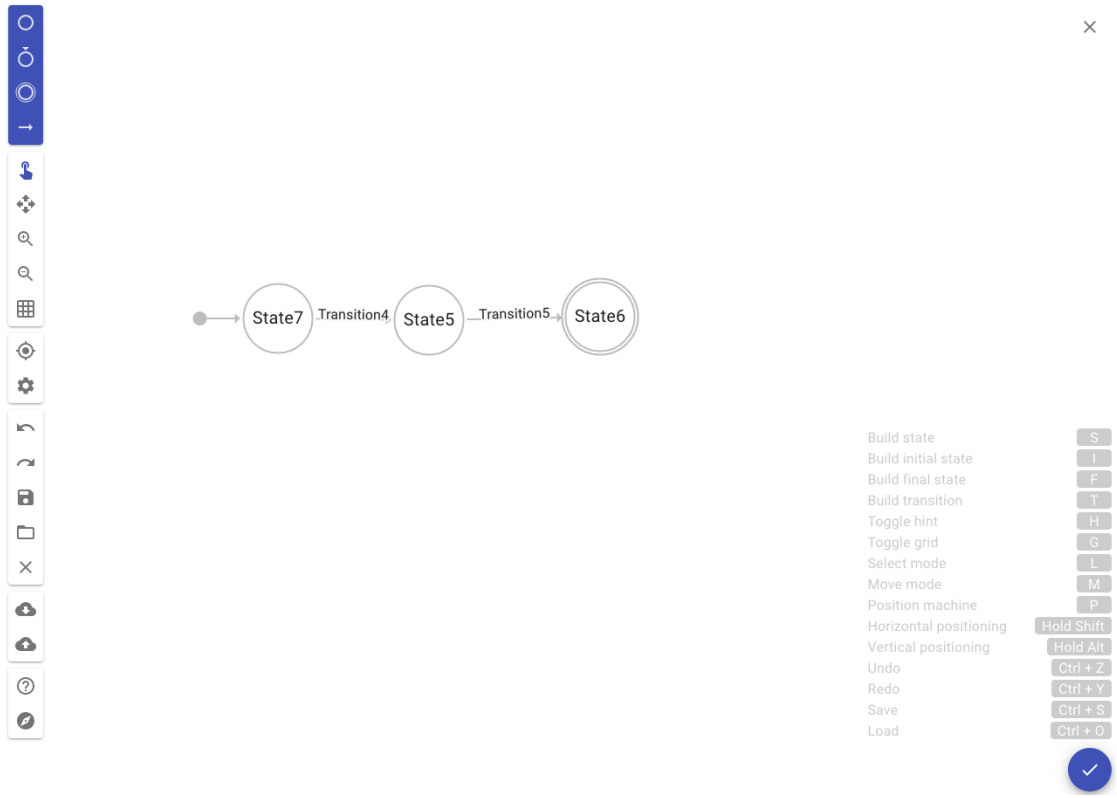


Figure 3.2 Statemaker screenshot

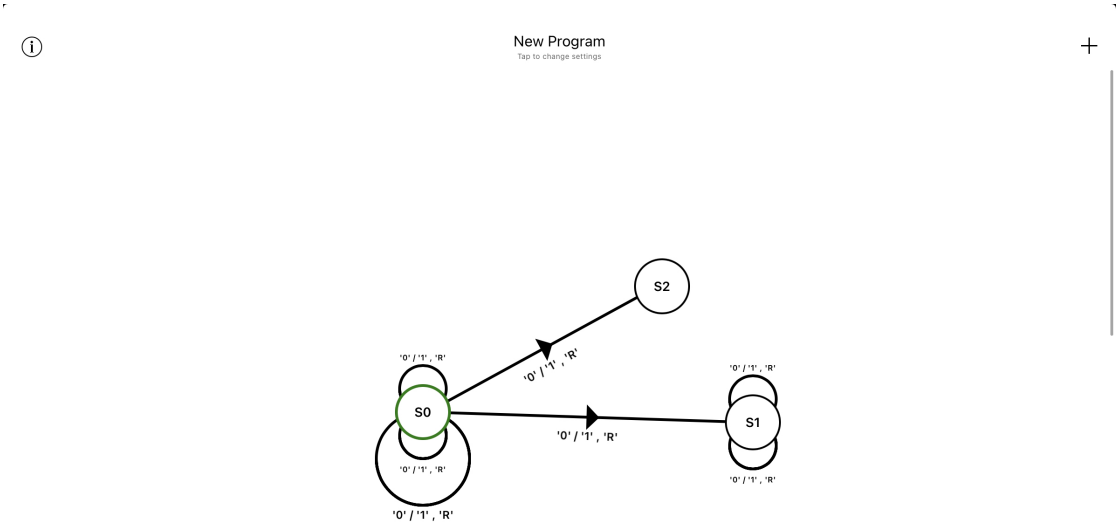


Figure 3.3 TuringSim interface screenshot

also features such as NFA determinization, simulating input, and minimizing DFA. Thus, it has a powerful feature set but one has to be already familiar with the FA theory to be able to utilize it fully. It should also be noted that the feature set is a subset of what the ALT web interface offers.

3.2 ALT

I will now go over ALT and its features that can be leveraged for simulating FA input. The code of ALT is available on GitLab [14] where there are multiple repositories in the group Algorithms Library Toolkit (webui-client, infrastructure, etc.) - that includes a repository Algorithms Library Toolkit Core [15], a library written in C++ [16]. There we can find algorithms that can later be used for the FA editor. The code is divided into multiple modules that are then built and linked together using CMake [17]. The most important modules for this thesis are *alib2data* and *alib2algo* where *alib2data* contains FA models and *alib2algo* algorithms for simulating input.

3.2.1 FA Model

Multiple FA types are supported by ALT - which includes a deterministic and a nondeterministic finite automaton, as well as a nondeterministic finite automaton with ϵ-transitions. There is also an extended NFA that has regular expressions as its transitions. These models serve as a description of an automaton - its states, transitions, etc. To create e.g. the NFA with ϵ-transitions, one can use its constructor where it is possible to specify its states and input alphabet. For adding transitions there is a method called `addTransition`. Both can be seen in figure 3.1.

Code snippet 3.1 EpsilonNFA example methods

```
// Creates a new instance of the Automaton
// with a concrete initial state.
explicit EpsilonNFA (
    ext::set < StateType > states,
    ext::set < SymbolType > inputAlphabet,
    StateType initialState,
    ext::set < StateType > finalStates
);
// Add a transition to the automaton.
bool addTransition (
    StateType from,
    common::symbol_or_epsilon < SymbolType > input,
    StateType to
);
```

3.2.2 FA Algorithms

ALT offers a multitude of algorithms that can be run on finite automata - such as minimization, determinization, and simulating input. Simulating input can be found in `Accept.h`, `Result.h`, `Occurrences.h`, and `Run.h` that contain classes of the same name. `Accept` has a method called `accept` that returns a simple boolean indicating whether the input has been accepted or rejected. `Result`'s `result` method returns a state where the simulation ended and `Occurrences`'s `occurences` method returns a set of indices where the automaton has passed a final state. `Run`'s returned value is then a combination of `Accept` , `Result` and `Occurences` where it contains the combined output of all those classes. The `Run`'s function `calculateStates` can be seen in 3.2.

Code snippet 3.2 Run's calcuteStates function

```
template < class SymbolType, class StateType >
static ext::tuple <
    bool,
    ext::set < StateType >,
    ext::set < unsigned >
> calculateStates (
    const automaton::EpsilonNFA <
        SymbolType,
        StateType
    > & automaton,
    const string::LinearString < SymbolType > & string
);
```

3.3 Strokes Recognition

The final prototype will include recognizing automaton elements from drawing. In this section I will go over available methods of how to achieve it.

3.3.1 Google ML Kit

Google offers a framework called ML Kit that includes what they call "Digital Ink Recognition". This lets you construct a stroke from points drawn on the screen and create an `Ink` object from those strokes. It also includes base models for recognizing text and even some basic shapes like

arrow and rectangle. If you want to create your own model with TensorFlow Lite [18], you are forced to use "Image Labeling". Since the editor should support creating cycles, it is necessary to create a custom model because that shape is not supported by any of the base models for Digital Ink Recognition.

3.3.2 Core ML

Apple's CoreML framework supports a variety of use cases - analysis of images, processing text, converting audio to text, and identifying sounds in audio [19]. It does not, however, support anything like Digital Ink Recognition - but for the editor it is suitable to use analysis of images because it is possible to create an image from the screen and pass that to the model. Apple also provides some models already in CoreML format [20] but not any of them are applicable for the FA editor's use case. Therefore, a custom model for CoreML would be necessary as well.

Considering that CoreML is bundled in the system and Google ML Kit needs to be installed separately, increasing the app's size and incurring the maintenance burden, I have opted for CoreML. This decision has also been made based on the fact that both frameworks do support TensorFlow, although, for CoreML it needs to be first converted to its format.

3.3.3 Creating the CoreML Model

There are multiple ways how to create a CoreML model, though, they generally fall into two categories:

- ML model created by ML libraries that are not from Apple such as TensorFlow or Keras [21] and then converted with coremltools [22] to the CoreML format.
- ML model created by framework or application that outputs CoreML directly.

For creating CoreML models directly there is either Create ML [23] or turicreate [24]. Create ML, at the time of writing, supports only image classification, whereas turicreate has built-in support for drawing classification. Although both image classification and drawing classification operate on images, the important distinction is that drawing classification takes 28x28 grayscale bitmap as input. The drawing classification is also tailored for inputs created by the Apple Pencil [25], thus I have chosen to use it instead of libraries such as TensorFlow. It should be noted, though, that turicreate leverages TensorFlow as a lower-level framework and aims to streamline the development of CoreML models.

Chapter 4

Automata Editor Design

In this chapter I will go over some decisions made, such as which technology I have decided to use, and over the design of the editor - how the app will look and how users will interact with the editor.

4.1 Touch Device

The main reason why this prototype is meant for a touch device is to simulate as much as possible the experience of drawing FAs on regular paper. There were three main options possible:

- create a touch-friendly web interface
- implement an app for Android
- target iPad devices

Creating a touch-friendly web interface would have the benefit of being universal and not tied to a specific platform. But native apps offer better precision and developers can tap into OS APIs that are tailored for touch. The choice between Android and iPad was less clear but iPad has the benefit of the Apple Pencil [26] that offers high precision that will make the user experience better.

4.2 Used Technologies

I will now cover which technologies are used in the app and why I have chosen them. A lot of decisions have been influenced by focusing on the iPad and the Apple Pencil.

4.2.1 Language

Choosing a language in which one will write the application is an important first step. For iOS applications I could have generally used:

- Objective-C
- Swift
- cross-platform framework

Objective-C was designed by Brad J. Cox at the start of the 1980s and was then licensed by NeXT Software in 1988. Then in 1996 NeXT Software was acquired by Apple - along with Objective-C. Apple has then chosen Objective-C as the main language for OS X and in 2007 for the new operating system iOS [27]. On the Apple developer website it is described as ”superset of the C programming language and provides object-oriented capabilities and a dynamic runtime” [28]. Objective-C has been thus the main programming language for years. Nowadays, Objective-C is not anymore that popular and ranks 23rd in the language popularity as per TIOBE index [29]. Apple has rather shifted their focus to Swift and some of the new frameworks, like SwiftUI [30], are only available in Swift. Swift is therefore a much better option if one is starting a new iOS app.

Using a cross-platform framework - such as React Native [31] or Flutter [32] that are written in Javascript and Dart, respectively - was also a possibility. But to leverage the Apple Pencil fully it was necessary to use PencilKit [33] and for interacting with that framework one would have to write native code. Thus, I have decided to use Swift as the main language.

4.2.2 UI Framework

At the time of writing it is possible to use two UI frameworks offered by Apple to write UI code. Those are UIKit [34] or, already mentioned, SwiftUI [30]. SwiftUI is a newer framework than UIKit, released in 2019 [35]. In the Thinking in SwiftUI book it is described as a ”radical departure from UIKit, AppKit, and other object-oriented UI frameworks” [36]. SwiftUI offers a more declarative approach, quite similar to React [37] used in the web development. Declarative UIs have the benefit of less code since it enables the framework to do more on behalf of the developer. This comes at a cost of lesser control. Getting back to SwiftUI, specifically, one of its major drawbacks is that not all components, that are written in UIKit, are available in SwiftUI. But there is very strong support for SwiftUI-UIKit interoperability [38] and thus it is always possible to use UIKit when necessary. The fact that SwiftUI offers faster development due to its declarative nature and SwiftUI previews [39] has made it a better candidate than UIKit, especially for a prototype. Therefore, I have decided to use SwiftUI as the main UI framework.

4.2.3 PencilKit

I have already mentioned the PencilKit framework but it is not the only way how to let users draw with either their finger or the Apple Pencil. One can also use CGContext which is a ”Quartz 2D drawing environment” [40]. For drawing in CGContext it is necessary to observe user's touches and draw a path based on those touches. CoreGraphics framework, which CGContext is a part of, does not bundle any component that would handle the drawing for the developer [41]. PencilKit, however, has a component called `PKCanvasView` [42]. By using this component, the PencilKit framework handles for us how to draw strokes made by the user. `PKCanvasViewDelegate` enables listening to changes of the drawing. The `drawing` property of type `PKDrawing` then allows for programmatic modifications of the drawing - which will be used to render the FA elements after they are recognized.

For the purposes of the prototype, PencilKit is a better fit because it offers more by default while still allowing for minor modifications. If some more custom behavior of e.g. rendering strokes was needed, though, it would be necessary to use the CoreGraphics framework. To make a possible migration between those frameworks as seamless as possible, the PencilKit framework should be well-contained.

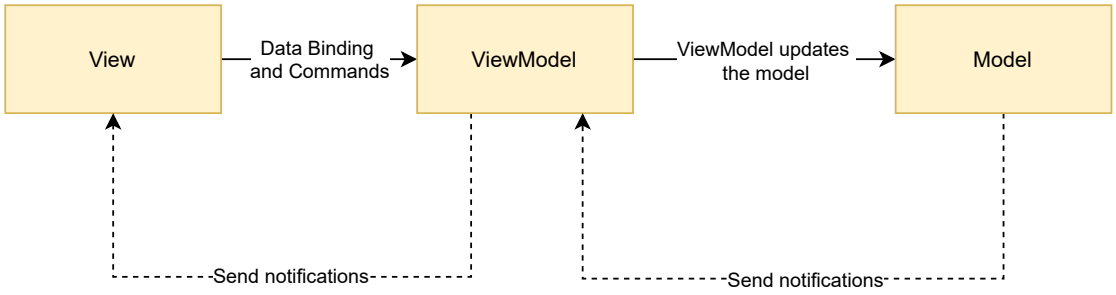


Figure 4.1 MVVM architecture diagram [43]

4.2.4 Architecture

There is no recommended architecture by Apple for apps written on top of SwiftUI frameworks. It is also entirely possible to create an app without adhering to any architecture. This code, though, is then more difficult to maintain for a longer period.

One possible architecture is MVVM [43]. A diagram of this architecture can be seen in figure 4.1. It enables developers to have a clear boundary between UI code and business logic and is a good option for either UIKit or SwiftUI applications. One of its drawbacks is that it can sometimes lead to imperative code where developers call a function and act based on its inputs. The alternatives are (among others) architectures inspired by Redux. A diagram of Redux is depicted in figure 4.2. The main difference between Redux and MVVM is that MVVM is event-driven, whereas Redux is data-driven. The data-driven approach is much more closer to the declarative programming since the state of the application describes how it should look. Thus I have decided that Redux-like architecture will be a better option.

One of concrete implementations of Redux architecture is the Composable Architecture by Point-Free [44]. This architecture is based on Redux but it has some modifications such as handling of side effects. It also makes testing more exhaustive by asserting that no action that you do not expect is run, as well as that the state is not changed in any other way than you describe in your test. Considering all of the above, I have chosen to use the Composable Architecture.

4.3 ALT Integration

I have already talked about ALT in chapter 3. I have not discussed there, however, how ALT will be integrated into the application. That is now possible as I have stated that the app will be written in Swift. As already mentioned, ALT is a library written in C++. There exists a Swift-C++ interoperability manifesto [46] but this manifesto only goes over what it would take to make C++ and Swift interoperable but not even all functionalities of C++ have their discussions of how they could be ported to Swift. However, there is well-supported interoperability between Swift and Objective-C [47]. For Objective-C and C++, interoperability is supported via a language iteration of Objective-C called Objective-C++. It is even e.g. possible to ”include pointers to Objective-C objects as data members of C++ classes” [48]. I first tried to integrate ALT directly and compile it right via Xcode. Since ALT is built via CMake [17] and does not have a simple setup, I then resorted to pre-building the necessary modules and afterwards bundling them in the application. I will go over the details in chapter 5.

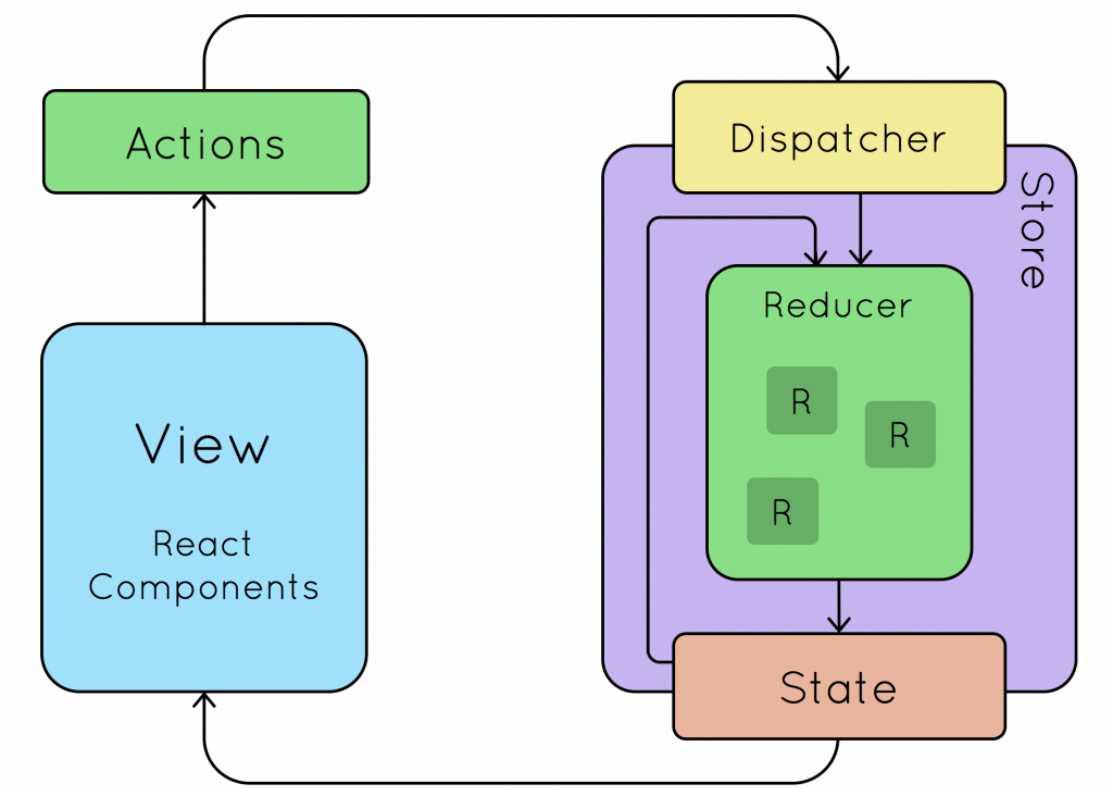


Figure 4.2 Redux architecture diagram [45]

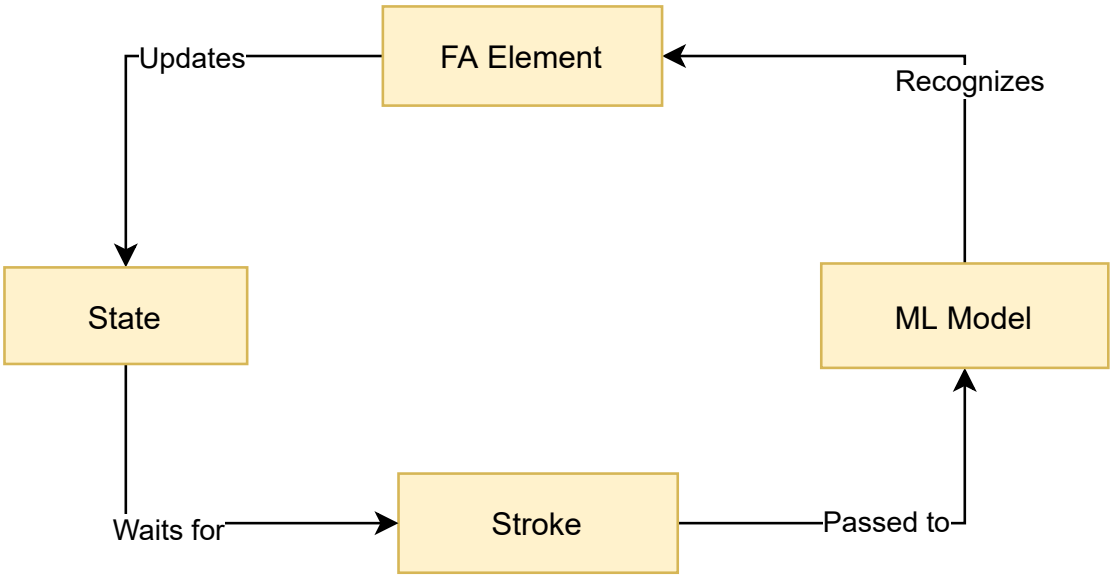


Figure 4.3 Flow of recognizing FA elements from strokes

4.4 User Interface

As a final section of this chapter, I will discuss the design of the UI. The design has been heavily influenced by the fact that one of the main goals was to imitate the experience of drawing FAs on a piece of regular paper. The app should let users to:

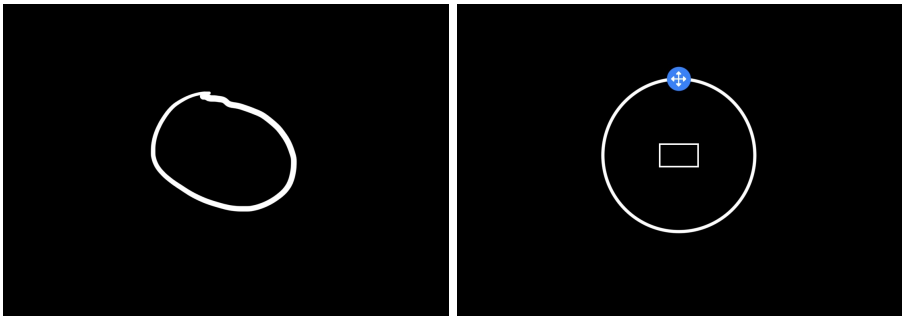
- create FA states, transitions, and cycles.
- delete and rearrange all of the above.
- name states.
- specify symbols for transitions.
- simulate input and see whether the input was or was not accepted by the automaton.

4.4.1 Canvas

Canvas is the most important part of the editor since it is the space where users can draw FA elements. After each stroke, a function will be run that will evaluate the stroke to determine which FA state the user has drawn. The flow of recognizing the FA elements is graphically represented in figure 4.3. The app first waits for the user to make a stroke, after a stroke is made its representation is sent to the ML model which recognizes the type of FA element, a state is updated with the new element and it is drawn on the canvas. Then the app again waits for another stroke.

4.4.2 State

The app needs to be able to work with three FA elements - state, transitions and a cycle (a special case of transition that starts and ends in the same state). These elements should be represented the same way as they are in the weighted directed graph representation. That means a state will be rendered as a circle. But it is necessary to also enable users to edit the name of the state.



(a) Stroke (b) Rendered state

Figure 4.4 Example of a state stroke (a) and how it is rendered (b)

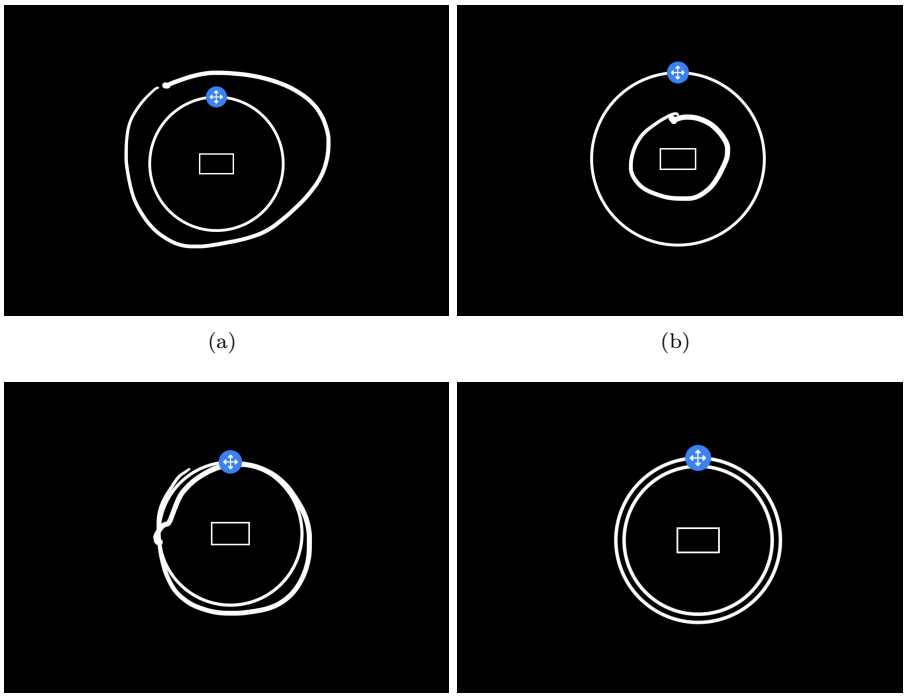
Thus, a text field in the center of the state will be shown. You can see an example of a stroke that should be rendered as a state and how it looks after being recognized in figure 4.4. Notice also a button at the top of the circle - this button is for dragging the state. To indicate that the state is final, users should be able to draw another circle where the stroke contains the center of the state that should be final. In figure 4.5 you should see examples of strokes that should be then rendered as a final state.

4.4.3 Transition

A transition is represented as a directed edge between vertices which has a shape of an arrow. The stroke can either be as an arrow or, more conveniently, just a straight line which is faster and easier to draw, especially when the shape must be drawn with a single stroke. Example strokes and a rendered transition are in figure 4.6. Note that the transition can be drawn in no matter which direction. The transition has also a text field positioned above its middle point. Apart from this text field where users can write symbols that the transition should occur on, there is also a button with a plus icon. This button allows users to add multiple symbols to a single transition. Leveraging a delimiter, such as a comma, was also considered but that could inhibit discoverability. Users can easily remove the transition symbols by tapping a cross symbol beside the symbol. To enable drawing a FA with ϵ-transitions there is also a button with ϵ. When it is tapped, ϵ is added as another symbol for that particular transition. A transition with multiple symbols and with an ϵ-transition are in figure 4.7. Similar to a state, there is a drag button to drag the middle point of a transition. This is especially useful when having multiple states on the same horizontal line with a transition going from the leftmost to the rightmost state.

4.4.4 Cycle

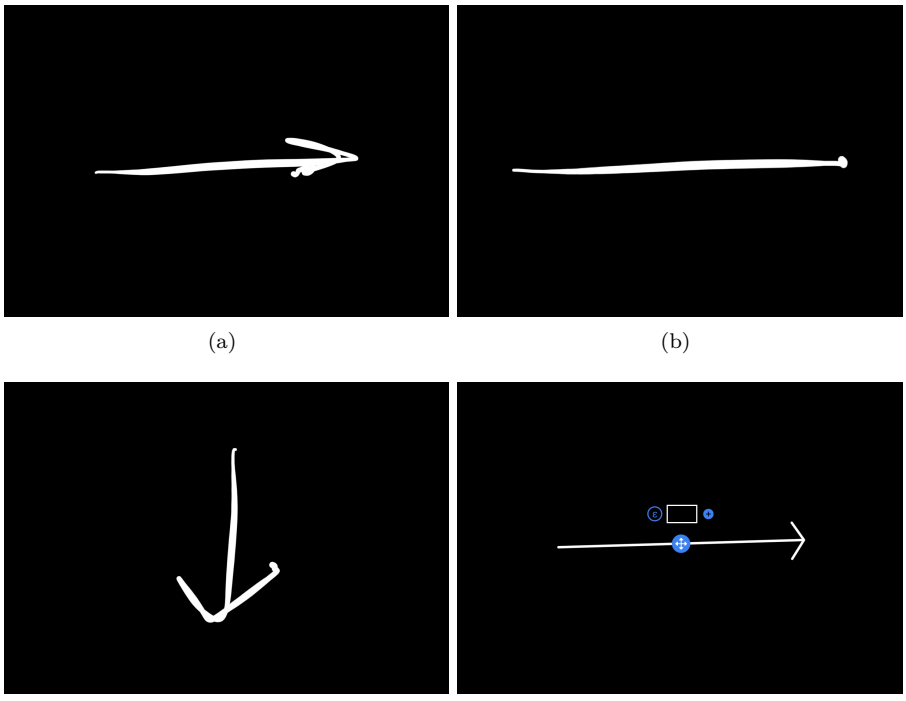
A cycle must be handled differently than a normal transition due to its shape - that means that it must be recognized as a different class in the ML model. The business logic tied to a cycle will not be the same, too. Alike a regular transition, it can be drawn in whichever orientation. The ML model should also support multiple variants of a cycle shape to accommodate most of the cycle strokes drawn by users. It should, however, have the same text field, the button for ϵ-transition, and multiple transition symbols as a regular transition. The possible strokes and how the cycle should then look like are in figure 4.8.



(a) (b)

(c) (d)

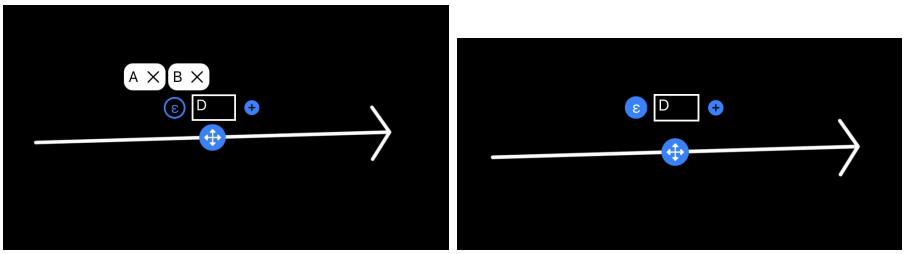
Figure 4.5 (a), (b), (c) are example strokes that should be rendered as a final state (d)



(a) (b)

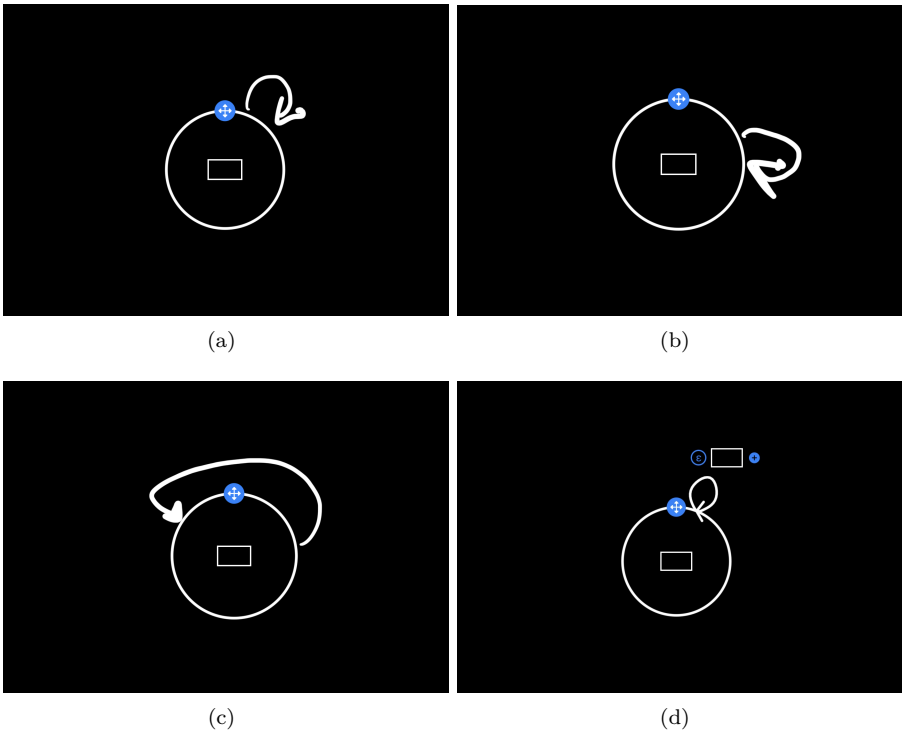
(c) (d)

Figure 4.6 (a), (b), (c) are example strokes that should be rendered as a transition (d)



(a) Transition with multiple symbols (b) ϵ-transition

Figure 4.7 Transitions with symbols



(a) (b)

(c) (d)

Figure 4.8 (a), (b), (c) are example strokes that should be rendered as a a cycle (d)

4.4.5 Connecting Transitions and States

So far, I have only described how transitions and states will be recognized in isolation. To create a FA, though, it is necessary to create valid connections between transitions and states. There are multiple cases that the app should support. Similarly to Statemaker [2], it should allow users to create a transition between two existing states. But because the app strives to imitate the experience of using regular paper, it should be less restrictive and enable creating a transition:

- without any state
- without a start state
- without an end state
- with both start and end states

For transitions that do not have states on both sides, it must be possible to connect a new state to their end. E.g. if a user draws a state close to a transition's end and it has no state there, it should connect it and make the transition an incoming edge of that state. Note that if the user draws a transition without a start state, it is a valid transition as it will be interpreted as the marking of an initial state.

4.4.6 Simulating Input

The only missing functionality that I have not yet touched upon is simulating input. Considering that a FA is drawn, the user should be able to write their desired input into a text field and see whether the input string has or has not been accepted. That can be done via tapping a button with the title "Simulate". If the FA accepts the input, the output is denoted with the checkmark emoji (4.9) and if not, it is denoted with the cross emoji (4.10). For convenience, there is a button to erase the last character.

As the currently edited automaton is saved into a custom model specific to the editor app, it is also necessary to convert it to ALT's model and then run the simulation on the ALT model. Before doing so, checks to ensure the automaton is valid are made - such as whether it has an initial state or if all the symbols of the input are in the FA's alphabet. The alphabet itself is constructed by going through the automaton's transitions and adding their symbols. After the conversion is done, the input is simulated on the ALT's FA model and then it is discarded. It is important to note that the ALT model is always recreated when the user simulates a given input.

4.4.7 Erasing

As you might have noticed in 4.9 and 4.10 there are two buttons right of the input text field. The left is for drawing mode and the right one lets users erase specific elements. If a user made a stroke over an element in the eraser mode, it would be deleted. On the left side, there is a "Clear" button that clears the whole canvas whenever it is tapped.

4.4.8 Text Fields

One thing I have purposefully omitted is how the user will interact with the app's text fields. There are two options:

- using a keyboard
- via Scribble feature

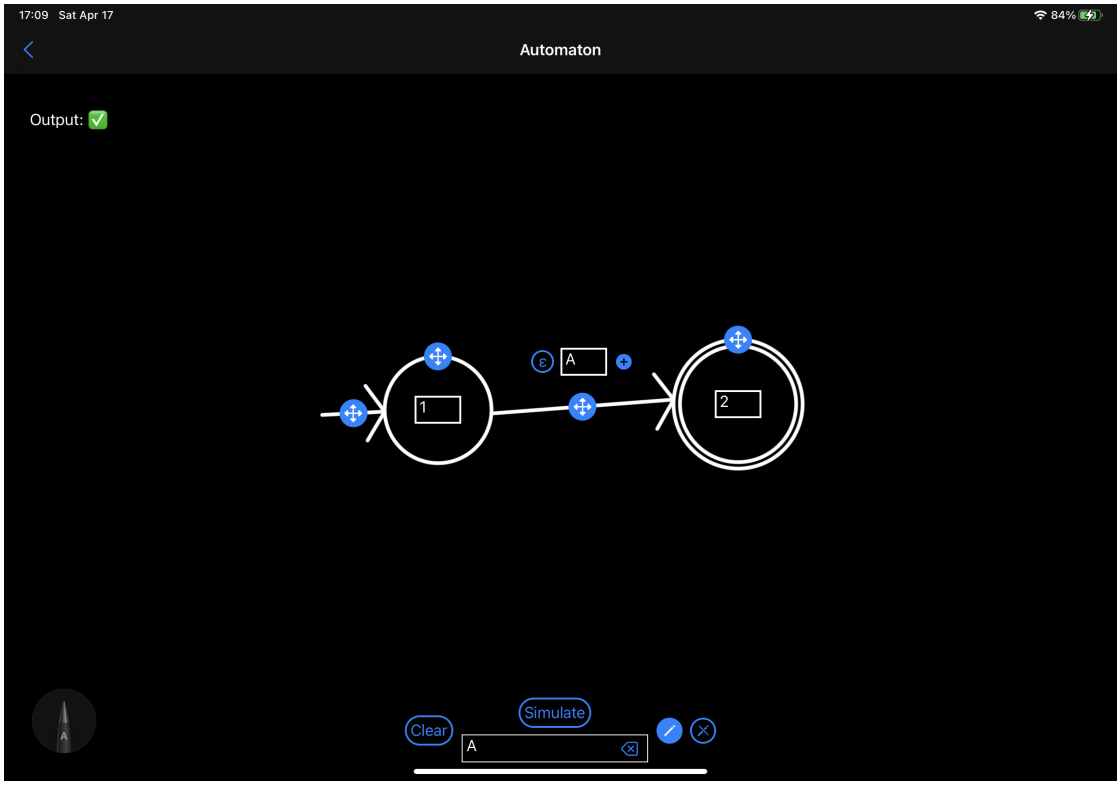


Figure 4.9 Accepted input

Using a keyboard is the standard way of input for text fields. But this breaks the flow of using the Apple Pencil and does not fulfill the goal of imitating the real life experience. However, from iOS 14 it is possible to use the Apple Pencil for writing in text fields directly [49]. This means that it was not necessary to use a custom ML model to achieve the same effect. It does have a drawback and that is the Scribble feature is not available on iOS versions prior to 14.0. It is also only available if the device's language is in English or Traditional and Simplified Chinese. But when the Scribble feature is not supported, users can still use the system keyboard as a fallback.

4.4.9 Document Based Apps

Users should also be able to save their automata and even share them with other people. The state of the editor is thus saved in a document and can be interacted with in what Apple calls document based apps [50]. In document based apps it is also possible to save changes done by the user as they happen, ensuring no progress is ever lost.

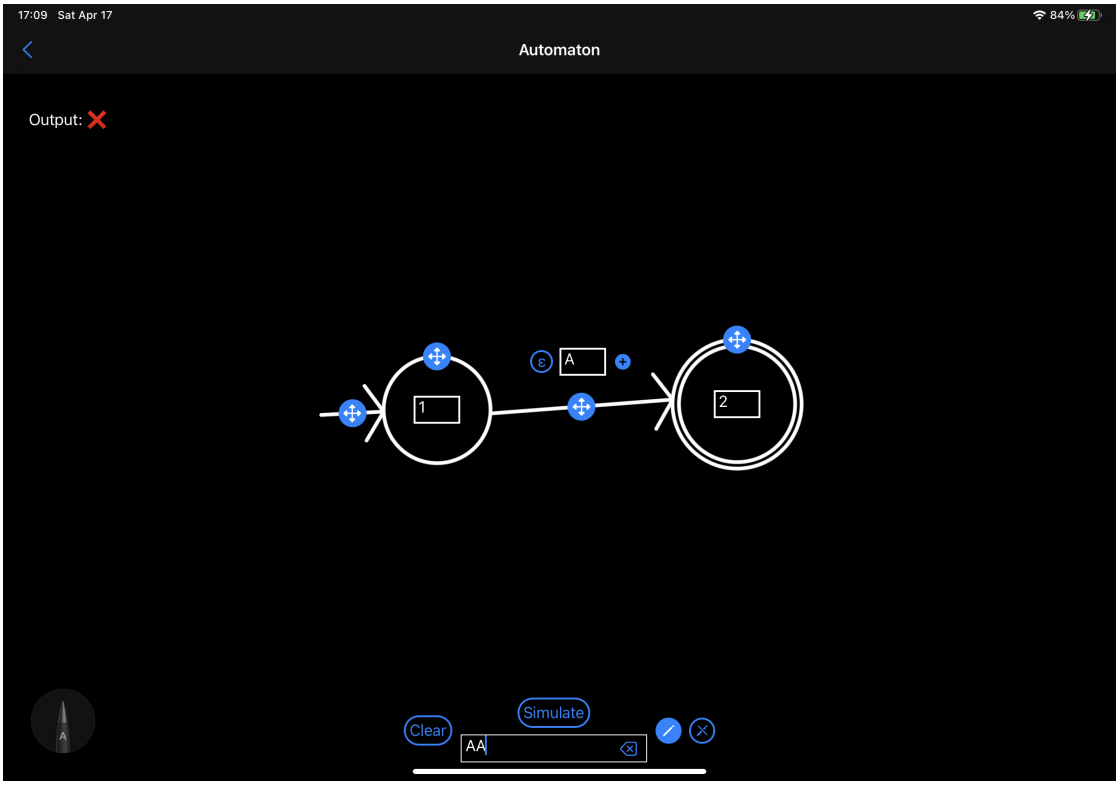


Figure 4.10 Rejected input

Chapter 5
Implementation

I have laid out a basic overview of how the app is designed to look and work. In this chapter, I shall go into details of how the most important parts of the app have been implemented. You can also see the source code either in the thesis' attachments or in the project's repository.

5.1 ML Model

The ML model for recognizing FA elements is a core part of the app. I will go over how it has been implemented and integrated.

5.1.1 Creating ML Model

For creating the ML model I have chosen to use turicreate [24], as discussed in chapter 3. The user guide [25] describes how to create the ML model for drawing classification using QuickDraw dataset [51]. This dataset has millions of drawings of various objects, shapes. The shapes do not include, however, arrow and what would resemble a cycle. Therefore, I had to create a custom dataset. To make the ML model as accurate as possible, I have used the same input method for creating the dataset as will be used for drawing the elements in the prototype. For this I have created an MNIST Maker app [52]. This app lets you draw shapes with either a finger or the Apple Pencil, scales it down to the desired size (I have used 28x28), and converts it to grayscale to make the image as small as possible and it is also what is used when creating the ML Model.

Once I have created a dataset, I could train the ML model. I have used the guide from turicreate and made some modifications like working with grayscale images exclusively (originally, the guide works with RGB images) and specifying which classes to work with. To make iterating on the model easy and quick, I have used a Jupyter notebook [53]. The final Jupyter notebook with the data used is available on Github. Once the ML model is trained, one can use turicreate's visualization to see how the ML model worked on the test dataset. In figure 5.1 you can see the visualization. From the results it can be seen that the model did not make a mistake in recognizing an arrow but it had mistaken a cycle for a circle and vice versa. This is due to the fact that circles and cycles do have a similar shape.

The final product, once all the commands in the Jupyter notebook are run, is a file *AutomataClassifier.mlmodel*. The extension hints at the fact that it is a ML model that can be used by the CoreML framework.

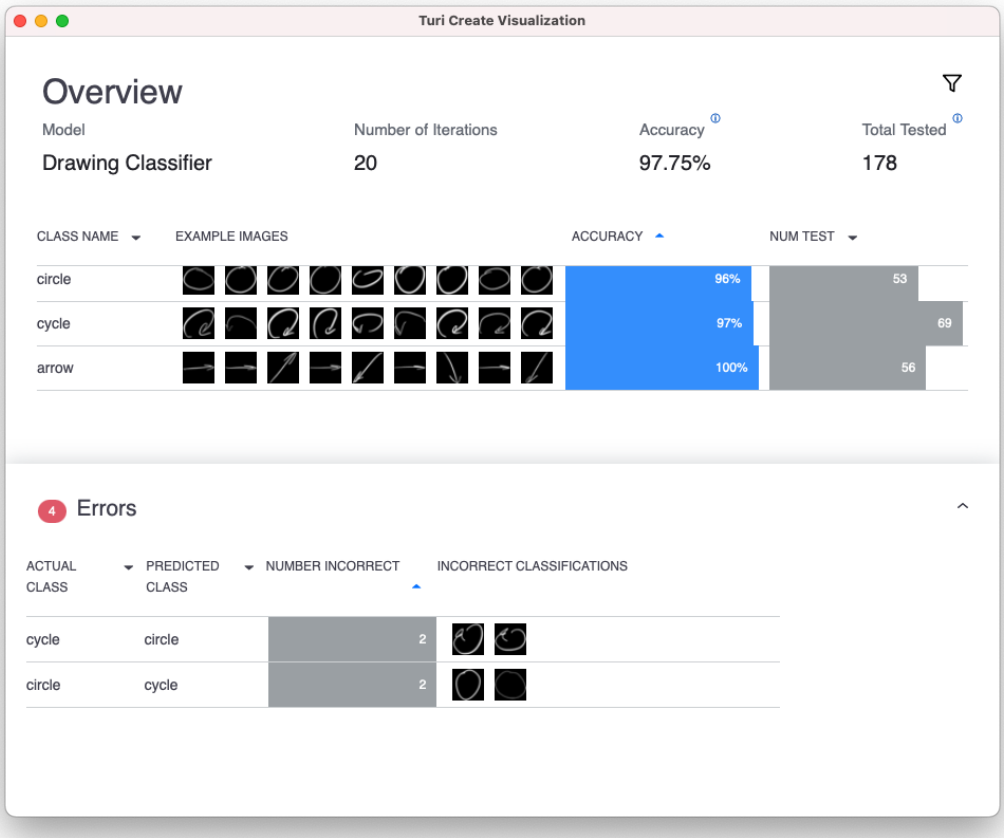


Figure 5.1 Test dataset ML model prediction results in turicreate

5.1.2 Integration of the ML Model

To integrate the ML model one has to simply drag and drop the file into Xcode [54], Apple's IDE, to create a reference in the project file. This will automatically bundle the model with the app and synthesize Swift code that can be later leveraged to interact with the model. The code for recognizing a stroke is located in `AutomataClassifierService` and the main logic is in the code snippet 5.1.

Code snippet 5.1 Automata classifier

```
// Convert stroke to cgImage
// ...
let input = try AutomataClassifierInput(
    drawingWith: cgImage
)
let classifier = try AutomataClassifier(
    configuration: MLModelConfiguration()
)
let prediction = try classifier.prediction(input: input)

guard
```

```
    let automataShapeType = AutomatonShapeType(
        rawValue: prediction.label
    )
else { return promise(.failure(.shapeNotRecognized)) }

switch automataShapeType {
case .arrow:
    promise(.success(.transition(stroke)))
case .circle:
    promise(.success(.state(stroke)))
case .cycle:
    promise(.success(.transitionCycle(stroke)))
}
```

5.2 Drawing FA Elements

Before FA elements can be recognized, they also must be drawn by the user, converted to a model that can be processed by the `AutomataClassifierService` and then re-drawn with a more exact shape once it is known which element the user has made.

5.2.1 Canvas

To enable drawing with the Apple Pencil, Apple offers PencilKit [33]. One of the components this framework defines is `PKCanvasView` - in the documentation described as a "view that captures Apple Pencil input and displays the rendered results in an iOS app" [42]. To use it in SwiftUI, `View` I had to wrap it into a view I called `CanvasView` conforming to `UIViewRepresentable` since `PKCanvasView` is only available in UIKit. The instance of `CanvasView` was then added to `EditorView` which is the main view of the app.

5.2.2 Strokes

When drawing in `PKCanvasView` the individual strokes are represented with `PKStroke`. It contains all of the available information about the given stroke - such as its bounds, type of ink (`PKInt`), and path (`PKStrokePath`). In order not to use PencilKit directly and also to make it easier to work with, I wrapped `PKStroke` into a custom model `Stroke` that contains control points (array of `CGPoint`) of `PKStroke`. I have not used `PKStrokePath` because it contains infinite amount of points. To get only a subset of them one can use a method `interpolatedPoints(by: CGFloat)`. The most recent `PKStroke` is then passed to the service called `AutomataClassifierService` as an image. The image is created from `PKStroke` as can be seen in 5.2 where `modelImage()` method simply converts the image from `PKDrawing` to grayscale and the desired size.

Code snippet 5.2 Automata classifier

```
let image = PKDrawing(strokes: [stroke.pkStroke()])
.image(
    from: stroke.pkStroke().renderBounds,
    scale: 1.0
)
.modelImage()
```

5.2.3 Drawing State

If the ML classifier predicts that the stroke being analyzed is a state, it is necessary to draw it more precisely than a user. A state is represented as a circle, so to draw it the app needs to know its center and radius. To obtain a center, I make an average of all the points available. Once the center is computed, radius is then calculated as an average of distances of all the points from the center. A new stroke can then be created as shown in 5.3.

It is also necessary to connect a state to an existing transition. There is a method for this called `closestTransitionWithoutEndState` that, as the name suggests, finds the closest transition that does not have an end state. If such a transition exists, a center is moved in the direction of the tip of the transition, so they have a single intersection point.

A final state is marked with a double circle. Therefore, if a stroke is a state, before making a new one I check whether a center of a different state is contained in a frame of the new state. If it is so, then the state is marked as final and a new circle around the previous one is made and the state is denoted in the internal state as a final one.

Code snippet 5.3 Circle stroke

```
extension Array where Element == CGPoint {
    static func circle(
        center: CGPoint,
        radius: CGFloat
    ) -> Self {
        stride(from: CGFloat(0), to: 362, by: 2).map { index in
            let radians = index * CGFloat.pi / 180

            return CGPoint(
                x: CGFloat(center.x + radius * cos(radians)),
                y: CGFloat(center.y + radius * sin(radians))
            )
        }
    }
}
```

5.2.4 Transition

To draw a transition, I take its first and last point (tip point). At the tip point two additional lines must be made, so the final shape looks like an arrow. For this I needed vectors, so it works for whichever orientation. Vectors are not offered by any of the bundled frameworks, so I have heavily inspired my implementation from a repository called VectorMath [55]. Once vectors were available, I was able to compute a perpendicular vector to the line from the start point to the tip point that has one common point with that line and that is a point on the transition's line in a pre-defined distance from the tip point. The bottom and top points are then on that vector with the pre-defined distance from the line as well. To add additional points, top and bottom vectors are created by connecting top and bottom points with the tip point as in 5.4.

Similarly to states, it is necessary to handle the case when a new transition should be connected to an existing state. Firstly, the closest states to the start point and the tip point are computed. Then the closer one is chosen as a start state or end state, respectively - if the distance is smaller than a given threshold.

Code snippet 5.4 Computation of top and bottom points, vectors

```
let vector = Vector(flexPoint ?? startPoint, tipPoint)
let anchorPoint = vector.point(
```

```
    distance: -arrowSpan / 3,
    other: tipPoint
)
let perpendicularVector = vector.rotated(by: .pi / 2)
let topPoint = perpendicularVector.point(
    distance: -arrowSpan / 2,
    other: anchorPoint
)
let bottomPoint = perpendicularVector.point(
    distance: arrowSpan / 2,
    other: anchorPoint
)
let topVector = Vector(tipPoint, topPoint)
let bottomVector = Vector(tipPoint, bottomPoint)
```

5.2.5 Cycle

The last class that the ML model classifies and that has not been discussed how to draw is a cycle. The cycle is different from a state and a transition since it can not exist without being connected to a state. If there is no state to connect it to, the stroke is just erased. Otherwise, once the closest state is found the cycle's shape can be created. Its shape is more complicated than the previous - but it is possible to use PencilKit's behavior that smoothes out a curve based on the points given. Thus, to recreate it I used the point at the intersection with the state and the state's center in 5.5.

Code snippet 5.5 Cycle stroke

```
extension Array where Element == CGPoint {
    static func cycle(
        _ point: CGPoint,
        center: CGPoint
    ) -> Self {
        let vector = Vector(point, center)
        let topPoint = vector.point(distance: -70, other: point)
        let startToTopVector = Vector(point, topPoint)
        let finalPoint = startToTopVector
            .rotated(by: .pi * 0.4)
            .point(distance: 5, other: point)
        return [
            point,
            startToTopVector
                .rotated(by: -.pi / 3)
                .point(distance: 10, other: point),
            startToTopVector
                .rotated(by: -.pi / 4)
                .point(distance: 40, other: point),
            topPoint,
            startToTopVector
                .rotated(by: .pi / 4)
                .point(distance: 40, other: point),
            startToTopVector
                .rotated(by: .pi / 3)
                .point(distance: 10, other: point),
```

```
        ] + .arrow(
            startPoint: finalPoint,
            tipPoint: point,
            arrowSpan: 30
        )
    }
}
```

5.3 ALT Integration

ALT integration has been discussed both in chapter 3 and chapter 4. In this section I will go into the implementation details.

5.3.1 Source Code

In chapter 4 I have already hinted that I had first attempted to integrate the source code directly. Swift Package Manager is a package manager by Apple that is integrated right in Xcode [56]. Therefore, if I was able to use that for building ALT, others could benefit from this and it would make the integration seamless. To create a package via Swift Package Manager one must create `Package.swift` manifest where the products and other targets are defined. It is also possible to define C++ headers and flags. But the headers in the ALT source code point to already built modules by CMake. This means that when I tried to build the code, I got compiler errors about `#include` directive pointing to non-existent headers. This was possible to circumvent by modifying `#include` directives to point to existing locations, instead of pointing to the built modules. But doing so would be a maintenance burden. Thus, I have decided not to integrate the source code directly but rather use only frameworks built with CMake.

5.3.2 Headers

ALT's current version only supports `gcc` compiler. But to build frameworks for iOS it is necessary to use clang [57]. At the time of writing, ALT is not compilable with the standard library in clang since ALT uses some C++20 features that are only implemented in gcc - for example `lexicographical_compare_three_way`, among others. It is possible to point clang to headers from gcc via `-cxx-isystem` flag and `-stdlib=libstdc++` that instruct clang to parse the headers as `libstdc++` instead of `libc++`. But this is not possible to do since Xcode 10 has dropped support for `libstdc++` and all C++ projects must use `libc++` standard library. In the future, it should be feasible to use the current version of ALT with some newer version of clang as they add support for C++20 features. ALT has only recently started using C++20 features, so to fix this issue, a new branch `libc++-17` was created by Jan Trávníček from a commit from before C++20 features have been introduced to the codebase.

5.3.3 CMake

There is support for building for iOS platform in CMake directly. To save some upfront work, I have used ios-cmake toolchain file that builds on top of CMake and makes the setup easier [58]. An example of how to build a framework for `OS64` (iOS) is in 5.6. The same could then be done for a simulator by specifying `SIMULATOR64`. This way two different frameworks are built. For combining multiple frameworks with different architectures there is XCFramework [59]. I have created a small shell script in 5.7 that creates a new XCFramework from frameworks necessary for the editor app. The built XCFrameworks could then be moved to Xcode.

Code snippet 5.6 CMake build instructions

```
mkdir build
cd build
cmake .. -G Xcode \
-DCMAKE_TOOLCHAIN_FILE=../../ios.toolchain.cmake -DPLATFORM=OS64
cmake --build . --config Debug
```

Code snippet 5.7 XCFramework shell script

```
#!/bin/sh
modules=(
    "alib2std" \
    "alib2measure" \
    "alib2abstraction" \
    "alib2common" \
    "alib2xml" \
    "alib2str" \
    "alib2data" \
    "alib2algo" \
)
for module in "${modules[@]}"
do
    xcodebuild -create-xcframework \
    -framework ../build/$module/Debug-iphoneos/$module.framework \
    -framework \
    ../simulator/$module/Debug-iphonesimulator/$module.framework \
    -framework \
    ../arm64/$module/Debug-iphonesimulator/$module.framework \
    -output $module.xcframework
done
```

5.3.4 Objective-C and Swift Wrappers

With XCFrameworks in Xcode, the next step was to write wrappers, so the C++ frameworks could be used in Swift code. As mentioned in chapter 4, it is not possible to interact with C++ code from Swift directly - for this Objective-C++ wrappers had to be written, exposing only Objective-C in its interface. To better encapsulate the wrappers, I have also created a separate module `SwiftAutomataLibrary`. The most important part in Objective-C is present in `NFA_objc` that interacts with ALT's `EpsilonNFA` class. Its interface is in 5.8. It is initialized with all necessary data to create the underlying automaton and has one method that takes a list of symbols for input and returns `bool` based on whether the input was accepted or rejected. All necessary conversions are done in the implementation file. E.g. states are passed to `NFA_objc` as `NSArray *` where the individual elements are `NSString *`. This had to be converted to `ext::set<std::string>`.

Code snippet 5.8 NFA_objc interface

```
- (instancetype)init: (NSArray *) states
    inputAlphabet:(NSArray *) inputAlphabet
     initialState:(NSString *) initialState
      finalStates:(NSArray *) finalStates
      transitions: (NSArray *) transitions;
- (bool)simulate: (NSArray *) input;
```

The wrapper between Swift and Objective-C was then more straightforward as conversions between a lot of types - such as `NSArray *` and `Array` - are done automatically. It also would be possible to use `NFA_objc` directly but to have better control I have created `NFA` as a simple Swift struct. This is the only part that was made public in the SwiftAutomataLibrary module.

5.4 App State

I have already explained how the FA elements are recognized, rendered and how the ALT C++ code is bridged to Swift. Now I would like to explain a little bit more the internal state of the app - of which the most substantial part is located in `EditorStore.swift`.

5.4.1 FA Models

There are two FA models - `AutomatonTransition` and `AutomatonState`. Notably, a cycle does not have its separate model but is rather an `AutomatonTransition`. But since it is rendered differently, it must be recognizable from a normal transition - for this there is `enum TransitionType` that has two types - `regular` and `cycle`. `AutomatonTransition` can be seen in 5.9. The `id` property is generated for each element at the moment when it is created, so it is easily distinguishable. The same is done for `AutomatonState`. Apart from what you see in the code snippet, `AutomatonTransition` has a set of computed properties like `stroke`. Computed properties are useful for keeping the state of the model consistent and should be used whenever a property's value can be safely and in a performant way derived from other properties of that model. `AutomatonState` is similar to the transition with some specific properties like `isFinalState` denoting whether a state is final or `center` used to determine a state's center.

Code snippet 5.9 Transition model

```
struct AutomatonTransition: Equatable, Identifiable, Codable {
    enum TransitionType: Equatable, Hashable, Codable {
        case cycle(
            CGPoint,
            center: CGPoint,
            radians: CGFloat
        )
        case regular(
            startPoint: CGPoint,
            tipPoint: CGPoint,
            flexPoint: CGPoint
        )
    }
    let id: String
    var startState: AutomatonState.ID?
    var endState: AutomatonState.ID?
    var currentSymbol: String = ""
    var symbols: [String] = []
    var includesEpsilon: Bool = false
    var type: TransitionType
    var currentFlexPoint: CGPoint? = nil
}
```

Code snippet 5.10 State model

```
struct AutomatonState: Equatable, Identifiable, Codable {
```

```
    let id: String
    var name: String = ""
    var isFinalState: Bool = false
    var center: CGPoint
    let radius: CGFloat
    var currentDragPoint: CGPoint
}
```

5.4.2 Simulating Input

Simulating the automaton's input is the main feature of the prototype. Once a user taps on "Simulate" button, `simulateInput` action is triggered. There some preliminary checks are made - such as whether the FA has an initial state (5.11. If a check failed, the app shows the user the exact reason why it failed to make it easier to correct it. In case all preliminary checks succeed, the action returns `Effect` with the result of the simulation. `Effect`s are a mechanism to interact with asynchronous functions and services in general. The initialization of the `Effect` is in 5.12. You can also see that an `alphabetSymbols` variable is used there. Automaton's alphabet is constructed from the symbols defined for the automaton's transitions. In `AutomataLibraryService`, simulation is run with `NFA` Swift wrapper that was discussed earlier. Only the mapping of `transitions` property needed multiple lines of code because a name of the state should be used instead of `AutomatonState`'s `id` for which some additional logic was needed.

Code snippet 5.11 Check for initial state

```
guard
    let initialState = state.initialStates.first
else {
    state.outputString = "No initial state"
    return .none
}
```

Code snippet 5.12 Effect for simulating input

```
env.automataLibraryService.simulateInput(
        input,
        state.automatonStates,
        initialState,
        state.finalStates,
        alphabetSymbols,
        state.transitions
    )
```

Code snippet 5.13 `NFA` initialization in `AutomataLibrarryService`

```
NFA(
    states: states.map(\.name),
    inputAlphabet: alphabet,
    initialState: initialState.name,
    finalStates: finalStates.map(\.name),
    transitions: transitions
        .compactMap { transition -> Transition? in
            guard
                let startState = states.first(
```

```
                    where: { $0.id == transition.startState }
                ),
                let endState = states.first(
                    where: { $0.id == transition.endState }
                )
            else { return nil }
            return Transition(
                fromState: startState.name,
                toState: endState.name,
                symbols: transition.symbols
                    + (
                        transition.currentSymbol.isEmpty ?
                        [] : [transition.currentSymbol]
                    ),
                isEpsilonIncluded: transition.includesEpsilon
            )
        }
)
.simulate(input: input)
```

5.5 Dragging

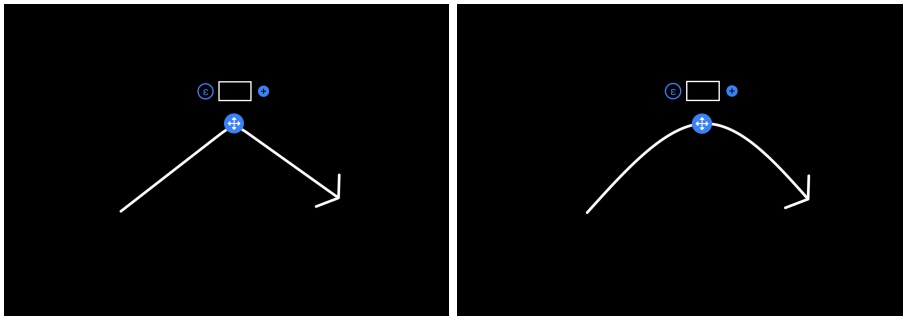
Dragging is an important feature that lets users rearrange the automaton's elements. Both states and transitions should be draggable. When a button for dragging, discussed in chapter 4, is moved by the user, an action is triggered - it can be either `stateDragPointChanged` or `transitionFlexPointChanged` depending on the FA element.

5.5.1 Dragging Transitions

Transitions are dragged with what I have called a `flexPoint`. The transition is thus constructed with three points instead of two - start point, flex point and tip point. But doing so makes the curve of the transition sharp at the flex point. To make the change of the direction taking place at the flex point smooth I have used a library called SwiftSplines [60]. A spline is "a piecewise polynomial in which the coefficients of each polynomial are fixed between 'knots' or joints", using a definition from the Cubic Splines paper [61]. You can see how the transition looked like before applying the cubic spline and after in 5.2. In 5.14 it is possible to see how it is created in code. Note the important parameter `resolution` that directs how "smooth" the curve will be where the higher resolution will increase the number of points used for the curve.

Code snippet 5.14 Creating spline points

```
let points = [startPoint, flexPoint, tipPoint].compactMap { $0 }
let spline = Spline(values: points)
let resolution = 100
let splinePoints: [CGPoint] = (
    0...(points.count - 1) * resolution
)
.map { offset in
    let argument = CGFloat(offset)/CGFloat(resolution)
    return spline.f(t: argument)
}
```



(a) (b)

Figure 5.2 Transition before using cubic spline (a) and after (b)

5.5.2 Dragging States

States are dragged as a whole and thus their shape is not changed during that process - the only variable in this process is a center of a given state. However, when a state has transitions connected to it, these transitions have to be changed as the state is dragged. There are three types of transitions that can be connected to a state - a transition that corresponds to an incoming or outcoming edge and a cycle. For cycles the app stores an angle between the single point that intersects with the state and a vector going from a center of the state and its topmost point. When the center is moved, the intersection point is recalculated with the angle that is stored in the app's state. For transitions other than cycles I take advantage of the `flexPoint` and recompute `startPoint` or `endPoint` (depending on the transition) as the intersection point on the shortest path between the new center and the aforementioned `flexPoint`. You can see that logic in 5.15 for transitions that are the incoming edge.

Code snippet 5.15 Calculation of new `tipPoint`

```
state.transitions
.filter {
    $0.endState == automatonStateID
    && $0.endState != $0.startState
}
.forEach { transition in
    guard
        let flexPoint = transition.flexPoint,
        let endStateID = transition.endState,
        let endState = state.automatonStatesDict[endStateID]
    else { return }
    let vector = Vector(endState.center, flexPoint)
    state.transitionsDict[transition.id]?.tipPoint = vector.point(
        distance: endState.radius,
        other: endState.center
    )
}
```

5.6 DocumentGroup

I have already outlined the intent to use a document based app approach to enable saving the editor's state into a file. Leveraging the benefits of using the Composable Architecture, I was able to save the whole app's state as JSON. To make that happen, I first had to conform `EditorState` to a `FileDocument` protocol. When a type conforms to this protocol it has to implement reading a file as well as saving it. The type must also declare on which extensions it will operate. For this app I have chosen the `automaton` extension. This also ensures that the user cannot open a file that does not have that same extension (which would lead to undefined behaviour). To make an app a document based one, it is necessary to wrap a root view into a `DocumentGroup` scene [62].

`DocumentGroup` is initialized with two parameters. `newDocument` that describes what data should a new document contain - for the FA editor it is a new editor state - and then `editor` closure that has the currently opened document as an input and it should return a view for editing the document. You can see how that is done for the automata editor here: 5.16. It should be noted that apart from initializing a new `EditorStore` for a given document, as well as modifying the file when the state changes, it is also desirable to save stores for documents that have already been opened in the current application run. This is to ensure that a store is not reinitialized when a state changes which would cancel all currently running effects. The stores are saved into a dictionary `stores` where each store is identified by the unique state's id. The UI for document based app is in figure 5.3.

Code snippet 5.16 DocumentGroup scene

```
private final class DocumentStore {
    var stores: [UUID: EditorStore] = [:]
}

@main
struct AutomataEditorApp: App {
    private var documentStore = DocumentStore()

    var body: some Scene {
        DocumentGroup(
            newDocument: EditorState()
        ) { file -> EditorView in
            let store = documentStore.stores[file.document.id]
            ?? EditorStore(
                initialState: file.document,
                reducer: editorReducer,
                environment: EditorEnvironment(...)
            )
            documentStore.stores[file.document.id] = store

            return EditorView(
                set: {
                    file.document = $0
                },
                store: store
            )
        }
    }
}
```

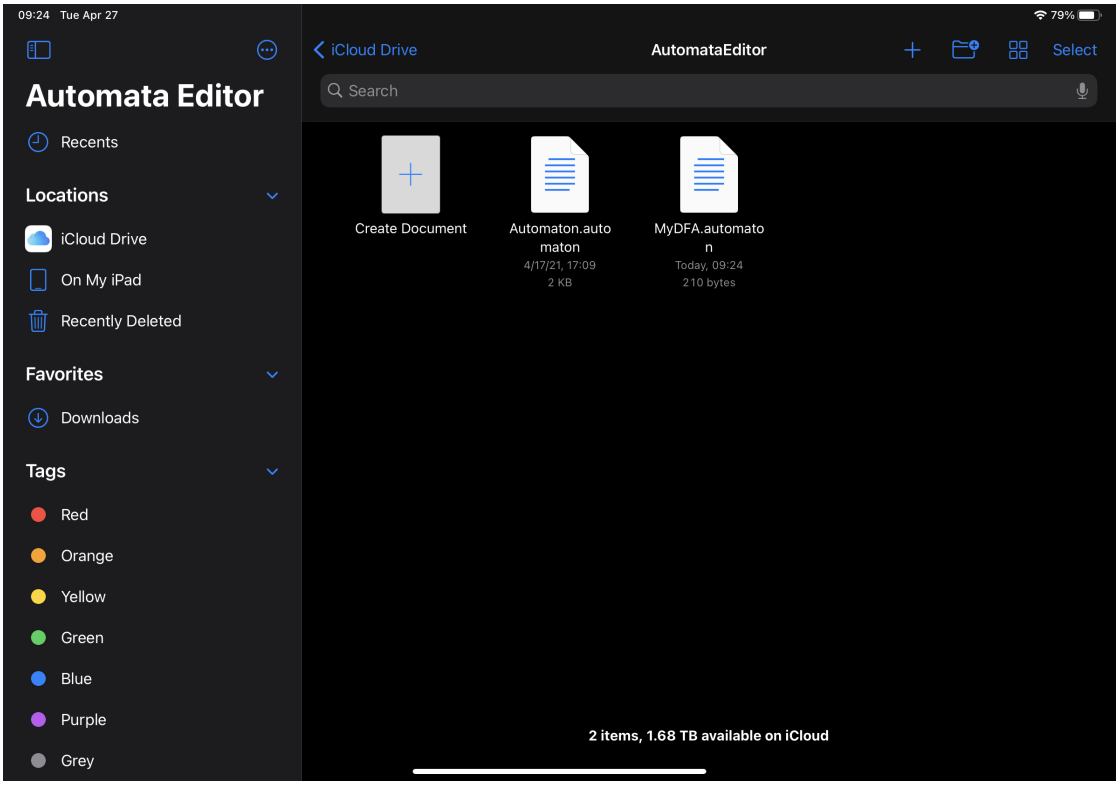


Figure 5.3 Document based app interface

Chapter 6
Testing

Firstly, I will discuss how the app has been tested with automated tests. After that, I will go into how the usability testing has been conducted and what were its results and outcomes.

6.1 Automated Testing

Automated testing is an important part of a software project to ensure everything is working correctly and no regressions are made during the development. It also increases the confidence of doing refactoring. The Composable Architecture unit testing is built on top of the XCTest framework that is the bundled framework for testing Swift or Objective-C in Xcode. The goal of the testing in the Composable Architecture was to make tests more exhaustive, ergonomic and concise.

The main component for testing in the Composable Architecture is `TestStore`. This component has a method called `send` that takes as an input an action and a closure in which you describe how you expect the state to be modified. There is also a `receive` method that asserts receival of a new action not triggered by the user, but rather returned from an effect. An example showcasing both `send` and `receive` methods can be seen in 6.1 - there `strokesChanged` action, that simulates a user drawing a stroke, is triggered first. When a stroke is drawn, it is sent to `AutomataLibraryService` that returns `Effect` with the recognized element (note that in the tests this component is mocked). When the new action `automataShapeClassified` is received, I assert that it is added to the state.

It should be noted that the paradigm of this style of unit testing is focused on flows, rather than individual functions where the test asserts output for a given input - although individual functions can still be tested by sending a single action to the test store and asserting the state's changes. This is to make tests more concise and be closer to how the app will be used by the user. It does not mean, however, that the tests can be less reliable because still a single component is tested and all non-deterministic functionality should be in the store's environment, which is mocked during the testing.

Code snippet 6.1 Testing creating of state

```
store.send(
    .strokesChanged(
        currentStrokes + [
            Stroke(controlPoints: [center])
        ]
    )
)
```

```
scheduler.advance()

store.receive(
    .automataShapeClassified(
        .success(
            .state(
                Stroke(
                    controlPoints: [center]
                )
            )
        )
    )
) {
    $0.automatonStatesDict[id] = AutomatonState(
        id: id,
        center: center,
        radius: radius
    )
}
```

6.2 Usability Testing

Usability testing does not have a strict definition but in general, it should refer to testing that evaluates a product or a system that is not automated. However, in the Handbook of Usability Testing the authors have narrowed this down to ”a process that employs people as testing participants who are representative of the target audience to evaluate the degree to which a product meets specific usability criteria” [63]. This book has also been used further in this section to define types of tests and their recommended techniques with which they should be conducted.

In the Handbook of Usability Testing the authors divide usability tests into four distinct categories: exploratory, assessment, validation, and comparison. Each test differs by its acceptance criteria and in which phase of the development it should be done. As the final product of this thesis is a prototype, the exploratory test is the most suitable one.

The primary focus of exploratory tests should be on the main functionalities of the app and to assess answers to questions such as how intuitive the design is, how valuable the product is for the user, which functionality will need to be documented via a tutorial or help, etc. This type of test is also different from the others in that it is expected that the moderator conducting the test will communicate with the user extensively - whereas the other tests should be possible to finish without the moderator stepping in.

6.2.1 Test Plan

Before tests could take place, it was necessary to create a test plan that was to be followed in the tests themselves. Test plans do not have to have the same components but the structure of the test plan for the Automata Editor app consists of:

- Methodology
- Participant characteristics
- Introduction to the session
- Task list

- Final feedback

As outlined above, the test was exploratory, implicating intensive interaction between the user and the mediator. Apart from that, it was measured how long it took for the participants to accomplish the individual tasks .

Before each test, the participants were sent a questionnaire to assess their characteristics. It included the following questions:

- How familiar are you with finite automata? (rated 1-5 where 5 is ”expert”)
- Do you have prior experience with software for creating and editing finite automata?
- How familiar are you with using Apple Pencil in productivity applications - such as note-taking, drawing, creating diagrams, etc.? (rated 1-5 where 5 is ”expert”)
- Could you provide examples of software that you use with Apple Pencil?

These questions, although self-evaluated, should give a clearer picture of the user's previous experience with either finite automata or iPad usage.

During the introduction to the test, participants were asked to think aloud, if they were comfortable with it. It is beneficial to know the thinking process as that can uncover more about the prototype. Then the users were asked to do five tasks. These tasks were of two types:

- Draw automaton based on a figure and test prescribed inputs.
- Draw automaton based on a formal definition of a language. The user has to come with inputs to test that it works properly.

The second type of tests (tasks D and E) is more complicated and therefore it was to be done after the first one (tasks A-C). The tasks for each type were also randomized. Below you can find the individual tasks:

- Task A:
  Create the automaton in 6.1 and test the following inputs:
  - AA
  - BA
  - B
  - AAA
  - ABA
  - A

  Then modify this automaton without clearing the canvas in such a way that it represents automaton in 6.2 and test the following inputs:
  - ABBA
  - ABAABA
  - AAA
  - BAB
  - BBB
- Task B:
  Create the automaton in 6.3 and test the following inputs:
  - DDE

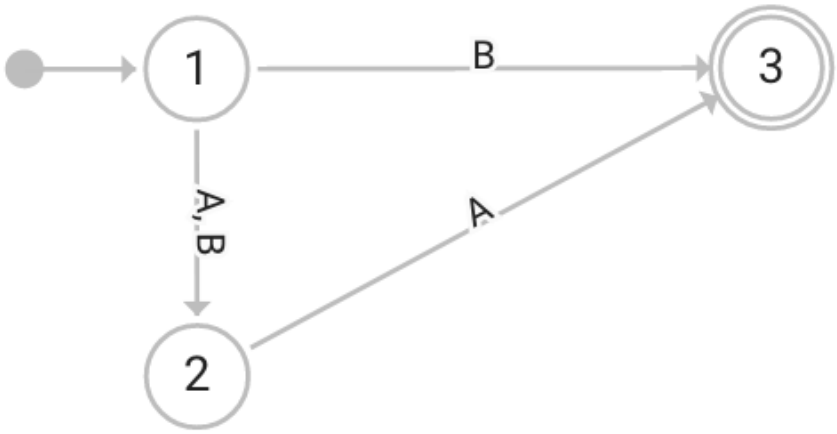


Figure 6.1 Task A automaton

- DEF
- DE
- EF
- EFE
- DDDDE
- DDDDF

- Task C:
  Create the automaton in 6.4 and test the following inputs:
  - A
  - BA
  - AB
  - ABB

- Task D:
  Create automaton for the following language:
  $L = \{(10)^n\text{: n} \in N\}$
  Simulate at least 4 inputs to test whether the automaton is behaving correctly.

- Task C:
  Create automaton for the following language:
  $L = \{\text{w: w} \in \{0,1\}^* \wedge \text{w starts with the string 011}\}$
  Simulate at least 4 inputs to test whether the automaton is behaving correctly.

6.2.2 Test Results

The tests have been done with a total of 8 participants. The results can be seen in 6.5. I have not included two questions in the table. One is whether the participant has prior experience

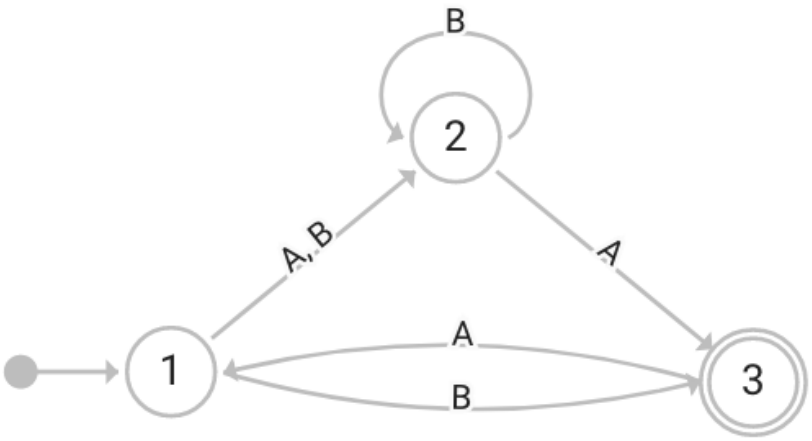


Figure 6.2 Task A modified automaton

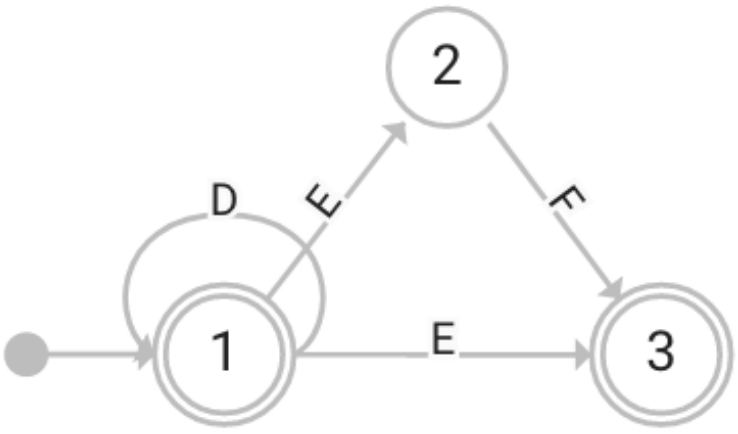


Figure 6.3 Task B automaton

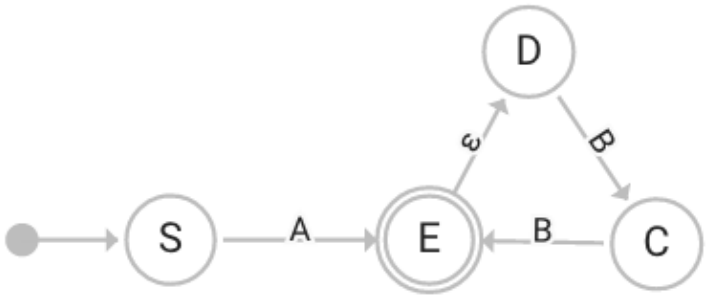


Figure 6.4 Task C automaton

with software for creating and editing finite automata - for which all answers have been ”No”. This indicates that most (in this case all) people have always used regular paper for sketching their automata. The second question not included is about examples of software that the participant has used with Apple Pencil: most of it were apps for making notes such as Notability or GoodNotes. From the time results the two fastest times are by people who rated themselves as FA ”experts”. In contrast with that is the Apple Pencil familiarity having not a great impact on the time results.

The following list summarizes what the participants have said they did not understand and where the design was not intuitive enough:

- Plus button for adding multiple symbols to a single transition.
- When a state is deleted it’s hard to reconnect it to transitions.
- Hard to understand output when it does not include any words.
- ϵ button was often not obvious.
- Clear was mistaken for back, deleting the whole automaton.
- Erasing button is not intuitive.
- It is hard to tell if the simulate button works if the output does not change.

Some of these points could be resolved by adding a quick tutorial and help that would introduce users into how the app should be used, rather than expecting users to figure it out on their own. The tutorial is also important since states and transitions are not added via a button and thus the users need to try and see how the app responds to their drawing. This can be a daunting first experience.

The help feature, among others, was also mentioned by the participants as one of the following suggested improvements:

- Missing switch gesture for Apple Pencil.
- Back button to undo last change.

- Larger bottom control panel.
- Help or quick tutorial describing how the app should be used.
- Keyboard is too intrusive and often overlays text field currently being edited.
- It is not possible to make a state smaller.
- Make naming states not mandatory.
- Step-by-step simulation would help with visualization of how the FA works.
- Edit and presentation mode to hide and show buttons and text fields.

From this list, most of the items are small improvements and could be easily implemented into the app. I would like to expand on "step-by-step simulation". This feature would help with understanding better why the automaton rejected the input strings. It would also make it a great tool for learning and understanding how the automaton works. Implementation for DFAs should be possible as the automaton is always at one state only, making the visualization straightforward. However, for NFAs this is not the case. Therefore, the app could either offer to limit the automaton to a specific type or limit the visualization to only one possible path leading to rejection or acceptance. The usability testing has also uncovered few bugs:

- Input by keyboard is sometimes inadvertently triggered.
- Cycle is often mistaken for a state.
- Eraser sometimes does not work.
- It is not possible to make a final state a non-final one without deleting it.
- When a user taps on the screen in quick succession, transitions appear in the same place.
- Text field for transitions does not trim newlines.

All of these bugs are tracked in the project's repository.

Apart from things that could be improved, at the end of each test I have asked the participants to say what they felt were good things about the app:

- Eases learning of FAs.
- Simulating inputs is quick.
- Ability to save the FA in a file.
- Saving to a file is automatic.
- Making transitions with a simple line is fast.
- Recognizing elements is almost instant.
- Creating automata is quick once the user knows how to use the editor.
- Connecting elements works well.
- Interacting with elements is intuitive.

The positive comments show that the basic interactions with the elements work well and offer an enjoyable experience for the users. And apart from the enjoyable experience it makes creating the finite automata quick - which is important to convince people to use the app, rather than resorting to regular paper.

Participant	Total time (mm:ss)	FA familiarity	Apple Pencil faimiliarity
A	29:28	1	4
B	10:30	5	1
C	08:48	5	4
D	15:26	3	1
E	19:30	3	2
F	25:12	3	4
E	33:23	3	4
G	25:58	5	3

Figure 6.5 Usability testing results

Chapter 7
Conclusion

The primary goal of this thesis was to implement a prototype of an automata editor for the iPad, built primarily for touch-based input and minimizing the need to use other UI controls such as buttons. The final prototype has focused on the Apple Pencil that offers a high level of precision while being easy to use. Most of the functionalities, like creating and editing FA elements, are done with the Apple Pencil, thus accomplishing the main goal. This was made possible by creating a custom ML model and using the CoreML framework to classify the drawn FA elements.

The chosen Composable Architecture has proved to be beneficial for the project as it allowed easy testing as well as a straightforward implementation of a document based app, which is an important feature for the users that was added on top of the originally outlined goals.

For simulating the input I have successfully leveraged the ALT library, proving that it can be integrated into a Swift-based project and used in iOS applications. This integration also opens many possible future directions this project can take as it can tap into the functionalities the library offers, such as export of automatons in DOT and PNG format or FA algorithms like determinization and removal of ϵ-transitions.

In the conducted usability testing, which has uncovered some minor issues with the prototype, it has also been suggested by some of the participants to add a step-by-step simulation of the input, so the users can better visualize how the particular FA operates. This functionality would also have to be implemented in the ALT library by e.g. returning a trace of the individual steps taken by the FA. Another feature that has been raised during the testing was adding support for a pushdown automaton.

One of the main motivations of this project was to help students learning FAs. To accomplish this, the prototype needs some smaller improvements but it is planned to publish it on the App Store before the winter semester of the 2021/2022 school year, so it can be suggested to students taking the BI-AAG course at FIT CTU in Prague.

The feedback I have been given in the usability testing has made it clear that the prototype, even in the current state, would be beneficial for students learning the FAs, therefore, I deem the prototype to be successful.

Appendix A

Acronyms

DFA	Deterministic Finite Automaton
FA	Finite Automaton
NFA	Nondeterministic Finite Automaton
ALT	Algorithms Library Toolkit
UI	User Interface
ML	Machine Learning
JSON	JavaScript Object Notation

Appendix B

User Instructions

To run the application, you will need Xcode installed. It can be installed via App Store or from Apple's developer website. Afterwards, open `automata-editor/AutomataEditor.xcodeproj` using Xcode. To run the application on a device, you will also need to set up code signing.

Bibliography

1. TRÁVNÍČEK, Jan; PECKA, Tomáš; PLACHÝ, Štěpán. *Algorithms Library Toolkit* [online] [visited on 2021-04-06]. Available from: `https://alt.fit.cvut.cz/`.
2. SVOBODA, Petr. *Webový editor konečných automat* [Bachelor's thesis]. 2019.
3. VRÁNA, Michael. *Knihovna algoritmů ALT-webové rozhraní* [Bachelor's thesis]. 2020.
4. ŠESTÁKOVÁ, Eliška. *Automata and Grammars: A Collection of Exercises and Solutions*. Faculty of Information Technology, Czech Technical University in Prague, 2020. ISBN 978-80-01-06462-7.
5. HOPCROFT, John E.; MOTWANI, Rajeev; ULLMAN, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation*. 2007. ISBN 9780321455369.
6. HOLUB, Jan. *BIE-AAG* [online] [visited on 2021-04-09]. Available from: `https://courses.fit.cvut.cz/BIE-AAG/`.
7. MOHRI, Mehryar; ROSTAMIZADEH, Afshin; TALWALKAR, Ameet. *Foundations of Machine Learning*. 2018. ISBN 9780262039406.
8. ČEPEK, Miroslav; VAŠATA, Daniel. *BIE-VZD* [online] [visited on 2021-04-10]. Available from: `https://courses.fit.cvut.cz/BIE-VZD/`.
9. BERGEN, Patrick van. *Pipe-And-Filter* [online]. 2020 [visited on 2021-04-10]. Available from: `http://www.dossier-andreas.net/software_architecture/pipe_and_filter.html`.
10. STANESCU, Cristian. *TuringSim* [online] [visited on 2021-04-10]. Available from: `https://homes.di.unimi.it/borghese/Teaching/AdvancedIntelligentSystems/ProjectDocuments/Progetto1_Stanescu.pdf`. App available at `https://apps.apple.com/us/app/turingsim/id563350412`.
11. VILELA, Plinio. *Finite Automata* [online] [visited on 2021-04-10]. Available from: `https://apps.apple.com/us/app/finite-automata/id1043670880`.
12. RANGEL-MONDRAGON, Jaime. *A Finite Automaton Editor* [online]. 2011-03 [visited on 2021-04-13]. Available from: `http://demonstrations.wolfram.com/AFiniteAutomatonEditor/`.
13. SHAWABKEH, Max. *Automata Editor* [online] [visited on 2021-04-13]. Available from: `http://max99x.com/school/automata-editor`.
14. *GitLab* [online] [visited on 2021-04-13]. Available from: `https://about.gitlab.com/`.
15. *Algorithms Library Toolkit* [online] [visited on 2021-04-13]. Available from: `https://gitlab.fit.cvut.cz/algorithms-library-toolkit`.
16. FOUNDATION, Standard C++. *C++* [online] [visited on 2021-04-16]. Available from: `https://isocpp.org/`.

17. KITWARE. *CMake* [online] [visited on 2021-04-13]. Available from: `https://cmake.org/`.
18. *TensorFlow* [online] [visited on 2021-04-13]. Available from: `https://www.tensorflow.org/`.
19. APPLE. *Core ML* [online] [visited on 2021-04-13]. Available from: `https://developer.apple.com/documentation/coreml`.
20. APPLE. *Core ML Models* [online] [visited on 2021-04-13]. Available from: `https://developer.apple.com/machine-learning/models/`.
21. *Keras* [online] [visited on 2021-04-13]. Available from: `https://keras.io/`.
22. APPLE. *coremltools* [online] [visited on 2021-04-13]. Available from: `https://coremltools.readme.io/docs/`.
23. APPLE. *Create ML* [online] [visited on 2021-04-13]. Available from: `https://developer.apple.com/machine-learning/create-ml/`.
24. APPLE. *turicreate* [online] [visited on 2021-04-13]. Available from: `https://github.com/apple/turicreate/`.
25. APPLE. *Drawing Classification* [online] [visited on 2021-04-13]. Available from: `https://apple.github.io/turicreate/docs/userguide/drawing_classifier/`.
26. APPLE. *Apple Pencil* [online] [visited on 2021-04-16]. Available from: `https://www.apple.com/apple-pencil/`.
27. KOCHAN, Stephen G. *Programming in Objective-C*. 2021. ISBN 9780321967602.
28. APPLE. *Objective-C* [online] [visited on 2021-04-16]. Available from: `https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html/`.
29. TIOBE. *TIOBE Index* [online] [visited on 2021-04-16]. Available from: `https://www.tiobe.com/tiobe-index/`.
30. APPLE. *SwiftUI* [online] [visited on 2021-04-16]. Available from: `https://developer.apple.com/documentation/swiftui/`.
31. FACEBOOK. *React Native* [online] [visited on 2021-04-16]. Available from: `https://reactnative.dev/`.
32. GOOGLE. *Flutter* [online] [visited on 2021-04-16]. Available from: `https://flutter.dev/`.
33. APPLE. *PencilKit* [online] [visited on 2021-04-16]. Available from: `https://developer.apple.com/documentation/pencilkit/`.
34. APPLE. *UIKit* [online] [visited on 2021-04-16]. Available from: `https://developer.apple.com/documentation/uikit/`.
35. APPLE. *Apple unveils groundbreaking new technologies for app development* [online]. 2019-06 [visited on 2021-04-16]. Available from: `https://www.apple.com/newsroom/2019/06/apple-unveils-groundbreaking-new-technologies-for-app-development/`.
36. KUGLER, Florian; EIDHOF, Chris. *Thinking in SwiftUI*. 2020. ISBN 9798626292411.
37. FACEBOOK. *React* [online] [visited on 2021-04-16]. Available from: `https://reactjs.org/`.
38. APPLE. *Interfacing with UIKit* [online] [visited on 2021-04-16]. Available from: `https://developer.apple.com/tutorials/swiftui/interfacing-with-uikit/`.
39. APPLE. *Structure your app for SwiftUI previews* [online] [visited on 2021-04-16]. Available from: `https://developer.apple.com/videos/play/wwdc2020/10149/`.
40. APPLE. *CGContext* [online] [visited on 2021-05-05]. Available from: `https://developer.apple.com/documentation/coregraphics/cgcontext`.

41. APPLE. *CGContext* [online] [visited on 2021-05-05]. Available from: https://developer.apple.com/documentation/coregraphics.
42. APPLE. *PKCanvasView* [online] [visited on 2021-04-20]. Available from: https://developer.apple.com/documentation/pencilkit/pkcanvasview/.
43. MICROSOFT. *The Model-View-ViewModel Pattern* [online] [visited on 2021-04-16]. Available from: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm/.
44. POINT-FREE. *The Composable Architecture* [online] [visited on 2021-04-16]. Available from: https://github.com/pointfreeco/swift-composable-architecture/.
45. BERGH, Michael Van den. *React Redux: Building Modern Web Apps with the ArcGIS JS API* [online] [visited on 2021-04-16]. Available from: https://www.esri.com/arcgis-blog/products/js-api-arcgis/3d-gis/react-redux-building-modern-web-apps-with-the-arcgis-js-api/.
46. *Interoperability between Swift and C++* [online] [visited on 2021-04-16]. Available from: https://github.com/apple/swift/blob/main/docs/CppInteroperabilityManifesto.md#exceptions/.
47. APPLE. *Importing Objective-C into Swift* [online] [visited on 2021-04-16]. Available from: https://developer.apple.com/documentation/swift/imported_c_and_objective-c_apis/importing_objective-c_into_swift/.
48. APPLE. *Using C++ With Objective-C* [online] [visited on 2021-04-16]. Available from: https://web.archive.org/web/20101203170217/http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocCPlusPlus.html/.
49. APPLE. *iPad User Guide* [online] [visited on 2021-04-17]. Available from: https://support.apple.com/guide/ipad/enter-text-with-scribble-ipad355ab2a7/14.0/ipados/14.0/.
50. APPLE. *Document Based Apps* [online] [visited on 2021-04-17]. Available from: https://developer.apple.com/document-based-apps/.
51. GOOGLE. *Quick Draw! The Data* [online] [visited on 2021-04-20]. Available from: https://quickdraw.withgoogle.com/data/.
52. FOŘT, Marek. *MNIST Maker* [online] [visited on 2021-04-20]. Available from: https://github.com/fortmarek/MNIST-Maker/.
53. JUPYTER, Project. *Jupyter* [online] [visited on 2021-04-20]. Available from: https://jupyter.org/.
54. APPLE. *Xcode* [online] [visited on 2021-04-20]. Available from: https://developer.apple.com/xcode/.
55. LOCKWOOD, Nick. *VectorMath* [online] [visited on 2021-04-20]. Available from: https://github.com/nicklockwood/VectorMath/.
56. APPLE. *Swift Package Manager* [online] [visited on 2021-04-20]. Available from: https://github.com/apple/swift-package-manager/.
57. LLVM. *Clang* [online] [visited on 2021-04-20]. Available from: https://clang.llvm.org/.
58. WIDERBERG, Alexander. *ios-cmake* [online] [visited on 2021-04-20]. Available from: https://github.com/leetal/ios-cmake/.
59. APPLE. *Binary Frameworks in Swift* [online] [visited on 2021-04-20]. Available from: https://developer.apple.com/videos/play/wwdc2019/416/.
60. FEILER, Konrad. *SwiftSplines* [online] [visited on 2021-04-23]. Available from: https://github.com/Bersaelor/SwiftSplines/.

61. JAMES, Antony. *Cubic Splines* [online]. Stanford [visited on 2021-04-23]. Department of Aeronautics and Astronautics, Stanford University. Available from: `http://aero-comlab.stanford.edu/Papers/splines.pdf/`.

62. APPLE. *DocumentGroup* [online] [visited on 2021-04-27]. Available from: `https://developer.apple.com/documentation/swiftui/documentgroup/`.

63. RUBIN, Jeffrey; CHISNELL, Dana. *Handbook of Usability Testing*. 2008. ISBN 9780470386088.

Contents of the Thesis' Attached Medium

- `automata-editor` ..... app's source code
- `thesis` ..... thesis' source code
- `thesis.pdf` ..... text of thesis in PDF format
- `readme.md` ..... description of the attached medium's contents