Replace the contents of this file with official assignment.
Místo tohoto souboru sem patří list se zadáním závěrečné práce.

Bachelor's Thesis

FINITE AUTOMATA IPAD EDITOR

Marek Fořt

Czech Technical University, Faculty of Information Technology
Department of Theoretical Informatics
Supervisor: Ing. Jiří Trávníček, Ph.D.
April 17, 2021

Czech Technical University in Prague
Faculty of Information Technology

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

Contents

List of Figures

List of Tables

Seznam výpisů kódu

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

Write abstract here!

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

Chapter 1
Introduction

1.1 Motivation, Focus of Thesis

Theory of finite automata is an important part of 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 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 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 try out algorithms library 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 web interface of ALT [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 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 theoretical concepts 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 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 user testing and its outcomes.
- **Conclusion** is the last chapter of this thesis where I will assess the success of fulfilling 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 describe languages. Below you can find how they are defined:

▶ **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:
$$\gamma A \delta \to \gamma \alpha \delta \ (\gamma, \delta \in (N \cup \Sigma)^*, \ a \in (N \cup \Sigma)^+)$$
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 recognized by Turing machine.
2. *recursively enumerable* if and only if ∃ unrestricted grammar which generates it
   - recognized by Turing machine
3. *context-sensitive* if and only if ∃ context-sensitive grammar which generates it
   - recognized by 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 finite automaton

In this thesis we will be mainly interested in regular languages / grammars 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 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 an end 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 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 *deterministic* finite automaton or *nondeterministic* finite automaton. This dictates the exact definition of δ - transition function. For deterministic finite automaton (DFA) the definition of δ is:

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

δ for 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 definition of the transition function:

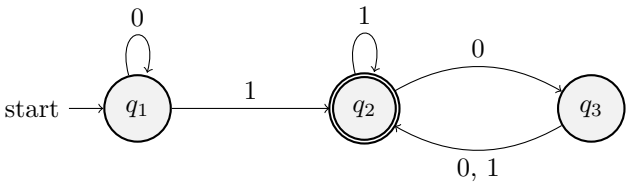


Figure 2.1 FA graph representation

δ_{NFA}	0	1
$\rightarrow S$	S	S, A
A	B	
$\leftarrow B$		

Figure 2.2 FA table representation

- DFA can only transition from one state to another, e.g. from q_0 to q_1 (q_0, $q_1 \in Q$)
- NFA can transition to a set of states, e.g. from q_0 to q_1 (q_0, 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 *nondeterministic finite automaton with* $\epsilon - transitions$.

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 incoming edge. The transitions are then directed edges between states. You can see FA represented as weighted directed graph in 2.1
- *Table*
  Table representation has in the first column all states where initial state is marked with $\rightarrow$ while final states is 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 form of weighted directed graph as that is what will the user edit in the app. This also concludes theory about finite automata and formal languages.

2.3 Machine Learning

Machine learning does not have an exact definition but e.g. in a 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, have defined a few learning scenarios, most notably supervised and unsupervised learning. 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 the 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, person can leave the bed - 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 edit finite automata
- ALT itself, focusing on design and drawing of finite automata
- possibilites 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 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 ouputs of the individual algorithms that ALT offers which can be seen in figure 3.1. Apart from ALT algorithms it also includes 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, as well as initial and final states
- add transitions between states
- edit transition string
- mark state as initial or final
- remove states and transitions
- import and export automaton in supported formats
- automatic positioning of transitions and states

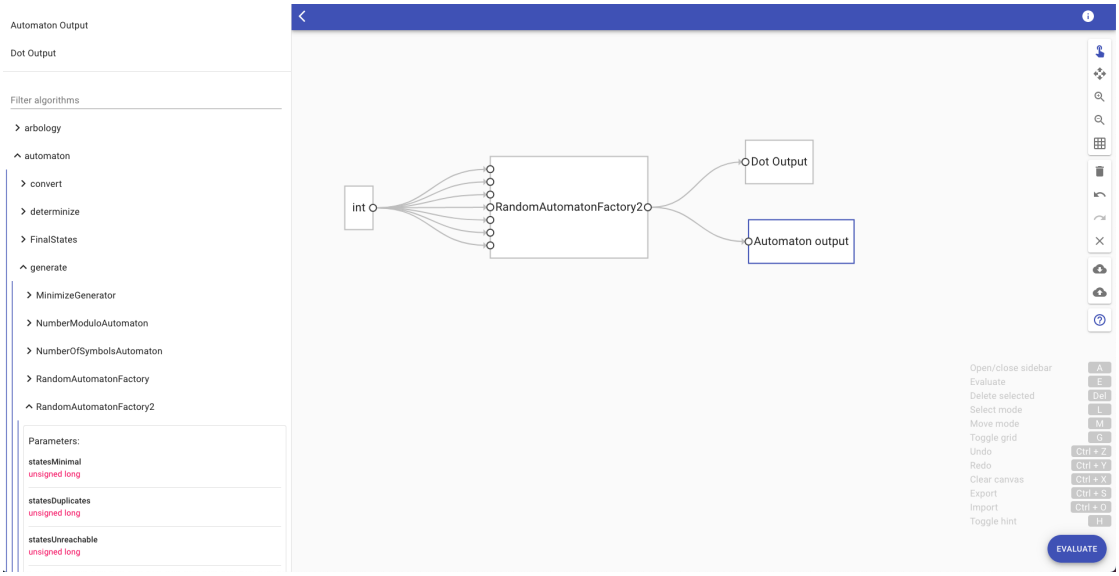


Figure 3.1 ALT web interface screenshot

All of the above features work reliably and are done in intuitive manner - user can quickly understand how to work with all the components. The most notable missing feature is easy simulation of input - this can be done via ALT web interface but if someone is looking for only editing FAs and simulating whether input string is accepted, they have to transition between two interfaces. The benefit is that they can then tap into all the other funcionality 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 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 editors 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. Unfortunately, the app on iPad is broken at the moment as it is missing bottom toolbar for simulating input.

There is also app called Finite Automata [11]. In this app user can not edit automata in their weighted graph representation but instead has to use a command line that takes individual command 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 a Finite Automaton Editor by Jaime Rangel-Mondragon that is available as interactive Wolfram notebook [12]. This app allows you to edit the automaton via a transition table and does not allow to simulate any input. There is also Automata Editor by Max Shawabkeh [13]. In this desktop applications users can create and edit their automaton either via a table representation or regular expression. There are also features such as NFA determinization, evaluating automata on strings, and minimizing DFA. Thus it has a powerful feature set but one has to be already familiar with FA theory. It should also be noted that the feature set is a subset of what ALT web interface offers.

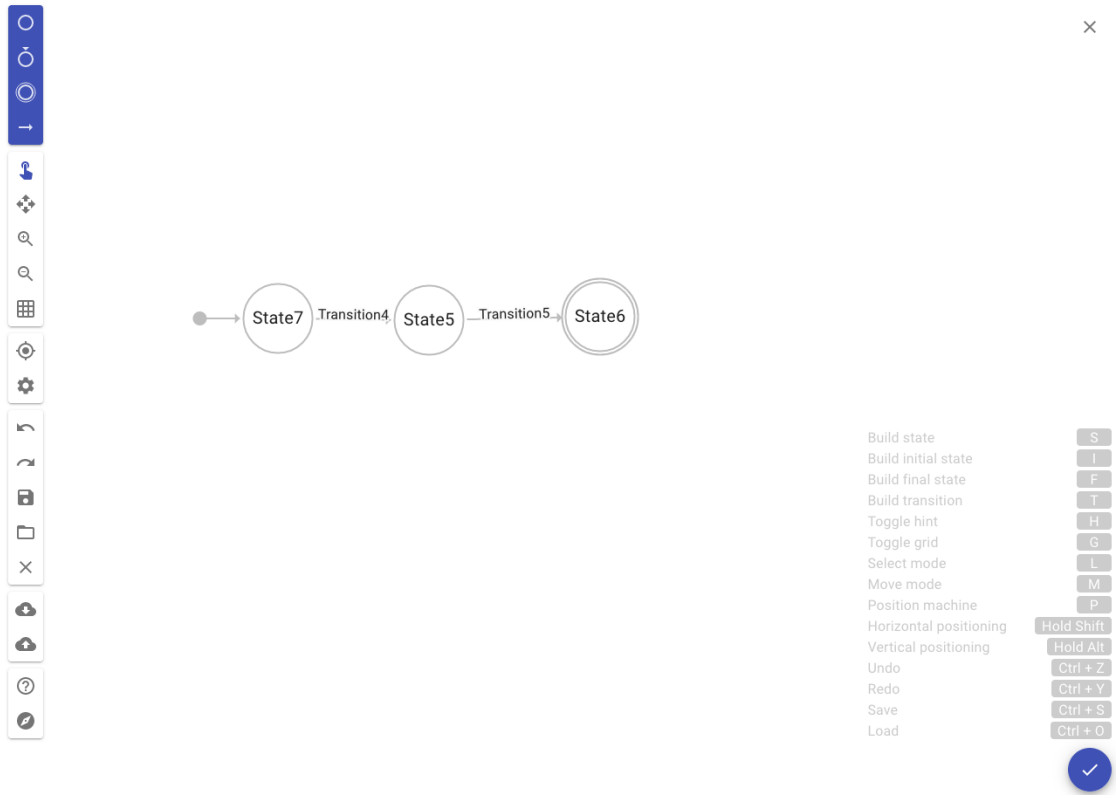


Figure 3.2 Statemaker screenshot

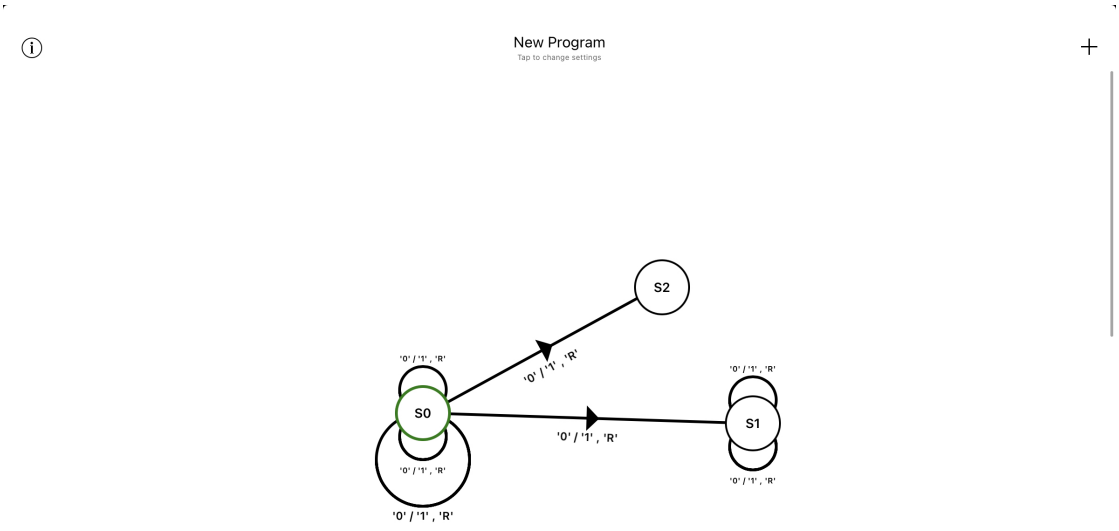


Figure 3.3 TuringSim interface screenshot

```
// 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
);
```

Figure 3.4 EpsilonNFA example methods

3.2 ALT

Now I will 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 group Algorithms Library Toolkit (webui-client, infrastructe, etc.) - that includes repository Algorithms Library Toolkit Core [15], a library written in C++ [16]. There we can find algorithms that can later be used for FA editor. The code is divided into multiple modules that are then built and linked together using CMake [17]. The most important module 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 - that includes deterministic and nondeterministic finite automaton as well as nondeterministic finite automaton with $\epsilon - transitions$. There is also extended NFA that has regular expressions as their transitions. These models serve as a definition of an automaton - its states, transitions, etc. To create e.g. NFA with $\epsilon - transitions$ 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 3.4.

3.2.2 FA Algorithms

ALT offers 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` and `Run.h`. Where the former is able to determine whether an input is accepted and the latter does the same but a part of its output is also e.g. in which states did the simulation end in.

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 `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 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. 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]. 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 app's size and incurring maintenance burden, I opted for CoreML. This decision was also 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 Creation of CoreML Model

There are multiple ways how to create 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].
- 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 28x28 grayscale bitmap as input. The drawing classification is also tailored for inputs created by 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 it should serve to streamline 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 a paper. There were three main options that were possible:

- create touch-friendly web interface
- implement 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 Apple Pencil [26] that offers high precision that will make the user experience better.

4.2 Used Technologies

I will now cover what technologies are used in the app and why I have chosen them. A lot of decisions have been influenced by focusing on iPad, and even more specifically, usage of the app on iPad along with 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 used:

- Objective-C
- Swift
- cross-platform framework

Objective-C was designed by Brad J. Cox at the start of 1980s and was then licensed by NeXT Software in 1988. Then in 1996 NeXT Software was acquired by Apple - along with Objective-C. Apple have then chosen Objective-C as a 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 at the 23rd place 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 thus a much better option to choose if one is starting a new iOS app.

Using a cross-platform framework - such as React Native [31] or Flutter [32] - was also a possibility. But to leverage Apple Pencil fully it was necessary to use PencilKit [33] and for this one would have to write native code. Therefore, 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, release in 2019 [35]. In Thinking in SwiftUI book it is described as ”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 benefits of less code since it enables the framework to do more on behalf of the developers. 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 also SwiftUI previews [39] has made it a better candidate than UIKit, especially for a prototype. Therefore, I decided to use SwiftUI as the main UI framework.

4.2.3 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 more difficult to maintain for a longer period.

One possible architecture is MVVM [40]. 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 alternative are (among others) architectures inspired by Redux. 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. Data-driven approach is much more closer to 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 [41]. 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 points given, I have chosen to use the Composable Architecture.

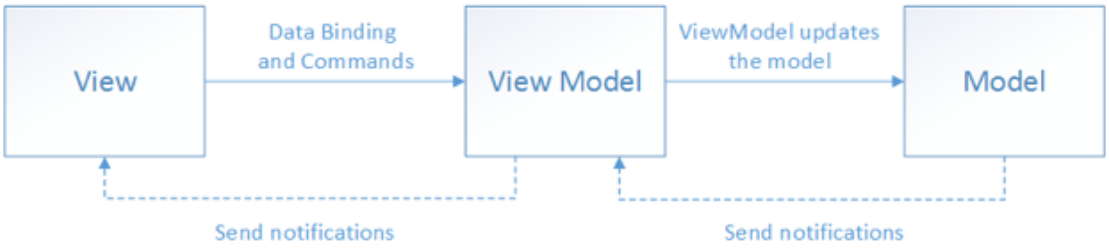


Figure 4.1 MVVM architecture diagram [40]

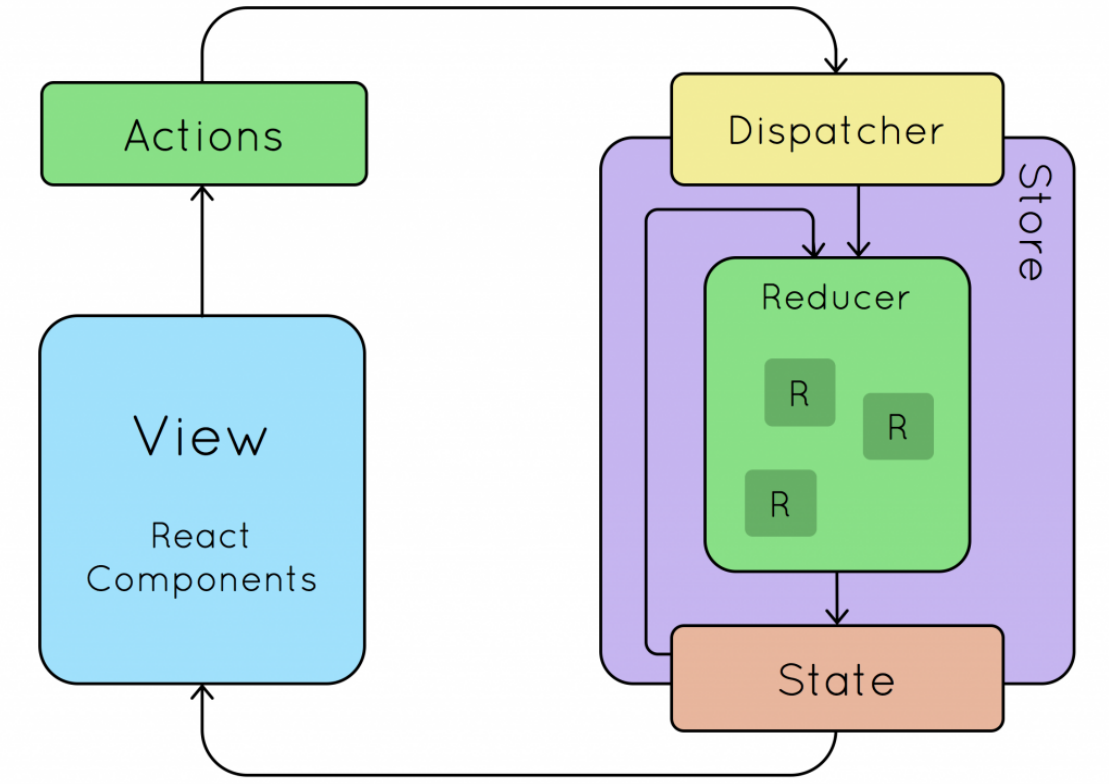


Figure 4.2 Redux architecture diagram [42]

4.3 ALT Integration

I have already talked about ALT in chapter 3. I have not discussed there, however, how ALT will be integrated in 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 [43]. This manifesto 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 a well-supported interoperability between Swift and Objective-C [44]. For Objective-C and C++ interoperability there is a language iteration of Objective-C called Objective-C++. It is even possible to "include pointers to Objective-C objects as data members of C++ classes" [45]. I first tried to integrate ALT directly and compile it right via Xcode but due to the fact that ALT is built via CMake [17] and does not have a simple setup, I then resorted to pre-building and then bundle the already built frameworks in the application. I will go over the details in 5.

4.4 User Interface

As a final section of this chapter I will go over the design of 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 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 user can draw FA elements. After each individual 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 user to edit the name of the state. Thus, a text field in the center of the state will be shown. You can see example of a stroke that should be rendered as a state and how it looks like 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, user 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.

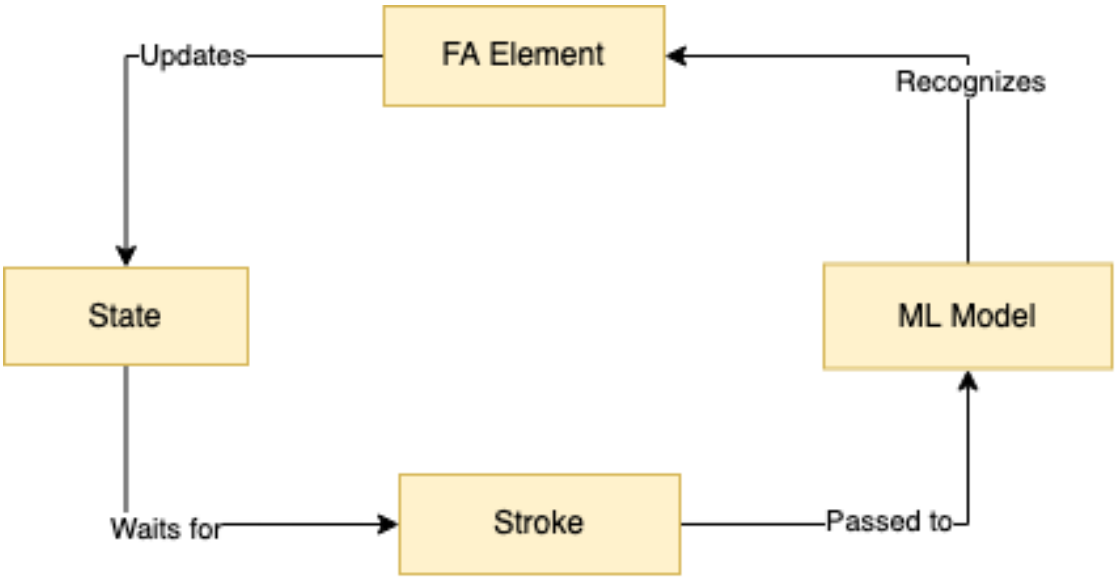
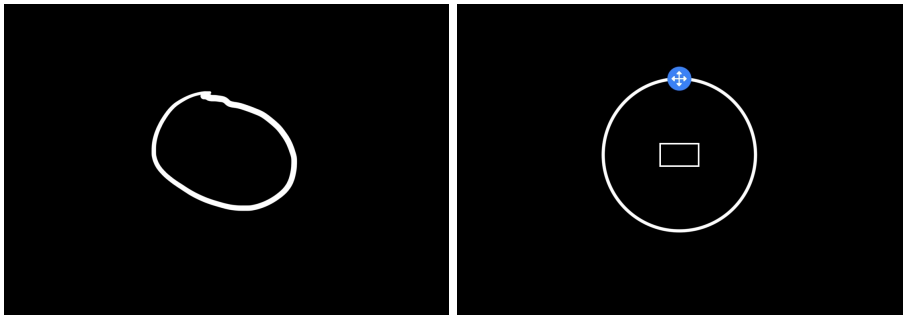
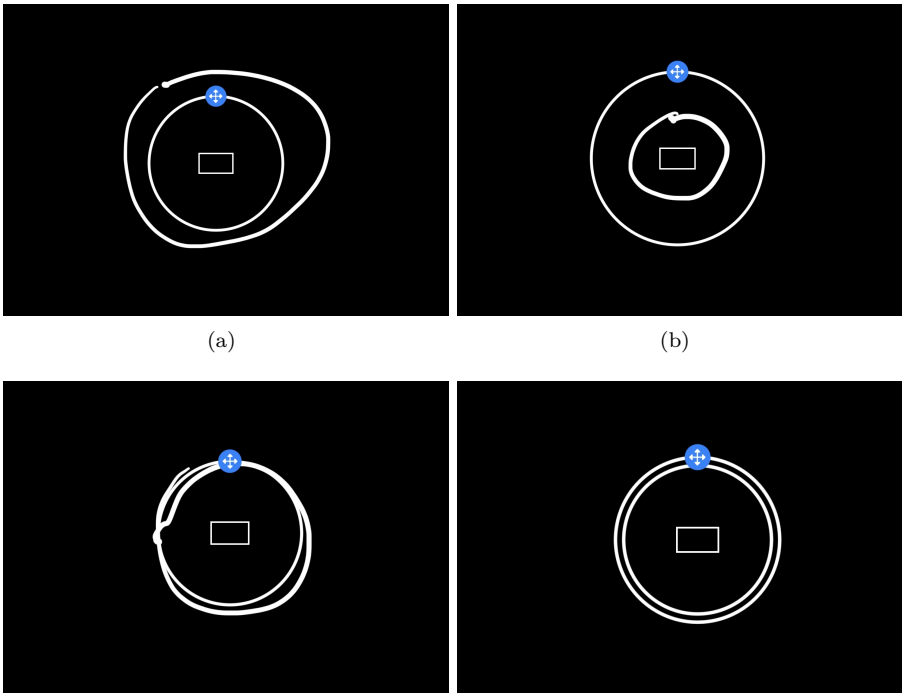


Figure 4.3 Flow of recognizing FA elements from strokes



(a) Stroke (b) Rendered state

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



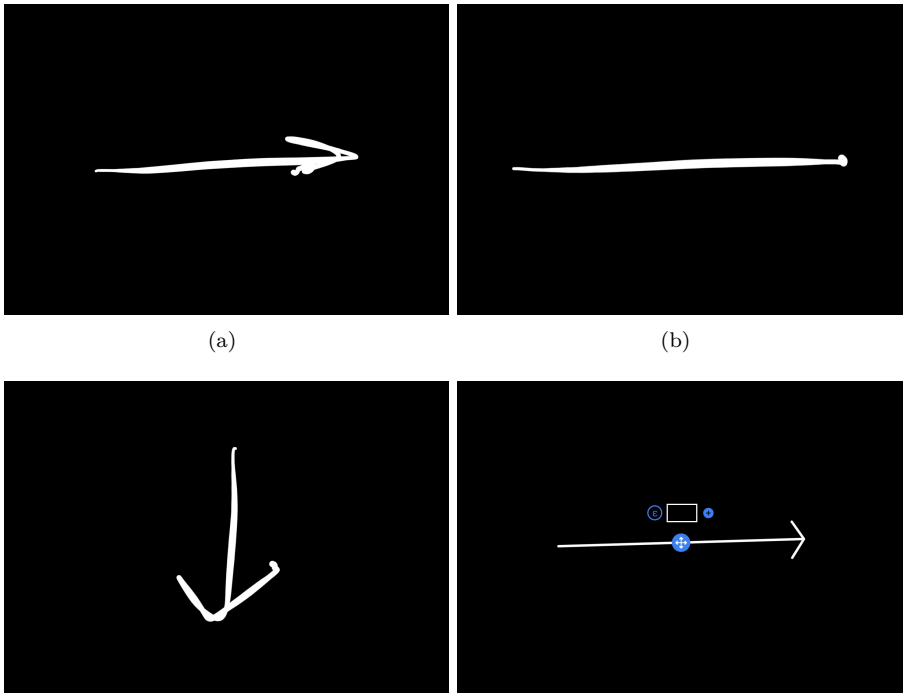
(a) (b)

(c) (d)

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

4.4.3 Transition

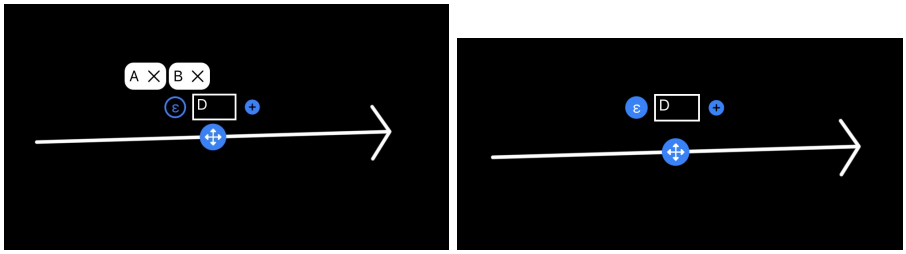
A transition is represented as a directed edge between vertices which is a shape of an arrow. The stroke can either be as an arrow or, more conveniently, just a straight line which is faster and easier to draw, especially when the shape must be drawn with a single stroke. Example strokes and a rendered 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 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 FA with $\epsilon - transitions$ there is also a button with ϵ. When it is tapped, it is added as another symbol for that particulat transitions. A transition with multiple symbols and with $\epsilon - transition$ 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.



(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) $\epsilon - transition$

Figure 4.7 Transitions with symbols

Chapter 5

Implementation

Chapter 6

User Testing

Chapter 7
Conclusion

7.1 Goals Assessment

Assess my goals.

7.2 Thesis Contribution

The contribution of this thesis is testing Algorithms Library Toolkit in practice and can now be pointed to for users who want to see its capabilities. The editor can now also be recommended in the course of BI-AAG at FIT CTU (and other universities) - for students and teachers alike.

7.3 Future Work

Algorithms Library Toolkit is an extensive library and there are still capabilites that are not implemented in the editor.

The choice of developing a native iOS application has resulted in good UX, but in the future it would be beneficial to broaden the possible audience and either develop a similar Android app or create a more ubiquituous web interface.

Chapter 8
List of Abbreviations

DFA	Deterministic Finite Automaton
FA	Finite Automaton
NFA	Nondeterministic Finite Automaton
ALT	Algorithms Library Toolkit
UI	User Interface

Příloha A

Nějaká příloha

Sem přijde to, co nepatří do hlavní části.

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. 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/.

41. POINT-FREE. *The Composable Architecture* [online] [visited on 2021-04-16]. Available from: https://github.com/pointfreeco/swift-composable-architecture/.

42. 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/.

43. *Interoperability between Swift and C++* [online] [visited on 2021-04-16]. Available from: https://github.com/apple/swift/blob/main/docs/CppInteroperabilityManifesto.md#exceptions/.

44. 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/.

45. 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/.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF