_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Predictive Text and Error Correction Using Weighted Finite State Transducers

Robin Staes

Tutor: Rafael Llobet Azpitarte

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València.

Curso 2015-16

Valencia, 24 de junio de 2016

Acknowledgments

I would like to thank my mother, my family and my friends for the endless support that they have given me during my studies and my Erasmus in Valencia. I also would like to thank my tutor, Mr. Llobet Azpitarte, at the polytechnic university of Valencia for the guidance and support while completing my thesis at the polytechnic university of Valencia.

Abstract
This thesis was carried out in the department of ETSIT and ETSINF at the “Universidad Politècnica de Valencia” (UPV). With growing demand of mobile devices and more efficient work environments, every aspect of the interaction with those devices and environments should be evaluated and if possibly, improved. The aim of this project is to research and implement efficient prediction and correction of the inserted text by using Weighted Finite State Transducers.

The project is comprised of three parts. The C/C++ program that shows the concept and use of the C/C++ OpenFST library. The Java program and Java library that allows the use of the text prediction and error correction system on vitrually any operating system and device due to the nature of Java. And finally the Android app that is an implementation of the concept and uses the previously created Java library.

Keywords: text prediction, error correction, Weighted Finite State Transducers, machine learning, C/C++, Java, Android, OpenFST, jOpenFST, Linux, Windows

Resumen
Esta tesis se llevó a cabo en el departamento de la ETSIT y ETSINF en la Universidad Politécnica de Valencia (UPV). Con la creciente demanda de dispositivos móviles y entornos de trabajo más eficientes todos los aspectos de la interacción con los dispositivos y entornos deben ser evaluados y, si es posible, mejorados. El objetivo de este proyecto es investigar y poner en práctica la predicción eficiente y corrección del texto insertado mediante el uso de Transductores de Estados Finitos Estocásticos.

El proyecto se compone de tres partes. El programa en C / C ++ que muestra el concepto y el uso de la biblioteca de C/C ++ OpenFST. El programa Java que permite el uso del sistema en cualquier sistema operativo y el dispositivo debido a la naturaleza de Java. Y por último, la aplicación para Android que es una implementación del concepto y utiliza la biblioteca de Java creada con anterioridad.

Palabras clave: predicción de texto, corrección de errores, Transductores de Estados Finitos Estocásticos x, aprendizaje automático, C/C ++, Java, Android, OpenFST, jOpenFST, Linux, Windows

Acronyms and initialisms

FST	Finite State Transducer
FA	Finite state Automaton
FSM	Finite State Machine
WFST	Weighted Finite State Transducers
OpenFST	The open source C/C++ FST library
JopenFST	The open source Java FST library
OS	Operating System
FA	Finite Automaton
FSA	Finite State Automaton
LM	Language model
PM	Prefix model
EM	Error model
Distro	Distribution (i.e. Linux distribution)
UML	Unified Modelling Language
IME	Input Method Editor
XML	Extensible Markup Language

Index

1. Introduction

The purpose of this document is to provide a study about Finite State Transducers, Weighted Finite State Transducers and a new efficient way to interact with current applications and systems. The project explains how a language model can be constructed using WFSTs and how the proposed system can learn new words and adapt its structure to them.

The project consists of different libraries in different languages implemented in different concept programs to enable a developer to learn more about the used libraries.

The project will explain how to implement and use the different libraries in the different languages.

1.1. Motivation

Since I started studying informatics, I have always been very fond of programming and the different challenges it brings. It always gives me a great satisfaction to successfully finish very challenging and at first seemingly impossible project.

The processing of natural language is a vast technology that always have been very compelling and challenging. It has always been present in the technology landscape and has kept many people busy throughout the years. It is a subject that still can evolve and be improved.

Thus it is my wish to research FSTs and how to implement them to create a natural language processing system. The project that I have chosen will make it easier for other developers to research and implement a text prediction and error correction system in different languages and different system.

1.2. Problem

The main problem with the current systems and technological landscape is that we have not made all the functionality of one device possible on another. This functionality, that is most of the time useful and even wanted, is often overlooked. In this case the functionality in question is the text prediction and error correction that can be found on smartphones.

One of the biggest problems when interacting with small devices like smartphones and tablets is data entry like text. Keyboards are limited to the device's form factor. The input of said data is very inefficient and there are a lot of errors during the entry process. It is also difficult to interact with such a limited keyboard.

These problems are not as prominently present on bigger devices, like desktops, as on mobile devices. But their possible solution has the possibility to improve the efficiency of the user and are in general a welcome addition to the current technology.

This project has been created on the fact that this feature has not been integrated, implemented in a desktop operating system or any of its applications. This could improve the user's workflow and create a more efficient way of typing long words or even papers.

1.3. The proposed solution

The proposed solution is a couple of libraries and concept programs that enables a developer to integrate the text prediction and error correction logic into other programs and get a better idea how these systems can be used. These libraries need to enable the developer to take the user's input and process is in an efficient and manageable way.

The solution is two concept programs and an Android app, one written in C/C++ and another written in Java. They are meant to give an idea on how to implement the whole logic. From those a Java class library is written that enables the integration of the logic in other Java programs.

1.4. Project structure

The project consists of three parts. The first part is a concept that was created in C/C++ on a Linux based OS that provides a concept based on the OpenFST library. This library is a C/C++ library which is created for the sole purpose of creating Finite State Transducers and interacting with them. The library has many useful operations that can be used on these models to create a language system. This library was the foundation for the Weighted Finite State class and the different model classes, specifically: LanguageModel, ErrorModel and PrefixModel. Which describe the different models used to create the text prediction and error correction system.

The second part of the project is the Java program and library that has been based on the first part of the project and uses the JOpenFST library, a ported version of the OpenFST library. Although the JopenFST has some limitations and small problems compared to the original OpenFST library this was still the best and most useful option for this language.

The third part of the project is the Android application that shows a real-world mobile implementation. The previous parts have been focused on the desktop side of informatics but the whole reason that text prediction and error correction exists is because it is difficult to type on such a small device. This implementation and real-world example proves that the whole system is versatile and can be used on virtually any system that supports Java (Java virtual machine). The application uses the library created in the second part of the project.

1.5. Objectives

The goal of this project is to create a uniform and globally useful system and program that can be used over different Operating systems and different devices.

The objectives are as followed:

- Create a concept that can predict text
- Expand that concept to correct errors
- Use that concept to create real-world use cases and concept programs
- Enable the system’s ability to learn (machine learning)

2. Scheduling

The scheduling of a project is one of the most important aspects. This will decide whether a project has been completed efficiently and within the right time frame.

2.1. Project management

The project has been split up in multiple parts since this is easier to handle than a whole project at once. The three parts consist of:

1. C/C++ Concept program
   1.1. Which can be seen as a research project to get to know the library and how WFSTs work.
2. The Java concept application & library
   2.1. Which is desired for creating an actual application and enabling the use of the library on different platforms.
3. The Android application
   3.1. This app shows a possible way to use the created library and also depicts the multiple platform part.

Each of the different parts have many other smaller parts to make the tasks easier. This has provided an easy and manageable way to create a successful project.

2.2. Task Distribution

The distribution of the different tasks were straightforward. Each sub part should take no longer than a week and each greater part should be finished in one month and a half.

2.3. Time diagram

Note that this table represent the planned time and not the time actually spend on the different parts.

Table 1 Project time table

Date	State and description	Required time
7-3-2016	Project start	/
13-3-2016	OpenFST research finished	One week
20-3-2016	Algorithm and models defined	One week
31-3-2016	C/C++ concept program finished	A week and a half
1-4-2016	Start Java port	/
10-4-2016	Create Java library	4 days
17-4-2016	Java port done	Two weeks
18-4-2016	Start Java concept program	/
24-4-2016	Java concept program done	One week
30-4-2016	Java library done	One week
22-5-2016	Android app finished	Three weeks

3. Background

To be able to understand the whole system with its code and their respective implementations, a basis knowledge about transducers, Finite State Transducers and Weighted Finite State Transducers needs to be obtained.

The function of the system is to transform an input string into a valid output string. In this case the input is represented or coded by means of a prefix model, it is the model that represents the user's data entry. The set of valid inputs are represented by the language model. And the set of edit operations that are allowed in the transformation, insert and delete, are defined in the error model.

In this chapter the basis knowledge around Transducers, States, Arcs, FSTs and WFSTs will be explained.

3.1. Transducer

When a search for "Transducer definition" [1] [2] [3] is performed on the internet, the most hits include the following definition:

> *A device that is actuated by energy from one system and supplies energy usually in another form to a second system. For example, a microphone is a transducer that transforms sound energy into electrical signals.*

This definition comes from the electronics field where a transducer can convert energy signals. This same principle can be applied in coding and thus it can be transformed to more IT related definitions:

> *Transducers, a powerful and composable way to build algorithmic transformations which performs a transformation over a sequential process.*

So to put it in lemans terms, a transducer is an algorithm that can transform information, data over a sequential process independent of the source to a wanted output. The word "transducer" itself can be split into two parts that reflect this definition: "transform" — to produce some value from another — and "reducer" — to combine the values of a data structure to produce a new one.

Transducers are a useful way and fast way to process data in large systems. The fact that it is a general implementation makes it that more useful because it can be used and reused on a lot of different data types and formed to different use cases.

3.2. State

A state [4] [5] is a component of a finite state transducer that can have none, one or multiple arcs. It represents the current state of a path in a model. A state can have a weight and thus can be final or transitional. A transitional state is represented with the weight of 0 where as a transitional state is represented with the weight of ∞.

3.3. Arc

An arc [4] [6] is the transition between two states. It can have one or two symbols depending on the technology used. In the case of finite state transducers an arc will have an input and an output symbol to represent the transition between one and the other. An arc can also have a weight or a frequency which indicates the cost of the transition or the amount of times the transition has been used.

3.4. Semiring

A semiring [6] [7] [8] [4] [9] is specified by two binary operations $\oplus$ and $\otimes$ and two designated elements 0 and 1. A semiring (K, $\oplus$, $\otimes$, 0, 1) is a ring that may lack negation. A semiring has the following properties:

- $\oplus$: associative, commutative and has 0 as its identity. It is used to compute the weight of a sequence (sum of the weights of the paths labelled with that sequence).
- $\otimes$: associative and has identity 1, distributes with respect to $\oplus$, and has 0 as an annihilator: $0 \otimes a = a \otimes 0 = 0$. It is used to compute the weight of a path (product of the weights of constituent transitions).

A left semiring distributes on the left; a right semiring distributes on the right. Table 2 displays all the possible semirings.

Table 2 Semiring summary

SEMIRING	SET	$\oplus$	$\otimes$	$\bar{0}$	$\bar{1}$
BOOLEAN	{0,1}	$\vee$	$\wedge$	0	1
PROBABILITY	$\mathbb{R}_+$	+	×	0	1
LOG	$\mathbb{R} \cup \{-\infty, +\infty\}$	$\oplus_{log}$	+	$+\infty$	0
TROPICAL	$\mathbb{R} \cup \{-\infty, +\infty\}$	min	+	$+\infty$	0
STRING	$\Sigma^* \cup \{\infty\}$	$\wedge$	.	∞	ϵ

$\oplus_{log}$ is defined by: $x \oplus_{log} y = -\log(e^{-x} + e^{-y})$ and $\wedge$ is the longest common prefix. The string semiring is a left semiring.

In this project all the models use the tropical semiring WFST $(K, \oplus, \otimes, \bar{0}, \bar{1})$ where K are negative log probabilities, $\oplus$ is the min operation, $\otimes$ is +, $\bar{0}$ is $+\infty$ and $\bar{1}$ is 0. This avoids underflow problems which would be encountered when working with probabilities. Therefore, the most probable path will be found using a lowest cost path search.

3.5. Finite State Transducers

FSTs [4] [8] [10] [5] have key applications in speech recognition and synthesis, machine translation, optical character recognition, pattern matching, string processing, machine learning, information extraction and retrieval among others. This project uses finite state transducers at its core and is thus a very important part of the project.

A Finite State Transducer (FST) is a Finite State Machine (FSM) which has transitions labelled with two symbols. One for the input and one for the output. This is different from the ordinary finite state automaton (FSA) because this has only one tape. A FST is an adaptation to the FA. Instead of just using one set of symbols, the FST maps between two sets of symbols, the input and the output symbols. And it is more general than a FSA which defines a language by defining a set of accepted strings. While a FST defines the relations between the strings in said set. An FST can be seen as a translator or a relater between strings in a set.

The figures below show the workings of an FST in a graphical way. The FST shown in the examples use the same input a→c.

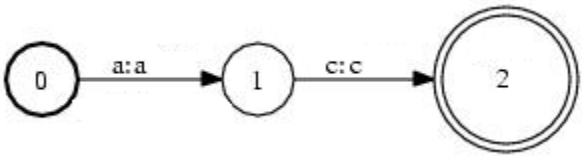


Figure 1 Simple input output FST
source: http://www.openfst.org/twiki/bin/view/FST/FstQuickTour

Figure 1 represents an FST that has the same input and output symbols, this is called an identity transducer. It is constructed of three states with 2 arcs between them. The arcs represent the transitions and the states the progress throughout the model.

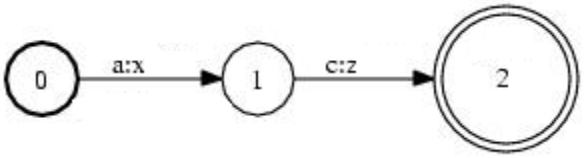


Figure 2 Different input output FST
source: http://www.openfst.org/twiki/bin/view/FST/FstQuickTour

Although at first glance Figure 2 looks the same as Figure 1, there is a small difference. The input symbols are mapped to different output symbols and thus displays the input and output trap more clearly.

3.6. Weighted Finite State Transducers

The previous chapters gave an explanation about what transducers are and what an implementation of transducers are, more specific FSTs. This chapter will explain what weighted finite state transducers are. These transducers are the final form of FSTs that will be used in the rest of the project.

WFSTs [6] [4] [8] have proven quite successful in many fields. This includes in particular written, spoken language processing and machine translating and pattern recognition. A WFST can be seen as an improved version of a Finite State Transducer. It allows us to add weights or frequencies to the different arcs and states which is useful when creating language patterns and models. These weight represent the cost for taking a certain path in the model. Using these weights and or frequencies, a determination of the shortest path or the path with the least cost can be determined.

Next to the weights and or frequencies in arcs we can also add a weight to a state to indicate if it is final or not. A final state will have a weight that is non-infinite while a state that is not final will have a weight of ∞. This means that when a WFST is being used, an iteration can be performed over states and their linking arcs until a state with a weight that is non-infinity and bigger than 0 is encountered. This state will indicate the end of the path that has been taken.

An example of a WFST:

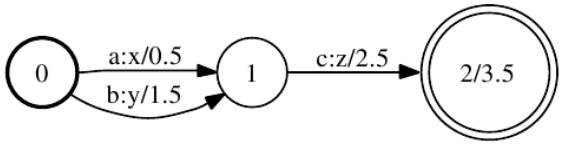


Figure 3 Full WFST
source: http://www.openfst.org/twiki/bin/view/FST/FstQuickTour

As shown in Figure 3, each state has its weight and its input and output symbol. This model transduces the input “ac” into “xz” with a cost of 3 and the input “bc” into “yz” with the cost of 4. When a determination of the shortest path or the path with the least cost would be made, the conclusion would be that the input “ac” resulting into “xz” is the least costly. This result and way of reasoning is very important for the rest of the project and is the key component in predicting text and correcting errors.

4. Methodology

To be able to use the OpenFST and jOpenFST library there was a need for some wrapper classes and classes that represents our models. Because a transducer can be used for many applications, there was a need to specify how the transducers, FSTs and WFSTs needed to be used. This means that the written classes needed to represent the natural language. And thus needed to represent the different models used in the prediction and correction of words. These are the Language model, the Error model and the Prefix model. These models are constructed in such a way that they have their own function and can be used together or separate to perform some tasks. The models are used in the following manner:

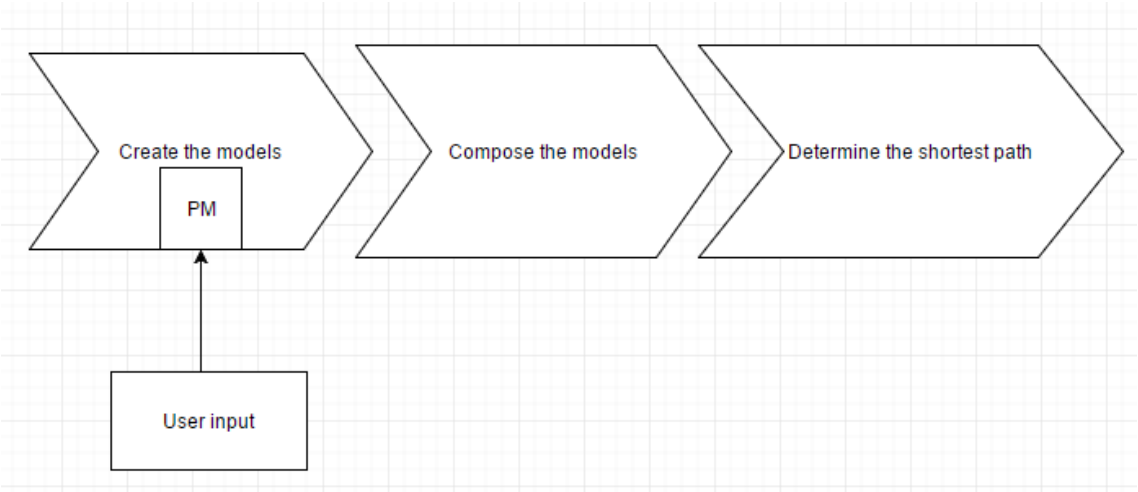


Figure 4 General prediction and error correction flow

First the models are being created, the prefix model has to be created using the users input since this represents the input word. Then the models are composed together to perform an error correction using the error model and a prediction using the language model. This results in an FST where the shortest path can be determined. The shortest path will then result into another FST where the output string can be extracted.

4.1. Language Model

The language model [8] [4] [11] is an identity transducer, same input and output symbol at each transition, which maps the different words from the word list with their respective weights. This can be seen as a mapping of the natural language used in the application. Therefore the name Language model.

Figure 5 displays a simple Language model.

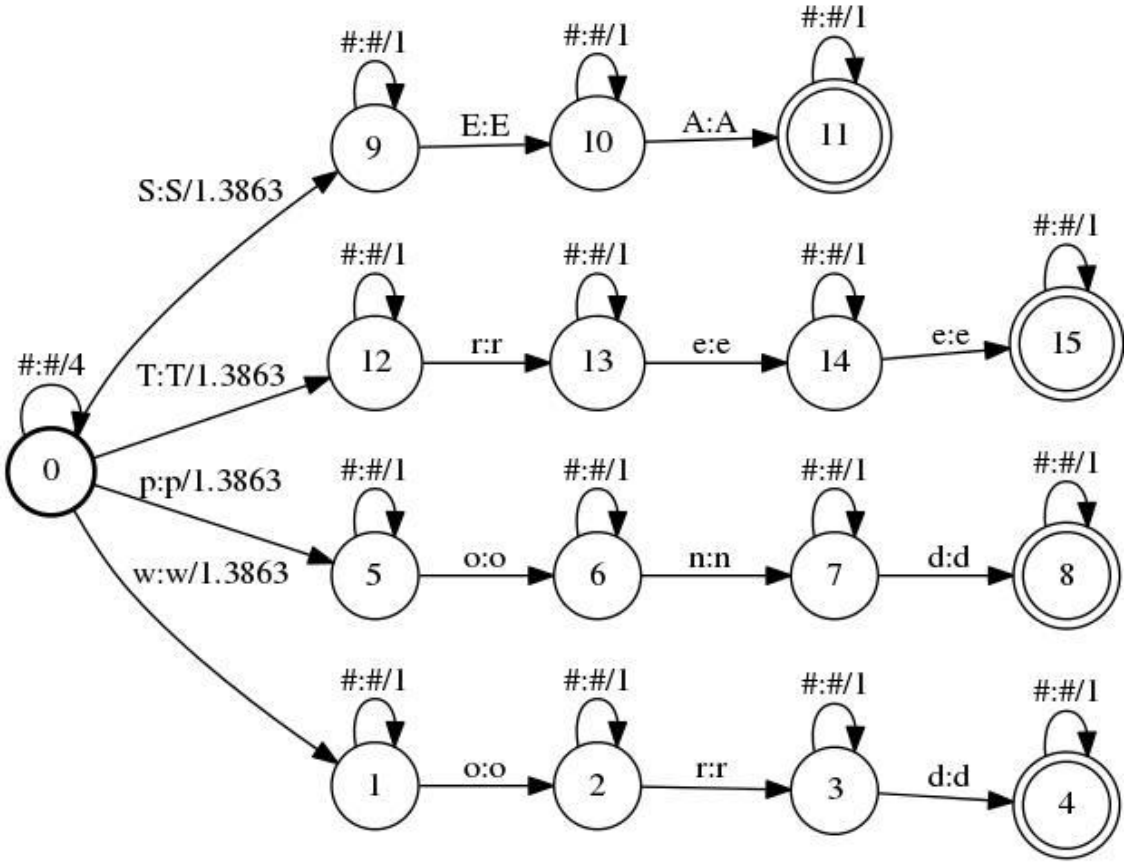


Figure 5 Simple LM

The model contains the following words:

- SEA
- Tree
- pond
- word

As seen in Figure 5, each word is connected using arcs and each arc has its respective weight. One of the big difficulties when creating a dynamic program is taking the previous results into account. Once the model is built, there is not enough information to reweight the affected paths when a new word is added.

That is why the states have an extra arc that is marked by the input and output label '#'. This arc, the dummy arc, is used by the application itself and is not being used by the OpenFST library. These dummy arcs are eliminated when composing the different models. The dummy arcs are used to keep track of the frequency of use of the states since once the model is built, each state can be used in different paths but for the program to calculate the exact weight, it needs to know how frequently a state can be passed when following a certain path. For example:

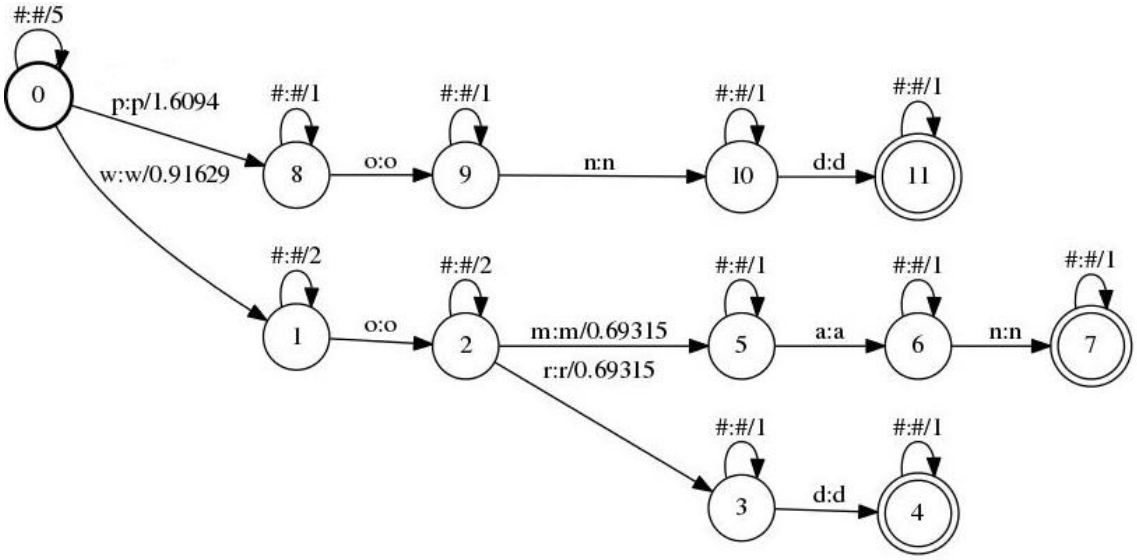


Figure 6 Three word LM

Figure 6 is a depiction of a Language model that has three words: pond, woman and word. Each state has the extra arc and its weight shows the frequency of the paths that pass through it. State 8's dummy arc has a frequency of 1 because there is only one path that goes through it. State 2's dummy arc has a frequency of 2 because the words "woman" and "word" go through that state.

As stated before, these dummy arcs are being used to calculate the exact weight (probability) of the arcs exiting that state. These weights are being calculated in the following matter:

Alphabet: S = {a, b, c, d… ε}

Symbol: S_i

Probability of S_i: P (S_i)

Frequency of S_i: F (S_i)

$$P(S_i) = \frac{F(Si)}{\sum_{S_j \in S} F(S_j)} \qquad T.F. = \sum_{S_j \in J} F(S_j)$$

From these formulas we can conclude that

$$P(S_i) = \frac{F(Si)}{TF} \quad \text{and} \quad F(S_i) = P(S_i) \times T.F.$$

4.2. Prefix Model

The prefix model is an identity transducer that maps the input from the user. This is the WFST that will be dynamically generated depending on the input of the user. The name prefix is used here because it is represents the user input so far, i.e. it represents the prefix of the text that a user has in mind to introduce. It is the starting point and from this model the output will be generated.

Figure 7 displays a prefix model:

Figure 7 Simple PM

The firsts states and arcs of the prefix model represent the symbols that the user has typed so far (i.e., the prefix of the desired word), while the arcs in the last state represent the set of symbols that can be appended to the given prefix

The prefix model can be described in two parts. The base or the first part represents the string that the user has inserted. It is a normal WFST that has default weights and X+1 states (where X = the amount of characters in the word). The first states and arcs represent the symbols that the user has typed so far, i.e. the prefix of the desired word.

The second part of the prefix model are the arcs on the final state. They represent the set of symbols that can be appended to the given prefix. This ensures that later on the correct and anticipated output will be generated.

4.3. Error Model

The error model is a WFST that maps symbol table. This means that it will create arcs for each symbol in the symbol table and each of its combination with each other symbol. It represents the possible edit operations: insertion, deletions and substitutions; including the substitution of a symbol by that same symbol.

An important symbol in the error matrix is "[]". This symbol represent nothing, it is the empty symbol. This symbol can be used to create an insertion of another symbol, for example []/s. It can also be used to delete a symbol, for example s/[]. An arc that has a transition with an input symbol s1 and a different output symbol s2 is called a substitution.

Figure 8 displays an error model:

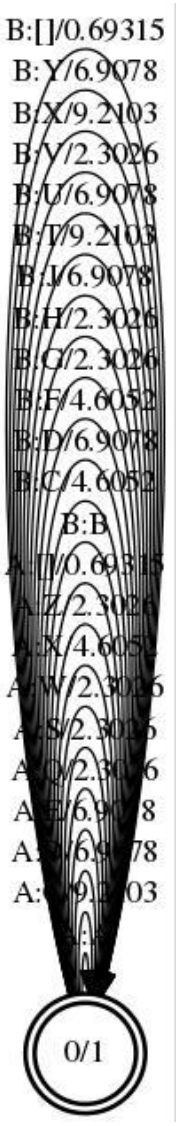


Figure 8 Simple EM

As stated earlier, the error model consists of a final state with arcs that ties every symbol to every other symbol with a specific weight. The weights are determined in the "errormatrix" file. These weights are mapped using a QWERTY keyboard layout. Each weight represents the distance between keys. That is, the probability of confusion between different symbols, considering that the probability of the confusion of two symbols increases as the distance of keys on a keyboard decreases.

4.4. Compose

The compose operation computes the composition of two transducers. If A transduces string *x* to *y* with weight *a*, and *B* transduces *y* to *z* with weight b_1, then their composition transduces string *x* to *z* with weight $a \otimes$ b.

The output labels of the first transducer or the input labels of the second transducer must be sorted. The weights need to form a commutative semiring, this is valid for the TropicalWeight and LogWeight for instance. The OpenFST library provides different versions of this operation which accept options that allow choosing the matcher, composition filter, state label and when delayed, the caching behaviour used by the composition.

A small example:

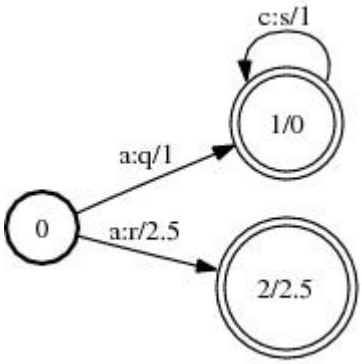


Figure 9 FST *a*
source: http://www.openfst.org/twiki/bin/view/FST/ComposeDoc

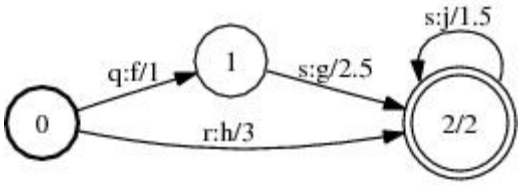


Figure 10 FST *b*
source: http://www.openfst.org/twiki/bin/view/FST/ComposeDoc

Considering FST *a* and FST *b*. Their composition would result in:

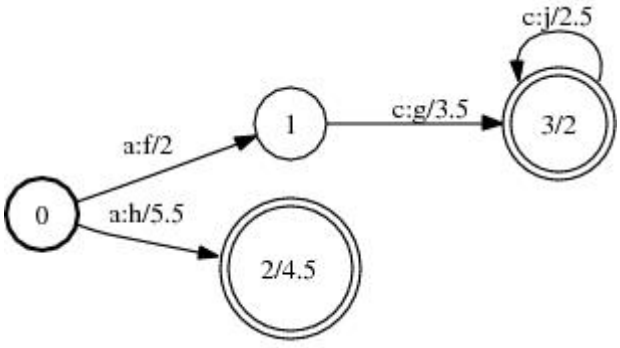


Figure 11 FST composition of *a* and *b*
source: http://www.openfst.org/twiki/bin/view/FST/ComposeDoc

As seen on the resulting FST, Figure 11, the different arcs are combined and matched via their input and output symbols. Note that if the models were switched in order, the resulting FST would be different since the output symbols from FST *a* are being used to match with the input symbols of FST *b*.

This example can be composed using the following OpenFST shell and C++ commands:

```
Compose(A, B, &C);
ComposeFst<Arc>(A, B);
fstcompose a.fst b.fst out.fst
```

4.4.1.Composing the PM, EM and LM

As stated before, the different models need to be composed to perform an error correction and a prediction starting from the user's input string. The composition is handled by the OpenFST and jOpenFST library.

The models need to be composed in a specific manner since this will affect how the output will be calculated.

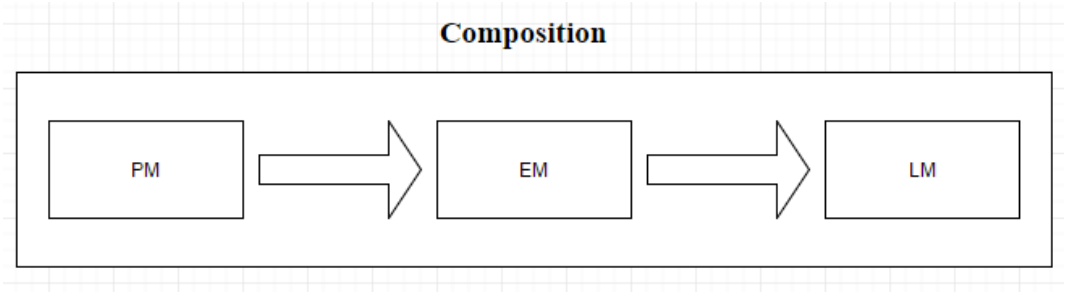


Figure 12 Composition diagram

First the EM and LM need to be generated.

Like explained earlier, the EM will use the keyboard mapping to generate an error model with the appropriate weights. The EM will look similar to Figure 13. Only it will have more arcs.

Figure 13 small EM

The LM will be generated using the word table. The generated LM will look like Figure 14. Note all the dummy arcs on the different states.

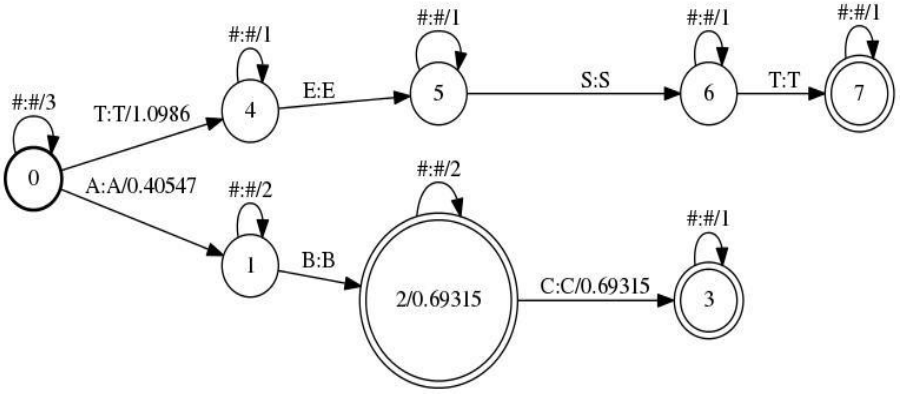


Figure 14 small LM

When these are generated, the PM needs to be generated whenever user input is present. This means that the application needs to regenerate the PM every time a user changes the input but the EM and LM will not be regenerated. The generated PM will look similar to Figure 15, only the last state will have more arcs. These arcs are needed to produce any suffix to the given prefix.

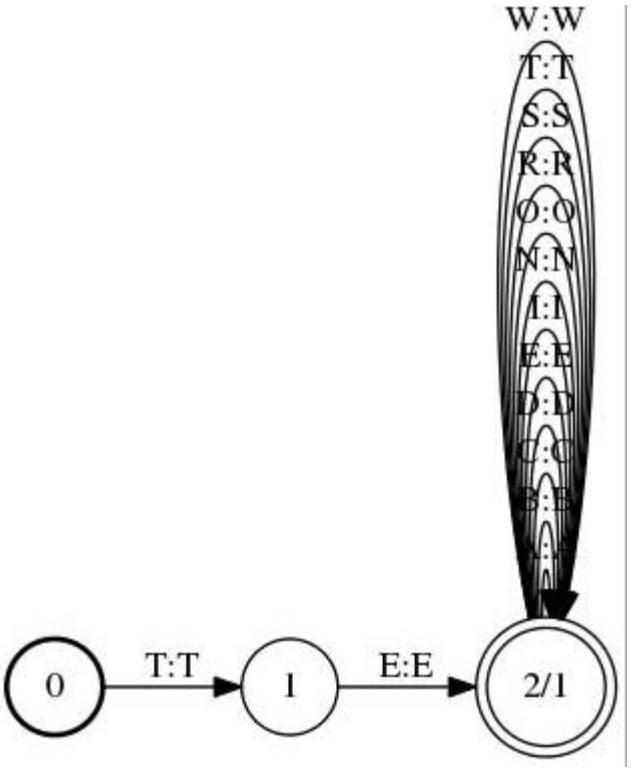


Figure 15 simple PM with reduced arcs

Then the PM needs to be composed with the EM. This will ensure that when the composition is made, the different operations of the EM can be applied. This will result in a new FST, Figure 16:

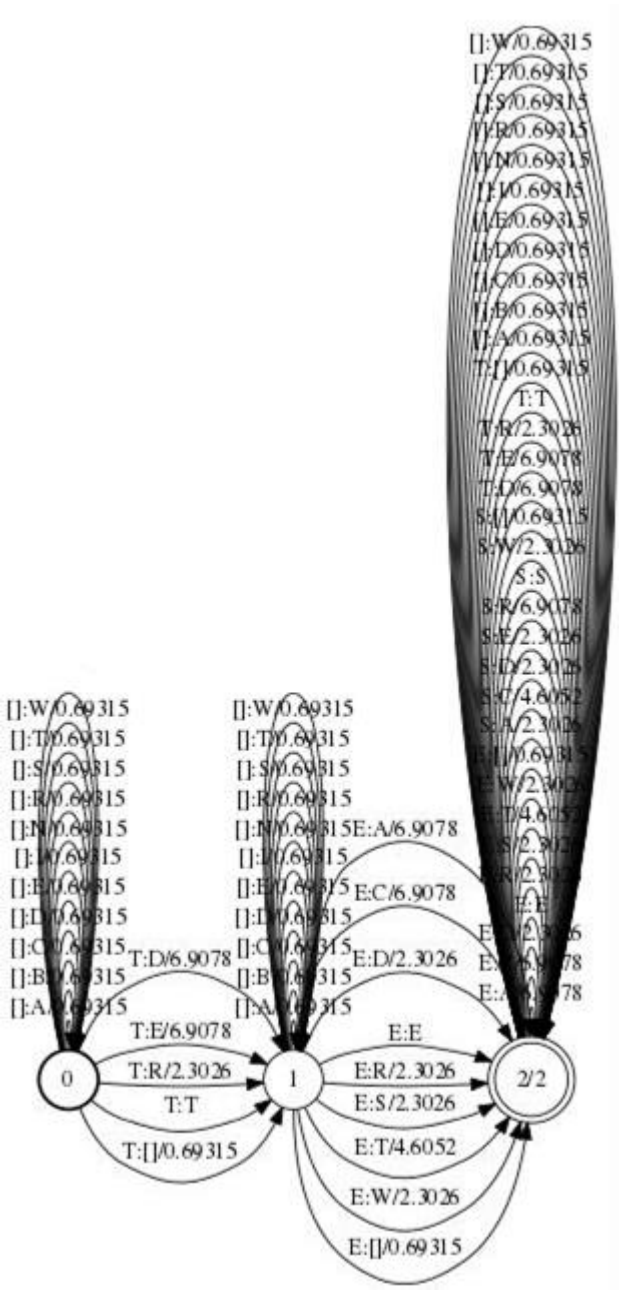


Figure 16 Composition of PM+EM

After the composition of the EM and PM, the result of that composition needs to be composed with the LM. This will result in a combined composition of the different models where the weights will be adapted.

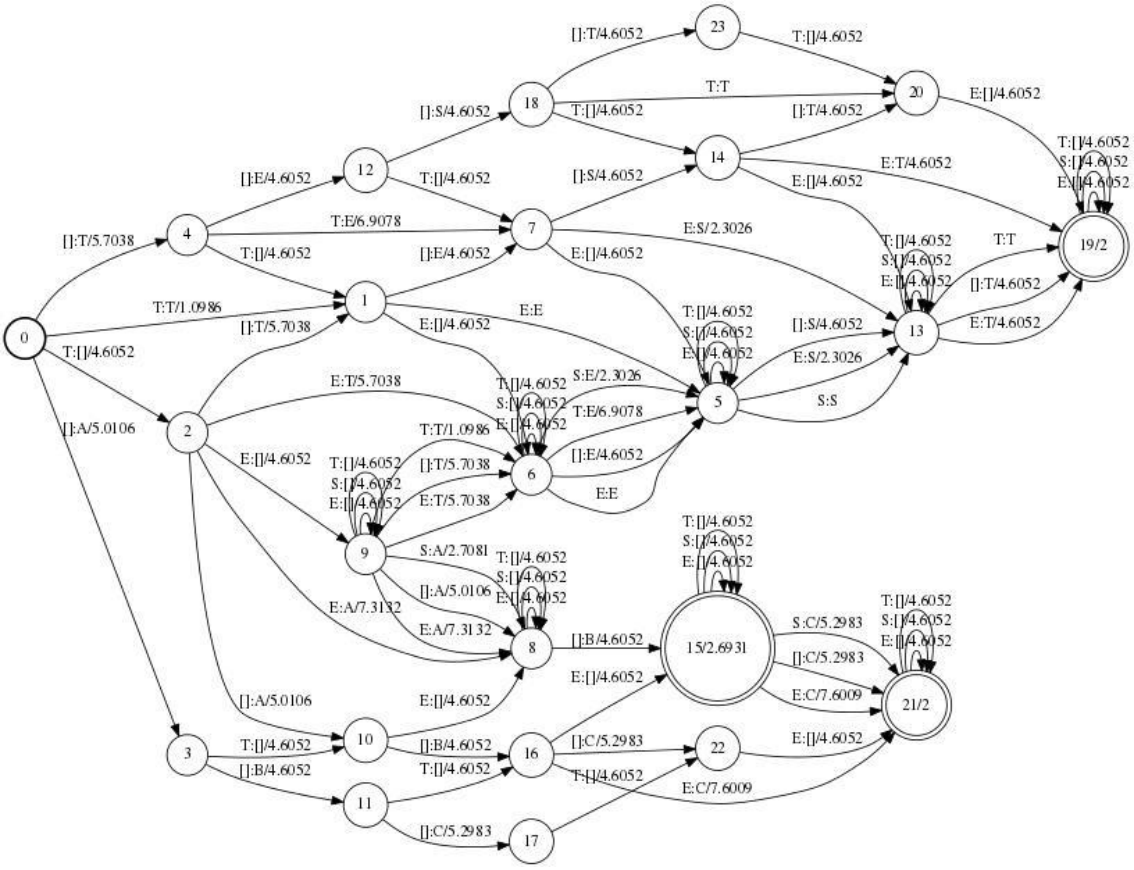


Figure 18 Composed model PM+EM+LM

After the composition of the different models, the shortest path can be determined to get the most likely transduction from the input string that has been coded in the prefix model into one of the output strings that has been coded in the language model, through the edit operations that were defined in the error model.

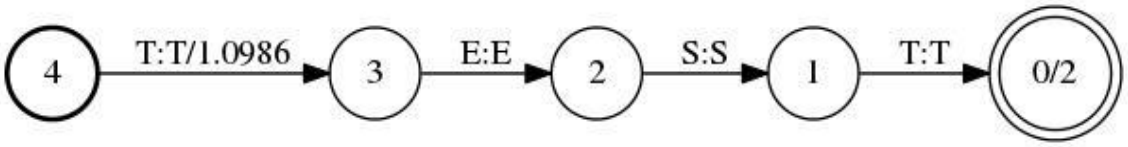


Figure 17 Resulting FST of the shortest path

4.4.2. Performance

The performance of the composition is very important in this project since the PM is generated on the fly by user input. The OpenFST library takes this into account when compiling by providing delayed composition.

The efficiency of composition can be strongly affected by several factors like:

- the operating system (Android, Linux, Windows)
- the use of the different libraries (OpenFST or jOpenFST)
- the choice of which transducer is sorted
    - sorting the FST with the greatest average out-degree
    - sorting both transducers allows composition to automatically select the best transducer to match against (per state pair)

      Note: stored sort properties of the FSTs are first checked in constant time followed by the minimum number of linear-time sort tests necessary to discover one sorted FST; thus composition may be unaware that both FSTs are sorted when those properties are not stored.

- the amount of non-determinism
- the presence and location of epsilon transitions - avoid epsilon transitions on the output side of the first transducer or the input side of the second transducer or prefer placing them later in a path since they delay matching and can introduce non-coaccessible states and transitions

Note: Compose and fstcompose trim their output, ComposeFst does not since it is a delayed operation. An FST is delayed (or lazy, on-the-fly, or dynamic) if the computation of its states and transitions occur only when requested. The complexity of a delayed FSTs constructor is constant-time, while the complexity of its traversal is a function of only those states and transitions that are visited. The OpenFST implements the delayed operation. It is not clear of the jOpenFST implement the same system since there is almost no documentation about the library.

Tests have suggested that the jOpenFST library does not implement delayed operations.

5. Implementation

This chapter will explain how the different parts of the project were developed using the different technologies and how they all work. It will also explain all the used libraries and their implementation and role in the project.

In order to deal with the proposed problems, the problems had to be divided in different sub problems. These sub-problems were then used to divide the project in sub-projects to be able to manage and solve a problem at a time.

The project has been divided in three parts. This made everything clear and manageable. The first part is the C/C++ concept program that is used to get an insight on how transducers work and how they can be manipulated and implemented in a Linux environment using the C/C++ language.

The second part is the Java concept program that shows a real world application of the prediction and error correction system. It shows the advantages and disadvantages of the system in a more user friendly and a graphical way. This enabled the creation of a Java library that enabled the third part of the project.

Lastly the third part is an Android application that is another real world application. It shows how the system can be used on different platforms and how they all tie together to enable a better user experience.

In Annex A is a list and short description of every library used in these concept programs. The next chapters describe the concept programs using these libraries.

5.1. Part One: C/C++

The C/C++ concept program has been developed on the Linux system Mint, using the text editor Sublime Text. The compilation was done by g++. The installation and the inner workings of the OpenFST library will be explained in further detail in the chapter OpenFST [12] [13] [14] [15] [16].

```
(1)Text prediction
(2)Add a word to the LM
(3)Save the created models to disk
(4)Print the word table
(5)Print the symbol table
(6)Load the LanguageModel fst file (make sure the file is present in the data ma
p)
(7)Settings
(esc)Exit
1

Text prediction:
insert your word
eor
compose the different models
Compose PM with EM
Compose PM+EM with LM
Perform ShortestPath in PM+EM+LM
saving the resulting FST...
Did you mean: word
The inserted word doesn't seem to exist in the LM

Do you want to add eor to the LM? press <1> to add or any other key to cancel
```

Figure 19 C/C++ concept program

5.1.1.OpenFST

OpenFST is a library for constructing, combining, optimizing, and searching weighted finite-state *transducers* (WFSTs). This library was developed by contributors from Google Research and NYU's Courant Institute. It is intended to be comprehensive, flexible, efficient and scale well to large problems. It has been extensively tested. It is an open source project distributed under the Apache license.

5.1.1.1. Installation

The OpenFST library is a C/C++ library that was meant to run on a Linux distribution because all the, limited, information gathered around the library indicated that the main used Operating system is a Linux based one. It can also be used on a Windows operating system but this requires the compilation of the project in Visual Studio which was not all too successful when tried.

The OpenFST library needs to be installed using the instructions on the OpenFST website. This basically explains that we need to compile the OpenFST library on the machine and copy them to the shell global variables so it can be used in the shell and by programs on the whole system.

After the installation of OpenFST it is also recommended to install the following programs:

- Graphviz
- pdfcrop
- ghostscript
- pdfedit

Installing these additional programs will make it able to create a visual representation of the created models using the OpenFST library. This makes debugging and in general getting a better idea of the models much easier. It is possible to convert the created binary models to an image, pdf and text format.

E.g.:

```
#!/bin/bash

FST=$1

# Draw
fstdraw --portrait=true $FST.fst |
dot -Tjpg > $FST.jpg

# Convert to PDF
ps2pdf14 $FST.ps /tmp/$FST.pdf
pdfcrop /tmp/$FST.pdf $FST.pdf
```

This bash script will convert a FST given the FST name to its image and pdf representation.

5.1.1.2. Utilization

Although at first it might seem a difficult task to use the library, in fact it is not that complicated. A good knowledge of C/C++ programming should be enough to understand and implement the library.

The example code will be based upon the following FST:

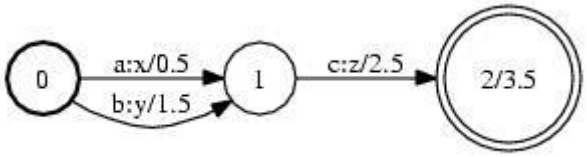


Figure 20 Simple FST

The initial state is label 0, there can only be one initial state. The final state is state no. 2 with a final weight of 3.5. As written earlier, a final state can only be final if the weight is non-infinite. There is an arc, transition from state 0 to 1 with input label a, output label x and a weight of 0.5. The FST can transduce the following path: “ac” to “xz” with a total weight of 6.5 (the sum of the arc and final weights). Note: this assumes the default weight type for this description.

FSTs can be created with constructors and mutators from C++ or from text files at the shell-level. Both ways will be explained.

5.1.1.2.1. C++

Creating an FST in C++:

```
// A vector FST is a general mutable FST
StdVectorFst fst;

// Adds state 0 to the initially empty FST and make it the start state.
fst.AddState();   // 1st state will be state 0 (returned by AddState)
fst.SetStart(0);  // arg is state ID

// Adds two arcs exiting state 0.
// Arc constructor args: ilabel, olabel, weight, dest state ID.
fst.AddArc(0, StdArc(1, 1, 0.5, 1));  // 1st arg is src state ID
fst.AddArc(0, StdArc(2, 2, 1.5, 1));

// Adds state 1 and its arc.
fst.AddState();
fst.AddArc(1, StdArc(3, 3, 2.5, 2));

// Adds state 2 and set its final weight.
fst.AddState();
fst.SetFinal(2, 3.5);  // 1st arg is state ID, 2nd arg weight
```

We can save this FST to a binary file with:

```
fst.Write("binary.fst");
```

5.1.1.2.2. Shell

One of the previously mentioned output formats for FSTs are Text files. Such a file can be used in the shell when creating and handling FSTs. It used the AT&T FSM format.

Such a text file can be created in the shell by using the following command:

```
# arc format: src dest ilabel olabel [weight]
# final state format: state [weight]
# lines may occur in any order except initial state must be first line
# unspecified weights default to 0.0 (for the library-default Weight type)
$ cat >text.fst <<EOF
0 1 a x .5
0 1 b y 1.5
1 2 c z 2.5
2 3.5
EOF
```

The internal representation of an arc label is an Integer and thus there must be a mapping provided between the symbols and their corresponding integers with a symbol table file that is also formatted in AT&T format.

```
$ cat >isyms.txt <<EOF
<eps> 0
a 1
b 2
c 3
EOF

$ cat >osyms.txt <<EOF
<eps> 0
x 1
y 2
z 3
EOF
```

Any string can be used as a label and any non-negative integer for the label ID. The zero label Id has been reserved for the epsilon (empty string) label. In this table 0 has been included even though it is not used in the given example. It is generally a good practice to include a symbol for it.

Before this FST can be used by the OpenFST library, it must be converted into a binary FST file.

```
# Creates binary Fst from text file.
# The symbolic labels will be converted into integers using the symbol table files.
$ fstcompile --isymbols=isyms.txt --osymbols=osyms.txt text.fst binary.fst

# As above but the symbol tables are stored with the FST.
$ fstcompile --isymbols=isyms.txt --osymbols=osyms.txt --keep_isymbols --keep_osymbols text.fst
binary.fst
```

Once this binary file is created, it can be used with the other shell-level programs (using the same machine architecture). It can be loaded into C++ using the following command

```
StdFst *fst = StdFst::Read("binary.fst");
```

5.1.2.The C/C++ concept program

The C/C++ concept program is a terminal application that enables the user to create and interact with WFSTs to perform a text prediction and/or error correction on the inserted word. The three WFSTs that are being used by the program are the language model, the error model and the prefix model. These model each present the different parts of the prediction and correction mechanism. The next chapters will explain the C/C++ concept program and its classes in more detail. The full UML diagram of the concept program can be found in appendix B1.

5.1.2.1. How does it work

The concept program makes use of different classes to interact with the different WFSTs. The different classes allows the program to create the different models and generate a prediction and perform a correction.

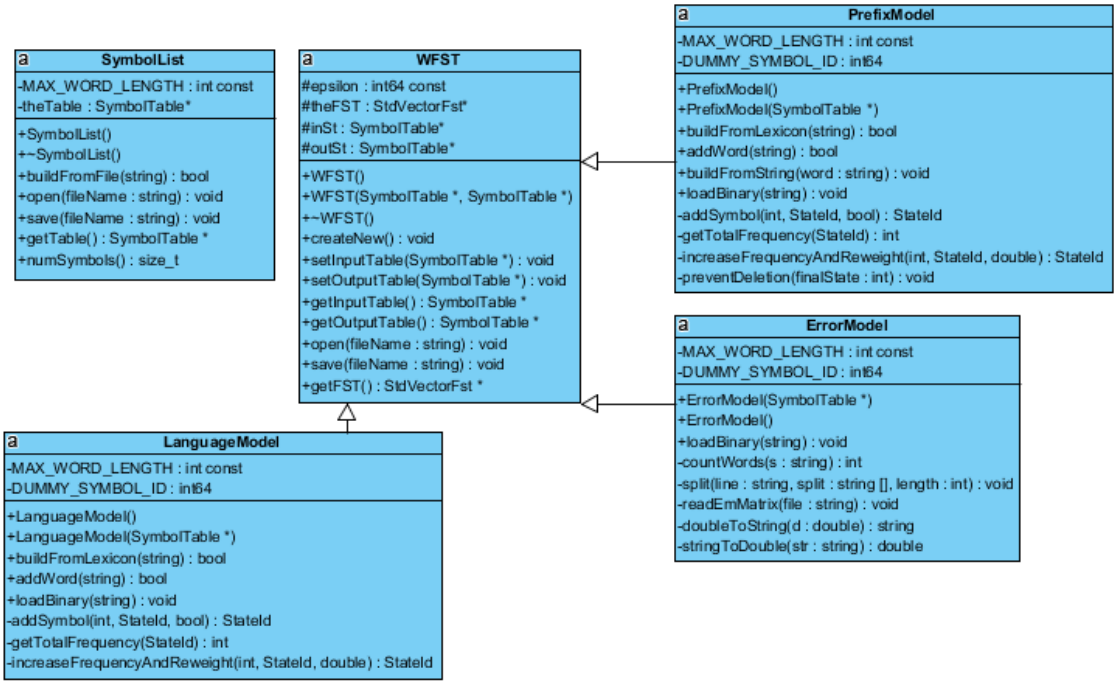


Figure 21 C/C++ UML class diagram

The different classes represent the different models. These models are then composed using different steps. First the prefix model will be composed with the error model, then the combined model of the prefix model and the error model are composed with the language model. This results in a WFST with the prediction and correction applied. After this the NshortestPath can be applied to determine the path with the least cost in the WFST. This will also result in a WFST from which the output can be extracted.

5.1.2.2. WFST

The WFST class is a class that is used as a wrapper for the StdVectorFST class in OpenFST. The WFST class has an FST as a variable and allows for advanced interaction with the FST. Note that the FST is actually a WFST since it uses weights. The WFST class has multiple functions that can be seen as helper functions for the FST to provide extra functionality.

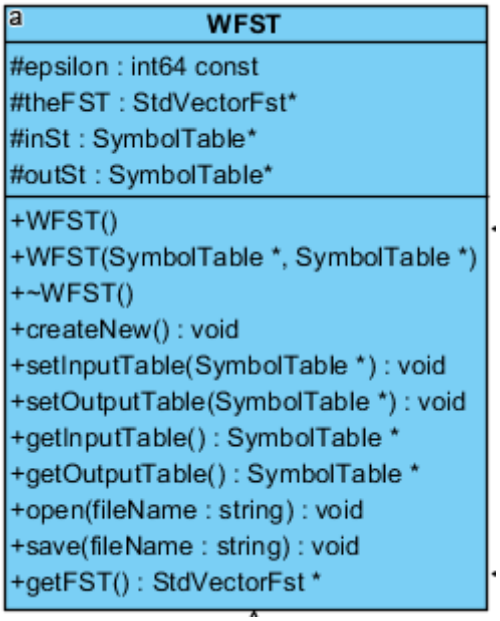


Figure 22 WFST UML class diagram

The WFST class consists of some getters and setters like the setInputTable and getInputTable but also provides functions to open and save binary FSTs. The function createNew initializes the FST with the previously defined SymbolTables.

The deconstructor which is indicated with "~" in C/C++ will make sure that the FST is deleted from memory when an instance of the class is destroyed.

5.1.2.3. LanguageModel

Each of the model classes are similar. They were all based on the language model class. The class consist of different functions to build and interact with the LM.

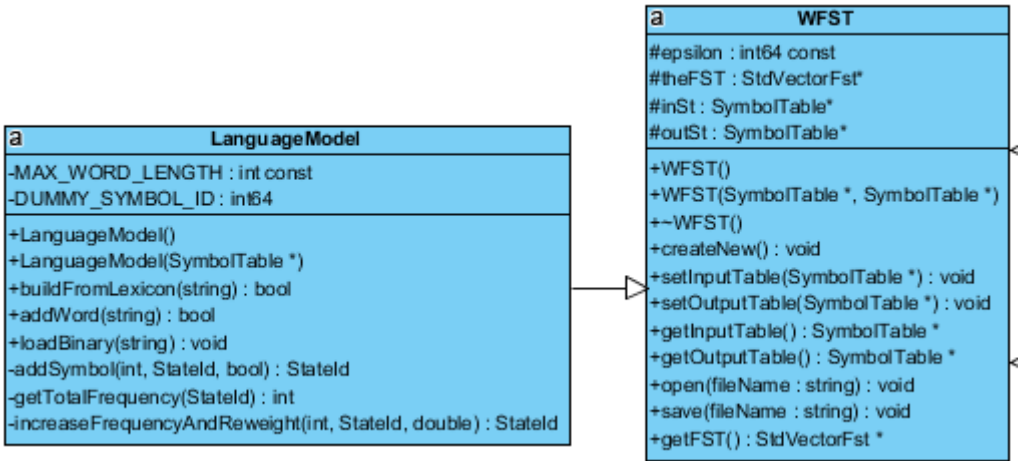


Figure 23 LM UML class diagram

The language model inherits from the WFST class and thus also has access to the same functions.

The main used function in the language model is the buildFromLexicon function. It will create the language model by reading the words in a word table. It does this by making use of the addWord function which adds a word to the language model using the addSymbol function which adds a symbol using an arc in between of states.

An example of a language model can be found below. The word table consists of three words:

- TEST
- WHY
- WORD

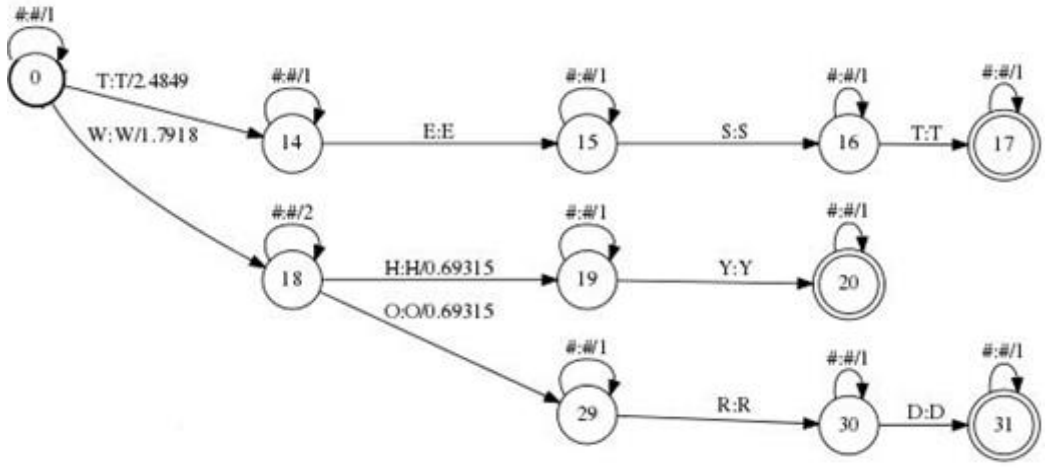


Figure 24 Simple LM example

Notice the extra dummy arcs on each state. These arcs hold information about the frequency.

Using these arcs, the probability and frequency can be calculated using the following formulas:

$$\text{Probability} = e^{-arc\ weight}$$

$$\text{Frequency} = \text{probability} * \text{total frequency}$$

Then the weight of the arc can be calculated using the following formula.

$$\text{Arc weight} = -\log\left(\frac{frequency}{(total\ frequency+1)}\right)$$

5.1.2.4. PrefixModel

The prefix model, as stated earlier is similar to the language model but still quite different. The prefix model consist of only one word with default weights and no dummy arcs.

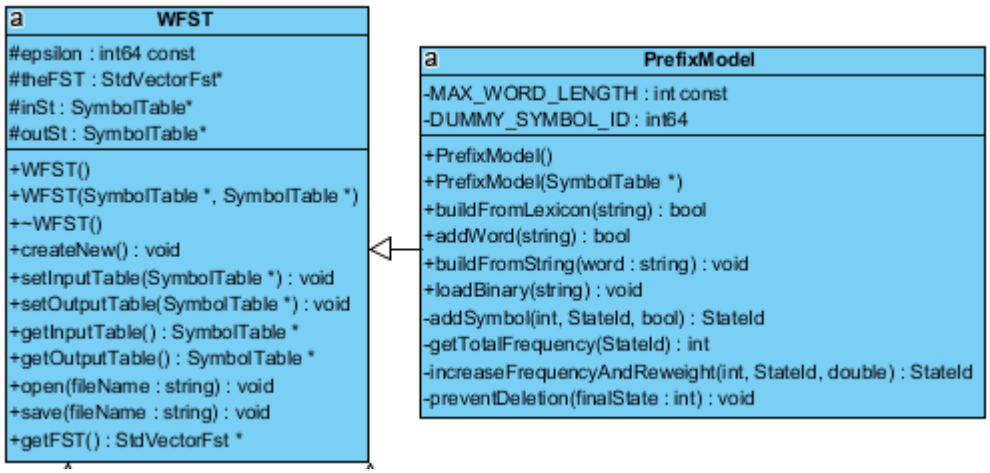


Figure 25 PM UML class diagram

The prefix model contains the same functions as the language model but they are implemented in a slightly different way. The functions are focused on handling one word and don not use the dummy arcs. These are not needed in the prefix model since there no words that needs to be added to the model. The prefix model has to be regenerated whenever a new prefix is used. This means that the prefix model code has to be as efficient as possible.

An example with the prefix “TE”, this could be a possible input when using the program.

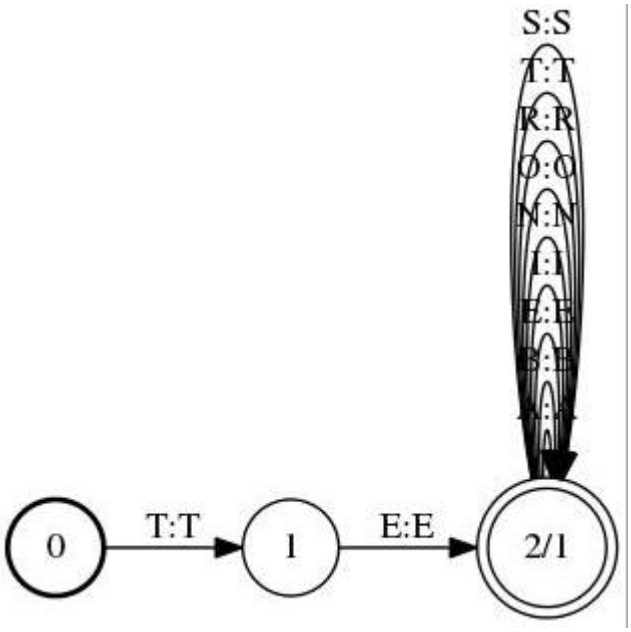


Figure 26 simple PM example

The model has been constructed out of three states, X+1 (where X = the amount of characters in a prefix), where each arc that connects the states have a default weight of zero and a probability of one.

Also note that the last state has a great amount of arcs. In a realistic situation the amount of arcs would be higher.

The arcs allow the composition to produce any word starting with “TE” and followed by zero, one or more symbols in the symbol table, {S, T, R, O, N, I, E, B, A}.

Note: these arcs are created by the preventDeletion function. Due to the nature of the composition of models, the OpenFST library will delete, replace and/or insert one or more characters just to make sure that there is a match with the least cost.

To prevent this unwanted behaviour each symbol in the symbol table needs to be added as an arc to the final state. Note that this is a one on one symbol mapping this means that it has the same input and output symbol. Some symbols need to be excluded because they are not necessary. These symbols are the epsilon (empty) symbol and the dummy symbol.

5.1.2.5. ErrorModel

The error model is a straightforward class that creates one state and adds arcs with input and output symbol combinations. This will enable the application to correct errors that were made by the user.

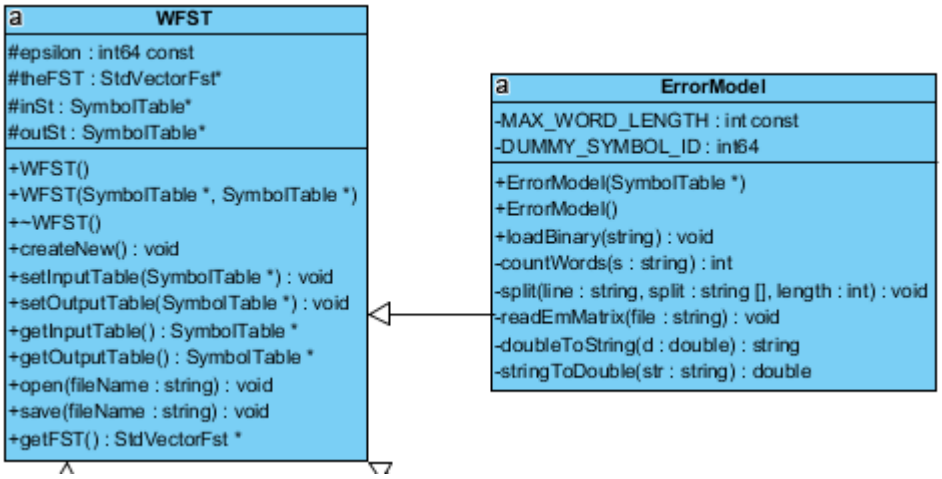


Figure 27 EM UML class diagram

Like every model class the error model inherits from the WFST class.

When creating an object of the ErrorModel class, the error matrix will be read and handled. This is a text file where every key on a QWERTY keyboard has been mapped with their corresponding weight, depending on the position of each key. The mapping is very important because they allow the application to correct each mistake that has been made in the prefix model depending on the user’s input and keypresses. The closer a key is to a certain letter the more likely that key was mistaken for that certain letter. This mapping can be of course be substituted for a different mapping depending on the preferences and keyboard layouts that are being used.

The first parts of the file consists of weight definitions. This allows a developer or user to adapt the error mapping and change each weight as desired.

CFM_SMOOTH_FACTOR 0

W1 1

W2 0.1

W3 0.01

W4 0.001

W5 0.0001

W6 0.00001

EP 0.01

The smooth factor is an option that has not been included in this project. The smooth factor allows for smoothing out the mapping when the probability is being calculated, i.e. a tolerance when calculating the probability.

Below the smooth factor are the definitions of the weights. There are 6 weights defined for a keyboard key position of up to 5 keys apart. This means that W1 stands for the key itself and W6 stands for a key that is in a 5 key radius around W1.

EP stands for epsilon, the empty symbol and defines the weight for the symbol.

The epsilon symbol is matched with each other symbol in each possible combination to provide the different operations. 'A' symbol deletion which is marked by the character and the epsilon symbol, for example: "A eps EP". In this case the epsilon symbol can be considered as the substitution for a missed key press that is transduced into a deletion.

The second part of the file consist of the mapping of each key on the QWERTY keyboard.

A A W1

A C W5

A D W4

A E W4

A Q W2

A S W2

A W W2

A X W3

A Z W2

A eps EP

The above is a depiction of the mapping of the key 'A'. Each line represent a key and its possible keypress with the probability of that keypress.

For example:

When a user presses the A-key, the probability that he meant to press the 'Z'-key is 0.1 .If the user pressed the 'A'-key again, the probability that he meant to press the 'C'-key would be 0.0001.

At the end of the A-series we can find “A eps EP”. This is the mapping for the epsilon symbol which means that we can delete the ‘A’ because the user did not mean to press the key. This probability is always 0.01 unless defined differently in the error matrix text file.

This sort of mapping goes on for every key on the QWERTY keyboard.

The last part of the error matrix text file is the epsilon mapping. The mapping of epsilon allows the application to insert certain characters into the other models. This is done for when the user forgets to insert certain characters

eps A EP

eps B EP

eps C EP

eps U EP

.

.

.

eps X EP

eps Y EP

eps Z EP

The weight of each arc between the states are calculated using the following formula:

$$\text{Weight} = -\log(\text{probability})$$

The probability is deducted from the earlier mentioned configuration file.

5.1.2.6. SymbolList

The SymbolList class is a class that has an OpenFST SymbolTable variable and provides extra functionality to interact with that variable.

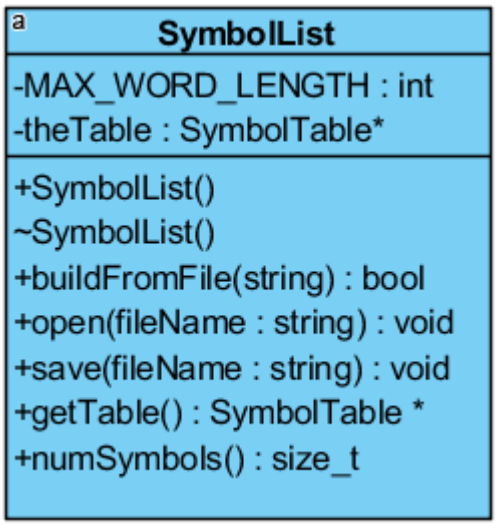


Figure 28 SymbolList UML class diagram

The class allows for creation of the SymbolTable using a text file. It is also able to read and save binary SymbolTable files.

5.2. Part Two: Java

The Java program [16] [10] [9] [17] has been developed on a Windows 8.1 system using NetBeans. This made it possible to create a concept program fast and eventually create a library with which a developer can easily create JopenFST applications. The Java concept application and the library are constructed in the same way as the C/C++ concept program. These programs will contain a lot of similarities.

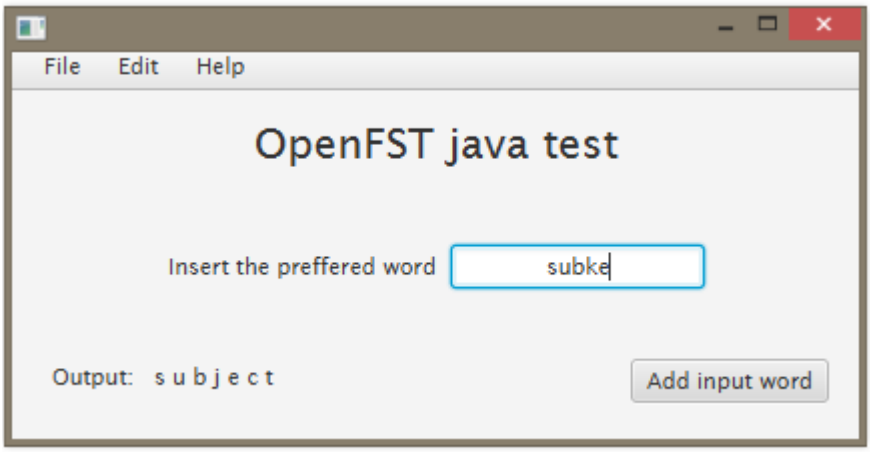


Figure 29 Java concept program

5.2.1.4.1.2 JOpenFST

5.2.1.1. Installation

JopenFST is the ported version of the OpenFST C/C++ library. This ported version is not completely the same as the OpenFST library and has its limitations. The installation of this library cannot really be seen as an installation but more like an import of the library into the NetBeans programming environment. Like every class library in Java, the jOpenFST library is a .JAR file that can be used in any Java program.

5.2.1.2. Utilization

The JopenFST can only be used in code since it is a Java library. The utilization of the library is like any other library but it is quite different from the C/C++ OpenFST library.

```
//create a FST object
Fst fs = new Fst();

//set the input and output symbols
fs.setIsyms(symbol.getSymbolsString());
fs.setOsyms(symbol.getSymbolsString());
```

Figure 30 Java code FST declaration and initialization

First of all a FST object has to be created. The JopenFST library does not provide a StdVectorFST, only an FST class and ImmutableFst class are present. When the FST object has been created the input and output symbols must be defined. This is necessary because otherwise it will be impossible to create arcs with the desired symbols.

```
//define the weight
float fnlst = Float.POSITIVE_INFINITY;

//define the different states
State s = new State(fnlst);
State s2 = new State(fnlst);
State s3 = new State(fnlst);

//any state with a weight different from infinity is final
State s4 = new State(0);
```

Figure 31 Java State declaration and initialization

After the creation of the FST object, the states can be created. In this example there are only four states where the fourth one is the final state. Note that the weight of the final state is different from infinity. The FST definition [14] states that:

Any state with non-infinite final weight is a final state.

This is why the final state has a final weight of 0.

```
//Add the arc's to the states. The word is "VAL"
//int ilabel int olabel float weight State nextState
s.addArc(new Arc(symbol.Find('V'), symbol.Find('V'), fnlst, s2));
fs.addState(s);
fs.setStart(s); //set the start state

//int ilabel int olabel float weight State nextState
s2.addArc(new Arc(symbol.Find('A'), symbol.Find('A'), fnlst, s3));
fs.addState(s2);
//int ilabel int olabel float weight State nextState
s3.addArc(new Arc(symbol.Find('L'), symbol.Find('L'), fnlst, s4));
fs.addState(s3);

s4.setArcs(new ArrayList<>()); //reset the final state's arcs
fs.addState(s4);
fs.setSemiring(new TropicalSemiring());
```

Figure 32 Java arc declaration and initialization

After the creation of the states, the arcs can be added to each state. Each arc must be an explicit object with the input and output symbols, the weight and the next state which also must be an object of the type State. After the addition of the states, the arcs of the final state must be initialized. This is a limitation or even an error in the JopenFST library. When creating a State object, the arcs list will not be initialized which generates errors when an operation is performed on that state. After all this the FST must have a semiring assigned. The semiring used in this application is the TropicalSemiring.

The interaction with library is different in this part. In the C/C++ library the arcs were created by the library itself, in Java a developer must explicitly create a new Arc and add it to an already existing state. The next state must also be a previously created state as it must be included when adding an arc to a state. Also a state can be set as the start state using the libraries predefined functions but not as a final state. These small differences can make it more difficult to create an application using the JopenFST library.

```
out = NShortestPaths.get(fs, 1, true);
```

Figure 33 Nshortestpath code example

The last operation that needs to be performed is the NshortestPath operation. This will determine the N-amount of shortest paths in the source FST and will return a FST as a result. The method's arguments are the source FST, the amount of paths that are desired and a Boolean that indicates if the input FST must be determinized prior to the operation or not. In this example it is determinized prior to the operation.

5.2.2. The Java concept program

The Java concept program had been developed using NetBeans and JavaFX. This concept program has made it possible to develop a Java class library which will be explained later on and to easily learn how to use the library in an application. The learning curve of this library was moderate to high since there is almost no documentation about the library nor are there code examples on how to use it. Both the projects contain similar code and have similar classes as the C/C++ concept program. The UML diagram of the concept program can be found in appendix B2.

5.2.2.1. WFST

The WFST Java class is based upon the C/C++ class and can be seen as a direct port of C/C++ to Java.

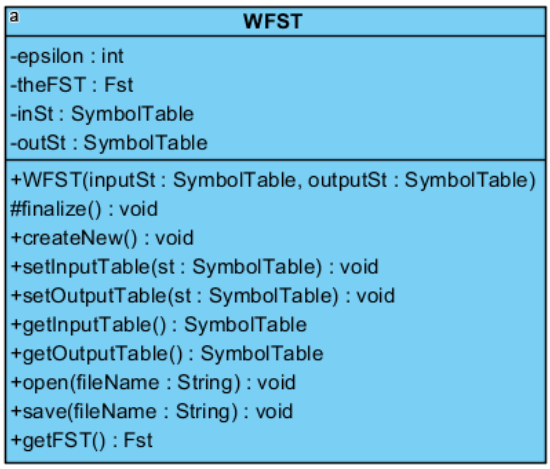


Figure 34 WFST UML class diagram

As shown in figure 34, the WFST Java class is very similar to the C/C++ WFST class. This class works in the same way as the C/C++ class but is adapted to work with Java.

Because of the differences between Java and C/C++ some of the code had to be altered. For example a deconstructor in Java is always written as an overwritten finalize method. This method will be called by the garbage collector on an object when garbage collection determines that there are no more references to the object. This method is not necessarily mandatory since the garbage collection in Java will take care of freeing up memory space anyway. In C/C++ the objects have to be specifically handled in code to free up space. The finalize method will set the FST object to null. If this for any reason fails the WFST object will be deleted using the parent's finalize method. In general it is not wise to rely on the finalize method or even the garbage collector since there is no assurance when these will be run. Thus when limited space is a real concern, other methods should be considered.

Another difference is that the semiring has to be specified. Whereas the semiring in the StdVectorFst class in C/C++ already is defined. The StdVectorFst uses the tropical semiring. Thus this is also what has been defined in the WFST class.

To open a binary FST file, the FstInputOutput class must be used. The Java binary FST file is not compatible with the C/C++ binary FST file. The encoding and serialization is handled by the FstInputOutput class in the jOpenFST library which is written in Java and has a different way of serializing objects. The open method makes also use of this class.

The saving of the model is also handled by the jOpenFST library. The library will save the Java binary FST file in the right format.

Note: the saved binary file is not compatible with the C/C++ OpenFST library. These are not interchangeable.

The rest of the methods are getters and setters that allow developers to get or set the different variables in the WFST.

5.2.2.2. SymbolTable

The SymbolTable class is based upon the same named class in the OpenFST C/C++ library. The jOpenFST library does not contain a SymbolTable class but expects a String array for the input and output symbols whereas the OpenFST library uses character arrays for the input and output labels.

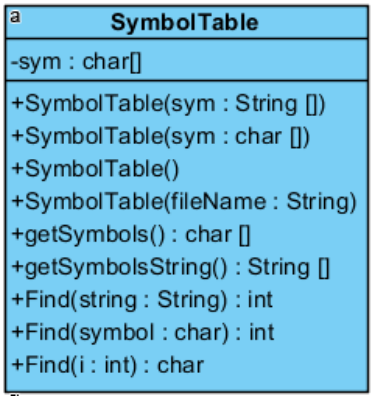


Figure 35 SymbolTable UML class diagram

The SymbolTable class has one variable which is the character array, it holds the symbol characters. The jOpenFST library requests a String array to represent the symbol table.

The SymbolTable class has two custom constructors to enable the use of the two different types.

The SymbolTable class has a method that converts the character array into a String array, this is the getter, getSymbolString(). It returns the converted symbol table as a String array.

Each symbol in the symbol table has an id. The id is defined by the position in the table. To add a symbol to an arc, the symbol must be converted to its id. This is done by using the Find method. A symbol can be searched by using its string, character and integer representation. This will search the symbol and return it as a character in the case of the integer representation and an integer in the case of the string and character representation.

5.2.2.3. LanguageModel

The language model is a direct port of the C/C++ OpenFST library. It inherits from the WFST class and represents the mapping of the word library.

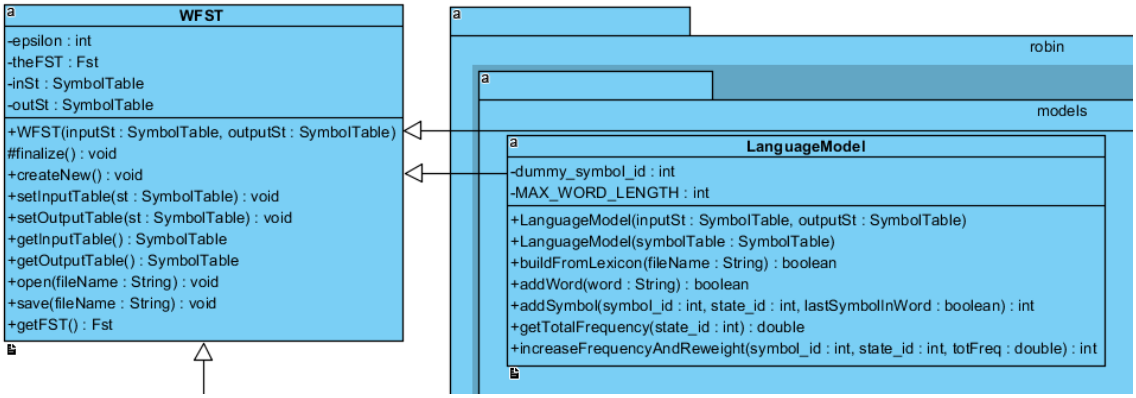


Figure 36 LM UML class diagram

The language model has been altered compared to the C/C++ version to be able to work in Java. The differences between the two libraries are quite significant when tring to interact and build FST models.

Each of the model classes share similar methods since they are all based upon the LanguageModel class. The LM class contains methods to build the LM from a file and dynamically add words on a character base.

The addWord method adds the word to the state using arcs after splitting the word in its individual character. It does this by calling the addSymbol method. The method will create an arc per character. It does this by increasing the frequency and weight. After which it calculates the weight for each symbol.

Note that the language model uses dummy symbols and arcs to keep track of the usage of a state.

An important note to make is that while most of the needed classes are present in the jOpenFST library, there are still some missing. Like the previously explained SymbolTable and also the StateId class. This means that the C/C++ functions needed to be altered to use and return integers to represent the state ID instead of the StateId type. This is noticeable in the logic that has been used in the methods. Also note that the logic had to be altered since the methods in jOpenFST have different inner workings.

5.2.2.4. PrefixModel

The prefix model is a direct port of the C/C++ OpenFST library. It inherits from the WFST class like the other models and represents the mapping of the user's input.

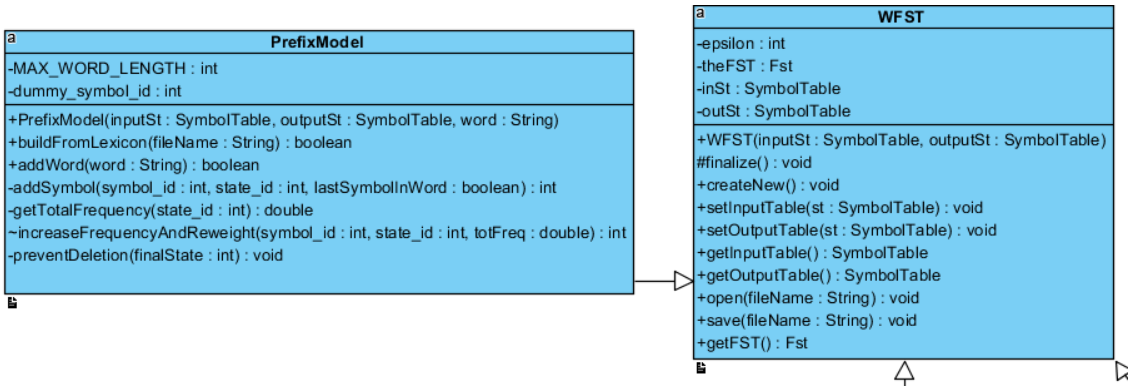


Figure 37 PM UML class diagram

The class contains all the main methods that the C/C++ class contains. It has the same functionality and use as the C/C++ class.

The addWord method adds the word to the state using arcs after splitting the word in its individual character. It does this by calling the addSymbol method. The method will create an arc per character. It also increases the frequency and weight after which it calculates the weight for each symbol. Since there are no dummy arcs, the frequency will always be one.

Note that the prefix model does not use the dummy symbols and arcs.

The prevent deletion method enables the addition of zero or more characters from the symbol table. The method will essentially add the input symbols to the final state using arcs. This does not include the epsilon symbol and the dummy symbol. Also arcs are one to one relations. This means that the input and output symbols are always the same since this is meant to preserve the symbol instead if deleting or substituting.

5.2.2.5. ErrorModel

The error model is a direct port of the C/C++ OpenFST library. It inherits from the WFST class and represents the mapping error matrix. It ensures that an insertion or deletion can be made.

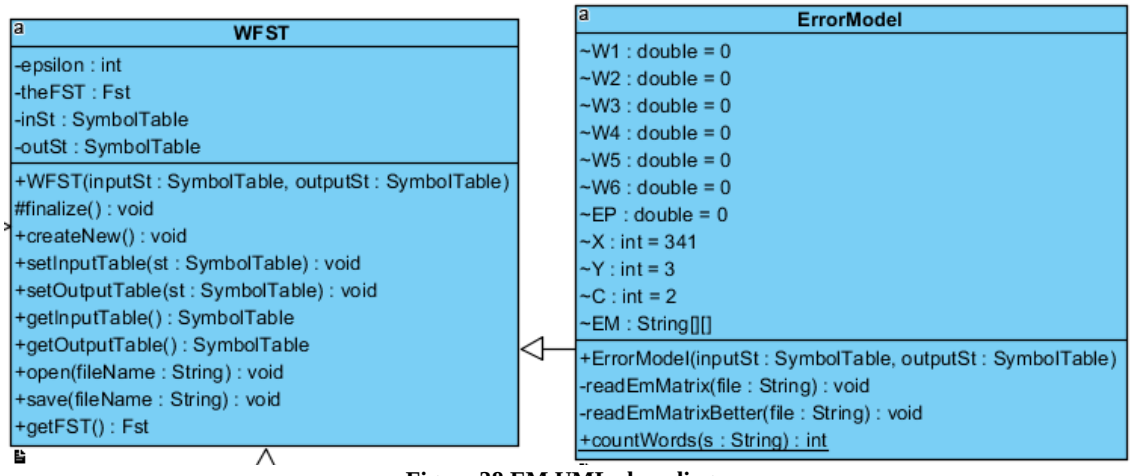


Figure 38 EM UML class diagram

The EM class makes use of the configuration in the "ErrorMatrix.txt" file that was previously explained. The configuration distinct itself from the actual mapping by being comprised out of

two separate strings. Whereas the actual mapping consists of three strings. This allows the program to distinct the two types in the configuration file.

The error model is constructed using one state with arcs which are created using the mapping inside the configuration file.

Note that there is only one state in the whole model. This is why the final weight is 0.

In C/C++ a whole work around was needed to be able to count the amount of characters in a word. In Java this code is simpler. The use of the String.Split method provides a word splitting method and returns a String array and since an array has a length, the word count can be easily found.

5.2.3.JavaFX

JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms. It allows developers to create a GUI program fairly quickly without having to spend too much time about designing the application. It is written in Java, and it substitutes Swing. It is bundled with the Java Runtime Environment (JRE), so the user does not need to install additional libraries

JavaFX application code can reference APIs from any Java library. For example, JavaFX applications can use Java API libraries to access native system capabilities and connect to server-based middleware applications. More information can be found on the oracle site.

5.2.3.1. FXMLDocumentController

The JavaFX [18] [19] application generates automatically an .fxml file which consists of the layout code, a document controller which contains the code that controls the layout in the .fxml file and a Java class which contains the main class.

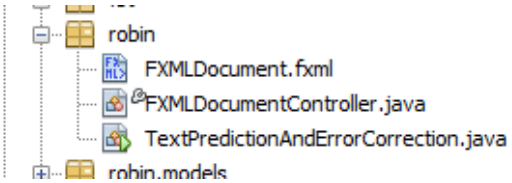


Figure 39 FXML document generation

The controller will contain a lot of the important code in the application. It will handle all the creations, compositions and printing of the different models.

The main class, TextPredictionAndErrorCorrection.Java, contains the code to start the application.

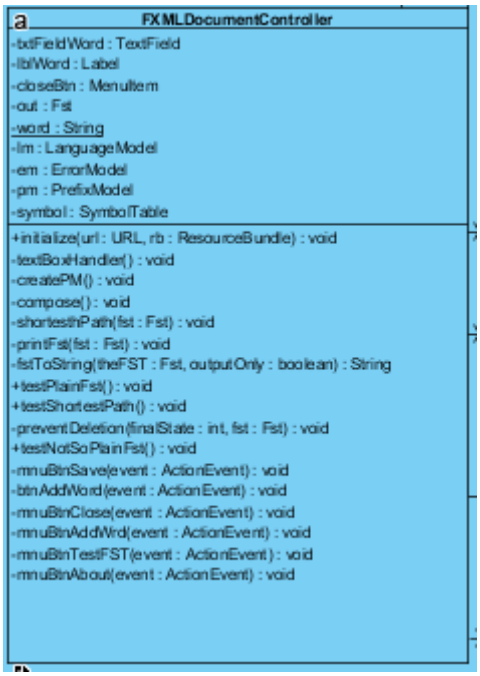


Figure 40 FXML DocumentController UML class diagram

In the main method, the launch method is called. This will trigger the start method and executes its code. The start method will first load the parent which is the FXMLDocument.fxml. This document contains the reference to the controller of that fxml file and also all the layout information. Then the scene is created using the parent object.

The scene is then set in the stage. After the initialization of the scene, the stage will be shown to the user. This will trigger the stage and thus the controller.

5.2.4.The Java class library

The Java class library is similar to the previous Java project except for the PredictionAndCorrection class. This is considered as the main class to interact with the library instead of the previously explained controller. The other classes are the same as in the Java concept implementation.

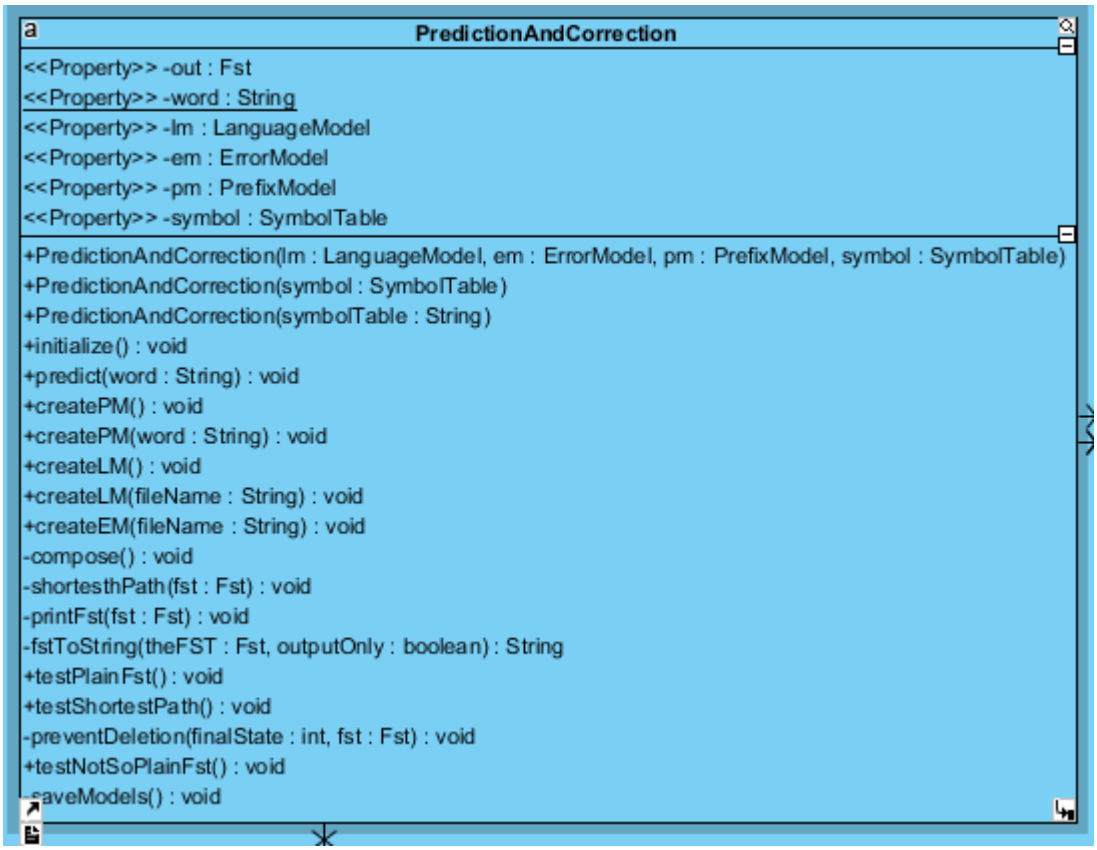


Figure 41 PredictionandCorrection UML class diagram

The class contains all the necessary methods to interact and create the different models. The Class allows a developer to take full advantage of the whole FST and WFST system and create a text prediction and error correction system that is able to run on any Java 7+ capable machine.

5.3. Part Three: Android

The Android application has been created using Android Studio and the previously discussed Java library. Android Studio is the native programming environment that has been made available by Google. It allows a developer to make native Android apps using Java. Note that Android only has support up to Java 7, this means that the library from part two had to be altered to support Java 7 instead of the new Java 8.

The Android application is comprised out of two parts. A simple custom Android keyboard to allow for simple text input without third party software and an app where the text prediction and error correction has been implemented. The simple custom Android keyboard has partially been created using a tutorial.

5.3.1.Android Studio

Android Studio is the official Integrated Development Environment (IDE) for Android app development. Android Studio offers a lot of features that enhance a developer's productivity when building Android apps.

Some of these are:

- A flexible Gradle-based build system
- A fast and feature-rich emulator
- A unified environment where you can develop for all Android devices
- Instant Run to push changes to your running app without building a new APK
- Extensive testing tools and frameworks
- C++ and NDK support
- Etc.

The C++ and NDK support in Android Studio means that the C/C++ code could have been converted into an Android app. But since this would take too much time and it would take too long to learn the correct syntax and how to correctly build an application the decision was made to use Java, since this is the native language in which most Android application are made.

5.3.1.1. Overview

When creating a project in Android Studio, multiple files and directories are automatically generated. Each project in Android Studio contains one or more modules with source code files and resource files.

Types of modules include:

- Android app modules
- Library modules
- Google App Engine modules

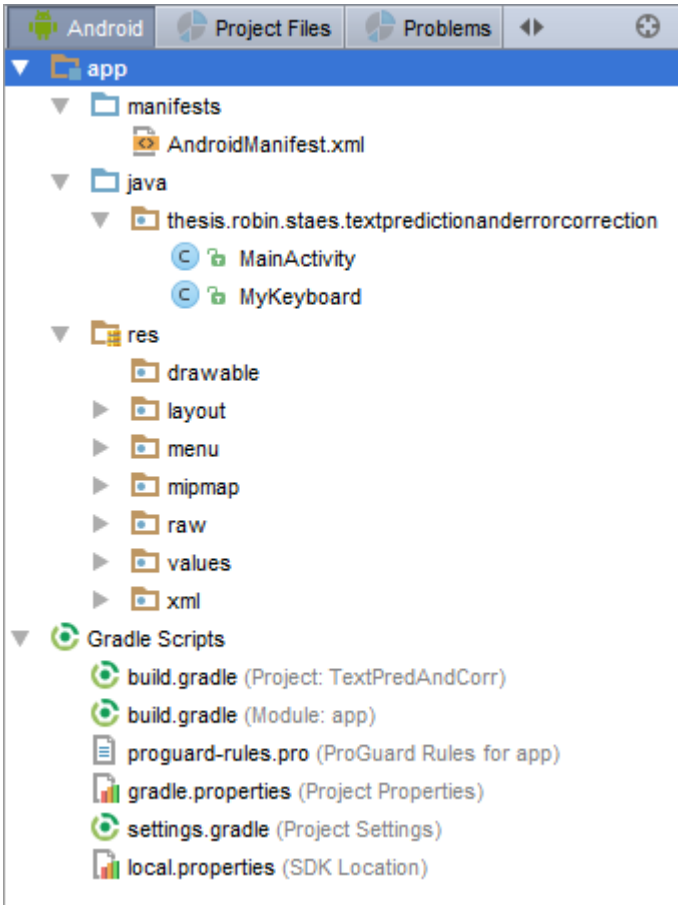


Figure 42 Android project view

By default, Android Studio displays your project files in the Android project view, as shown in figure 42. This view is organized by modules to provide quick access to a project's key source files.

All the build files are visible at the top level under Gradle Scripts and each app module contains the following folders:

- Manifests: Contains the AndroidManifest.xml file.
- Java: Contains the Java source code files, including JUnit test code.
- Res: Contains all non-code resources, such as XML layouts, UI strings, and bitmap images.

5.3.1.2. Importing the library

Importing an external library in Android Studio is not as straightforward as in for example NetBeans. To import a Java library (.Jar file) in Android Studio, the file need to be imported first. This can be done by creating a new directory in the file structure and copy pasting the file into that directory. Android Studio will automatically detect the changes and update the project to include those files.

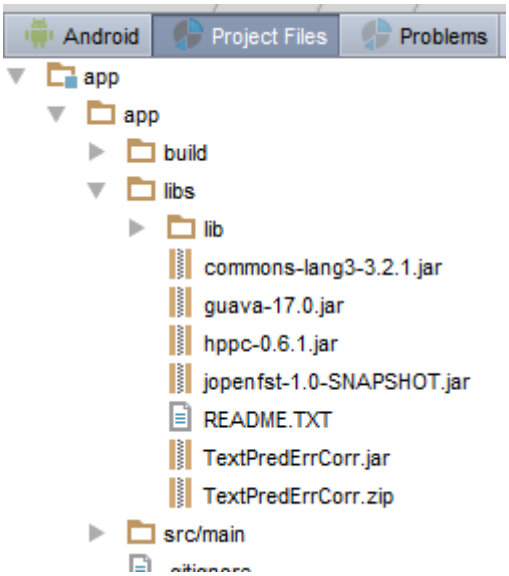


Figure 43 Imported libraries file structure

After which the selected library can be added to the project as a library by right clicking it and selecting “add as library”.

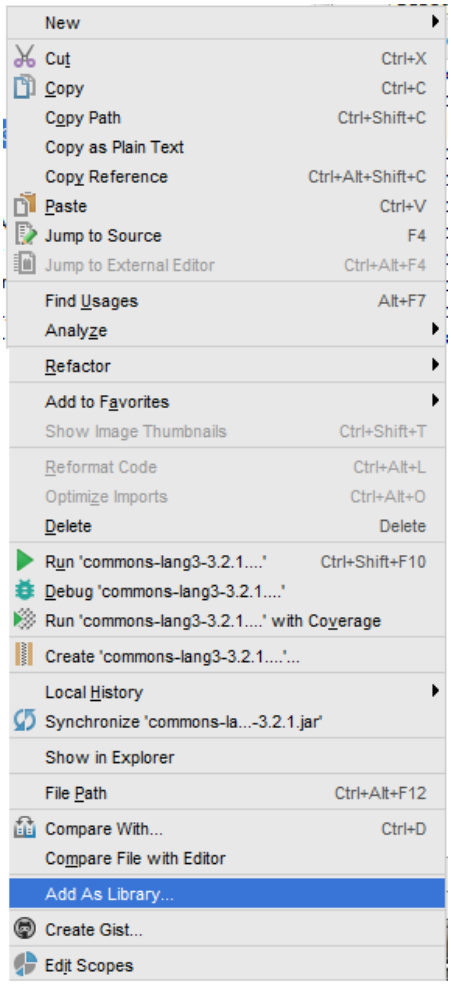


Figure 44 Android Studio "Add as library"

5.3.2. The app

As mentioned earlier, the app consists of two parts. A standard looking app that implements the text prediction and error correction system. And a simple custom Android keyboard that allows a user to input text without third party software. The app is as the previously discussed library written in Java.

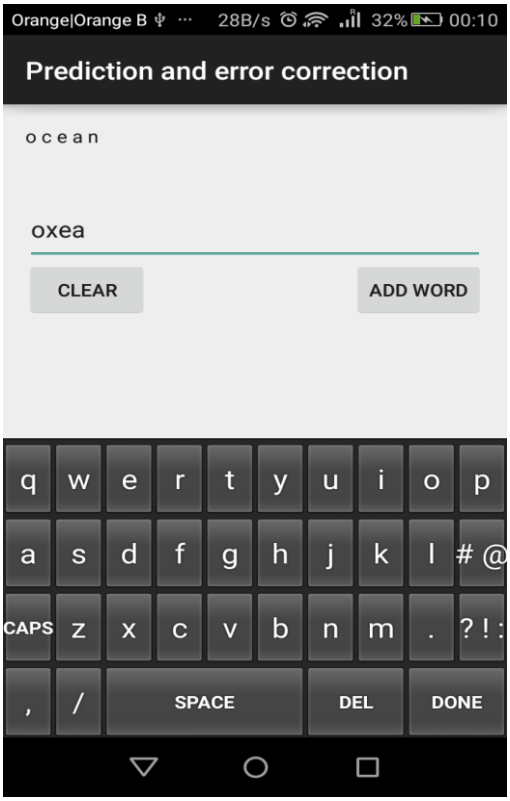


Figure 45 Text prediction and error correction

5.3.2.1. Simple custom Android keyboard

Most mobile devices, whether or not they run Android, do not have a physical keyboard. Instead they rely on virtual or soft keyboards to allow a user to insert data. Android provides the possibility to personalize and create a custom soft keyboard.

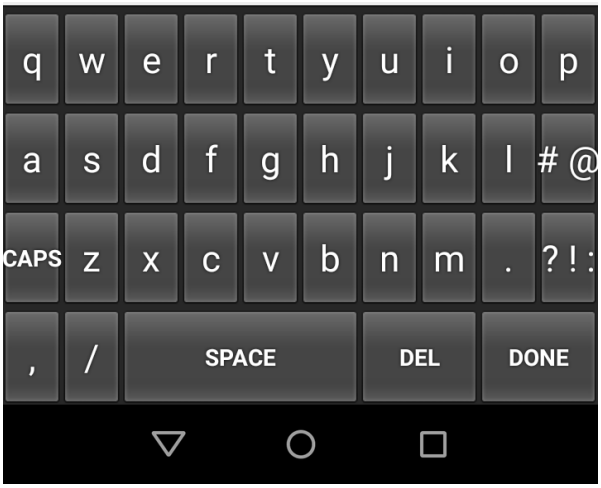


Figure 46 Simple custom Android keyboard

The Android SDK allows a developer to quickly create a soft keyboard without a lot of effort. This is not code intensive because a lot of the low level tasks like recognizing key presses, drawing the keyboard and establishing connections between the soft keyboard and possible input fields are handled by the SDK.

The following chapters explain the function and the content of the most important sections and files concerning the custom Android keyboard.

Note: in the following chapters it is assumed that there is already a version of Android Studio installed and that there is a basic knowledge of creating projects and building apps using Android Studio.

5.3.2.1.1. The manifest

The Android operating system considers soft keyboards as an input method editor (IME). This is declared as a service in the AndroidManifest.xml file and it uses the BIND_INPUT_METHOD permission to grant access to the user's input. It responds to the Android.view.InputMethod action.

To enable this permission, the manifest which can be seen in figure 47 must be altered. This alteration comprises of a couple of lines inside the application tag.

```
        <service
            android:name="thesis.robin.staes.textpredictionanderrorcorrection.MyKeyboard"
            android:label="MyKeyboard"
            android:permission="android.permission.BIND_INPUT_METHOD">

            <meta-data android:name="android.view.im"
                android:resource="@xml/method" />

            <intent-filter>
                <action android:name="android.view.InputMethod" />
            </intent-filter>
        </service>
    </application>

</manifest>
```

Figure 47 Manifest code snippit

5.3.2.1.2. Method.xml

To ensure that the Android operating system recognizes the service as a valid IME service, the manifest file must contain a meta-data-tag that references an XML file.

```
<meta-data android:name="android.view.im"
    android:resource="@xml/method" />
```

Figure 48 XML meta-data-tag

That XML file provides details about the input method and its subtypes. For this keyboard the en_US locale has been chosen since this is a well-known and universal locale.

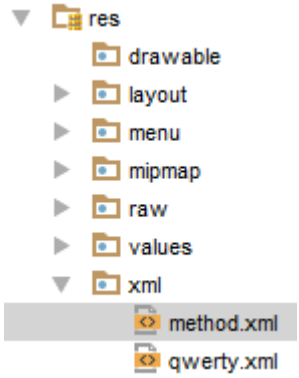


Figure 49 Res/xml folder structure

The XML file should be placed into the res/xml folder. If Android Studio has not automatically generated the file structure then it should be manually created. There the XML file should be placed to ensure that it can be found.

```
<?xml version="1.0" encoding="utf-8"?>
<input-method xmlns:android="http://schemas.android.com/apk/res/android">
    <subtype
        android:label="@string/sybtype_en_US"
        android:imeSubtypeLocale="en_US"
        android:imeSubtypeMode="keyboard" />
</input-method>
```

Figure 50 Locale file, method.xml

Inside the XML file the XML code in Figure 50 can be found. This contains all the locale settings.

Note: locale en_US ensures the user and the application that the keyboard is intended for the English language.

5.3.2.1.3. Keyboard layout

The keyboard layout is defined in the keyboard.XML file. This file contains the layout properties and components concerning the keyboard layout. The layout is made out of a keyboardview. The keyboard will appear on the bottom of the screen if the property layout_alignParentBottom is set to true.

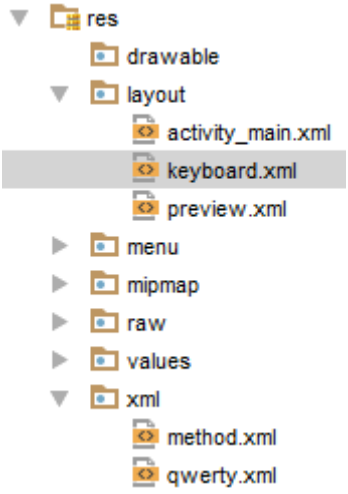


Figure 51 keyboard layout XML file

The XML file is placed under res/layout as seen in Figure 51. Again if this directory has not been generated by Android Studio, it should be created manually.

5.3.2.1.4. Keyboard Keys

The details of the keyboard keys and their positions are specified in an XML file.

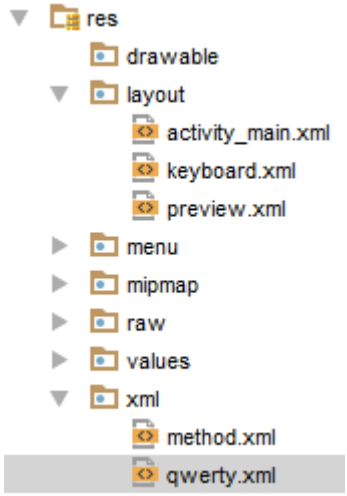


Figure 52 Keyboard key Layout XML file

Every key has the following attributes:

- KeyLabel: This attribute contains the displayed text.
- Codes: This attribute contains the Unicode value of the character of that key.

```
<Row>
    <Key android:codes="113" android:keyLabel="q" android:keyEdgeFlags="left"/>
    <Key android:codes="119" android:keyLabel="w"/>
    <Key android:codes="101" android:keyLabel="e"/>
    <Key android:codes="114" android:keyLabel="r"/>
```

Figure 53 Key layout code example

This ensures that the keys have the right mapping. A list of Unicode mappings can be found online

5.3.2.1.5. Service class

The MyKeyboard class is the class that contains the code that handles the keyboard. It should extend the InputMethodService class and implement the OnKeyboardActionListener interface which contains the methods that are called when a user interacts with the keys of the soft keyboard.

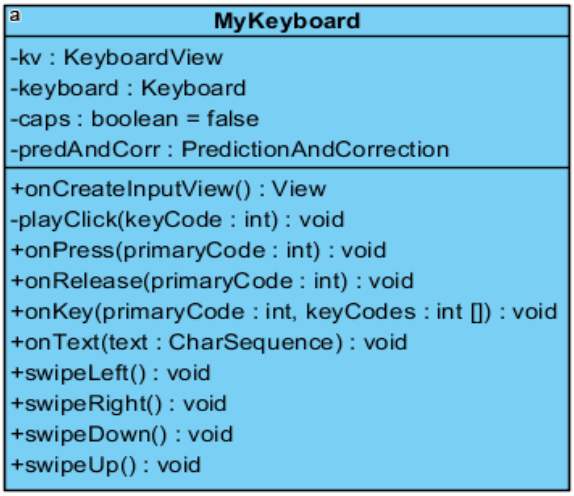


Figure 54 Android Keyboard UML class diagram

The MyKeyboad class contains the different by Android Studio auto generated methods. These methods are the extensions of the InputMethodService class and implementation of the OnKeyboardActionListener interface.

The important methods are:

- onCreateInputView
- onKey
- playClick

The onCreateInputView method is called when the keyboard is created. All the member variables of the service should be initialized in this method.

The onKey method is called whenever a key on the keyboard is pressed. This means that all the key presses are handled in this method. Thus this method contains code that reads the key input and plays a sound, the playClick method, accordingly to the different keys. If the keypress was a special key like shift or delete then the right action will be triggered.

5.3.2.2. Text prediction and error correction app

The basic app that contains the actual text prediction and error correction system is a simple app that has been created using the blank app template in Android Studio. The app contains a TextView which is the label that displays the generated output, an EditText which is essentially a text input field and two buttons. One to add an inserted word to the LM and one that cancels the current prediction. Ideally, in a real keyboard, new words should be added automatically to the LM if, for example, a given property is checked.

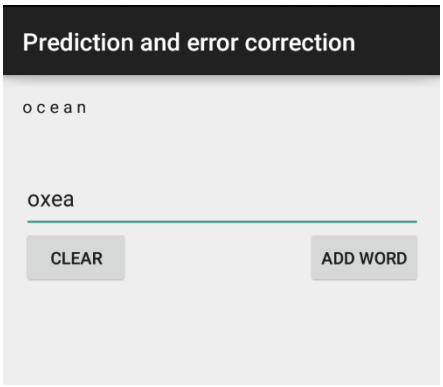


Figure 55 App layout

This app has been made this simplistic because it does not need more functionality to provide a real world example. The app will predict and correct any prefix that is inserted into the text input field. As Figure 54 shows, the prefix "oxea" will be corrected and predict the correct word ocean.

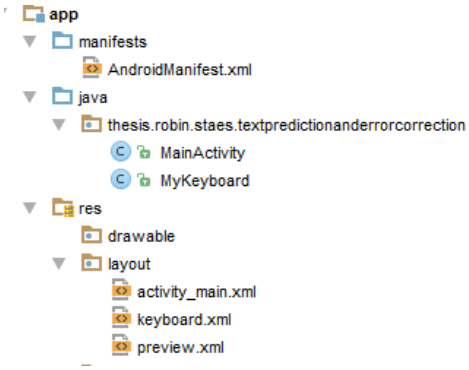


Figure 56 Basic app file structure

The basic app contains, like the keyboard part, a layout file. This layout file, called activity_main.xml as shown on Figure 56, contains all the components and attributes of the class.

5.3.2.2.1. The main class

The MainActivity class which is the main class for the text prediction and error correction part contains very little code. This is because all the code is handled by the Java library.

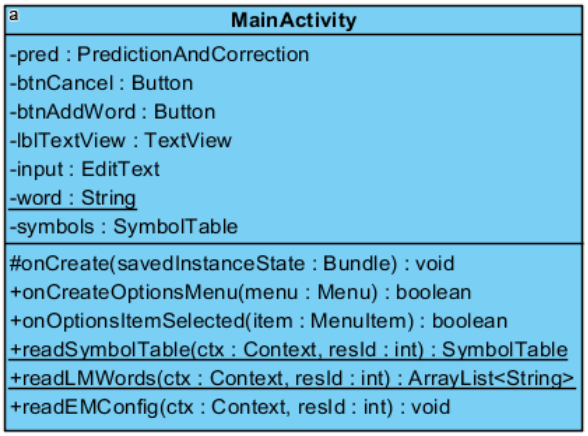


Figure 57 Android MainActivity class UML diagram

In the onCreate method which is called when the app is launched, the different models are initialized using the configuration files, which reside in the res/raw folder. These configuration files are being handled by three methods inside the MainActivity class because of the way how Android handles the reading of files. This could not be integrated into the app and are thus handled by the following methods:

- readSymbolTable: which reads the symbol table configuration file and builds a symbol table that will be used when creating the different models.
- readLMWords: which reads all the words in the words.txt file. Using these words a LM is generated.
- readEMConfig: this method reads the error matrix and builds a EM using that matrix.

The code to use the Java library is fairly straightforward. Firstly the symbol table needs to be initialized where after the PredictionAndCorrection object can be created

```
symbols = MainActivity.readSymbolTable(this.getBaseContext(),R.raw.symbols );
pred = new PredictionAndCorrection(symbols);//symbol table
```

Figure 58 Android example code library usage

After the initialization of the main class object, the different models need to be initialized.

```
//create EM
readEMConfig(this.getBaseContext(),R.raw.emmatrix);

//Create LM
pred.setLm(new LanguageModel(symbols,symbols));
pred.getLm().buildFromArray(MainActivity.readLMWords(this.getBaseContext(),R.raw.words));

//create PM
try {
    pred.createPM("TES"); //first word
} catch (Exception e) {
    e.printStackTrace();
}
```

Figure 59 Android code example model initialization

Note: the last step, the initialization of the PM is not mandatory. Since the createPM will be called every time a new prefix is entered. And thus will be re-initialized dynamically when a user provides input.

6. Problems

While making this project, a lot of problems were encountered. The main problems will be explained below alongside with their possible solutions.

The earliest problem that was encountered has to do with installing the OpenFST library. Since Windows is the main operating system that most people use on a day to day basis, an installation on Windows has been tried. This did not work as planned since the OpenFST library was intentionally created to be compiled using a shell command line interface. This should be solved when using Windows 10 since there is a shell integrated in the operating system that a developer can enable. There also exists a Visual Studio project that contains a C++ program. This should enable a developer to compile the library for Windows but this was again not successful.

After these struggles a decision was made to change to Mint, a Linux distribution. This would have been, in theory, the best option since it is a commonly used distribution by programmers. This did not go without hurdles. Most of the necessary tools were made for kernel 3 instead of 4 which made it more difficult to install the necessary tools. This has been resolved by installing the right programs to support the necessary legacy programs. In hindsight, Ubuntu would have been the better choice since this is also a commonly used distro and it supports all the necessary tools.

The second problem was using C/C++. This is because firstly C and C++ are not the most comprehensive languages, due to the use of pointers and references. Also because the OpenFST library does not have a lot of information and examples which can be used to create a program in conjunction with the different models.

The more the project advanced and became more complicated, the more difficult it became to keep a clear head and still understand what was going on. The learning curve to FSTs and WFSTs was quite high and the inner workings are very mathematical. This caused the project to slow down and created problems that took a long time to solve.

After the creation of the C/C++ concept program, there was a need to extend the project to other platforms. This was also difficult since the OpenFST library only work with C/C++ code and the library only runs on Linux. After a long search, jOpenFST was discovered. This library is written in Java and is able to run on any Java capable machine.

The problem with this library is that there is almost no documentation or examples available. This meant that it took a long time to understand how the library works and how it can be implemented. Also the library does not contain all the same classes as the OpenFST library and appears to miss some useful classes and functionality.

All these problems meant that the project was delayed significantly and thus some choices and compromises had to be made.

7. Conclusion

7.1. General

Finite state transducers and weighted finite state transducers enable developers to develop a system that transduces one thing into another. It is a very capable and broad applicable system that can be used on virtually any platform.

The library made in this project allows a developer to create a text prediction and error correction system that can be integrated in any application, on condition that one of the supported languages are being used. The concept programs provide an insight in how to create such a program and how it can possibly be implemented.

The libraries were made with ease of use and performance in mind. The Java class library does not officially use the delayed composition. The reason is because it is not clear if the jOpenFST library makes use of it since there is almost no documentation about it. The OpenFST library does make use of delayed composition. This means that it will only compose the paths that need to be visited instead of the whole model every time. The code in the projects has been made as efficient as possible in the provided time frame.

The possible application for transducers are virtually endless. It could be used in applications to predict user input but it could also be used as a word translator where each word in a language is linked to another word in another language. Or it could be implemented in an operating system like on mobile phones. Then it would be applicable to all the installed applications on the system that uses user text input.

7.2. Personal

Despites the problems that were encountered in the project, the project was still finished in time and the wanted goals were met. The project could have been improved if there was more time available but the current result of the project gives a good depiction on how the whole system works and how it can be implemented using different languages and environments.

When making the project, some of the problems were rather personal. This is because it has been quite a while since I have used C/C++. And I have not used C/C++ in a Linux environment before this project. This caused a small delay in the execution, since I needed to learn how to use the environment and refresh my knowledge of C/C++. This took a couple of weeks but was overcome quite easily.

There were also some hurdles when switching from one stage to another, this was not a big problem since I have a good knowledge of Java.

The project has broadened my idea around natural language processing and the possible applications for such a system. The first idea, thought for the project was the possibility to incorporate a text prediction and error correction system on computers. The possible efficiency that could be acquired by this are quite significant.

Working and studying abroad is not easy but it has enabled me to increase my personal abilities and my management skills when it comes to such a big project. The environment and people at UPV enabled me to finish the project without too much problems and within the given time frame.

8. Future

This project can have a lot of future extensions, revisions and implementations since text prediction and error correction is a modern technology that already has been implemented on mobile phones and tablets but not on desktop systems or desktop applications. Transducers are not limited to the implementation of this project. They can be used to implement totally different applications like translations or even just neural networking. It can also be used for OCR post processing (to correct de output of an OCR engine) [11]. In general, it can be used to parse any information that can be represented as a sequence of symbols (for example a DNA chain).

This project can be extended by implementing the following improvements:

A better real world application that allows to make use of all the features and implements them in a user friendly way.

Integration of multiple languages. This is already possible by replacing the words file with words written in a different language. This could be integrated into the application to allow for better usability. Also the application could be expanded to be able to support different LMs for the different languages.

A better and more professional looking GUI for the Java concept program and a GUI for the C/C++ concept program that allows a user/developer to take more advantage of the whole system and allows them to play around with it.

Improve the code, make it more resilient so it catches possible errors and exceptions better and make the code more general so it can be applied and reused in more situations and environments.

Adapt the project to be able to handle multiple words and different symbols so that the LM and symbol table can be expanded with more words and abbreviations. This enables the project to provide more and better predictions in all the possible use cases.

Alter the jOpenFST library to enable delayed functions when composing the different models. There is also the possibility to make us of multi-threading and/or hyper threading since this can improve the systems speed when using large data structures.

This small list of improvements can still be extended since there are a lot of possible use cases for such a project.

Bibliography

[1] A. Atria, “transducers cppcon,” [Online]. Available: http://sinusoid.es/talks/transducers-cppcon15/#/2. [Accessed 5 june 2016].

[2] “CSP and transducers in JavaScript,” [Online]. Available: http://phuu.net/2014/08/31/csp-and-transducers.html. [Accessed 1 june 2016].

[3] E. Shira, “Understanding Transducers,” [Online]. Available: http://elbenshira.com/blog/understanding-transducers/. [Accessed 10 June 2016].

[4] M. R. J. S. W. S. M. M. Cyril Allauzen, OpenFst: A General and Efficient Weighted Finite-State Transducer Library.

[5] WeightedFiniteStateTransducers, “Lecture 1 Introduction to Finite Automaton,” [Online]. Available: https://www.youtube.com/watch?v=1aEinrlyp8w&list=PLxbPHSSMPBeicXAHVfyFvGfCywRCq39Mp. [Accessed 10 april 2016].

[6] edobashira, “Speech and Natural Language Processing,” [Online]. Available: https://github.com/edobashira/speech-language-processing. [Accessed 20 april 2016].

[7] R. L. J. A. J.-C. P.-C. J. Ramon Navarro-Cerdan, Composition of Constraint, Hypothesis and Error Models to improve interaction in Human–Machine Interfaces.

[8] M. R. ,. S. ,. S. ,. M. Cyril Allauzen. [Online]. Available: http://www.stringology.org/event/CIAA2007/pres/Tue2/Riley.pdf. [Accessed 2 may 2016].

[9] “Shortest distance in the log semiring for a weighted automaton,” [Online]. Available: http://stackoverflow.com/questions/9074929/shortest-distance-in-the-log-semiring-for-a-weighted-automaton. [Accessed 2 may 2016].

[10] J. Salatas, “Porting openFST to java: Part 1,” CMU Sphinx, [Online]. Available: http://cmusphinx.sourceforge.net/2012/05/porting-openfst-to-java-part-1/. [Accessed 12 april 2016].

[11] J. R. N.-C. ,.-C. P.-C. a. J. A. Rafael Llobet, Efficient OCR Post-Processing Combining Language, Hypothesis and Error Models.

[12] "FstGltTutorial part 2," [Online]. Available: http://www.openfst.org/twiki/pub/FST/FstHltTutorial/tutorial_part2.pdf. [Accessed 15 april 2016].

[13] “Headers and Includes: Why and How,” [Online]. Available: http://www.cplusplus.com/forum/articles/10627/. [Accessed 20 march 2016].

[14] openfst, “FstExamples,” [Online]. Available: http://www.openfst.org/twiki/bin/view/FST/FstExamples. [Accessed 15 april 2016].

[15] [Online]. Available: http://ubuntuforums.org/showthread.php?t=2151633. [Accessed 10 march 2016].

[16] I. G. a. t. U. o. Illinois, “OpenFst Lab Cheat Sheet,” [Online]. Available: http://www.ifp.illinois.edu/~pjyothi/jsalt/tutorial/fstcheat.html. [Accessed 3 may 2016].

[17] “minicourse,” [Online]. Available: http://www.isle.illinois.edu/sst/courses/minicourse/2009/. [Accessed 16 april 2016].

[18] M. Pawlan, “What is JavaFX,” Oracle, [Online]. Available: http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm. [Accessed 6 june 2016].

[19] “JavaFX overview,” Oracle, [Online]. Available: https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm. [Accessed 6 june 2016].

[20] "Introduction to Finite State Transducers," [Online]. Available: http://white.ucc.asn.au/Kaldi-Notes/fst-example/. [Accessed 15 may 2016].

[21] O. H. &. S. University, "JHU tutorial," [Online]. Available: http://www.cslu.ogi.edu/~hollingk/JHU_tutorial.html. [Accessed 2 may 2016].

[22] “OpenFst Glossary,” openfst.org, [Online]. Available: http://www.openfst.org/twiki/bin/view/FST/FstGlossary#SemiringDef. [Accessed 10 may 2016].

[23] “User guide,” [Online]. Available: http://pyfst.github.io/user_guide.html. [Accessed 19 march 2016].

[24] CMUSphinx, “CMUSphinx Wiki,” [Online]. Available: http://cmusphinx.sourceforge.net/wiki/. [Accessed 2 april 2016].

Appendices

Appendix A

Implementation libraries

A.1 OpenFST - C/C++

An FST library written in C/C++. It allows a developer to create and interact with transducers

OpenFST: http://openfst.org/twiki/bin/view/FST/WebHome

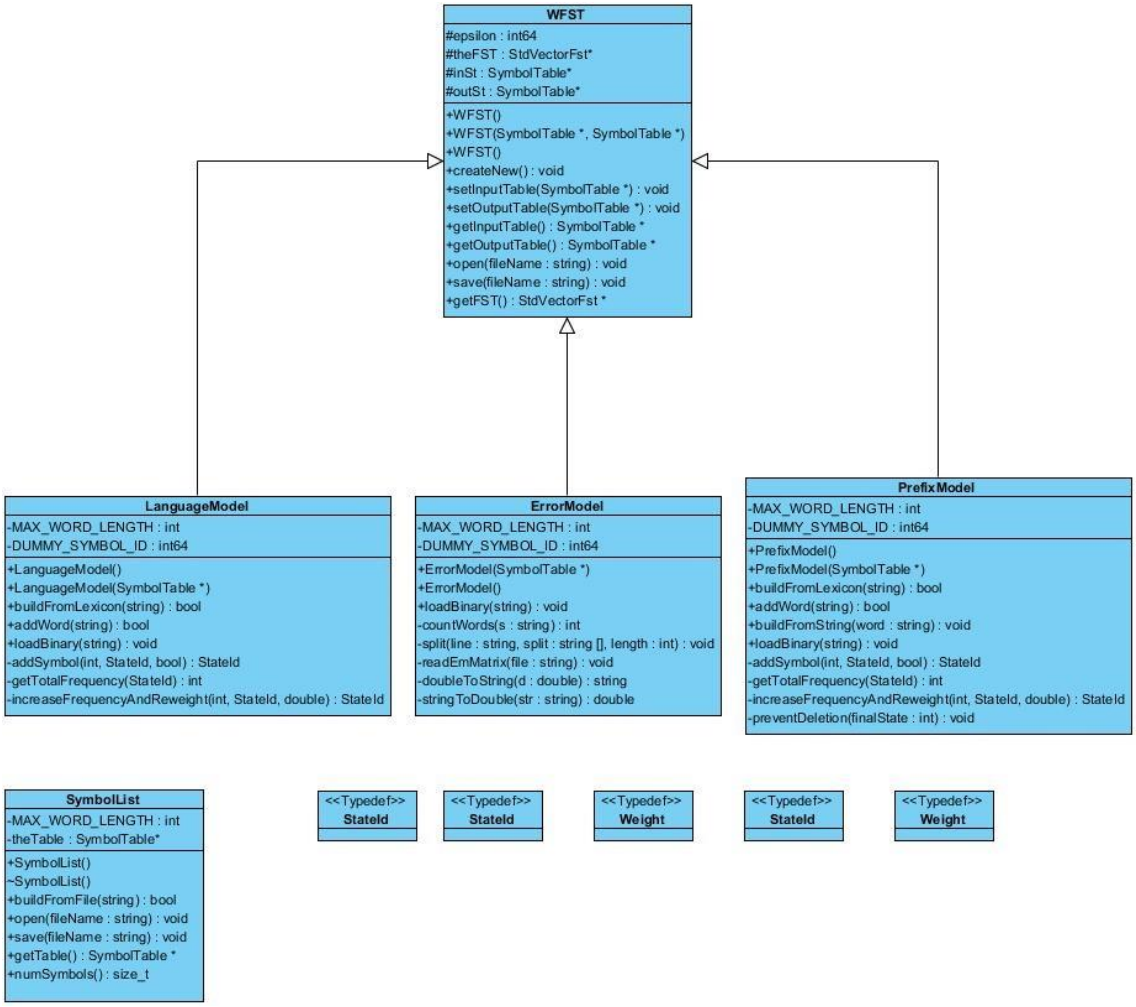
A.2 jOpenFST - Java

An FST library written in Java. It allows a developer to create and interact with transducers in Java. This is supposed to be a direct port of the previously defined OpenFST library.

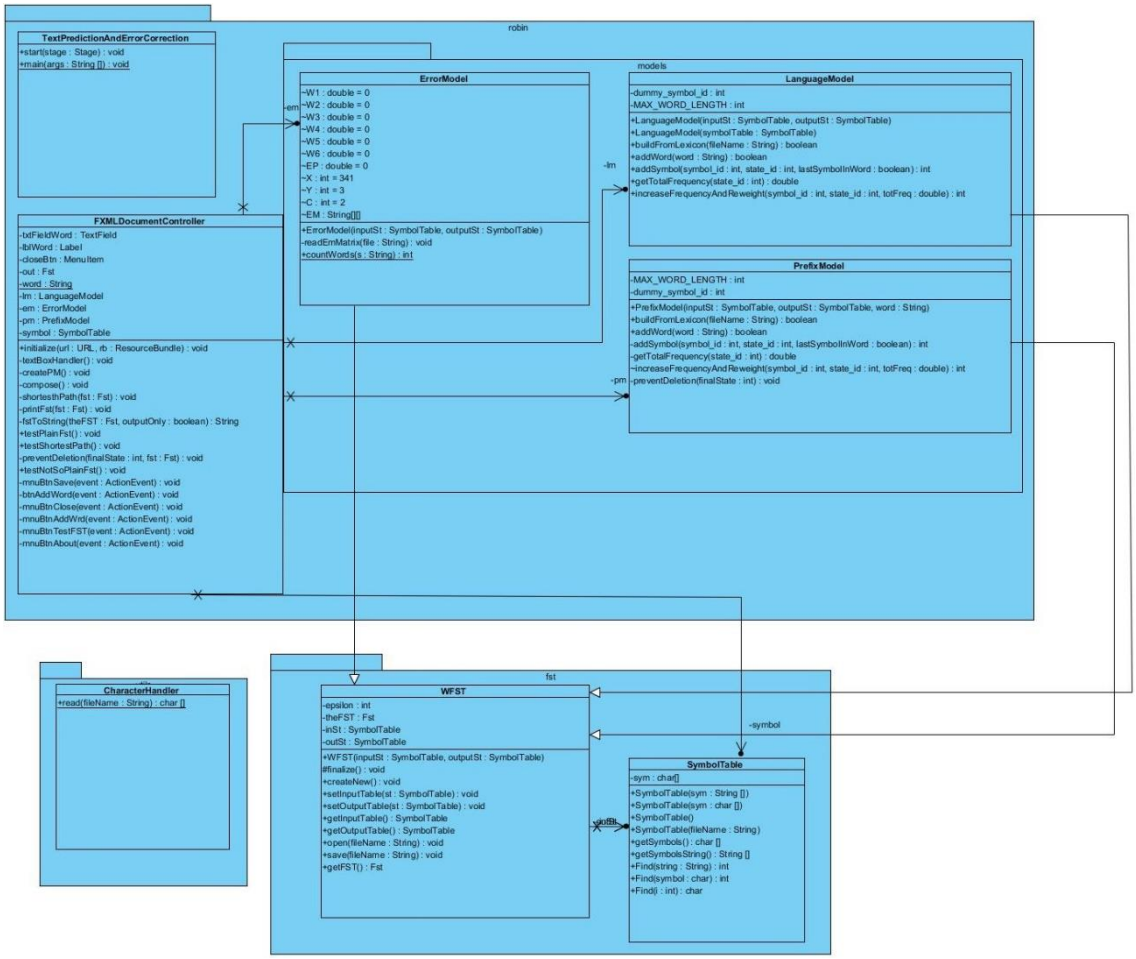
JopenFST: https://github.com/steveash/jopenfst

Appendix B

B 1 C/C++ concept program UML diagram



B.2 Java concept program UML diagram



B 3 Java Text prediction and Error correction library UML class diagram