Technical University of Crete

Diploma Thesis

Development of an Educational Graphical Tool for Finite State Machine Simulation

Author:
George Koykoympedakis

Supervisor:
Michail G. Lagoudakis
(Associate Professor)

Committee:
Antonios Deligiannakis
(Assistant Professor)
Katerina Mania
(Associate Professor)

A thesis submitted in partial fulfilment of the requirements for the Diploma of Electronic and Computer Engineer

in the

School of Electronic and Computer Engineering

July 2013

Πολυτεχνειο Κρητης

Διπλωματικη Εργασια

Ανάπτυξη Εκπαιδευτικού Γραφικού Εργαλείου Προσομοίωσης Πεπερασμένων Αυτομάτων

Επιβλέπων:
Μιχαήλ Γ. Λαγουδάκης
(Αναπληρωτής Καθηγητής)

Επιτροπή:
Αντώνιος Δεληγιαννάκης
(Επίκουρος Καθηγητής)
Αικατερίνη Μανιά
(Αναπληρώτρια Καθηγήτρια)

Συγγραφέας:
Γιώργος Κουκουμπεδάκης

Εκπόνηση διπλωματικής εργασίας προς μερική διεκπεραίωση των προαπαιτούμενων για το Δίπλωμα του Ηλεκτρονικού Μηχανικού και Μηχανικού Υπολογιστών

στην

Σχολή Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

Ιούλιος 2013

Abstract

Automata (also known as state machines) are abstract mathematical models used widely in theoretical computer science for the study of computability, but also in several practical applications. In the literature, they are oftentimes depicted as directed graphs. Their graphical representation makes automata an ideal intuitive way of communicating fundamental concepts of the Theory of Computation. Many academic institutions use educational software tools that simulate automata in an interactive way to facilitate the teaching process. This thesis describes our approach to creating such a software tool, called FA-Sim, which is a graphical interactive simulator for the most basic type of automata, called Finite Automata or Finite State Machines. Our tool is consistent with and customized for the course of Theory of Computation taught at our institution. The user (student or instructor) is able to graphically create, edit, and simulate any deterministic or nondeterministic finite automaton on any given input and observe the resulting computation step-by-step. Additionally, our tool features automatic conversion from any nondeterministic finite automaton to an equivalent deterministic one. Finally, our tool supports import of automata specified in the text-based JSON open standard and export of any automaton's graphical representation to several image file formats. Our implementation is based entirely on the Java programming language, following the principles of object-oriented programming. The graphical user interface was developed using Swing, the native Java library for user interfaces, and JUNG, a third-party Java framework for manipulating graphs. The internal representation of automata for the purposes of storage and retrieval is based on GraphML, an XML-based file format for graphs. The user evaluations we conducted with students from the Theory of Computation class revealed several user interface suggestions and preferences, which were taken into account and incorporated into the latest version of FA-Sim. All student participants agreed that FA-Sim would indeed be a useful tool in the better understanding of the concept of finite automata.

Περίληψη

Τα αυτόματα (επίσης γνωστά και ως μηχανές καταστάσεων) είναι αφηρημένα μαθηματικά μοντέλα που χρησιμοποιούνται ευρέως στη θεωρητική επιστήμη των υπολογιστών για τη μελέτη της υπολογισιμότητας, αλλά και σε πολλές πρακτικές εφαρμογές. Στη βιβλιογραφία, συνήθως απεικονίζονται ως κατευθυνόμενοι γράφοι. Η γραφική τους αναπαράσταση τα καθιστά ένα ιδανικό διαισθητικό τρόπο μετάδοσης θεμελιωδών εννοιών της Θεωρίας Υπολογισμού. Πολλά ακαδημαϊκά ιδρύματα χρησιμοποιούν εκπαιδευτικά εργαλεία λογισμικού που προσομοιώνουν αυτόματα με διαδραστικό τρόπο προς διευκόλυνση της εκπαιδευτικής διαδικασίας. Η παρούσα διπλωματική εργασία περιγράφει την δική μας προσέγγιση για τη δημιουργία ενός τέτοιου εργαλείου λογισμικού, που ονομάσαμε FA-Sim, και αποτελεί έναν γραφικό διαδραστικό προσομοιωτή για το πιο βασικό είδος αυτομάτων, τα πεπερασμένα αυτόματα ή μηχανές πεπερασμένων καταστάσεων. Το εργαλείο μας είναι συνεπές με το μάθημα της Θεωρίας Υπολογισμού που διδάσκεται στο ίδρυμά μας και προσαρμοσμένο στις ανάγκες του. Ο χρήστης (μαθητής ή διδάσκων) είναι σε θέση να δημιουργήσει, να επεξεργαστεί και να προσομοιώσει γραφικά οποιοδήποτε ντετερμινιστικό ή μη ντετερμινιστικό πεπερασμένο αυτόματο σε οποιαδήποτε δεδομένη είσοδο και να παρατηρήσει τον υπολογισμό βήμα προς βήμα. Επιπλέον, το εργαλείο μας υποστηρίζει την αυτόματη μετατροπή από οποιοδήποτε μη ντετερμινιστικό πεπερασμένο αυτόματο σε ισοδύναμο ντετερμινιστικό. Τέλος, το εργαλείο μας υποστηρίζει την εισαγωγή αυτομάτων που περιγράφονται με το ανοικτό πρότυπο κειμένου JSON και την εξαγωγή της γραφικής αναπαράστασης οποιουδήποτε αυτομάτου σε διάφορες μορφές αρχείων εικόνας. Η υλοποίησή μας βασίζεται εξ ολοκλήρου στην γλώσσα προγραμματισμού Java και ακολουθεί τις αρχές του οντοκεντρικού προγραμματισμού. Το γραφικό περιβάλλον χρήστη αναπτύχθηκε χρησιμοποιώντας την βιβλιοθήκη Swing της Java για περιβάλλοντα χρήστη και το εργαλείο διαχείρισης γράφων JUNG. Η εσωτερική αναπαράσταση αυτομάτων για τις ανάγκες αποθήκευσης και ανάκτησης βασίζεται στο πρότυπο αρχείων για γραφήματα GraphML. Οι αξιολογήσεις των χρηστών που διεξήχθησαν με φοιτητές από την τάξη της Θεωρίας Υπολογισμού αποκάλυψαν αρκετές προτάσεις και προτιμήσεις σε θέματα διεπαφής χρήστη, οι οποίες ελήφθησαν υπόψη και ενσωματώθηκαν στην τελευταία έκδοση του FA-Sim. Όλοι οι συμμετέχοντες συμφώνησαν ότι το εργαλείο FA-Sim θα μπορούσε πράγματι να φανεί χρήσιμο για την βαθύτερη κατανόηση της έννοιας των πεπερασμένων αυτομάτων.

Acknowledgements

I thank caffeine, stackoverflow, louis ck, and web crawlers for their support and contribution. I also thank all my friends that helped me through rough times, even without knowing so.

Special thanks go to:

- My advisor, Michail G. Lagoudakis, who entrusted this project to me, and kept providing purposeful remarks and constructive ideas until the end.
- My friends John & Nick Agadakos (for the moral boost), Vaggelis Gavalakis (for the mpalidia) and Nick Papoulias (for the fish).
- Argiro, Manolis and Kostas, my family.

Contents

List of Figures

Chapter 1

Introduction

1.1 A World of Finite States

According to *Michael Sipser*, as stated in his book *Introduction to the Theory of Computation* [1], there are three central areas in the theory of computation: *automata*, *computability*, and *complexity*. Moreover,

> "... (these three areas) are linked by the question:
>
> *What are the fundamental capabilities and limitations of computers?*
>
> This question goes back to the 1930s when mathematical logicians first began to explore the meaning of computation." [1, p. 1]

Needless to say, hadn't humanity asked that question, our respected readers would be staring now at a blank piece of paper authored by a student of a non-existent school. In other words, *computers as we know them would not have existed.*

As for the three areas *M. Sipser* mentions, each relates to a wide scientific area. There is no coincidence in the fact that every computer scientist is being taught the fundamentals of these areas and that every curriculum in *Computer Science* contains at least one related course.

Our institution (*Technical University of Crete*) could be no exception to that. In the respective *School of Electronic and Computer Engineering*, the undergraduate course on *Theory of Computation* covers the fundamentals of exactly these three areas.

Despite how beautifully interesting they might be, the second and third areas (computability and complexity) will not concern us further in this text. Only the first one

(automata) does related to our purpose, and even so, not in its entirety. So, let us take a closer look at automata.

Usually, in computation-related courses, *Automata Theory* is used by instructors to convey the basics of computation and provide a solid, logical *link* between *theoretical* and *practical* aspects. Automata are truly appropriate for this task, as they were traditionally used by mathematicians to study computation, but quite often are used to solve practical problems as well. Their field of applications spans across many scientific areas, such as *programming languages*, *linguistics*, *natural language processing*, *artificial intelligence*, and even *biology*. For example, one of the student assignments in our course involves creating the first stages of a programming language compiler, using automata to perform the *lexical analysis*.

However, when it comes to showcasing the importance of *automata* and their applications, there is noone better than **Stephen Wolfram**. In his book *"A New Kind of Science"* [2], he studies how sets of finite, simple rules enforced upon a computational unit with finite states (such as an automaton) can lead to unbelievable levels of complexity, impossibly large degrees of variation, and apparently random processes. This way, he claims, we can study a multitude of branches of science such as *fluid flows, snowflake and crystal formation, chaos theory, cosmology* and many more. It is impressive that automata theory can contribute to a better scientific understanding of these areas. *Stephen Wolfram* is considered to be a modern-day pioneer of a specific type of automata called *cellular*, and he even goes to the extend to argue that the ***entire universe*** might eventually be describable as a machine with finite sets of states and rules and a single initial condition.

Unfortunately though, the entire universe cannot be contained in our thesis, so we digress a bit. What **is** contained though, is the fact that Automata Theory plays an important role in Computer Science, and all the students learn about it one way or another. In order to facilitate this educational process, many universities around the world employ dedicated software tools, called *Automata Simulators*. These are tools that accept a description of an automaton as input and simulate its computation providing it as output. Furthermore, many authors in recent and past scientific literature argue that students actually ***prefer*** these tools to be accompanied by a *Graphical User Interface* and a form of *visualization* to depict automata and their computations; and to **that** our university was indeed an exception. As of July 2013, there was no such tool available for our course, tailored after our specific needs and in accordance to the textbook we use, called *Elements of the Theory of Computation* [3]. Thus, we decided to cover that need, and the topic for this thesis was born.

The result we came up with, was not to be a complete solution to the problem, after all. Due to time constraints, lack of experience and other factors, we were forced to create a visual automata simulator that only covers the simplest of all automata: ***Deterministic Finite Automata*** and ***Nondeterministic Finite Automata***. However, despite their simplicity, these two types are not to be taken lightly. They have numerous important applications in all domains mentioned above and they adequate as a *starting ground* for someone who wants to study *Automata Theory*. Finite Automata are able to bear important, fundamental concepts related to the *Computation Theory*.

The way we see our software tool, the fruit of our work, is quite similar to that: A starting ground to build upon, towards a complete visual automata simulator; but also, a starting ground that encompasses more than enough functionality to fulfill it's role as a ***Finite Automata Simulator, used as a pedagogical tool***.

1.2 FA-Sim

We called our tool *Finite Automata Simulator (FA-Sim)*. It was implemented in the **Java** programming language, while various third-party tools (also written in Java) were used to facilitate our efforts. The current version comes with a simple, minimalistic, intuitive *Graphical User Interface (GUI)*, which was implemented using the **Swing** library. The GUI itself contains an area responsible for editing and displaying content, i.e. an area that serves as our *canvas*. In this area, the user can specify a finite automaton (DFA or NFA) *in a graphical way*, provide the appropriate input, and simulate its computation. If this automaton is an NFA, the user has the option to *convert it to a DFA*, as well. In any case, the result includes visual feedback on the *acceptance* or the *rejection* of the input, as well as visual feedback on each of the automaton's *execution steps* separately.

Additionally, the user can ***save or load*** his design to/from the hard drive, ***export*** the graph to various image file formats, and ***import*** an automaton described using the ***JSON*** language. This last feature is quite interesting, because JSON proved to be a very natural, human-readable means to describe an automaton, even though it had not been used for that purpose before.

1.3 Overview

If reading the previous sections, made you wonder *"what is he talking about?"*, do not despair. In Chapter 2 we explicitly provide all the required *background information*,

including terminology and basic concepts of Automata Theory. We also list all the *tools* we used to implement our automata simulator with appropriate descriptions.

In Chapter 3, we will be discussing our *problem specifications* and elaborating about the important role that *visual software tools* play in education.

Chapter 4, title *Approach and Implementation*, is the most technical one. Here you will find descriptions of the thought process behind our approach, the details of our solutions, a description of the various obstacles that hindered us along the way, and the means we used to overcome them.

A thesis document about the development of a graphical, educational tool, could not be missing a chapter about *User Evaluation*. In Chapter 5 we explain how we used the *"think-aloud evaluation protocol"* to conduct an **informal** evaluation, why we did that, and how our implementation was affected by our first users' feedback.

In Chapter 6, titled *Related Work*, we put our application to yet another test as we describe various related projects we got our hands on, providing corresponding screenshots. It is up to you, our readers, to decide whether our application withstands competition or not.

Our concluding Chapter 7, mainly summarizes our thoughts from the previous two chapters, and displays the tasks we included into *future work*. The fellow beginner programmer might find something useful in the last section about the *lessons we learned* during the whole process of development.

Finally, one can find the complete *User Manual* of our tool in Appendix A. Students and teachers might prefer to isolate and/or print these few pages, for convenience.

Chapter 2

Background

In order to delve deeper into the specifics of this application, we must first get accustomed with some fundamental concepts which derive from the *theory of computation*, *graph theory*, and *linguistics*. In short, this chapter will provide some brief explanations, covering terms such as: *finite state machines*, *formal languages*, *regular languages*, *deterministic* and *nondeterministic finite automata* and a few others as well.

Secondly, our reader will be able to find a brief overview of the tools we used in our implementation, the core being *Java*, a class-based, object-oriented programming language. This second section not only will refresh the memory of those with related former experience, but will also provide a starting ground for the uninitiated as well.

Those of you who already know *what a Finite Automaton is*, *what "nondeterminism" means*, and *what Java and Swing are*, are encouraged to skip this chapter and return to it in case you find a missing piece.

So, let us indulge!

2.1 Automata (or acting by one's self)

Our typical polymath reader has already met the term *automaton* under several different contexts, for sure. Some of them might have included: Mathematics, linguistics, computer science, programming, hardware design, artificial intelligence, biology, cosmology, maybe even sci-fi literature. Whether the occurrence of this term in such a broad field of scientific endeavor intrigues us or not, one thing is certain: *automata*, and their study called *Automata Theory*, have played a role (from insignificant to leading) in humanity's

greatest technological feats for almost over one and a half century. Humanity still carries —and will continue to carry— the automata theory in its luggage. After this brief introduction, one should have asked by now: *What is an automaton?*

Harry Lewis and Christos Papadimitriou, in their book *Elements of Theory of Computation* [3], provide us with a disarmingly simple definition of an automaton, in a humble footnote:

> "An *automaton* (pronounced: o-to-ma-ton, plural: *automata* is a machine designed to respond to encoded instructions; a robot." [3, p. 55]

We will try to expand upon that a bit. Automata theory originates from mathematics, where we find the notion of an abstract, mathematical object, called *abstract machine*[1] which proved to be quite useful in studying and computing the solutions to specific categories of problems. In other words, this abstract machine is a *mathematical model of computation*. There are many different kinds of automata and we tend to organize and classify them by the kind of problems they help us model, study, and solve.

2.1.1 Automata and Formal Languages

Automata theory is also closely related to *formal language theory* and it is very common to describe an automaton as one that *recognizes* or *accepts* a formal language. In fact, automata are often classified by the class of formal languages they are able to recognize. So, before proceeding to more formal definitions of automata, we must first make a quick pass through the basics of formal language theory.

In mathematics, computer science and linguistics, a *formal language* is a set of strings of symbols that may be constrained by rules that are specific to it. Different, distinct sets of rules define different and distinct types of formal languages, respectively. From all the types of formal languages we can find, we are only interested in the —so called— *regular languages*, the ones that can be described and defined by *regular expressions*, but more on that later. The *alphabet* of a formal language is the set of symbols, letters or tokens from which the strings of the language may be formed; frequently it is required that the alphabet is *finite*. The strings formed from this alphabet are called *words*.

The field of formal language theory studies primarily the purely syntactical aspects of such languages, or in other words, their internal structural patterns. Formal language theory sprang out of linguistics, as a way of understanding the syntactic regularities of

[1] The terms *Machine* and *Automaton* are considered equivalent and interchangeable. However, computer scientists tend to prefer using the term *Automaton* (with the exception of *Turing Machines*).

natural languages. In computer science, formal languages are used among others as the basis for defining *programming languages* and formalized versions of subsets of natural languages in which the words of the language represent concepts that are associated with particular meanings or semantics [4].

According to the book *Elements of theory of computation* [3]: "A central issue in the theory of computation is the representation of languages by finite specifications". Later we read that: "There are many ways of representing a language, each more powerful than the last in the sense that each is capable of describing languages the previous one cannot." [3, p. 47]

In fact, it turns out that Automata are a great, powerful way to represent and depict formal languages and study the words that formulate under that language's restrictions. This is the reason why Automata theory and formal language theory are so closely related and the reason why automata have vast, enormous applications in computer text editing, compiler design, natural language processing, programming language design and artificial intelligence.

2.1.2 Definition

Having said all that, we can attempt another informal definition of an automaton:

An *automaton* is a mathematical object that is supposed to run on some given sequence of inputs in discrete-time steps. An *automaton* reads one input at each time step that is picked up from a set of symbols that belong to an alphabet. At any time, the symbols so far fed to the automaton as input, form a word.

An *automaton* contains a finite set of *states*. At each step of some run, the automaton is *in* one of its states, starting from the *starting state*. At each time step when the automaton reads a symbol, it *transits* to a next state that is decided by a function that takes current state and the symbol currently read as parameters. This function is called the *transition function*. The automaton *reads* the symbols of the input word one after another and *transits* from state to state according to the transition function, until the word is read completely.

Once the input word has been read, the automaton is said to have been stopped and the state at which automaton has stopped is called *final state*. Depending on the final state, it is said that the automaton either *accepts* or *rejects* an input word. There is a subset of states of the automaton, which is defined as the set of *accepting states*. If the final state is an accepting state, the automaton *accepts* the word. Otherwise, the word

is *rejected*. The set of all the words accepted by an automaton is called the *language recognized* by the automaton [5].

Finally, all computational problems that can be represented in a finite length of symbols and are reducible into the accept/reject question on words can be studied and solved using automata. Here lies the reason *Automata Theory* plays such a crucial role in computational theory. Also, as we mentioned earlier as well, automata are categorized by the classes of languages they recognize, because languages themselves model different types of problems. An indicative, partial list of types of automata, and their respective languages is given in Figure 2.1.

Automaton	Recognizable language
Nondeterministic/Deterministic Finite state machine (FSM)	regular languages
Deterministic pushdown automaton (DPDA)	deterministic context-free languages
Pushdown automaton (PDA)	context-free languages
Linear bounded automaton (LBA)	context-sensitive languages
Turing machine	recursively enumerable languages
Deterministic Büchi automaton	ω-limit languages
Nondeterministic Büchi automaton	ω-regular languages
Rabin automaton, Streett automaton, Parity automaton, Muller automaton	ω-regular languages

FIGURE 2.1: Partial list of automata types and the languages they recognize. [5]

Visual representation

There are certain conventions when it actually comes to depicting an automaton and we should clearly state what the common consensus is regarding that matter. In the *Elements of the Theory of Computation* [3], and almost everywhere else, *states* are visually represented by a *circle* and an identification string. *Starting states* have a small arrow (or an arrowhead) pointing to them from anywhere, while *accepting states* are represented by a *double* circle. *Transitions* are represented by a *directed arrow*. A string of symbols (usually a single character) is written nearby, to indicate the appropriate input that must be given for the machine to perform that particular transition. Our application, of course, fully complies to all of the above. Figure 2.2 shows a simple automaton depicted under these notational conventions.

2.1.3 Educational Value

There is no coincidence in the fact that every computer scientist in the world has taken a course on Automata theory at least once and probably used one directly or through

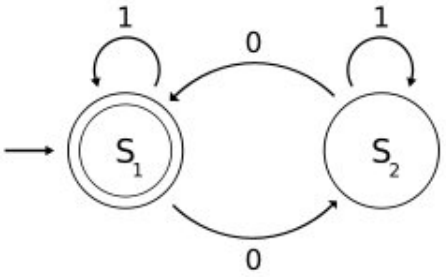

FIGURE 2.2: A simple example of an automaton. [5]

its vast array of applications. Automata theory, as a pedagogical tool, is exemplary in teaching, learning, and researching computation theory. As Michael Sipser states:

> "Automata theory is an excellent place to begin the study of the theory of computation. The theories of computability and complexity require a precise definition of a *computer*. Automata theory allows practice with formal definitions of computation as it introduces concepts relevant to other non-theoretical areas of computer science." [1, p. 3]

Automata simulators, accordingly, play their own important role in teaching, learning, and researching automata theory and have been used in universities for decades. Citing from the abstract of the review article titled *Fifty years of automata simulation: a review* [6],

> "The article concludes with an advocacy for continuing research on simulation of automata and integration of automata simulators in teaching."

Undoubtedly, the development of a similar tool for our needs, in full compliance with the terminology and notation in the book *Elements of theory of computation* [3], will truly enhance the way *Theory of Computation* is being taught at our institution. Our own application, falls under the category of *visualization-centric automata simulators*, in contrast to the *language-based automata simulators*[1].

2.1.4 Deterministic/Nondeterministic Finite Automata

The subject of this thesis does not concern automata in general, but only two specific types: [2]

[1] A classification of the automata simulators broadly into these two categories on the basis of their design paradigms has been developed by Chakraborty et al. [6].

[2] Strictly speaking there is only one distinct type, the Finite Automaton, with or without the feature of nondeterminism.

- *Deterministic Finite Automata (DFA)*
- *Nondeterministic Finite Automata (NFA)*

DFAs and NFAs, also known as deterministic/nondeterministic finite state machines, are the simplest of their kind. Quoting from the *Elements of Theory of Computation*:

> "Here we take up a severely restricted model of an actual computer called a **finite automaton**, or **finite-state machine**. The finite automaton shares with a real computer the fact that it has a "central processing unit" of fixed, finite capacity. It receives its input as a string, delivered to it on an input tape. It delivers no output at all, except an indication of whether or not the input is considered acceptable. It is, in other words, a language recognition device ... What makes the finite automaton such a restricted model of real computers is the complete *absence of memory* outside its fixed central processor." [3, p. 55]

Definitions

Our first ***informal*** definition derives from trying to describe the operation of a *deterministic finite automaton*. Quoting from the *Elements* once again:

> "Strings are fed into the device (finite automaton) by means of an **input tape**, which is divided into squares, with one symbol inscribed in each tape square. The main part of the machine itself is a "black box" with innards that can be, at any specified moment, in one of a finite number of distinct internal **states**. This black box —called the **finite control**— can sense what symbol is written at any position on the input tape by means of a movable **reading head**. Initially, the reading head is placed at the leftmost square of the tape and the finite control is set in a designated **initial state**. At regular intervals the automaton reads one symbol from the input tape and then enters a new state *that depends only on the current state and the symbol just read*; this is why we shall call this device a *deterministic* finite automaton, to be contrasted to the nondeterministic version introduced in the next section. After reading an input symbol, the reading head moves one square to the right on the input tape so that on the next move it will read the symbol in the next tape square. This process is repeated again and again; a symbol is read, the reading head moves to the right, and the state of the finite control changes. Eventually the reading head reaches the end of

> the input string. The automaton then indicates its approval or disapproval of what it has read by the state it is in at the end: if it winds up in one of a set of **final states** the input string is considered to be **accepted**. The **language accepted** by the machine is the set of strings it accepts." [3, p. 57]

Taken from the same source, the formal, mathematical definition of a DFA is shown in Figure 2.3.

Definition 2.1.1: A **deterministic finite automaton** is a quintuple $M = (K, \Sigma, \delta, s, F)$ where

K is a finite set of **states**,
Σ is an alphabet,
$s \in K$ is the **initial state**,
$F \subseteq K$ is the set of **final states**, and
δ, the **transition function**, is a function from $K \times \Sigma$ to K.

FIGURE 2.3: Formal definition of a *Deterministic Finite Automaton* [3, p. 57]

The Feature of Nondeterminism

By adding the feature of nondeterminism to a DFA, we get the ***N****ondeterministic* ***F****inite* ***A****utomaton*. This is a new, powerful upgrade in our armament, which vastly facilitates our design and study of finite automata. However, we must clarify something first.

> *"NFAs are not meant as realistic models of computers. They are simply a useful notational generalization of finite automata, as they can greatly simplify the description of these automata."* [3, p. 64]

A first, simplistic way to describe them, would be by noting that although DFAs have a unique computation for a given input every time, NFAs have multiple parallel computation runs for the same input. But, let us listen to Lewis and Papadimitriou once again:

> "*Nondeterminism* is essentially the ability to change states in a way that is only partially determined by the current state and input symbol. That is, we shall now permit several possible "next states" for a given combination of curent state and input symbol. The automaton, as it reads the input string, may choose at each step to go into any one of these legal next states; the

choice is not determined by anything in our model, and is therefore said to be *nondeterministic*. On the other hand, the choice is not wholly unlimited either; only those next states that are legal from a given state with a given input symbol can be chosen.“ [3, p. 63]

Michael Sipser brings even more light to the issue, by using an example. Figure 2.4 and the text that follow, are both taken from his book, *Introduction to the Theory of Computation*:

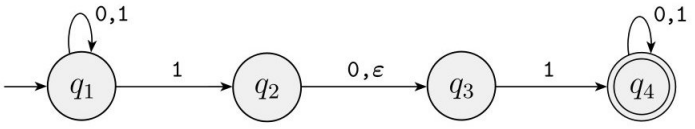

FIGURE 2.4: The nondeterministic finite automaton N_1.

“ ...First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. The NFA shown in Figure 2.4 violates that rule. State *q1* has one exiting arrow for 0, but it has two for 1; *q2* has one arrow for 0, but it has none for 1. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol.

Second, in a DFA, labels on the transition arrows are symbols from the alphabet. This NFA has an arrow with the label ϵ. In general, an NFA may have arrows labeled with members of the alphabet or ϵ. Zero, one, or many arrows may exit from each state with the label ϵ.

How does an NFA compute? Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. For example, say that we are in state q1 in the NFA N_1 (Figure 2.4) and that the next input symbol is a 1. After reading that symbol, the machine splits into multiple copies of itself and follows *all* the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if *any one* of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an ϵ symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple

> copies, one following each of the exiting ϵ-labeled arrows and one staying at the current state. Then the machine proceeds nondeterministically as before." [1, p. 48]

Viewing nondeterminism as a kind of *parallel* computation, is one way of thinking, but in this application we adopted another view, mostly for implementation-related reasons, by considering the operation of NFAs as a *tree of possible computations*. Michael Sipser also mentions that, later on the same page:

> "The root of the tree corresponds to the start of the computation. Every branching point in the tree corresponds to a point in the computation at which the machine has multiple choices. The machine accepts if at least one of the computation branches ends in an accept state..." [1, p. 48]

Figure 2.5 shows the formal definition of an NFA, which is a modified version of the one about DFAs (Figure 2.3), in order to accommodate the feature of nondeterminism:

Definition 2.2.1: A **nondeterministic finite automaton** is a quintuple $M = (K, \Sigma, \Delta, s, F)$, where

K is a finite set of **states**,
Σ is an alphabet,
$s \in K$ is the **initial state**,
$F \subseteq K$ is the set of **final states**, and
Δ, the **transition relation**, is a subset of $K \times (\Sigma \cup \{e\}) \times K$.

FIGURE 2.5: Formal definition of an *Nondeterministic Finite Automaton* [3, p. 65]

2.1.5 Equivalence of DFA and NFA

In the *Elements of Theory of Computation* [3, ch. 2.6], it is proven that for each NFA, there exists an equivalent DFA, such that both recognize the same regular language. In fact, there is a conversion method that allows us to construct an equivalent DFA from any given NFA. That method is called *powerset construction*, or *subset construction*, or *Rabin-Scott powerset construction* [7].

Regarding the fact that DFAs and NFAs are equivalent, *Michael Sipser* states:

> " Such equivalence is both surprising and useful. It is surprising because NFAs appear to have more power than DFAs, so we might expect that NFAs

> recognize more languages. It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language." [1, p. 54]

Our application features the algorithmic conversion of NFAs to DFAs, using the aforementioned algorithm of *powerset construction*. More details about the algorithm and our implementation may be found in the respective chapter.

2.1.6 Regular Languages, Regular Expressions

As mentioned earlier in *Automata And Formal Languages* (section 2.1.1), the only class of formal languages that concerns us, is the one DFAs and NFAs recognize, called *regular languages* (Figure 2.1).

In both the *Elements of Theory of Computation* [3] and the *Introduction to the Theory of Computation* [1, p. 66] it is proven that: *"A language is regular if and only if some regular expression describes it."* So, let us talk about what a *regular expression* is and what the *regular operations* that define it are.

Regular Operations

Quoting directly from the *Introduction to the Theory Of Computation* [1, p. 44]:

> In arithmetic, the basic objects are numbers and the tools are operations for manipulating them, such as + and ×. In the theory of computation, the objects are languages and the tools include operations specifically designed for manipulating them. We define three operations on languages, called the *regular operations*, and use them to study properties of the regular languages.

DEFINITION 1.23

Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows:

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

FIGURE 2.6: The *formal* definition of regular operations. [1, p. 44]

> You are already familiar with the union operation. It simply takes all the strings in both A and B and lumps them together into one language.

The concatenation operation is a little trickier. It attaches a string from A in front of a string from B in all possible ways to get the strings in the new language.

The star operation is a bit different from the other two because it applies to a single language rather than to two different languages. That is, the star operation is a **unary operation** instead of a **binary operation**. It works by attaching any number of strings in A together to get a string in the new language. Because "any number" includes 0 as a possibility, the empty string ε is always a member of A^*, no matter what A is.

For example, let the alphabet Σ be the standard 26 letters {a, b, ..., z}. If A = {good, bad} and B = {boy, girl}, then

$$A \cup B = \{good, bad, boy, girl\},$$
$$A \circ B = \{goodboy, goodgirl, badboy, badgirl\}, \text{ and}$$
$$A^* = \{\varepsilon, good, bad, goodgood, goodbad, badgood, badbad,$$
$$goodgoodgood, goodgoodbad, goodbadgood, goodbadbad, \ldots\}.$$

Regular Expressions

Quoting from the same book [1, p. 63]:

"In arithmetic, we can use the operations + and × to build up expressions such as

$$(5 + 3) \times 4$$

Similarly, *we can use the regular operations to build up expressions describing languages, which are called regular expressions.* An example is:

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

The value of the arithmetic expression is the number 32. The value of a regular expression is a language. In this case, the value is the language consisting of all strings starting with a 0 or 1 followed by any number of 0*s*."

Here is a second example of a regular expression from the same source [1, p. 64]:

"Another example of a regular expression is

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

> It starts with the language $(0 \cup 1)$ and applies the $*$ operation. The value of this expression is the language consisting of all possible strings of 0s and 1s."

Later, Michael Sipser adds:

> "Regular expressions have an important role in computer science applications. In applications involving text, users may want to search for strings that satisfy certain patterns. Regular expressions provide a powerful method for describing such patterns. Utilities such as *awk* and *grep* in UNIX, modern programming languages such as Perl, and text editors all provide mechanisms for the description of patterns by using regular expressions."

2.1.7 Equivalence with Finite Automata

In both the *Elements of the Theory of Computation* and the *Introduction to the Theory of Computation* it is proven that **regular expressions** are equivalent to **DFAs** and —therefore— to **NFAs** as well [1, Sec. 1.3.2][3, Sec. 2.3].

Quoting Lewis and Papadimitriou:

> "The class of languages accepted by finite automata, deterministic or nondeterministic, is the same as the class of *regular languages* —those that can be described by regular expressions, ..."

And Michael Sipser adds:

> "Regular expressions and finite automata are equivalent in their descriptive power. This fact is surprising because finite automata and regular expressions superficially appear to be rather different. However, any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa."

So to summarize, ***every problem that relates to, or can be modeled after a regular language, can be described equivalently and interchangeably by***:

- *a regular expression*,
- *a deterministic finite automaton*,

- *a nondeterministic finite automaton.*

Our own application does not support regular expressions at the moment, although it does feature the algorithmic conversion of NFAs to DFAs and vice-versa[2]. Conversions to/from regular expressions may be supported in the future.

2.2 Tools of the Trade

The task of choosing a programming language and its related tools, was not an easy one. There are many libraries and frameworks regarding graphical user interface (GUI) design in desktop applications, spanning a huge array of programming languages. Not to mention, of course, the evolution in Web design technologies that provide quick and elegant solutions in web application design.

What proved to be a significant factor for us though, was the learning curve and the overhead from getting accustomed with new tools of choice. In that respect, considering our former experience with object-oriented programming with *Java* and *C++*, the scales were tipped towards that direction. Moreover, object-oriented languages have proved their adequacy, efficiency and power in GUI design and programming, time and time again. Even the concepts we now call *Design Patterns* (a popular and efficient program designing doctrine which is very common in GUI programming) originate from the very foundations of *Object-Oriented Programming* (OOP). As an added bonus to that, our application would become platform independent, an attribute that most object-oriented languages facilitate with their runtime environments.

Secondly, another factor was the available documentation, as well as community support and third-party frameworks and libraries. With no doubt, this is the field that Java excels at. By being so popular in industry and academia for over a decade, Java has been nurtured by many people and grew up to be a sure-fire solution to most common problems. Of course, Java has its strong/weak points, but if the problem at hand is of moderate difficulty, it is guaranteed that the programmer can get a lot of help. From Oracle's tutorials to community forums and to third-party libraries, the programmer is provided with such support that it is unlikely he/she will hit a brick wall. However, all these goodies do not come without side-effects. Java is a well-known trouble-maker when it comes to strict demands in performance and resource management. The freedom other languages (e.g. C, C++) provide regarding that matter is superior. Also, combining many third-party, general solution packages and libraries into a single working application, is not as simple as it sounds. Except the costs in performance that come

[2]The conversion of a DFA to NFA is trivial, since any DFA is also an NFA.

from using general and not specific-purpose code, one also runs the risk of dealing with difficult-to-trace bugs, especially if the authors did a sloppy job at documenting and debugging their code.

The third factor were the specifications of our problem, which are discussed in the next chapter. The only thing that we are going to mention here, is that our demands in performance were not really that high. *Data visualization, data representation, the study of graphs (Graph Theory)* and other related fields, are vast and oftentimes highly demanding in computational and memory costs when it comes to their applications. The demands of our own application, on the other hand, would seem childish from a graph theorist perspective. Our common user will probably never design a graph with more that thirty or forty nodes (states), and our algorithms will never challenge his/hers CPU and RAM with their computational complexity. Networking researchers, for example, could easily be put to the task of finding the shortest or the cheapest paths, inside a graph of as many as 200,000 nodes.

Taking all the above considerations into account, we decided to use *Java*. After a long search, we also settled on the additional tools that would supplement *Java* in the creation of our application. Here is a list of everything we used, with short descriptions following next:

- Java (http://www.java.com/en/)
- Java SE (http://www.oracle.com/technetwork/java/javase/overview/)
- Swing (http://en.wikipedia.org/wiki/Swing_(Java))
- JUNG (http://jung.sourceforge.net/)
- GraphML (http://graphml.graphdrawing.org/)
- JSON (JSON.simple) (http://code.google.com/p/json-simple/)
- FreeHEP (http://java.freehep.org/)
- NetBeans IDE (http://netbeans.org/)

2.2.1 Java

Java is a general-purpose, concurrent, class-based, object-oriented computer programming language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers “write once, run anywhere”

(WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java applications are typically compiled to bytecode (class file) that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture.

Java was originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

As of May 2007, in compliance with the specifications of the Java Community Process, Sun relicensed most of its Java technologies under the GNU General Public License. Others have also developed alternative implementations of these Sun technologies, such as the GNU Compiler for Java and GNU Classpath.

There were five primary goals in the creation of the Java language:

1. It should be "simple, object-oriented and familiar"
2. It should be "robust and secure"
3. it should be "architecture-neutral and portable"
4. It should execute with "high-performance"
5. It should be "interpreted, threaded and dynamic"

One characteristic of Java is portability, which means that computer programs written in the Java language must run similarly on any hardware/operating- system platform. This is achieved by compiling the Java language code to an intermediate representation called Java bytecode, instead of directly to platform-specific machine code. Java bytecode instructions are analogous to machine code, but they are intended to be interpreted by a virtual machine (VM) written specifically for the host hardware. End-users commonly use a Java Runtime Environment (JRE) installed on their own machine for standalone Java applications, or in a Web browser for Java applets. A major benefit of using bytecode is porting. However, the overhead of interpretation means that interpreted programs almost always run more slowly than programs compiled to native executables would [8][9][10][11].

2.2.2 Java SE

Java Platform, Standard Edition or Java SE is a widely used platform for development and deployment of portable applications for desktop and server environments. Strictly

speaking, Java SE is a platform specification. It defines a wide range of general purpose APIs (Application Programming Interfaces) and also includes the Java Language Specification and the Java Virtual Machine Specification. One of the most well known implementations of Java SE is Oracle Corporation's Java Development Kit (JDK). Another well-known implementation is OpenJDK, which is the official Java SE 7 reference implementation [12][13][14].

The Java Runtime Environment (JRE) and Java Development Kit (JDK) are the actual files downloaded and installed on a computer to run or develop Java programs, respectively. For the development of our application, we used *Sun/Oracle's JDK Version 6*.

2.2.3 Swing

Swing [15] is the primary Java GUI widget toolkit. It is part of Oracle's Java Foundation Classes (JFC) — an API for providing a graphical user interface (GUI) for Java programs. Swing was developed to provide a more sophisticated set of GUI components than its predecessor, the Abstract Window Toolkit (AWT). Swing provides a native look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to exhibit a look and feel independently of the underlying platform. It has more powerful and flexible components than AWT. In addition to familiar components such as *buttons*, *check boxes*, and *labels*, Swing provides several advanced components such as *tabbed panel*, *scroll panes*, *trees*, *tables*, and *lists*.

Swing is architectured after the *Model-View-Controller* design pattern, which we will cover in Chapter 4, and follows a single-threaded programming model. Additionally, this framework provides a layer of abstraction between the code structure and graphic presentation of a Swing-based GUI.

2.2.4 JUNG

JUNG (the Java Universal Network/Graph Framework) [16] [17] is an open source graph modeling and visualization framework written in Java, under the BSD license. The framework comes with a number of layout algorithms built in, as well as analysis algorithms, such as graph clustering and metrics for node centrality.

JUNG's architecture is designed to support a variety of representations of entities and their relations, such as directed and undirected graphs, graphs with parallel edges, and hypergraphs. In our case, we are interested in representing the *states* and *transitions* of DFAs and NFAs, so we need to use a *directed graph*, one that also *allows parallel edges*.

By *directed* we mean that the *edges* (representing transitions) are painted as *arrows*, to denote direction. "*Allowing parallel edges*" means that linking a specific pair of nodes with more than one edge is *allowed*, since multiple transitions are also allowed between states in NFAs.

JUNG has enormous capabilities and provides the most important features to support areas, such as *graph theory*, *data mining* and *social network analysis*. It includes implementations of a number of algorithms for clustering, decomposition, optimization, random graph generation, statistical analysis, and calculation of network distances, flows, and importance measures. For those of you likely to stumble upon such matters in the future, we strongly recommend keeping JUNG in mind.

In our case, however, it is true that none of the above advanced features was of any use. So, considering that our graphs will likely be very simple and would not be subject to assiduous analysis, the use of JUNG may seem to be an "overkill". Nevertheless, in fact, JUNG provides a visualization framework that makes it easy to design graphs and interact with them. We can use one of the layout algorithms provided or use the framework to create our own custom layouts and, in addition, we can use it to easily customize the appearance of all graph components (e.g. changing the size, shape, and color of nodes and edges). All these features come in full compliance with Java's Swing, as intended by JUNG's creators.

One last thing to note, is that we ended up using JUNG after trying several alternatives. JUNG ended up being superior to all of them for various reasons. Our other attempts included: JGraph, JGraphT, and Grappa.

2.2.5 GraphML

GraphML [18] [19] is an XML-based file format for graphs, which is already integrated into JUNG and can be easily used with the appropriate customizations. The GraphML file format results from the joint efforts of the graph drawing community to define a common format for exchanging graph structure data. It uses an XML-based syntax and supports the entire range of possible graph structure constellations including *directed*, *undirected*, *mixed graphs*, *hypergraphs*, and application-specific attributes.

A GraphML file contains a graph element, within which there is an unordered sequence of node and edge elements. Each node element should have a distinct id attribute and each edge element has source and target attributes that identify the endpoints of an edge by having the same value as the id attributes of those endpoints. Figure 2.7 shows what a simple NFA with two nodes (q0 and q1) and one edge between them will look

like in GraphML. GraphML was the file format of our choice, in order to support the feature of saving/loading our graphs to/from the hard drive.

```
%, title=\emph{A simple graphml example.}, label=graphml_example]
<graph edgedefault="directed">
<data key="input"> </data>
<data key="alphabet"> </data>
<data key="type">dfa</data>
<node id="q0">
<data key="isStarting">false</data>
<data key="isAccepting">false</data>
<data key="y">258.0</data>
<data key="x">299.0</data>
</node>
<node id="q1">
<data key="isStarting">false</data>
<data key="isAccepting">false</data>
<data key="y">245.0</data>
<data key="x">446.0</data>
</node>
<edge source="q0" target="q1">
<data key="symbols">a</data>
</edge>
</graph>
</graphml>
```

FIGURE 2.7: A simple GraphML example.

2.2.6 JSON

Apart from storage and retrieval using GraphML for representation, our application supports the extra feature of *importing a graph from a JSON file*. *JSON* or *Javascript Object Notation* [20] [21] is a text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting languages for representing simple data structures and associative arrays, called *objects*. Despite its relationship to JavaScript, it is language-independent, with parsers available for many languages. As far as Java is concerned, we opted for the *JSON.simple* toolkit [22], which makes encoding and decoding JSON text in Java very easy. For example, to parse a .json file using JSON.simple, we just included the library's .jar file in the project's *classpath*, imported *org.json.simple.** package in our respective Java class, and typed something like:

```
JSONParser myParser = new JSONParser();
JSONObject obj = (JSONObject)myParser.parse(someFile.json);
```

The JSON format is often used for serializing and transmitting structured data over a network connection. It is used primarily to transmit data between a server and web application, serving as an alternative to XML.

JSON's basic types are:

Number (double precision floating-point format in JavaScript, implementation-dependent)

String (double-quoted Unicode, with backslash escaping)

Boolean (true or false)

Array (an ordered sequence of comma-separated, any-typed values, enclosed in square brackets; the values do not need to be of the same type)

Object (an unordered collection of comma-separated `key:value` pairs with the `:` character separating the key and the value, enclosed in curly braces; the keys must be strings and should be distinct from each other)

null (empty)

Non-significant white space may be added freely around the "structural characters" (i.e. brackets, colons, and commas).

The example in Figure 2.8 shows the JSON representation of a simple DFA with two states, 'q0' (starting and accepting state) and 'q1'. The alphabet that forms this automaton's language contains only two characters, 'a' and 'b'. As for the transitions, if the automaton reads 'a' it stays at the same state and if it reads 'b' it transits to the other state. For example, this automaton starts from 'q0' and if it reads 'b' it transits to state 'q1', while if it reads 'a' it stays at 'q0'.

```
%,title=\emph{An example of using JSON to describe Finite Automata.}]
 {
   "type":"dfa",
   "states":["q0","q1"],
   "transitions": [["q0","a","q0"],
                   ["q0","b","q1"],
                   ["q1","a","q1"],
                   ["q1","b","q0"]
                  ],
   "startState": "q0",
   "acceptStates":["q0"],
   "alphabet":"ab",
   "input": "aababbaab"
 }
```

FIGURE 2.8: An example of using JSON to describe Finite Automata.

Note how easy it is for a human to read JSON text. To our knowledge, JSON has never been used as a *Finite Automata Description Language* before.

2.2.7 FreeHEP

We already covered what formats our application *imports* from, but what about *exporting*? Well, our users will be delighted to find out that our application supports exporting to four different image formats (one vector and three bitmap formats), with more to be added in the future. So far these are:

- SVG (Scalable Vector Graphics)
- BMP (Bitmap Image File)
- PNG (Portable Network Graphics)
- GIF (Graphics Interchange Format)

The library that allows us to realize this feature is called *FreeHEP* [23] [24]. It is an open-source Java library initially designed to make programming high-energy physics applications easier. Nonetheless, many self-contained APIs (Application Programming Interfaces) in the library are generic and suitable for other applications. In particular, we used a package called *VectorGraphics* which completely met our needs.

The way FreeHEP is used, is as simple as it gets: After including the library's `.jar` file in the project's classpath, import the package `org.freehep.util.export.ExportDialog` in your desired Java class, and then create and show a new *ExportDialog* instance, as shown below:

```
ExportDialog export = new ExportDialog();
export.showExportDialog(frame,"Export graph as...",wvv,"export");
```

Here, the class *ExportDialog* is a *JOptionPane* subclass, which in turn is a native Swing component.

2.2.8 NetBeans IDE

Last, but not least, the glue that held our project together, was the IDE (Integrated Development Environment) we used. Our choice, *NetBeans* [25] [26], along with *Eclipse*, are the most popular IDEs for developing primarily with Java, but also with other languages, in particular *PHP, C/C++*, and *HTML5*. It is also an application platform framework for Java Desktop applications and others.

NetBeans also features its own *GUI design tool* (formerly known as *Project Matisse*), which enables developers to prototype and design Swing GUIs by dragging and positioning GUI components on a canvas-like frame.

NetBeans is written in Java, therefore it can run on Windows, OS X, Linux, Solaris and other platforms supporting a compatible Java Virtual Machine (JVM). The version we used to develop our application is *NetBeans IDE 7.3*.

Chapter 3

Problem Statement

In this chapter we will be asking ourselves some important questions, and hopefully, in an attempt to answer them, we will define the specifications of our problem as well.

Our intention is to develop a *finite automata simulator*, with a *graphical user interface* (GUI), with special emphasis on its use as an *educational tool* for our institution's course on *Theory of Computation*. Therefore, we ought to ask ourselves:

- Why do we need a *finite automata simulator*?
- What are the benefits of having a *GUI*?
- Why should it be used as a *pedagogical tool*?
- What are the *necessary functions* it has to support, in order to be considered such a tool?

Our problem specifications derive from these questions and we should keep them in mind throughout the whole process of development.

3.1 Motivation

In the previous chapter we mentioned how important *Automata Theory* is in the study of the *Theory of Computation* and in Computer Science in general. In fact, every curriculum on Computer Science has at least one course on *Automata Theory*. Consequently,

> "...simulation of automata for pedagogical purposes is an important topic in computer science education research."

This quote is taken from an article titled *Fifty Years of Automata Simulation* [6]. In its introduction we read:

> "Educationists were early to understand that it is difficult to teach and learn automata theory. They thought that perhaps the best way to teach and learn automata theory is to take help of pedagogical tools. Since automata theory revolves around abstract machines and processes, automata simulators were conceived as the most common form of pedagogical tools on automata theory."

This article provides a historical overview on Automata Simulators throughout a period of time spanning five decades, especially emphasizing on those used at universities for pedagogical purposes. From *Curtis' Turing Machine Simulator* [27] (used in *Wesleyan University* in 1965), to the "*Tool suite based on Finite Automaton Description Language*" [28] (used at Jawaharial Nehru University in 2011), this survey lists over thirty different implementations and more than twenty universities that used them.

As of June 2013, the course that covers the fundamentals of *Automata Theory* at our institution[1] is called *Theory of Computation* and it is being taught by Associate Professor *Michail G. Lagoudakis*. However, this is done ***without using a pedagogical tool to model and simulate automata***.

The instructor himself took the initiative to propose and supervise this thesis topic in order to accommodate that particular need, since the benefits of using such tools are thoroughly documented in scientific literature [6] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38]. To put it simply, we hypothesized that many of the mistakes students make while solving their exercises *could easily be avoided*, had there been a tool to provide intuitive, visual feedback.

Sometimes it is as plain and simple as that. The teacher of this course had also noticed that students commit "silly" mistakes quite oftenly, like ommitting to assign a starting state, or ommitting to designate a transition for a particular symbol. In addition to that, when the occassional exercise involved the special case of the *nondeterministic automaton*, our professor noticed that even more pitfalls hindered the students, as it is very tricky to trace an NFA's computation *on paper*[2].

Thus, the existence of an intuitive, interactive tool to provide the students with *visual feedback* on the computational steps of a running automaton, would allow them to experiment and get accustomed with the fundamentals of Automata Theory unhindered.

[1] *Technical University of Crete* (www.tuc.gr), *School of Electronic and Computer Engineering* (www.ece.tuc.gr)

[2] See chapter 2.2.5, section *The Feature of Nondeterminism* (p. 11) as to why that might happens.

3.2 Yet Another Finite Automata Simulator

First of all, we decided that our simulator would only cover *DFA* and *NFA* and ***not*** the rest of the automata that this course covers, which includes *pushdown automata* and *Turing Machines*. This decision was taken mostly due to our time constraints. A future version of this simulator would have to support the other two types as well, for it to be considered *completed*.

Secondly, it would have to be intuitive, interactive, free and open - source, and support a number of desired features that we will discuss about in detail, in the next section called *Problem Specifications*.

Thirdly, the terminology, notations and the symbols we would use, should comply to the book this course is based upon, Lewis' & Papadimitrious' *Elements of Theory of Computation* [3].

Taking the above into account, and after trying out some of the simulators that already existed, we decided that we should make a new finite automata simulator of our own. In a later chapter called *Related Work*, we will provide a more detailed comparison of our solution with the most popular alternatives.

3.3 Problem Specifications

In this section we will attempt an enumeration of the parameters that specify our problem. These mostly include *features* that our simulator should have, in order to fulfill its role as a pedagogical tool to model and simulate finite automata.

So, our simulator:

- Should be ***free*** and ***open source***.
- Should be ***intuitively easy to use*** and ***user-friendly*** in appearance and behaviour, in order not to distract from its main purpose: the study of automata.
- Should ***represent*** an automaton by means of a ***graph***, in which the ***states*** are represented by ***nodes*** having the shape of a ***circle***, and the ***transitions*** by ***edges*** having the shape of an ***arrow***.[3]

[3]Since we are depicting automata using *graphs*, oftentimes we will be using the terms *nodes* and *edges* to denote the automaton's *states* and *transitions* respectively. This accords with the terminology of *Graph Theory*.

- Should depict the ***starting state*** as a normal node with a small ***arrowhead*** pointed towards it from anywhere, and the ***accepting states*** as a node with the shape of a ***double circle***. The user should also be able to change the contents of the ***input tape*** and the ***alphabet***, which in turn should be visualized in an obvious location.

- Should be ***interactive***, not only during the design of the automaton but during its computation as well.

- Should support the ***graphical editing*** of automata, as well as the feature of ***importing*** from a file, using some simple ***finite automata description language***.

  By ***graphical editing*** we mean that the user should be able to ***create***, ***name*** and ***edit*** nodes and edges, as well as arrange the graph's topography by moving the elements at will, via mouse clicks and gestures in a visible area inside the window of our application that will act as a ***canvas***.

  Importing from a file, on the other hand, implies that we had to settle on a ***simple***, ***easy to read*** description language for finite automata first. Afterwards, our application should be able to ***parse*** files containing text written in that language and describing a finite automaton, and then lastly, ***render*** the graph that represents it ***visible***. For the procedure to be simple and efficient, the user shouldn't have to type the locations of all the nodes manually, so the ***layouting*** of the graph should be done ***algorithmically***.

Moving on,

- Our simulator should provide the user with ***visual feedback of the automaton's computational run***, regarding the ***current state***, the ***input symbol about to be read*** and the respective ***transition*** that has been performed, and all of these at the ***beginning***, ***during the time***, and at the ***end*** of the automaton's execution on the given input.

- It should support a ***step-by-step playback*** of the run, with the options of going to ***start***, ***next step***, ***previous step*** and ***end***, encompassing ideas from programming tools for general programming, like backwards-in-time debugging and tracing.

- It should support the feature of ***nondeterminism***, not only by editing and running an NFA, but also by giving users the option to ***convert an NFA to a DFA***, using the algorithms that appear in *Elements of Theory of Computation* [3]. This last feature is considered to be of paramount educative value when studying and teaching NFAs.

There is also a number of *secondary features* that would be welcomed by teachers and students alike. These include:

- The capability of ***saving to*** and ***loading from*** the hard drive.
- ***Exporting*** the visual representation of the automaton in a variety of common image file formats.
- The option to have a ***textual representation*** of the automaton's computational run accompanying the visualization in a log-like manner, in full compliance with the notation Lewis & Papadimitriou use in their book *Elements to Theory of Computation* [3] (see page 67 for example).
- The option to set the ***designation*** or ***identification*** of the nodes to ***automatic***, which means that the user wouldn't have to name each and every new node he/she creates. The application would do that for him/her, using a given prefix and incremental numbering as a suffix to construct the new id.

Conclusion

In our opinion, a tool that adheres to the above would be a significant teaching assistant in our professor's disposal and would help the students grasp the fundamentals of Automata Theory, through obtaining a hands-on experience on deterministic and nondeterministic finite automata. Moreover, it would deepen the students' level of engagement to this course by encouraging them to be active participants in the educational process and not just passive receivers. The related references in section **Motivation** 3.1 strongly support this assumption.

Chapter 4

Approach and Implementation

Every programmer knows, that the process of developing *useful* software is a dynamic, living one, full of missteps and breakthroughs, mental slumps and creative bursts, disapointments and delights, ebbs and flows. Alas, this road is paved with knowledge and, in the end, the reward is immense. Nothing can substitute hands-on experience, for in this domain there is only one currency, *Time* - and you have to spend lots of it.

Nonetheless, there is still a lot to learn from someone that went through that process and decided to tell the story. Tales of disastrous mistakes, enormous blunders and unfortunate accidents, but also of paradigms of ingenuity, moments of brilliance and triumphs of reasoning, are what enable us to stand on the shoulders of other people (hopefully gigantic) and aim even higher.

In this chapter we share our humble experiences through the process of developing this tool. May others learn from our oversights and appreciate what we achieved. This is: *Our Approach & Implementation.*

4.1 Two Degrees of Decomposition

Our first approach involved *decomposing* the problem into different parts, a common strategy in programming, which has proven its worth amongst fellow programmers. This helped us clear the confusion that comes along when dealing with a problem *as a whole* for the first time. It also allowed us to focus our attention on the *smaller, more managable world* of our respective parts.

Since we wanted to develop a finite automata simulator with a GUI, at first we roughly divided the problem into two parts:

Machine

As in *finite state machine*, this part is the one responsible for simulating the *behaviour* of a finite automaton.

GUI

The part responsible not only for the *visual representation* of the automaton, but for the *user interaction* as well.

This was the *first degree of decomposition*, and frankly, more than enough for a kick-start.

As for the place to start, we opted for the **machine**. This was intended to be our "*backend*", or the piece of software supposed to stay hidden behind the scenes of our GUI, and the one that would provide us with all the functionality of a finite automaton. The user would never have to know exactly *how* his automaton actually runs, but to us it is an important raw material and building block. Having a working simulation of a machine in our disposal would allow us to experiment freely in the coming steps. We only had to make sure that our machine would feature a simple interface, such that would allow us to provide the machine with an *input* (describing a finite automaton, defining the alphabet and the content of the input tape) and receive the result of the computation as *output* (listing the machine's *states* per computation step for the given input and whether that input was accepted or not).

In short, we wanted to implement a prototype version of our machine and then treat it as a ***"black box"*** for the rest of the development.

The opposite would be quite troublesome indeed. Having a GUI without a machine would be of no help at all. The machine provides our core functionality and cannot - and *should not* - comply with assumptions and conventions made whilst developing a graphical user interface. In addition, our GUI would be incomplete because we wouldn't be able to visualize the actual execution of the machine but only its states and transitions in a static way. In any case, a GUI ***cannot be treated like a "black box"***.

As we revisited these issues in a later iteration though, we ended up decomposing the aforementioned parts even further, reaching a *second degree of decomposition*:

Machine was submitted to *refactoring* that led to a different decomposition to accomodate various changes and additions, such as the feature of nondeterminism or the conversion of an NFA to a DFA,

GUI was decomposed to:

- parts responsible for the *visualization* and *appearance* of the graph,

- parts responsible for the visualization of the various *GUI components* (buttons, textfields, menus, etc),
- parts responsible for listening and responding to *user generated events*, such as mouse and keyboard clicks, mouse gestures etc.

In later sections we examine these parts in detail.

These *revisions* to our initial decomposition were not only *expected*, but *desired* as well. Each iteration that resulted in improvements in our GUI gave feedback regarding the machine and vice versa. We strongly advocate following this strategy, in contrast to other approaches that treat the initial design as something *absolute* and *separated* from the implementation.

4.2 Machine

Let us begin then, by describing how the machine works. This will happen in three steps. First, we expose our *core*, a machine that supports only *deterministic* finite automata. In the second step we add to that, ending up with a machine that supports the feature of *nondeterminism* as well, and lastly, in a third step we add two auxiliery functions, the *conversion of an NFA to a DFA* and the *validation of user input* with the appropriate *error messages*.

4.2.1 Our Core

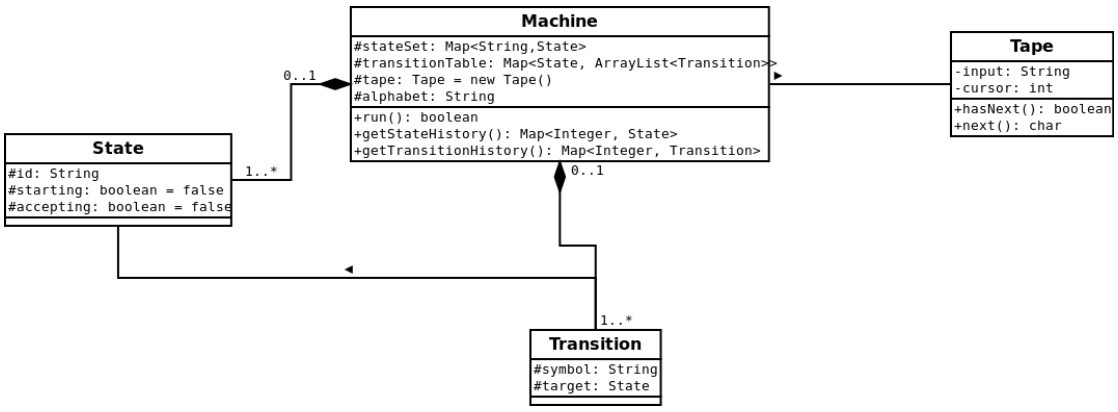

FIGURE 4.1: Class diagram of the Machine's core components. (https://en.wikipedia.org/wiki/Class_diagram)

Our Machine's core components consist of four basic classes. These classes are responsible for the modelling of a ***DFA*** and the simulation of its computational run:

1. *State*,
2. *Transition*,
3. *Tape*,
4. *Machine*.

Let's take a closer look:

State

Instances of this class model our automaton's ***states***. Every state should have an ***id*** of type: String, which must be unique per automaton. Also, each state *may or may not* be an ***accepting state*** and *may or may not* be a ***starting state***. However, for the automaton to be considered valid, there must be ***one and only one starting state***. There is no such restriction in the number of accepting states. A state can be *both accepting and starting at the same time.* This information is stored by means of two variables of type: boolean, which are defaulted to ***false*** and remain such, until the user defines otherwise.

Transition

Instances of this class model our automaton's ***transitions***. Each transition stores a reference to a single ***target state***, and the respective ***symbol*** that must be read in order for the automaton to perform this particular transition. The variable we used for that, was also called ***symbol***, and it was of type: ***char*** initially. In a later iteration though, we refactored it to ***String*** to account for the fact that users should be able to define multiple symbols that can lead the automaton to perform the exact same transition. This should be done in the same manner as in our book *Elements of Theory of Computation*[1]. Note that our transitions ***do not*** hold a reference to their source state.

Tape

This is the class that allows us to handle our automaton's ***input string***, or in other words, a ***word*** formed from a combination of letters from the user's desired ***alphabet*** which the automaton is supposed ***to compute on***. It has a variable called ***input*** of type: String, and a variable called ***cursor*** of type: int (or Integer).

[1]Such as: *'a,b,c, . . . '* etc, see [3, p. 61] for example.

The first variable holds the input word and the second variable keeps track of the symbols already read, by pointing to the next symbol in line. A ***Tape instance*** provides our automaton with the next symbol at every step, and can tell us if the cursor has reached the end (automaton has finished). The methods[2] responsible for these are called ***next()*** and ***hasNext()*** respectively.

Machine

This is our main class regarding the modelling and simulation of an automaton. To initialize an instance of this class, we have to provide it with the following references:

1. the ***alphabet***, stored in a variable of type: String,
2. the ***input***, also stored in a variable of type: String,
3. the ***automaton's states***, stored as values in a *Map*[3] called ***stateSet***, where state ids act as keys (in Java: *Map<String, State>*),
4. the ***automaton's transitions***, stored as values in a *Map* called ***transitionTable***, where all the keys are references to State objects (*Map<State, Transition>*). These objects represent the ***source*** states for all the transitions. The values of the ***transitionTable*** are *Lists*[4] containing ***Transition Object references***. We used a *List* because we wanted to group the transitions by source state, for implementation reasons. Think of this *Map* as a collection of triplets: {*source state, symbol, target state*}, grouped by source state:

```
state A -> { 0, state B}
state A -> { 1, state C}
state B -> { 0, state C}
state B -> { 1, state D}
state C -> { 0, state A}
state C -> ...
.
.
.
(etc)
```

After initializing our Machine, we can produce our automaton's result by calling the ***run()*** method. The output consists of a *Map<Integer, State>* called ***stateHistory*** and a second *Map<Integer, Transition>* called ***transitionHistory***. The integers that act as keys are an incremental numbering of the automaton's steps.

[2] Also called *functions*, or *operations* in other contexts.

[3] In Java, a *Map* is a collection of {*key:value*} pairs. Keys and values can be *Objects* of any type. In this case the keys are of type 'String' and the values are of type 'State'.

[4] A *List* is a simple, general-purpose container. We used Java's *ArrayList* in our case.

Combining them with their respective values, we get all the information we need about the automaton's ***step history***.

From an algorithmic perspective, our core functions are quite simple. For example our ***run()*** method looks something like:

```
1  public boolean run(startingState)
2  {
3    currentState = startingState;
4    int stepIndex = 1;
5
6    while (tape.hasNext())
7    {
8      char symbol = tape.next();
9      newState = nextState(currentState, symbol);
10     currentState = newState;
11     stepIndex++;
12   }
13
14   if (currentState.isAccepting())
15     return true;
16   else
17     return false;
18 }
```

Pseudocode for Machine's run() method.

Method ***nextState()*** in line 9, takes as arguments the automaton's current state, the symbol that was just read from the *Tape*, and an integer to denote the number of steps so far. Then, inside that method we retrieve the value stored in our Map: *transitionTable* using *currentState* as a key. Java uses its own *hash* function when it comes to the retrieval of data from a Map[5] by using the appropriate key. As we mentioned earlier, this value is a collection of references to Transition Objects (*ArrayList<Transition>*). So, we iterate this collection until we find the Transition that has the same symbol as the one we just read from the *Tape*. Finally, that Transition's *target State* becomes our *nextState*.

4.2.2 Nondeterminism

We talked about nondeterminism in Chapter 2: *The Feature of Nondeterminism*. There we quoted *Michael Sipser*, as he described thinking of nondeterminism as *" parallel computing"*, or as a *" tree of possibilities"* at the automaton's disposal per every step. In brief, here are the main differences between a DFA and an NFA:

[5] As a matter of fact, *Map* in Java is *abstract*, it cannot be *instantiated*. Instead we used one of Java's available implementations of a *Map*, called *HashMap*.

- In NFAs, for a given source state and a given input symbol it is allowed to have ***multiple target states***, or ***none***. In DFAs there must be ***one and only one***.
- In NFAs, it is allowed for the automaton to transit to another state ***without reading any input***. In that case we say that the automaton has ***"empty"*** transitions, or 'ε' transitions, or 'e' transitions[6]. Think of these as transitions available *"for free"*, that our automaton can perform *without* spending a computational *cycle*. An NFA that has ***'e'*** transitions as well as normal ones, is able to perform ***both*** upon reading an input symbol.

Let us see now in detail, what are the modifications that we had to make to our *core*, in order to account for these differences:

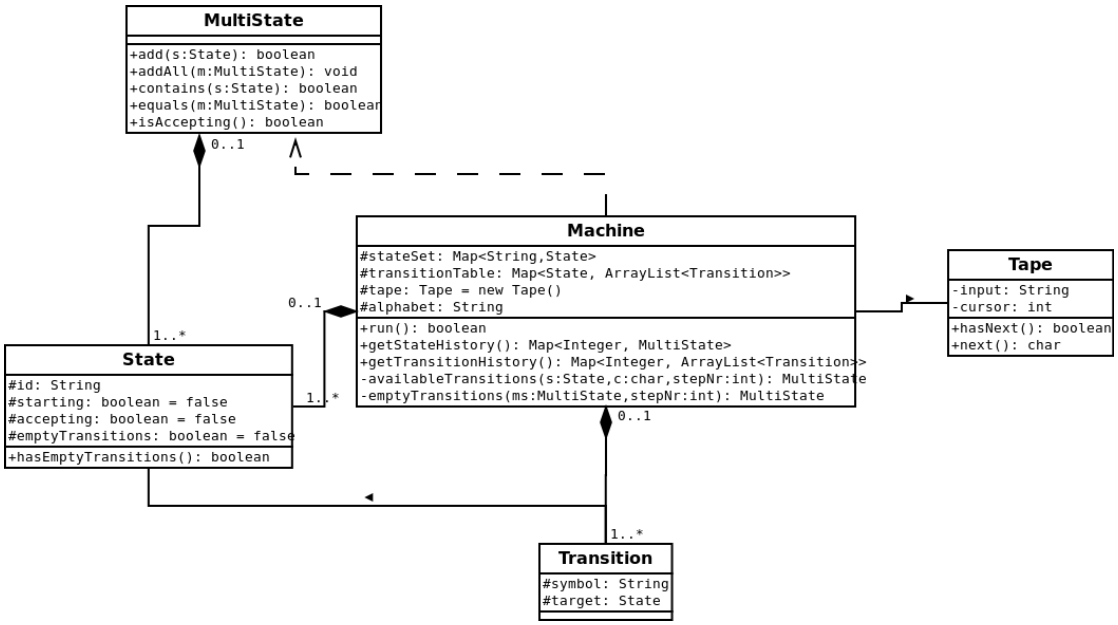

FIGURE 4.2: Class diagram of our modified *core*, to accomodate *Nondeterminism*.

MultiState

To initialize our new Machine, we provide the input exactly like before, but our Machine now handles it differently. A new class which *extends* [7] Java's *ArrayList* is created, called ***MultiState***. This is our own customized version of an *ArrayList*, created to contain only references to *State Objects* and nothing else. The names derives from the idea that when our NFA has more than one possible states to

[6] As a convention, from now on we will be using the letter '***e***' to denote *empty transitions*.

[7] This refers to an important concept in *OOP*, called ***inheritance***. See: https://en.wikipedia.org/wiki/Inheritance for more.

choose from, we construct a ***powerset***[8] containing all these possible states and consider it to be our automaton's ***Transition target***. In simple words, while a DFA transits to a *single* state per step, an NFA transits to a *set* of states. From now on, we will refer to that as a ***MultiState***.

When calculating what the automaton's next step should be, we first read the next symbol from the input tape, exactly like before. We then iterate through its current MultiState, and for every State that it contains we find the respective transition from the *transitionTable* and store a reference. If any of these States has a registration for an ***'e'*** transition in the *transitionTable*, we store it as well. In the end, we construct the automaton's ***new MultiState*** by adding together all the target states from the stored transitions, after removing the ***duplicates***. One thing to note about the duplicates, is that it is *conceptually* wrong to have duplicate entries in our MultiState, while, on the other hand, it is programmatically *inefficient*. In concept we only need one reference to a State, so that we can include it in our next step's computation. Therefore, despite the fact that our algorithm would work with duplicate entries, it would be inefficient to compute for the same State - symbol combination more than once.

Finally, the automaton accepts the input if its last MultiState contains ***at least one*** accepting State and rejects it otherwise.

State

It was necessary to add something to our *State* class as well. We included another ***boolean*** variable, called ***emptyTransitions***. This variable is defaulted to *false* and acts as a *flag*, set to *true* only when a State has 'e' transitions defined in the *transitionTable*. An accessor method accompanies this variable, called ***hasEmptyTransitions()***. The reason we need such a flag, is that there is a different piece of code handling the case of empty transitions, and it would be inefficient to execute it each time we searched for available transitions. Moreover, as we mentioned earlier, because of the fact that we view empy transitions as *"free"* transitions, we handle them after the normal steps have finished. This means that the algorithm performing the next step computation, finishes only *when 'e' transitions are handled*, **or** *when normal transitions are handled and 'e' transitions don't exist.*

Transition

The only change related to our *Transition* class, is that we reserved ***'e'*** as a special symbol. This means that we *cannot use it as a normal input symbol, or as part of an alphabet*, because from now on, it denotes empty transitions.

[8] See: https://en.wikipedia.org/wiki/Powerset

Machine

Our *Machine* class couldn't stay unaffected, either. Modifications were made so that it could handle Objects of type: *MultiState* instead of *State*, where a single State would now be represented by a MultiState containing only one entry. This allowed us to have *backwards compatibility* with the code that simulates DFAs.

Moreover, since our automaton can now perform more than one transition per step, our ***TransitionHistory*** was refactored from ***Map<Integer, Transition>***, to ***Map<Integer ArrayList<Transition>>***.

Finally, we created two additional methods, ***availableTransitions()*** and ***emptyTransitions()***. Method ***availableTransitions()*** is responsible for searching through the transitionTable for a given *source State* and a given *symbol*, returning a *State Object* as a result. This method doesn't deal with *MultiState* Objects, so we could say in a way that it encompasses the simple DFA functionality we described in our previous section: *Our Core*. Now that we are dealing with MultiState Objects, the idea is to break down our "nondeterministic" search to a lot ot "deterministic" searches and call the ***availableTransitions()*** to handle them. These calls are being made from within the method ***nextState()*** (figure 4.1) we saw in the previous section. The second method, ***emptyTransitions()***, is responsible for the production of all the target states that derive from empty transitions and it uses availableTransitions() to do that. By treating the symbol ***'e'*** (that denotes an empty transition in our transitionTable) as if it were a normal symbol, we break down our "nondeterministic" search for empty transitions to a series of "deterministic" searches for normal transitions.

The final result is the *union* of all the states that resulted from the modified ***nextState()*** and those resulted from ***emptyTransitions()***, and is stored in a *new MultiState Object*.

Let us take another look on our new Machine, this time from an algorithmic perspective. In order to do that though, first we have to explain what *Breadth-First Search* (BFS) is:

In graph theory, ***breadth-first search*** is a strategy for searching in a *graph* or a *tree*. The ***BFS*** begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on. [39]

In our application we employed a simple implementation of a ***BFS*** algorithm, to search for transitions in every step of the automaton's computational run. There are other *equivalent* strategies, such as the *Depth-First Search*[9], but the BFS solution came more

[9]See: en.wikipedia.org/wiki/Depth-first_search

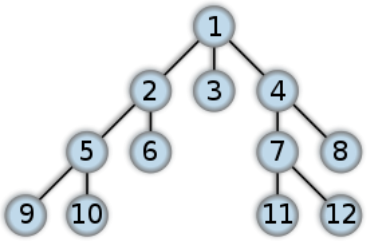

FIGURE 4.3: Breadth-first search

naturally. As we mentioned in section *Multistate* (page 36), our MultiState extends Java's *ArrayList* structure, thus inheriting a method called *add()*. This method is responsible for adding new elements in the collection, and it does so by adding the new element *at the end* of the List. This functionality provides a seamless implementation of a BFS algorithm, which normally uses a *queue* data structure to store intermediate results as it traverses a graph.

Besides having the functionality of a *queue*, we also wanted to avoid having duplicate states in our MultiState collection. The term *duplicate state* in our case, is not referring to duplicate *references* to the exact same *State Object*, but to two different *State Objects* sharing a common *id* String. Thus, we created another *add()* method which *overrides* ArrayList's native one, and checks for duplicates every time a new State is being added in a MultiState collection.

Lastly, let's see how our handling of normal transitions differs from that of empty ones, codewise:

```
1  public MultiState nextState(MultiState currentMultiState, char symbol)
2  {
3    ...
4
5    MultiState result_normal = new MultiState();
6    for (every State in currentMultiState)
7    {
8      result_normal.add(availableTransitions(State, symbol));
9    }
10
11   ...
12
13   return result_normal;
14 }
```

Excerpt from nextState()

Note how MultiState ***result_normal*** is a new, empty MultiState which is created locally (line 3). Also, note how the argument ***currentMultiState*** stays ***unmodified***

throughout this method. Finally, this method returns all the target states that derive from normal transitions, in *result_normal*.

```
1  private MultiState emptyTransitions(MultiState statesWithEmptyTransitions)
2  {
3    MultiState result_empty = statesWithEmptyTransitions;
4
5    for (every State in statesWithEmptyTransitions)
6    {
7      result_empty.add(availableTransitions(State, 'e'));
8    }
9  }
10
11   return result_empty;
12 }
```

Pseudocode for emptyTransitions

Note how MultiState ***result_empty*** is *not* a new MultiState this time, but gets initiated by referencing all the states with outgoing empty transitions (line 3). Also, this set of states gets modified inside the *for loop* (line 5), after adding the respective target states. This causes the statesWithEmptyTransitions size to grow in *runtime*, thus ***extending the duration of the for loop***. The added states constitute the first *child nodes* and they will be accessed immediately after the root nodes. This *for loop* exploits some of Java's innate characteristics in order to obtain the *queue* functionality we mentioned earlier, thus implementing a BFS search.

One last thing to underline is that the depth of search per computational step is only ***one*** regarding normal transitions, while that's not the case regarding the empty ones. The automaton can perform empty transitions *"for free"* and the depth can be more than that. How much more? Since there is a restriction on the occurrence of duplicate states, the depth is limited by the total number of states our automaton can be in. More than that would mean that at least one node (State) was visited for a second time.

Lastly, the pseudocode for the ***availableTransitions()***:

```
1  private MultiState availableTransitions (State currentState, char symbol)
2  {
3    MultiState result = new MultiState();
4    ArrayList transitionList = transitionTable.getValueFor(currentState);
5
6    for (every Transition in transitionList)
7    {
8      if (Transition.symbol == symbol)
9        result.add(Transition.targetState)
10   }
11
12   return result;
13 }
```

Pseudocode for availableTransitions()

This method takes a reference to a *State Object* as an argument (currentState) and then proceeds to locate all of its *outgoing transitions* in our *transitionTable* (line 4). The result is an ArrayList called *transitionList*, which contains references to *Transition Objects*. Afterwards, transitionList is iterated looking for the transitions that match the symbol we just read from the tape. The resulting target states are returned in a new MultiState. As you can see in ***emptyTransitions()*** (line 7), we are handling the empty transitions as if they were normal transitions for the symbol ***'e'***.

4.2.3 User Input Validation & NFA to DFA Conversion

During our third revision of the Machine's development we added two more necessary functions:

- Checking user's input for errors (with appropriate error message system),
- Converting an NFA to a DFA (using the *powerset construction algorithm*).

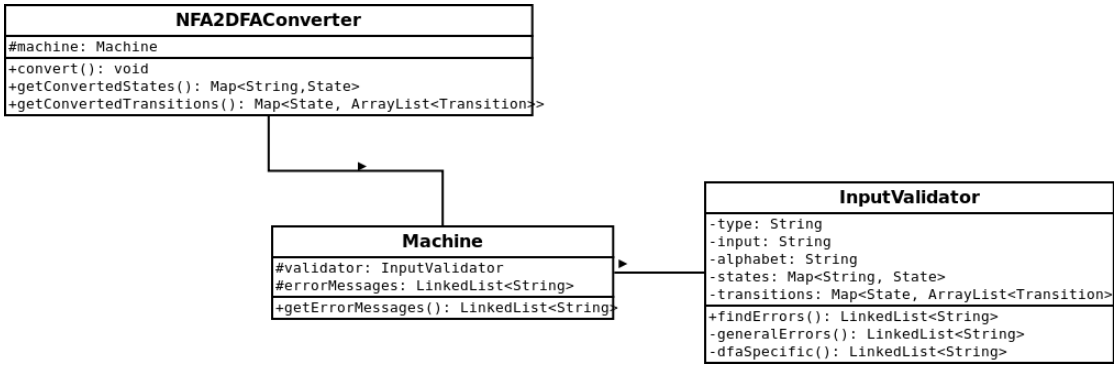

FIGURE 4.4: Class diagram of the Machine's additional features.

User Input Validation

In order for the defined automaton to be considered a *Finite Automaton* (deterministic or nondeterministic), it must fulfill some requirements. In case it doesn't, the user must be informed via error messages, before executing the computation. Here follows a list

of all the error messages and the errors they represent. There are eight errors regarding Finite Automata in general and only one specifically for DFAs:

"Empty Alphabet": User has to define an alphabet.

"Alphabet does not contain the symbol x**"**: Both the symbols used in transitions and the symbols used in the input tape must belong to the alphabet to be considered valid.

"Untitled State": A State *id* has been omitted.

"Duplicate State": User has used the same *id* to name more than one states.

"No Starting State": User has omitted to determine what is the automaton's *starting state*.

"Can't have more than one Starting State": Only one *starting state* should be determined per automaton.

"Undefined Transition Symbol": User has omitted defining a sumbol for a transition.

"Duplicate Transition": There are at least two transitions that start from the same state, use the same symbol and lead to the same target state as well. These transitions are considered to be equal, therefore rendundant.

And the DFA-specific error:

"There is a Transition missing from State X": In DFAs every State must have an outgoing transition for every symbol in the alphabet.

The way our input validation works is very simple. An *InputValidator* instance gets initialized with the same input as *Machine*. Then, in a method called *findErrors()* it iterates through the input data, while performing all the necessary checks. In the end it returns a *List* (Java's *LinkedList* in particular) containing all the error messages in the form of Strings.

Conversion of an NFA to a DFA

As we mentioned in Chapter 2: Background (section 2.1.5), we can construct a DFA from any given NFA using an algorithm called ***powerset construction***[10]. This is a

[10]See the same ref. from Chapter 2 as to what a *powerset* is.

standard method in the *Theory of Computation* and in *Automata Theory*, for *converting* a *nondeterministic finite automaton* into a *deterministic finite automaton* which recognizes the same *formal language*. This fact establishes that NFAs, despite their additional flexibility, are unable to recognize any language that cannot be recognized by some DFA. It is also important in practice for converting easier-to-construct NFAs into more efficiently executable DFAs. However, if the NFA has n states, the resulting DFA may have up to 2^n states, an exponentially larger number, which sometimes makes the construction impractical for large NFAs. This particular method was first published by *M.O. Rabin* and *Dana Scott* in 1959[40][41].

Intuitive Description

To simulate the operation of a DFA on a given input string, one needs to keep track of a single state at any time: the state that the automaton will reach after seeing a *prefix* of the input. However, to simulate an NFA, one needs to keep track of a set of states: all of the states that the automaton could reach after seeing the same prefix of the input, according to the nondeterministic choices made by the automaton. If, after a certain prefix of the input, a set S of states can be reached, then after the next input symbol x the set of reachable states is a deterministic function of S and x. Therefore, the sets of reachable NFA states play the same role in the NFA simulation as single DFA states play in the DFA simulation, and in fact the sets of NFA states appearing in this simulation may be re-interpreted as being states of a DFA. [41]

A (more) Formal Description

The powerset construction applies most directly to an NFA that does not allow state transformations without consuming input symbols ("e-transitions"). Such an automaton may be defined as a *5-tuple* $(Q, \Sigma, T, q0, F)$, in which Q is the set of states, σ is the set of input symbols, T is the transition function (mapping a state and an input symbol to a set of states), q0 is the initial state, and F is the set of accepting states. The corresponding DFA has states corresponding to subsets of Q. The initial state of the DFA is q0, the (one-element) set of initial states. The transition function of the DFA maps a state S (representing a subset of Q) and an input symbol x to the set $T(S, x) = \cup T(q, x | q \in S$, the set of all states that can be reached by an x-transition from a state in S. A state S of the DFA is an accepting state if and only if at least one member of S is an accepting state of the NFA.

In the simplest version of the powerset construction, the set of all states of the DFA is the powerset of Q, the set of all possible subsets of Q. However, many states of the

resulting DFA may be useless as they may be unreachable from the initial state. In our implementantion we used a version of the algorithm that creates only the states that are actually reachable. For an NFA with e-moves, the construction must be modified somewhat. In this case, the initial state consists of all NFA states reachable by e-moves from q0, and the value T(S,x) of the transition function is the set of all states reachable by e-moves from $\cup T(q, x)|q \in S$. [41]

Example

The NFA below has four states; state 1 is initial, and states 3 and 4 are accepting. Its alphabet consists of the two symbols 0 and 1, and it has *e*-moves.

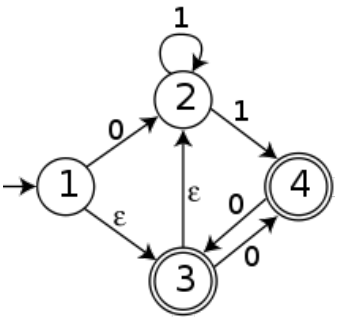

FIGURE 4.5

The initial state of DFA constructed from this NFA is the set of all NFA states that are reachable from state 1 by e-moves; that is the set {1,2,3}. A transition from {1,2,3} by input symbol 0 must follow either the arrow from state 1 to state 2, or the arrow from state 3 to state 4. Additionally, neither state 2 nor state 4 have outgoing e-moves. Therefore $T(\{1, 2, 3\}, 0) = \{2, 4\}$, and by the same reasoning the full DFA constructed from the NFA is as shown below.

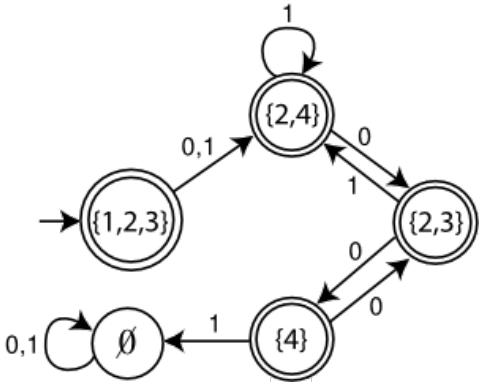

FIGURE 4.6

As can be seen in this example, there are five states reachable from the start state of the DFA; the reamining 11 sets in the powerset of the set of NFA states are not reachable.[41]

Complexity

Because the DFA states consist of sets of NFA states, an n-state NFA may be converted to a DFA with at most 2^n states. For every n, there exist n-state NFAs such that every subset of states is reachable from the initial subset, so that the converted DFA has exactly 2^n states. A simple example requiring nearly this many states is the language of strings over the alphabet 0,1 in which there are at least n characters, the nth from last of which is 1. It can be represented by an (n+1)-state NFA, but it requires 2^n DFA states, one for each n-character suffix of the input. [41],[3, p. 103],[1, p. 55]

Our Implementation

Our class *NFA2DFAConverter* (see figure 4.4), has an instance of *Machine* as an attribute. It also has all the necessary input that defines an automaton; states, transitions, input and alphabet. In order to find and construct all the necessary powersets, it provides the machine with the respective *prefix* of the actual input and sets it to a different starting state each time. In that way , by executing multiple 1-step computational runs in the machine, we get all the subsets of the automaton's states that we need. This class doesn't need to handle e-moves in a special way, as our Machine has the innate capability to handle them by itself.

Finally our NFA2DFAConverter takes the result and formats it in a way that it describes a DFA. As a proof of concept, we converted the same NFA from the previous example so that we could test and compare our results:

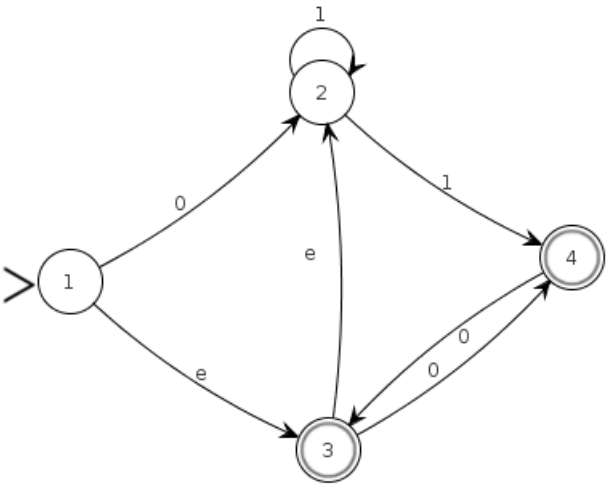

FIGURE 4.7: The NFA to be converted, as rendered by our application

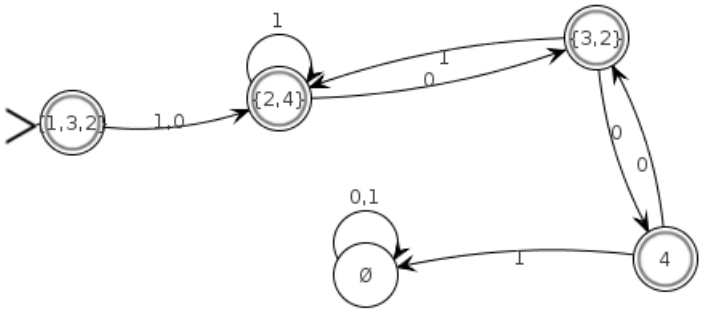

FIGURE 4.8: The resulting DFA

4.2.4 Using JSON as a Finite Automata Description Language

There was a last piece missing from our Machine-backend, before we could start testing and experimenting with it. This was a simple, intuitive *description language* for automata. This is where ***JSON*** came into play, as it could fulfill that role with impressive efficiency. You can find more information about JSON and why it suits our purpose so well, in Chapter 2 (ref JSON here). Suffice to say, that it is a language intended to be very simple and easy for a human to read. Here follows a description of an NFA that accepts strings containing either the sequence ***'bb'*** or ***'bab'***:

```
 {
    "type":"nfa",

    "states":["q0","q1","q2","q3","q4"],
    "transitions": [
                     ["q0","a","q0"],
                     ["q0","b","q0"],
                     ["q0","b","q1"],
                     ["q1","a","q3"],
                     ["q1","b","q2"],
                     ["q2","e","q4"],
                     ["q3","b","q4"],
                     ["q4","a","q4"],
                     ["q4","b","q4"]
                    ],
    "startState":"q0",
    "acceptStates":["q4"],
    "alphabet":"ab",
    "input":"bb"
}
```

Using JSON to describe an NFA

In our opinion, any further explanations would be redundant. This is how the described NFA looks like (as rendered by our application):

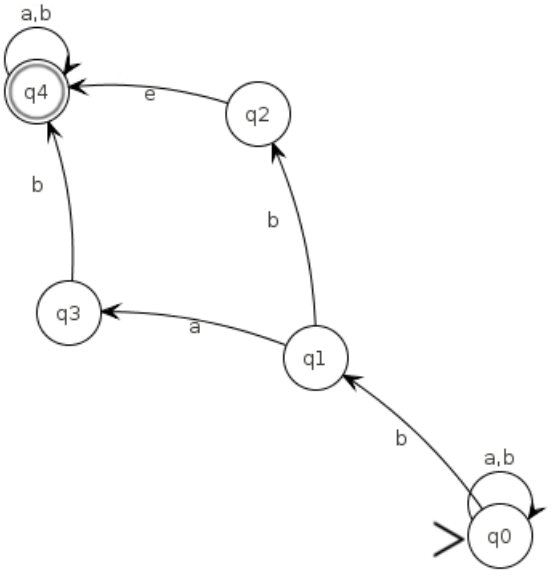

FIGURE 4.9: Visualization of the described NFA

The fact that JSON is a natural, efficient, out-of-the-box solution to describe Finite Automata, is utterly interesting. To our knowledge it has ***never been used in such a way before.***

4.3 Graphical User Interface (GUI)

In the previous section we talked about our *backend*, or the parts of our application that stay hidden behind the curtains, performing their duties silently. Now it is time to talk about our *frontend*, or all these things that are visible to the user, and available for him to interact with. This section discusses *our Graphical User Interface.*

4.3.1 Design Patterns: Model-View-Controller

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns can speed up the development process by providing tested, proven development paradigms.

Some of the tools we used were already architectured after a design pattern. This was not a problem for us, or anyone that intends to use third-party source code. In fact, this is an important benefit that comes with the use of a design pattern. The process of assimilation of someone else's code can be accelerated if one is already accustomed with the design pattern used by the developer.

Although we didn't use a specific design pattern in the development of our application, our design was loosely based on the famous *Model-View-Controller* pattern.

Model-View-Controller

Model-view-controller (MVC) is a software architecture pattern which separates the representation of information from the user's interaction with it. The *model* consists of application data, business rules, logic, and functions. A *view* can be any output representation of data, such as a chart or a diagram. Multiple views of the same data are possible, such as a bar chart for management and a tabular view for accountants. The *controller* mediates input, converting it to commands for the model or view. The central ideas behind MVC are *code reusability* and *separation of concerns.*

In addition to dividing the application into three kinds of components, the MVC design defines the interactions between them:

- A ***controller*** can send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document). It can also send commands to the model to update the model's state (e.g., editing a document).
- A ***model*** notifies its associated views and controllers when there has been a change in its state. This notification allows the views to produce updated output, and the controllers to change the available set of commands.
- A ***view*** requests from the model the information that it needs to generate an output representation to the user.

MVC was one the seminal insights of the early field of *graphical user interfaces* in the 70s and it still hasn't lost its power. Although many popular variations exist nowadays, the initial concept still echoes today. The terms ***model***, ***view*** and ***controller*** alone, represent a condensation of meanings easily communicated among programmers as part of our collective consciousness. So, even though we were only inspired by the MVC design pattern, we are still going to borrow its terminology to make our implementation more explicit. [42]

4.3.2 Our Design

In this section we will be exposing the details of our design and the central ideas that governed our approach. Figure 4.10 shows a class diagram that will act as our point of reference.

The elements that we called *MenuBar*, *ToolBar* and *TapeDisplay* can be considered to belong in our ***view***. They are clearly separated from the rest of the code and one could easily modify/substitute them to change our application's window appearance. We find their names quite self-explanatory as to what parts of our window they visualize. MenuBar is the visualization of our *file-menu*, ToolBar stands for our mainframe's *toolbar* and TapeDisplay is responsible for the area which an automaton's *input tape* is displayed in.

There three components can be accessed through their respective controllers, *MenuBarController, TapeDisplayController* and *ToolBarController*. These three , along with *MainController*, form a ***chain of access*** by the use of set/get methods. Besides providing a sequence of accessors, they also contain the code that handles user-generated events on these components. For example, if the user presses the "run" button in our ToolBar, a piece of code in ToolBarController gets executed, handling that event. Additionally, we also want this event to force an update upon our TapeDisplay component, in order to display the contents of the (now running) automaton's input tape. So, through the chain of access, TapeDisplayController addresses our MainController which in turn informs our TapeDisplayController that an update is needed. Finally, the TapeDisplayController calls our TapeDisplay to complete the task.

Therefore, in MVC terms, the four "controller" components (MainController, MenuBarController, TapeDisplayController, ToolBarController) constitute our architecture's ***controller***. This separation is not clear, however. The component that perplexes the situation is *GraphVisualization*. This component encompasses the third-party tool that we used to facilitate us in graph creation, editing and visualization. In Chapter 2 (section 2.2.4), we saw that this third-party tool is called ***JUNG*** and comes with its own architectural design. Since its primary task is everything graph-related, JUNG has already its own *model-view-controller* decomposition. It has a model, responsible for knowing *what to display*, and a hybrid view/controller responsible for knowing *how to display it*. The view/controller is implemented by a ***Graph*** class (node id's, node connections, edge labels etc) and a ***Layout*** class (relative and/or absolute location of nodes and edges in a visualization). The second is implemented by a very important class, called ***VisualizationViewer***, which performs all the painting, as well as numerous other tasks. The

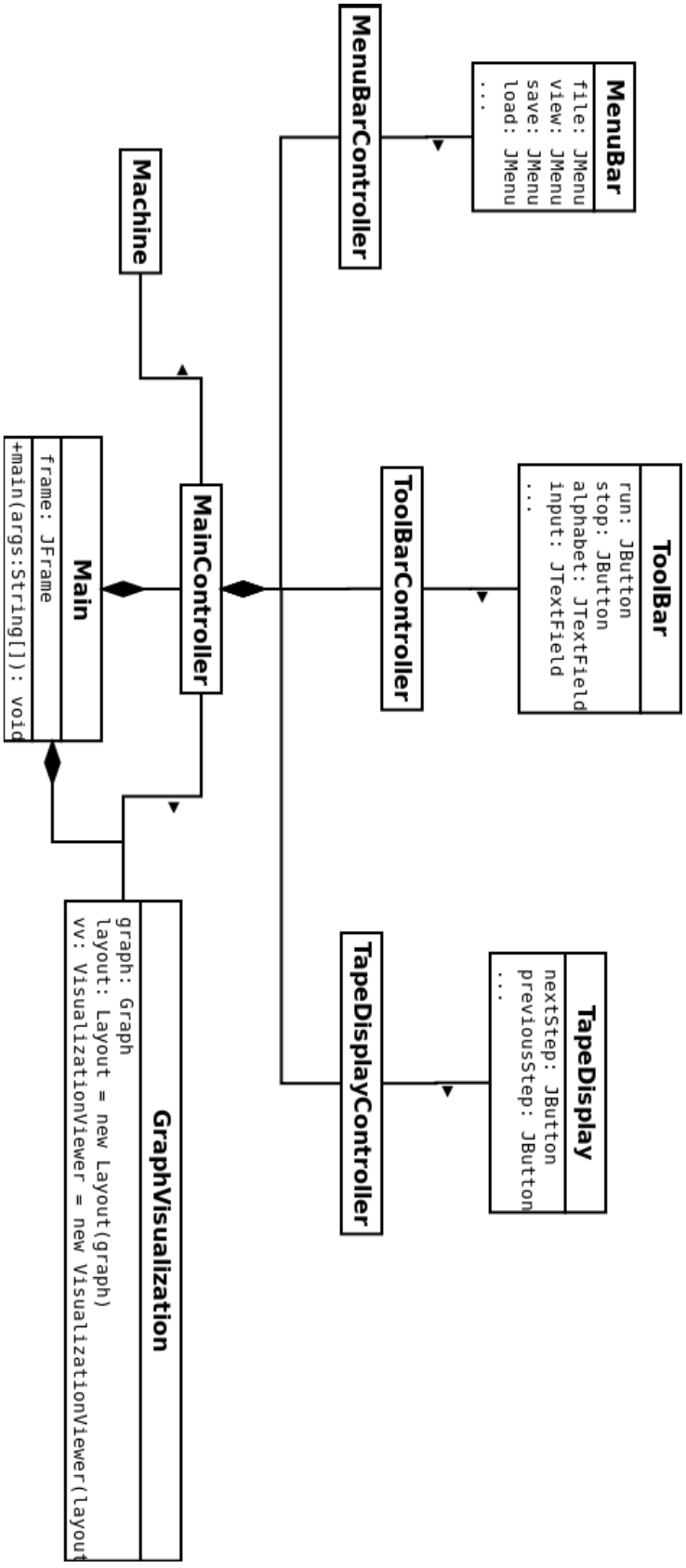

FIGURE 4.10: Our design

VisualizationViewer class also handles the events that derive from the interaction between the user and the graph. Nonetheless, we still had to associate our MainController with the VisualizationViewer, mostly to assimilate it into our chain of access structure.

Last but not least, our ***model*** (in MVC terms), should be viewed as two *different* models, actually. The first is our *computational* model, or our simulation *logic*, which resides within our *Machine*. User-generated events can cause the simulation code to be executed and the results cause the GUI to be updated to display them. The second model is the one we mentioned already, JUNG's model, which is responsible for all the information regarding the graph visualization. This model must be informed and updated every time the user creates new states/transitions, edits them, changes their location, selects and deletes them etc. The communication between the two models and all of our visual components (view) is facilitated by the synergetic usage of our controllers.

4.3.3 GUI Components & Layouting

Before moving on, we have to discuss a little bit more about the part that we called *view* in the previous section. We talked about four decomposed elements that constitute it, called *MenuBar*, *ToolBar*, *GraphVisualization* (JUNG's *VisualizationViewer)* and *TapeDisplay*.

In our actual implementation however, we ended up having *five* such elements. For the sake of simplicity we have been omitting to mention the fifth element, a class called *LogDisplay*. This class follows the exact same logic as the others, i.e. has it's own *LogDisplayController* and is responsible for a separate area in our GUI's main frame that we called *Step History*. This is an area containing a simple textual representation of the automaton's steps, and the user can enable/disable it at will.

Now that we settled that, let us see the five main areas that constitute our *main frame* (figure 4.11):

1. *MenuBar*,
2. *ToolBar*,
3. *Canvas*,
4. *Step History*,
5. *Tape Display & Controls*.

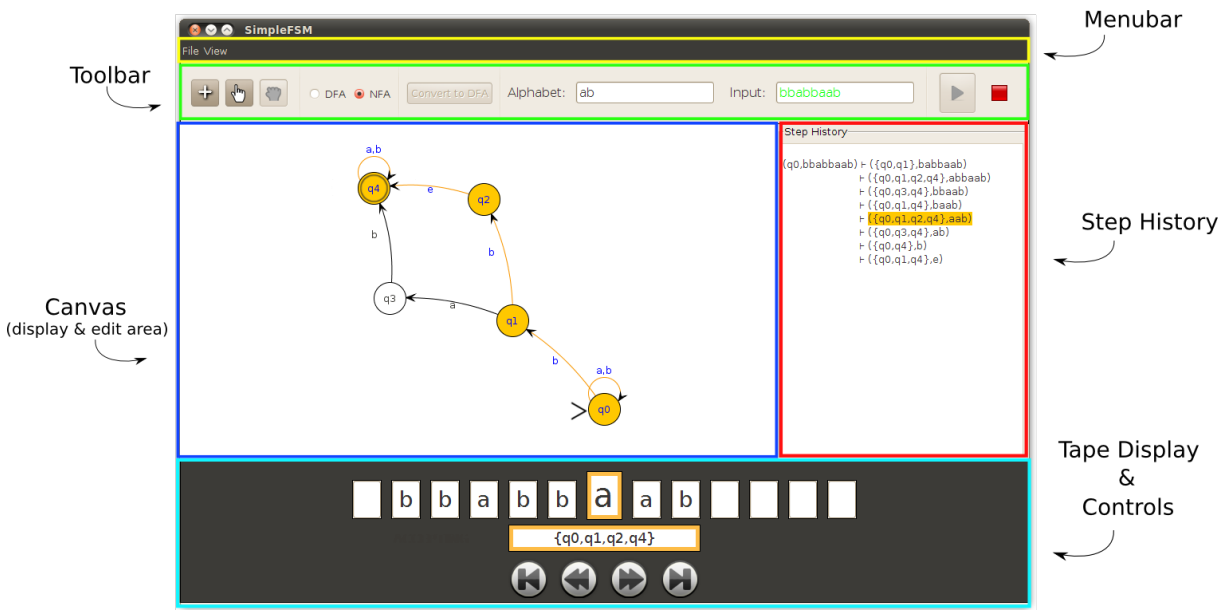

FIGURE 4.11: The five main areas

In our implementation each one of the above is represented by Java/Swing components, mainly *JPanel*.

MenuBar: A simple *JMenuBar* containing *JMenu* and *JMenuItem* objects.

ToolBar: A *JPanel* containing objects of type: *JButton, JRadioButton, JLabel, JTextField* and *JSeparator*.

Canvas: JUNG's *VisualizationViewer* which actually extends *JPanel*. We will see more about it in a following section.

Tape Display & Controls: A *JPanel* containing objects of type *JTextfield* and four *JButton* objects for the controls.

Step History: A *JPanel* containing a *JTextArea* object, with an additional *JScrollPane* to account for the scroll functionality.

These four panels and the menubar are laid out inside our main frame using one of Swing's simplest layout managers, called *BorderLayout*. This manager divides the frame into five areas (north, south, west, east and center) and appoints each component accordingly.

As for the basic components that our respective panels contain (such as textfields, buttons, e.t.c.), they were laid out using *NetBeans GUI Builder* (formely known as project *Matisse*). This tool enables the developer to customize and layout components *graphically*, with the capability of displaying a preview of the result.

To develop our protorype we used a *free design* layout, which means that we placed our components manually inside the panels. There is a disadvantage to that, however. When there isn't any layout manager involved the application doesn't have *dynamic behaviour*. This means that the various components cannot adapt to changes in the main window's *size*, like when the user resizes it. Thus we disabled that option, although half-heartedly. In future versions of this application we will replace our free design with Java's *GridBagLayout*, the most complex and flexible layout manager that Java possesses. This would allow us to program a dynamic resizing behaviour, but would demand a complete remake of some of our panels first.

4.3.4 Graph Visualization

Now we will be discussing about the component that visualizes everything that is graph related; a white, rectangular panel in the center of our application which acts as our *canvas*. This component is JUNG's ***VisualizationViewer***, a class that extends Java's *JPanel*.

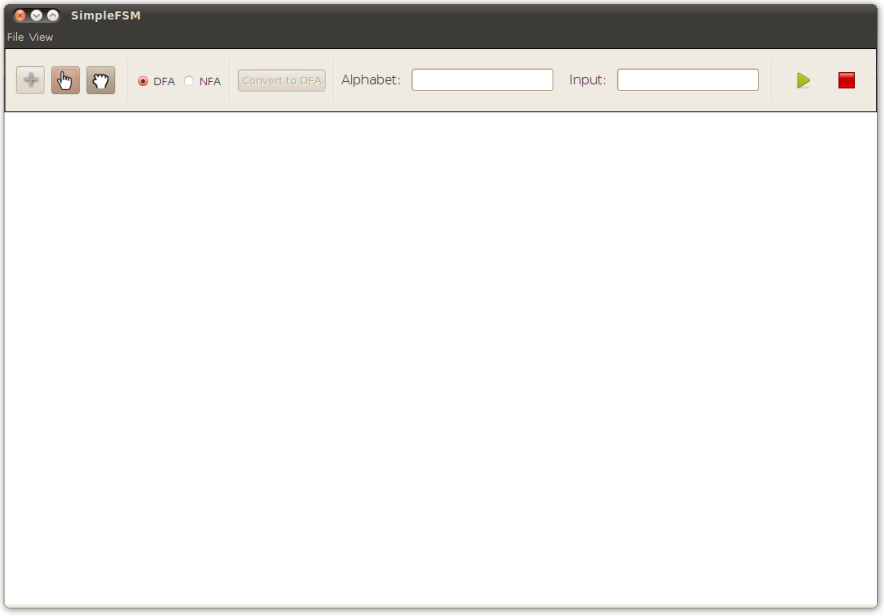

FIGURE 4.12: A screenshot of an empty canvas.

The VisualizationViewer is very important as it has a number of different tasks. It tracks the ***Renderer*** (class responsible for drawing) and the graph model, it handles the mouse when it acts inside its jurisdiction, and it applies the developer's customizations to either the view or the layout, allowing us to modify numerous variables that affect the appearance of the graph.

There is an excellent tutorial on JUNG, written by Greg Bernstein, and we will cite from it in order to get a better grasp on how we used it to serve our purpose. You can find it in http://www.grotto-networking.com/JUNG/JUNG2-Tutorial.pdf.

To start off, there is an interface *Graph<V,E>*. *V* stands for *Vertex* (or node) and *E* stands for *Edge*. In JUNG we can assign both of these roles to any Objects we want. In our application for example, the automaton's *states* are the graph's *vertices* and its *transitions* are the graph's *edges*. The *Graph<V,E>* inteface defines the basic operations that you can perform on a graph. These include:

1. Adding and removing edges and vertices from the graph and getting collections of all edges and vertices in a graph.
2. Getting information concerning the endpoints of an edge in the graph.
3. Getting information concerning vertices in a graph including predesessor and successor vertices.

The specific Graph implementation that we used is the *DirectedOrderedSparseMultigraph*. In Chapter 2 (section 2.2.4) we explained what are the special characteristics of this implementation. An example of adding vertices and edges in a Graph instance:

```
Graph<State,Transition> g
= new DirectedOrderedSparseMultigraph<State,Transition>();

g.addVertex((MyVertex) 1);
g.addVertex((MyVertex) 2);
g.addVertex((MyVertex) 3);
...

g.addEdge("Edge-A", 1, 2);
g.addEdge("Edge-B", 2, 3);
...
```

After initializing our Graph, we need to use it to instantiate an implementation of the *Layout<V,E>* interface. This interface is responsible for the location of all the graph's vertices and edges. It can be done either manually or algorithmically. When we wanted a manual layouting for our application we used an implementation called ***StaticLayout***, and when we needed an algorithmic one we used ***FRLayout***[11]. The way to instantiate a Layout is very simple:

```
Graph<State,Transition> g;
StaticLayout<State,Transition> layout = new StaticLayout<State,Transition>(g);
```

or

[11] An implementation of the *Fruchterman-Reingold force directed algorithm* for node layout.

```
FRLayout<State,Transition> layout = new FRLayout<State,Transition>(g);
```

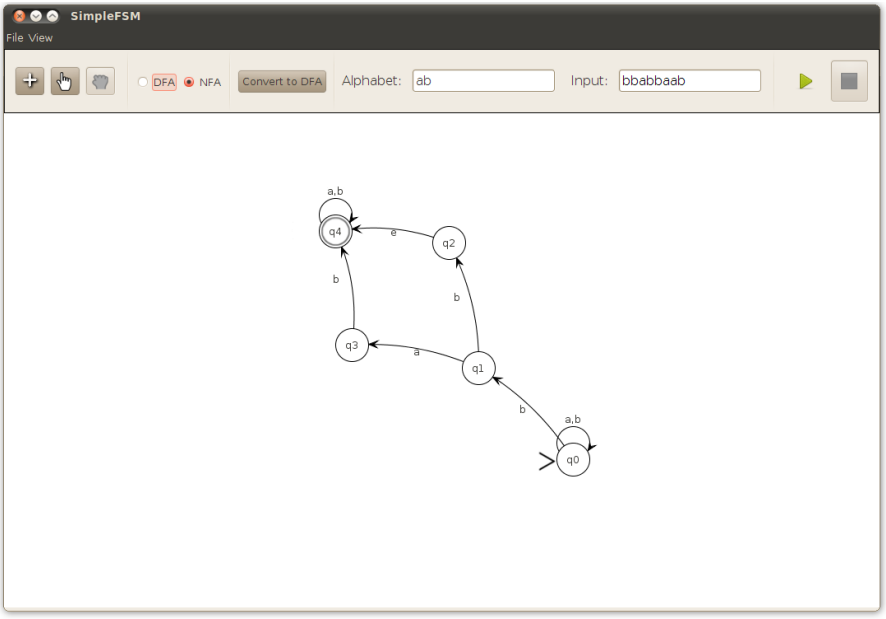

FIGURE 4.13: An example of a graph laid out by *FRLayout()*

Next comes the ***Transformer*** interface. This is an interface from the *apache commons collections*[12] that JUNG uses. It usually comes with the use of generics, for example Transformer<E,String>. The only method in this interface is ***String transform(E e)***, hence given an edge class of our creation, we will need to come up with an appropriate (and usually very simple) Transformer class to extract a String (denoting -let's say- a *label* for that edge). JUNG's VisualizationViewer and its Renderer make extended usage of the Transformer interface as a means for the developer to customize many variables, like the *shapes* of vertices/edges, their *color*, the labels' *fonts* etc.

After initializing a Graph and a Layout we simply instantiate our VisualizationViewer, customize its variables appropriately and add it in a *JFrame*. All these happen in our *Main* class.

```
VisualizationViewer<State,Transition> vv
= new VisualizationViewer<State,Transition>(layout);

/**
*
* Set up and customizations here.
*
**/

JFrame frame = new JFrame();
frame.add(vv);
```

Most variables can be found and modified through methods that reside in:

[12] See http://commons.apache.org/proper/commons-collections/.

- *VisualizationViewer* (vv),
- *Renderer* (accessed through method *vv.getRenderer()*),
- *RenderContext* (accessed through method *vv.getRenderContext()*).

The JUNG renderers are used to actually draw four different items: (a) edges, (b) edge labels, (c) vertices, and (d) vertex labels. JUNG supports the notion of pluggable renderers so that one can substitute different renderers for the default. However, many of the changes we wanted to make did not demand us to make a new Renderer, since they can accept a number of "parameters". How can someone supply all these parameters to the graph renderers? Where the default values kept? This is the job of *RenderContext*. Each VisualizationViewer contains a *RenderContext* object that we can access to set these various rendering "parameter" values, most of the times by using an implementation of the *Tranformer* interface that we talked about earlier.

As a simple example, let us see how can someone:

1. Change the vertex color from the default to green.
2. Make the line used in the edges a dashed line.
3. Display a label for both the edges and vertices.
4. Center the vertex label within its corresponding vertex.

For the first three we will use the following calls to the RenderContext: *setVertexFillPaintTransformer(), setEdgeStrokeTransformer(), setVertexLabelTransformer(),* and *setEdgeLabelTransformer()*. Each takes a *Transformer* class argument that converts an edge or a vertex to the type of information needed by the renderer. This means that it is easy for a developer to change the viual aspects of the graph based on attributes of custom edge and vertex classes.

We will go through the main customizations that we had to do.

- We increased the *size* of vertices. The shape was defaulted to a *circle*, but we wanted it to be a little bit bigger.
- We set their fill color to *white* and their stroke color to *black*.
- We created two custom icons and combined them with JUNG's default circle to represent *starting states* and *accepting states*.

- For vertices that are *selected* (or *picked*) by the user, we set their fill color to become *orange*. We also use this to animate a running automaton, as we programmatically pick states and transitions as the computational steps progress.
- We also changed the picked edge color to *orange* and their label color to *blue* to be more visible. Programmatically picked edges also participate in the animation of a running automaton.
- We set the edge label position to an appopriate distance from the edge's arrow, both in a parallel axis and a perpendicular one.
- We changed VisualizationViewer's background color to *white*.
- We set the vertex label position (state id) to be inside its circle. We also gave the option to change this position to the user, via menu "File -> Preferences".

You can find all the code that performs the above customizations inside our *Main* class.

4.3.5 Getting Interactive with our Graph

Again we have to underline the importance of Greg Bernstein's tutorial on JUNG as it was of tremendous help to us regarding interactivity in JUNG. From his tutorial we learn that JUNG provides GUI features to let users interact with graphs in various ways. Most interactions with a graph will take place via a mouse. Since there are quite a number of conceivable ways that users may want to interact with a graph, JUNG has the concept of a modal mouse, i.e., a mouse that will behave in certain ways based on its assigned mode. Typical mouse modes include: picking, scaling (zooming), transforming (rotation, shearing), translating(panning), editing (adding/deleting nodes and edges), annotating, and more.

To deal with a multitude of mouse modes JUNG uses the concept of a "pluggable mouse", i.e., a mouse that accepts various plugins to implement the currently supported modes. For example, *EditingGraphMousePlugin, LabelEditingPlugin, EditingPopupGraphMousePlugin*, and so on. For that reason, it wasn't necessary to create a new Graph Mouse of our own. We modified an existing implementation, and created our own mouse plugin instead. Before we describe it though, we must first define *what kind of actions should our mouse support*. These include:

- Creation of new nodes (states) at desired location via mouse left-clicks.

- Creation of new edges (transitions) between two nodes, by pressing and holding the left-click on the source state and moving the mouse pointer to the target state before releasing.
- Editing of a state's or transition's attributes (such as *id, transition symbol, starting/accepting status*), via right-clicking on the corresponding graph element and selecting the desired option in a *popup menu*.
- The option of *deleting* a state or transition, by using the same popup menu we mentioned in the previous case.
- Translating (panning), scaling (zoom) and rotating the graph, by holding down the left-click in combination with some keyboard button, and performing a mouse gesture afterwards.
- Both *single* selection and *box* selection, for nodes and edges. After they have been selected, the user should be able to change their location by holding down the left-click, moving the mouse pointer to the desired destination and then releasing. As a second option, a popup menu appearing in response to a right-click, would allow the user to *delete* the selections if he/she desires so.

One of JUNG's implementations of the Graph Mouse was very close to our goals. It is called ***EditingModalGraphMouse*** and we built upon it to cover all our needs. Since this mouse will be used to create new vertices and edges in respose to user actions, its constructors require that we furnish *factory* classes (derived from the Factory<V> and Factory<E> interfaces[13]). Also, the user will be deciding via the mouse where vertices should be placed, so we utilized a *StaticLayout*, to hold the locations. Most of the remaining actions in our user's disposal could also be covered by using the EditingmodalGraphMouse. In order to do that, we enabled three of its available *modes*. The user can change modes at will, by using three radio buttons in the mainframe's toolbar. (figure 4.14)

1. ***Editing Mode***: This mode allows to create new states by left-clicking on empty space. New transitions are created when left-clicking on a source state, holding it, and releasing after moving the pointer to the target state. It is represented by the *crosshair* icon.
2. ***Picking Mode***: This mode allows to pick (select) a single, or multiple nodes and change their location. Single left-click on a node selects it. Relocate it by holding down left-click, moving to the desired destination and releasing. The user can

[13]See: http://en.wikipedia.org/wiki/Factory_method_pattern

move multiple nodes in the same way. In order to select them, either hold left-click on empty space to perform a *box selection*, or select all the nodes individually by holding the ***shift*** button. This mode is represented by the *pointing hand* icon.

3. ***Transforming Mode***: While in this mode, the user can *pan* the screen. To do that, press and hold the left-click and move the mouse to the desired position before releasing. If the user does so, while holding the ***shift*** button at the same time, the graph *rotates*. By doing the same while holding the ***ctrl*** button, the action called ***shearing*** is performed. Finally, the user can *scale* the graph (zoom in/out) by using the mouse's *scroll wheel*. This mode is represented by the *grabbing hand* icon.

FIGURE 4.14: The three mouse mode buttons: *Editing, Picking and Transforming*

The actions related to the mouse's *right-click* on the other hand, are independent to mouse modes. This means that right-clicking causes the same pop-up windows to appear every time, only defined by the component that was right-clicked on. In order to implement that, we created our own plugin called ***PopupVertexEdgeMenuMousePlugin*** and we added it to our *EditingModalGraphMouse*. Three different custom menus were created and associated with that plugin, one for each component that our user can right-click on. The ***first*** menu appears when right-clicking on empty space. This menu has only one item, representing the option to *delete* all the elements that we might have selected. The ***second*** menu appears when right-clicking on a transition. This menu contains the option to *delete* the particular transition, and the option to *edit its symbol*. The ***third*** and final custom menu appears when right-clicking on a state. It provides the user with four options, (a) *delete* the respective state, (b) set it to be a *starting* state, (c) set it to be an *accepting* state, and (d) *edit the state's id*.

4.3.6 Running An Automaton

Now that we learnt how to design a new automaton, how can we *run* it? First we must make sure that we have defined an *Alphabet* and an *Input* for our automaton, in the respecting textfields at the top of the screen. Second, make sure we have selected the correct automaton's *type* (dfa or nfa), using the appropriate buttons that reside next to the "mouse modes" buttons in our ToolBar. This will only affect our error checking code, and *not* the way our Machine actually runs, as it is the same piece of code that

handles dfa's and nfa's alike. Third and last, we press the *run* button that can also be found in ToolBar (second from the end), thus starting the simulation of our automaton's computation.

In case the user omitted important information, or committed mistakes during the design and definition of the automaton, the Machine never starts simulating. Instead, the user gets an *error dialog* (JDialog in Java) that contains all the appropriate error messages. Here follows an example of a dfa where we made mistakes on purpose:

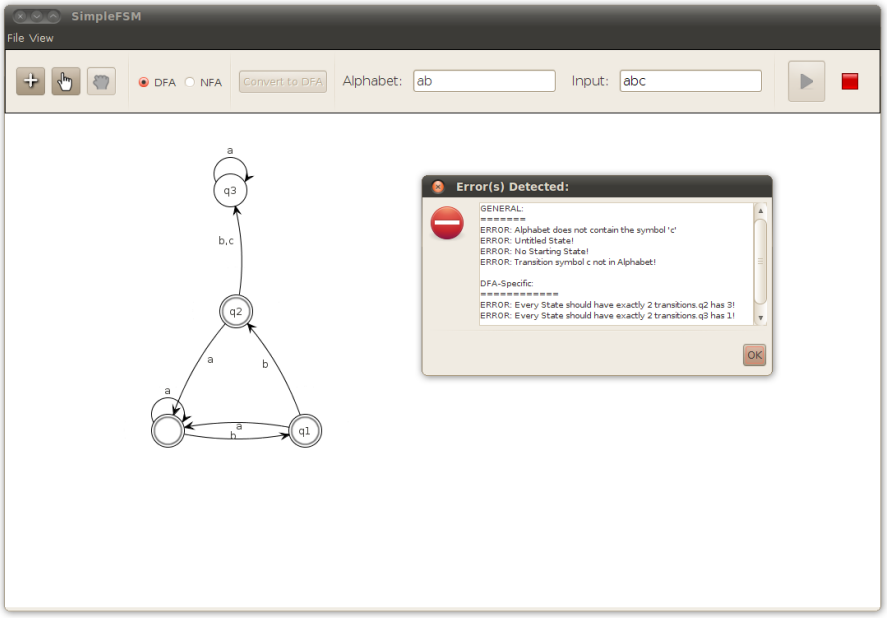

FIGURE 4.15: An example of design errors and their messages

When the errors have been resolved and the user's presses the *run* button once again, our simulation commenses. Our window then *changes*, as a new panel appears at the bottom of the screen to represent the automaton's *tape* and provide navigational controls to iterate it. Controlling the tape's progression is a way to navigate through the automaton's *running steps*.

In addition, we have included another panel called *LogDisplay*(labelled *Step History* on screen), which contains a log of the automaton's steps in textual format. In contrast to the TapeDisplay, which becomes visible only when the user presses the *run* button, LogDisplay's visibility can be toggled on/off via menu *View -> Step History* in MenuBar.

There are four available contols (implemented as JButtons) in our user's disposal, in order to control the tape's progression. From left to right: (a) go to start, (b) previous step, (c) next step, and (d) go to end. If you examine figure 4.16 carefully, you might notice that *as the user navigates* throughout the steps, a number of things are happening. We will describe them in respect with the GUI component they relate to.

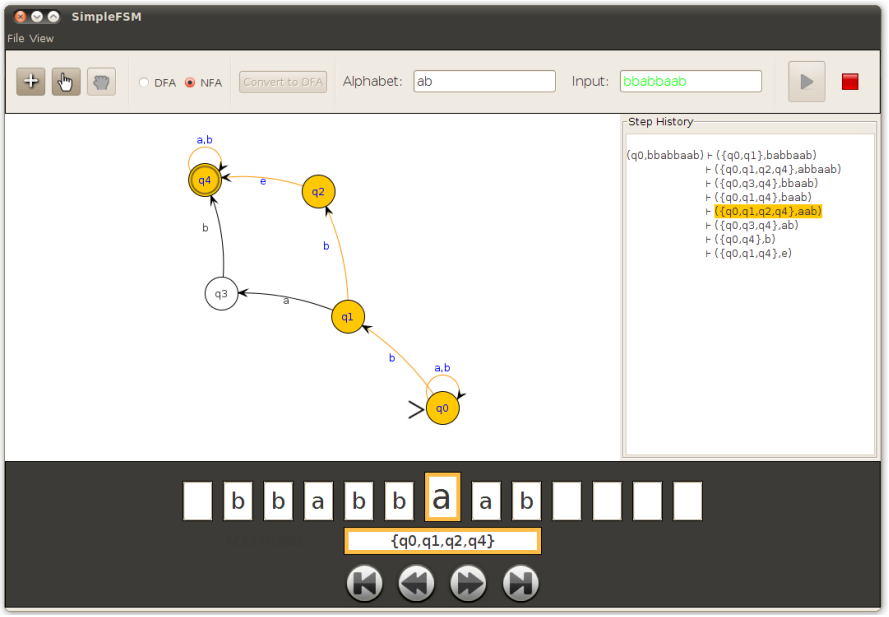

FIGURE 4.16: A screenshot of a *running* automaton

TapeDisplay

The ***first*** row is our visualization of the automaton's tape. There are thirteen cells (implemented by JTextFields), containing one symbol each. The middle cell has an orange border, and symbolizes the automaton's *reading head*, or the symbol that the automaton *is going to read in its next step*. As the tape progresses, we re-arrange and re-load the letters of the input in the corresponding cells to simulate the sensation of *movement*. If the input's length is more than our textfield group can display, the redundant symbols get omitted. After trial and error we found out that thirteen cells will suffice in the vast majority of cases.

The ***second*** row consists of a single cell (JTextField) that displays the id of the automaton's *current state*. In case the current state consists of multiple single states (in other words, a *MultiState*), then a composite id is created. This consists of all the single states' ids separated by commas and enclosed in brackets ({a,b} etc). This is called *powerset* notation (see page 13).

Graph

The automaton's *current state* gets highlighted with an **orange** color. If the current state is constituted by multiple states, all of those get highlighted as well. The *final states* (automaton's states in its last step) are an exception, as we change the highlighting color to indicate that the computation has reached its end. All *accepting* final states turn **green** and all *non-accepting* turn red. This means that if there is *at least one green state* among the final states, the automaton has ***accepted the input***. This further enhances the visual feedback on the result.

In any case, the set of highlighted states correspond to the contents of our *currentState* textfield, at the second row of our TapeDisplay. The states' ids get highlighted with a blue color to be easier recognized.

Futhermore, the transitions (one or more) that led to the current state from the previous step gets highlighted as well. The transitions *labels* (transition symbols) get highlighted with a blue color, as they are an indicator of the symbols that caused these particular transitions accordingly.

This appearance of *animation* is implemented by programmatically *picking* or *selecting* the appropriate elements, after defining the appropriate colors for picked states and picked transitions.

LogDisplay

All the above are accompanied by the corresponding text-highlighting inside our LogDisplay (Step History Panel). This can easily be done in Swing's *JTextArea* components, as the one representing our Step History display area.

The last form of visual feedback that we want to underline, is the change of the *input String's color*, inside the Input textfield. When an automaton has been run successfully, the input String is colored ***green*** if the automaton accepted it and ***red*** if the automaton rejected it. Notice its color in figure 4.16.

4.3.7 Saving/Loading

In Chapter 2.2.5 we saw GraphML, an XML-based language that we used to implement the saving/loading features. We also mentioned that this was done easily, since JUNG provides intergrated GraphML support. JUNG developers have created two classes called *GraphMLWriter* and *GraphMLReader2*, which encode/decode graphs to/from GraphML respectively. An example taken from our application:

```
GraphMLWriter<State,Transition> graphWriter
= new GraphMLWriter<State,Transition>();

...

graphWriter.save(graph,filename);
```

and

```
GraphMLReader2<Graph<State,Transition>,State,Transition> graphReader
= new GraphMLReader2<Graph<State,Transition>,State,Transition>(filename);

...

graphReader.readGraph();
```

The information that we ended up encoding in GraphML includes:

- The automaton's type (dfa or nfa),
- State id's,
- State status (accepting/rejecting),
- State positions (in the form of xy-coordinates of type Double),
- Transition id's (symbols),
- Input,
- Alphabet.

The dialog window that visualizes this function was a simple implementation of Java's *JFileChooser* class. This is a GUI Component made to serve this specific purpose and comes with all the necessary features.

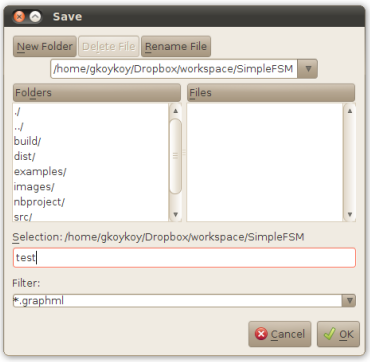

FIGURE 4.17: Java's FileChooser (GTK)

One last thing to note is that if the described automaton contains duplicate states, not only it cannot be run, but it cannot be saved as well. For that reason, we resolve duplicate states programmatically before saving the graph, by adding an incremental numbering suffix next to conflicted state ids (2,3,... etc).

4.3.8 Exporting to Image

Again in Chapter 2.2.7 we mentioned that we used *FreeHep* to export our graph to an image file format. So far the four supported types are ***svg, png, bmp and gif***.

FreeHep is fully compatible with Swing and can export every object that extends *JComponent* class. That is the case with our *VisualizationViewer* which extends *JPanel*, a subclass of *JComponent*. Since FreeHep already comes with a dialog window of its own, called *ExportDialog*, the code to use it in your program is very simple:

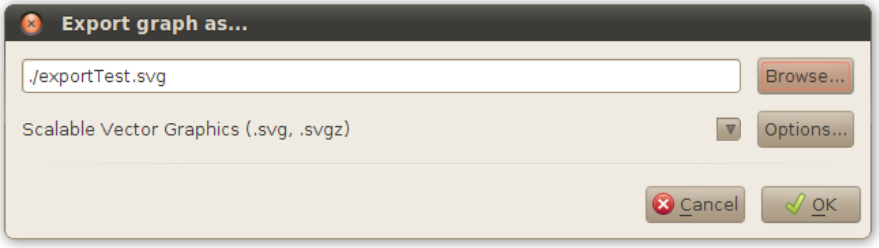

FIGURE 4.18: FreeHep's *ExportDialog*

```
ExportDialog export = new ExportDialog();
export.showExportDialog(VisualizationViewer);
```

This feature is a "last-minute" solution and we consider it to be in *beta*. In future versions of this application we will fix some minor bugs, we will add support for other image file formats (like **.eps or *.ps*) and we will provide a way for the user to *crop* his graph before exporting it.

4.3.9 Preferences

In page 57 we mentioned our File -> Preferences menu. This is a simple menu that gives access to two secondary features.

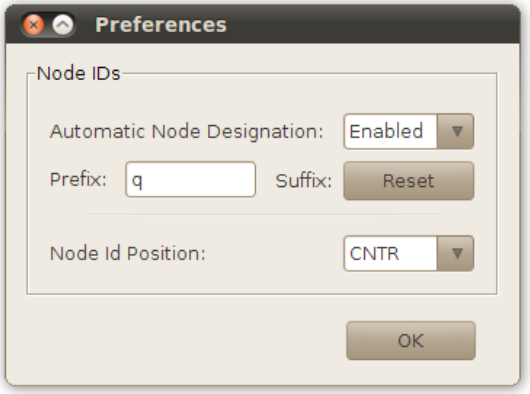

FIGURE 4.19: Our *Preferences* dialog

- ***Automatic Node Designation:*** An incremental integer gets initialized at zero. If the respective checkbox (shown in figure 4.19) is set to *'Enabled'*, then state ids become *auto-generated* upon creating a new node. The id derives from the combination of the desired *prefix* that has been defined in the correspondent *textfield* and the integer's current value as a *suffix*. If the *reset* button is clicked, the increment restarts from zero.
- ***Node Id Position:*** This is a simple *JComboBox*, i.e. a drop-down list of options, which contains the abbreviations of all the posible *relative* positions that a state id

can be drawn at. These are abbreviations of directions, such as 'N' for North, 'E' for East, etc. There is also an option to set it in *'AUTO'*. This is implemented by a simple call to JUNG's ***vv.getRenderer().getVertexLabelRenderer().setPosition()*** method. This feature is useful in case of very long state ids that don't fit inside node circles. This is quite common when *converting* an NFA to a DFA, as the DFA's state ids are *powersets* of NFA states.

Chapter 5

User Evaluation

By the time this document is being written, it is mid-summer and the university has entered it's summer break. Classrooms and dorm rooms are empty and students are mostly away on vacation. This is bad news for us though, as an educational application such as ours should be judged inside a classroom, with teachers and students giving the final verdict. A proper, in-depth user evaluation is not a matter to be taken lightly, especially since our application carries pedagogical value.

With that being said, we decided to perform a simple, informal user evaluation, only to try and apply some last-minute improvements before making this tool public. We tried to find as relative subjects as possible, i.e. people that had been taught *Theory of Automata* at least once in the past. They tested a previous version of our application and their remarks and suggestions were incorporated in the current one.

5.1 Method Description: Think-Aloud Evaluation

For our evaluation we employed the *"Think aloud"* user evaluation protocol, which we found suiting our needs and will be described in the next section. Citing from the corresponding article in Wikipedia[1],

> "***Think-aloud protocol*** (or think-aloud protocols, or TAP) is a method used to gather data in usability testing in product design and development, in psychology and a range of social sciences (e.g., reading, writing, translation research, decision making, and process tracing).
>
> Think-aloud protocols involve participants thinking aloud as they are performing a set of specified tasks. Users are asked to say whatever they are

[1] See:http://en.wikipedia.org/wiki/Think_aloud_protocol.

> looking at, thinking, doing, and feeling as they go about their task. This enables observers to see first-hand the process of task completion (rather than only its final product). Observers at such a test are asked to objectively take notes of everything that users say, without attempting to interpret their actions and words. The purpose of this method is to make explicit what is implicitly present in subjects who are able to perform a specific task."

The task our users were asked to perform was to ***re-create a given NFA using our application, and run it on various inputs.*** We used the NFA from Chapter 4, page 47, figure 4.9 as an example.

5.2 User Feedback

The feedback we received was overall ***positive*** and very encouraging. Most of our users liked the appearance and the simple, minimalistic interface. Those who had a better understanding of finite automata appreciated our *Step History* panel (which represents the automaton's steps in textual form), the backwards iteration on graphs and the error messaging system.

Of course, negative remarks were not absent. We will focus on these more, describing what measures we had to take to balance out some of them.

The **first** -and most important- negative response was usually related to our *modal* mouse. While most editing applications (including the majority of automata simulators) approach mouse modality through the notion of having different *"tools"* (i.e. state-creation tool, transition-creation tool, panning tool e.t.c.), our application uses only three *modes*, each specialized on a different set of tasks. In fact to design a new graph, one only needs the mode called *"Editing Mode"*, which our applications defaults at.

Our users, however, couldn't have known that. Usually when they failed to perform a task they had in mind (for example, draw a new transition), they couldn't derive if they failed due to their own *missclick*, or due to having an inappropriate mouse mode selected. Obviously, the names of our modes (which appear only in the tooltips of their corresponding buttons) didn't help.

Most of our users got around that problem somewhere in the first five minutes, although they found the process frustrating. To account for that, we included a *Help* menu on the menubar, displaying a summary of the application's *User Manual*. There one can find an explicit description of the three mouse modes and how to use them. This fits with our natural response to seek for a Help menu, when stuck in a application.

We didn't want to change our approach altogether. We strongly believe that after a few minutes, when the user finally gets accustomed, this approach is quite comfortable. The important difference from other automata simulators, is that you don't have to change many mouse modes, or tools, when designing a new automaton in our application. In fact, most important tasks are performed using only our *"Editing Mode"* and it is the mode which the user will be using most the time.

By adding the user manual into the application, we also hope to have covered the **second** common negative remark that we received. The fact that nowhere in the application was explicitely stated that the symbol 'e' is reserved as a special character, only to be used to denote an empty transition. Most of our users just guessed it and were not sure if it would work before running the simulation.

Third, most users felt that the visual feedback at the automaton's *last* step, the step that represents the computation's *result*, didn't have enough impact. That happened because in the previous version of our application, the states' highlight color didn't vary at all. Instead it remained **orange** throughout all the steps. To account for that, we changed it only for the last step. So in the current version, when the computation reaches its end, the automaton's *final states* turn **green** in case they are accepting, and **red** if not. This means that if there is at least one green state among them, the automaton has accepted the input. This information is now clearly represented *in color*.

A **fourth** category of negative responses came up, when some users tried to display the transition popup menu (which allows to edit the transition symbols). It appears that our mouse is too location-sensitive, with little to none intolerance in missclicks when trying to display that menu. The user has to click either exactly on the line, or on the arrowhead of the arrow that represents the corresponding transition. It is true that this fact can be very frustrating. However only a minority of our test users mentioned it, so taking measures against it was not of high priority. We may revisit this issue in a future update.

5.3 Conclusion

As we mentioned earlier, the overall responses were positive and encouraging. The first impression we shared with our test users, inclines us to believe that this application has what it takes to fulfill the role of a finite automata simulator used as a pedagogical tool.

Nonetheless, such a tool was meant to be evaluated by teachers and students. Thus, we believe that the most important feedback will be arriving at the start of the next

semester. We hope it will encourage us even further, to extend, evolve and enhance our application's educative value.

Chapter 6

Related Work

According to the article *"Fifty years of automata simulation: a review"*[6], automata simulators can be classified based on their design paradigms into ***language based*** automata simulators and ***visualization centric*** automata simulators. The first category encompasses simulators where the definition of an automaton is written in a predefined symbolic language and processed using tools like *compilers* and *interpreters*. This is not the case with our application though. Our simulator belongs to the second category. It is a *visualization centric automata simulator*, which means that it accepts the specification of an automaton and simulates its working ***graphically***.

The same article further classifies the **visualization centric automata simulators** into two categories:

1. **Those accepting structured input**: Simulators that accept specifications of automata in predefined structured formats. Such a format often comprises of a table to store the transition function. The user fills in a form providing the necessary details of an automaton and promptly starts the simulation process.

2. **Those accepting diagrammatic input:** Simulators that need the user to draw the transition diagrams of the automata. These tools typically provide a canvas where states and transitions are added and positioned by clicking and dragging the mouse. This gives the user a feel of drawing an automaton on paper. Oftentimes animation is employed during simulation to enhance pedagogy.

Our application belongs to the second sub-category which, in fact, is favored by the students (acm Inroads, 2011 December, Vol.2, No.4, pg. 64).

So to sum up, our application is a ***Visualization Centric Finite Automata Simulator, Accepting Diagrammatic Input.*** Therefore search for related projects was focused on simulators that belong to the same category as well.

Before moving on, we have one important thing to note: Many researchers publish in journals and conferences describing their solutions; in "Fifty years of automata simulation" alone, one can find more than ten projects that relate to ours. However, not all of these are accessible by simple internet searches using the application's title, or the author's name, or the corresponding paper's title, or by searcing their university's page. This goes to show that the accessibility of an educational tool is an important factor in its deployment.

In the following sections we will see other simulators that belong in the same category as ours and are *easily* accessible with a simple internet search using their keywords.

6.1 FLAP & JFLAP

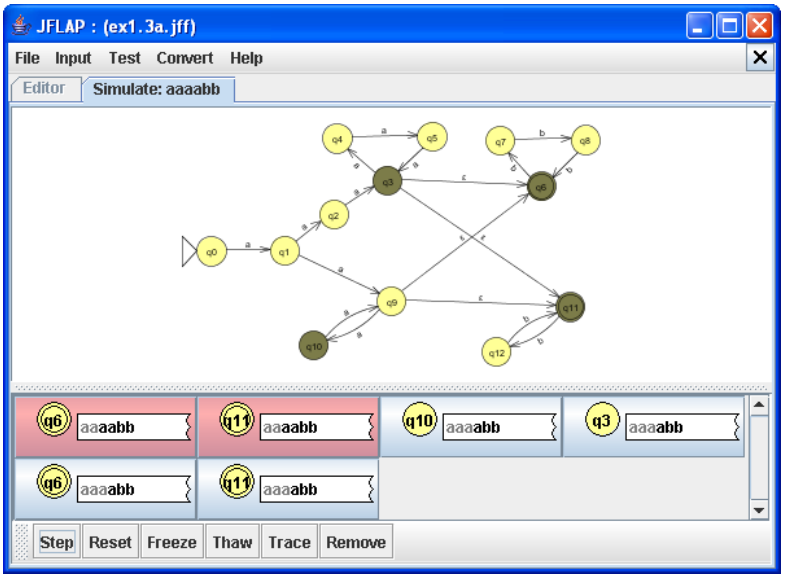

FIGURE 6.1: A running NFA in JFLAP

In 1993, Professor Susan H. Rodger of Duke University and co-researchers developed a Formal Language and Automata Package (FLAP) to design and simulate finite automata, pushdown automata and Turing machines. This tool had been successfully used for teaching at the Duke University. Few years afterwards, Rodger and co-researchers developed a Java Formal Languages and Automata Package (JFLAP[1]), as an evolution from its predecessor. In the last two decades, the tool has been under continuous

[1]You can find it at: www.jflap.org

enhancement and new features have been added regularly. This tool supports several deterministic and nondeterministic variants of finite automata, pushdown automata, and Turing machines as well as Mealy machines and Moore machines.

This tool is undoubtedly the most widely used tool for simulation of automata developed to date. Thousands of students have used it at numerous universities in more than a hundred countries[2]. It is licenced under its own licence (JFLAP 7) and one has to fill a form created to track JFLAP usage, in order to get the application for free. You can find it in `www.jflap.org`.

For additional screenshots see figures 6.5,6.6 and 6.7.

6.2 Automaton Simulator

In 2001, Burch, C. developed an Automaton Simulator to visually design and simulate finite automata, pushdown automata and Turing machines. In this tool, an automaton can be simulated either stepwise or instantaneously. An option for rewinding the simulation process is also available. It is distributed under **GPLv2** Licence and you can find both the application and the source code at: `http://ozark.hendrix.edu/~burch/proj/autosim/`.

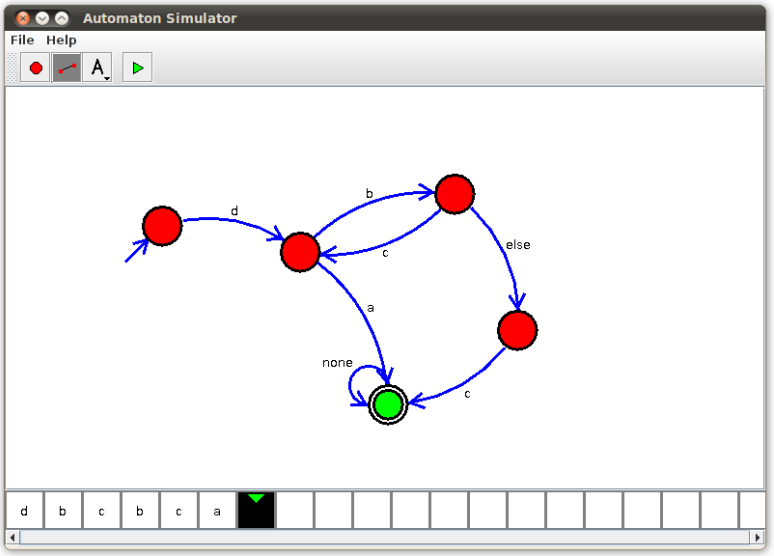

FIGURE 6.2: Screenshot of Automaton Simulator 1.2

In our opinion it has some flaws. First of all, you can't name the states and you can only use six different transition symbols: 'a', 'b', 'c', 'd', 'none' and 'else'. Second, the simulation animation happens real-time, as the user types the input tape which can be

[2]See `http://www.jflap.org/stats2008/`

a bit confusing. Finally, the author has abandoned the development of this project a long time ago.

6.3 Visual Automata Simulator (VAS)

In 2004, Jean Bovet developed the Visual Automata Simulator (http://www.cs.usfca.edu/~jbovet/vas.html). This application supports visual designing and simulation of finite automata and Turing machines. It features multiple tapes, batch testing for multiple files, opening of multiple documents at the same time, exports in EPS file format and more. It is used in D.Galles course on the *Automata Theory* at the University of San Francisco, Department of Computer Science. It is written exclusively in Java and Swing, and is licenced under BSD.

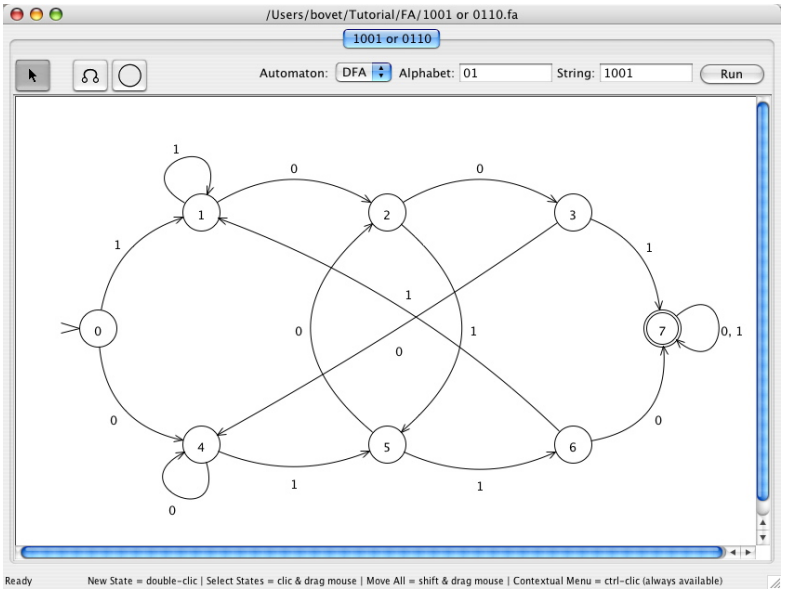

FIGURE 6.3: Designing a DFA using VAS

In general, VAS is a very intuitive application. The only thing that is counter-intuitive is the step-by-step animation. When an automaton runs the result is printed on screen immediately. For the user to iterate through the steps, he/she must first enter "debug" mode using the menu and then move back and forth using keyboard shortcuts. Also VAS doesn't support importing files written in some kind of description language, like we did with JSON. Like the previous case, this project seems to be unmaintained as well.

For additional screeshots see figures 6.8 and 6.9.

6.4 Java Finite Automata Simulation Tool (JFAST)

White, T. and Way, T. in 2006, developed a Java Finite Automata Simulation Tool to design and simulate visually finite automata, pushdown automata, Turing machines, and other types of automata. The tool supports both deterministic and nondeterministic machines. The tool allows designing complex automata by integrating simpler sub-machines. This tool is also written in Java, and is licenced under a slightly modified GPL. You can find it at: http://jfast-fsm-sim.sourceforge.net/

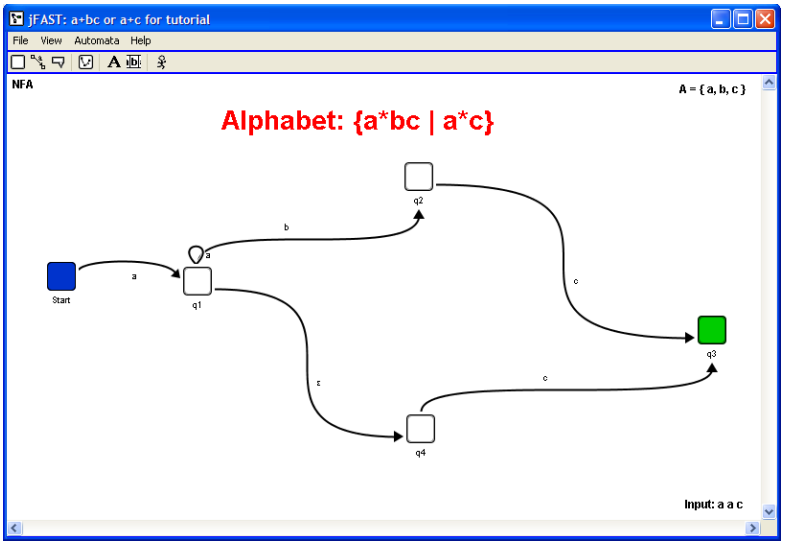

FIGURE 6.4: JFAST Editing Interface

Like most of the previous projects, this one seems to be unmaintained since 2006. The authors are aware of some weaknesses of this application, and mention them under section "User suggested improvements" in their homepage. These include the optional ability to enter an input as a string (i.e. using the keyboard) instead of selecting from a list, and redesigning the simulation interface to be more intuitive.

To see how JFAST looks like when simulating, go to figure 6.10.

6.5 Additional Screenshots

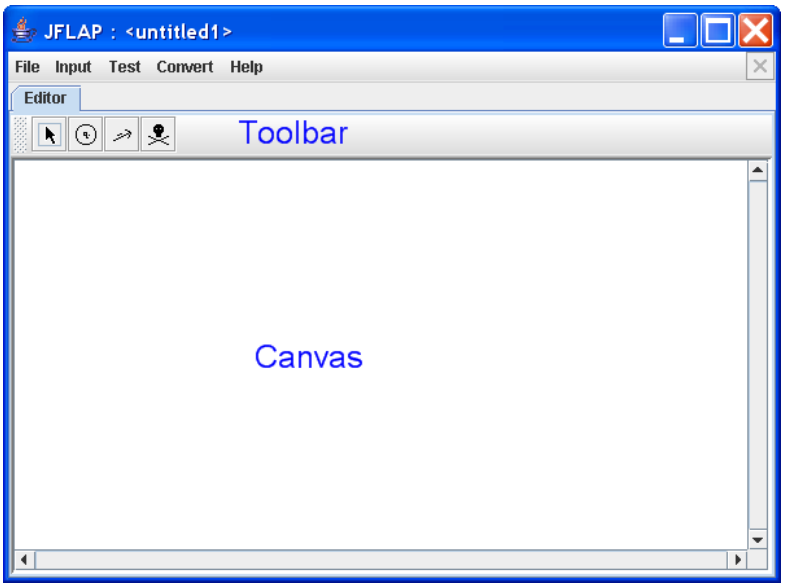

FIGURE 6.5: JFLAP editor window

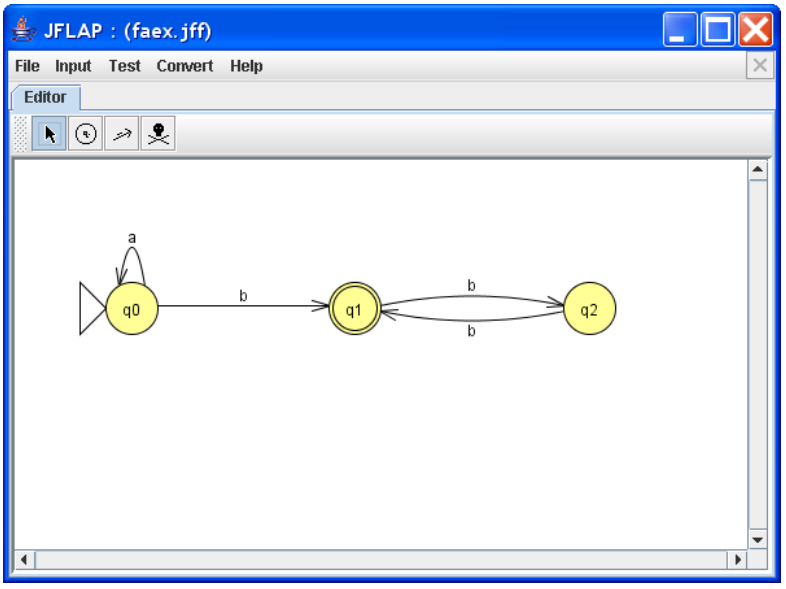

FIGURE 6.6: Designing a DFA in JFLAP

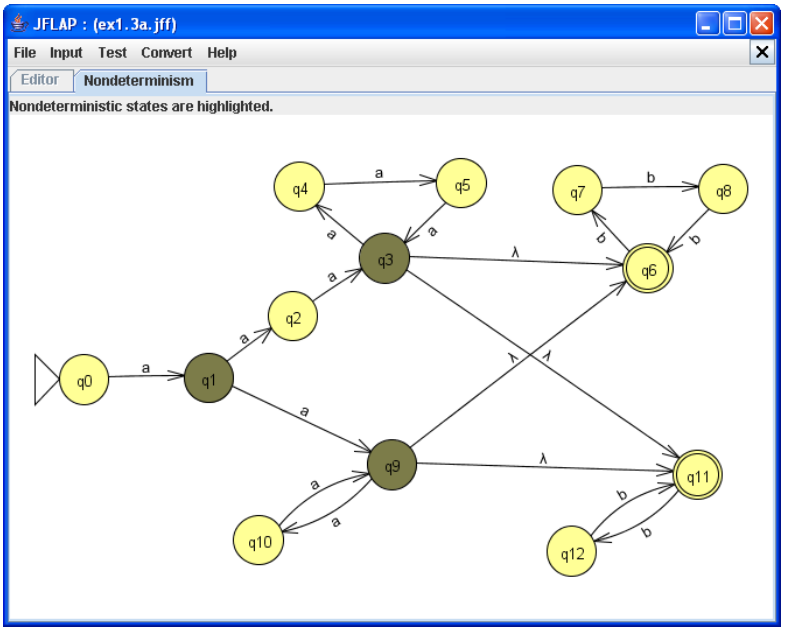

FIGURE 6.7: An NFA in JFLAP (nondeterministic states highlighted)

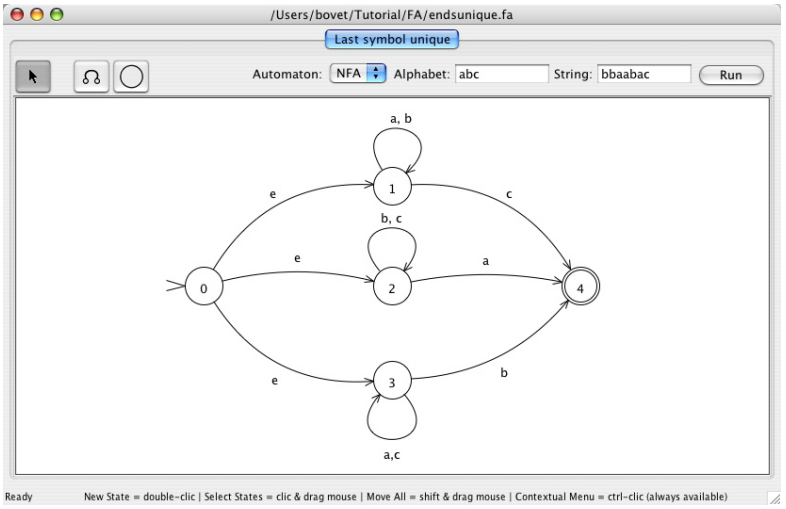

FIGURE 6.8: Designing an NFA using VAS

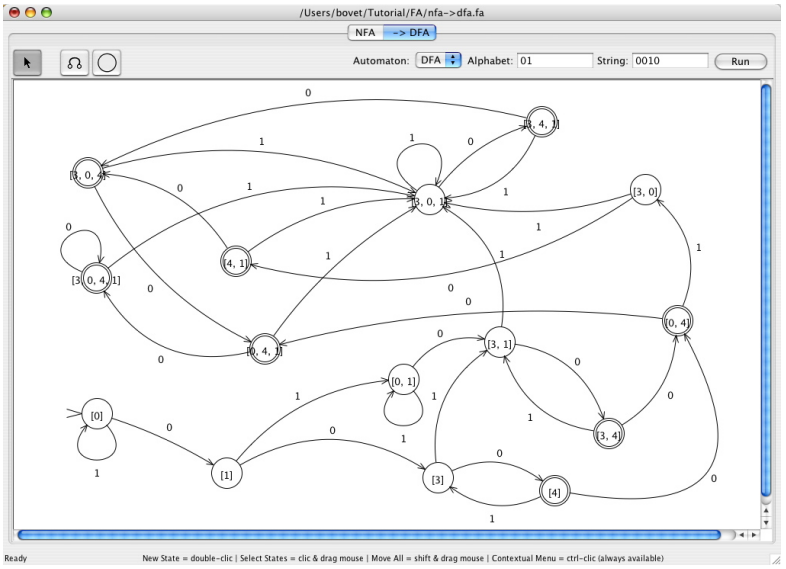

FIGURE 6.9: NFA to DFA Conversion

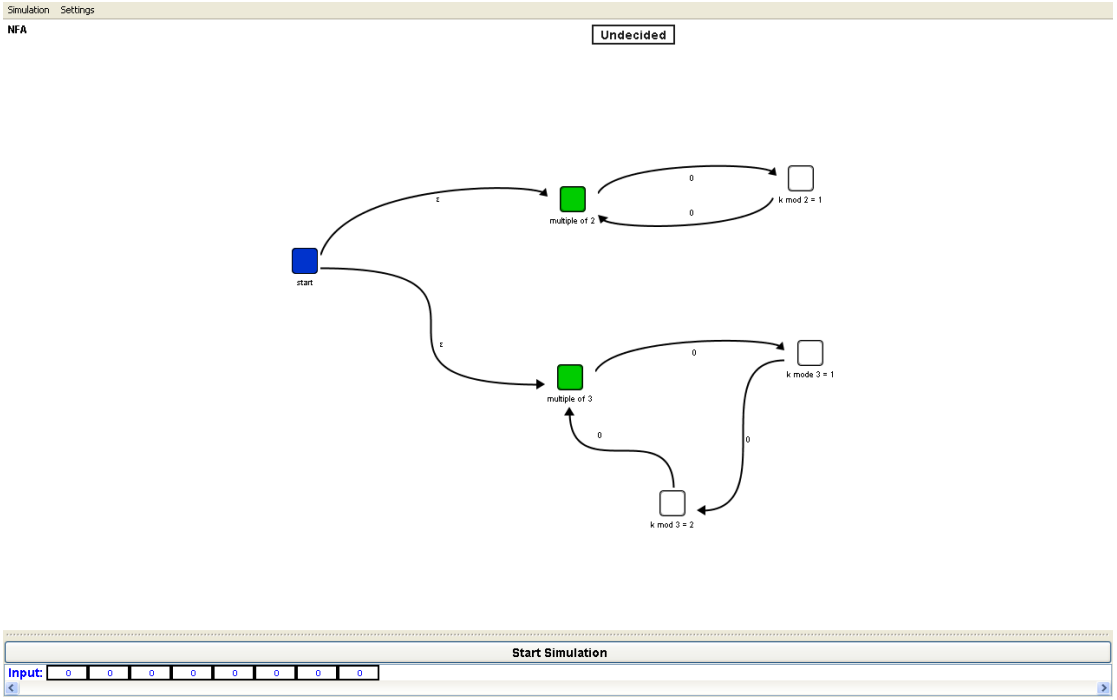

FIGURE 6.10: JFAST Simulation Interface

Chapter 7

Conclusions

7.1 Discussion

In Chapters 2 and 3 we made an attempt to justify our reasons behind making yet another finite automata simulator. We also attempted to justify its use it as a *pedagogical tool* in our University's course *Computation Theory*. With the appopriate literature backing us, we think we succeeded in that. The final confirmation will be delivered as soon as the summer break ends, and the students return to the classrooms to perform the decisive test. We are eager to know, *in what way* will students and teachers find our application to be useful, and whether it will enhance their understanding of fundamental concepts of the *Automata Theory* at all.

In Chapter 4 we saw how we used Sun/Oracle's Java language to implement our tool, and various third party libraries (like JUNG, a graph visualization framework). Overall, we do not regret that decision. It might not have been the optimal choice aesthetically, nor performance-wise, but it was a sure-fire solution that guaranteed we wouldn't get stuck behind some kind of obscure bug, undocumented code, depricated libraries and such. In addition to providing cross-platform execution, the benefits of using Java became apparent each time we rushed to a community forum full of questions, to find out that they had already been answered.

Then, in Chapters 5 and 6, we put our application to the test. **First** with users that provided an informal - and yet important - initial evaluation, and **second** by comparing it to other related projects that we could get our hands on. From these two chapters our reader can derive that our application is on par with the other simulators (at least as fas as DFAs and NFAs are concerned), and that it can fulfill its role as a pedagogical tool. We believe that it offers some minor advantages as well, such as compliance to the book

taught in our university's *Computation Theory* course, comfortable mouse functionality, and obvious visual and textual feedback on the automaton's computation and tape. Additionally, it can import automata described in the JSON language, and it is free software (licenced under GPLv3).

7.2 Future Work

We consider this project to be far from complete. Improvements, additions and modifications are in due, in order for it to become what we have envisioned. As we mentioned before, we expect that when it is finally tested in the classroom, feedback from teachers and students will bring up even more tasks for us to accomplish.

In an attempt to prioritize though, we will describe these tasks that would vastly contribute towards our purpose, which is *to provide a visual automata simulator used as a pedagogical tool*.

- A festidious, full-fledged *User Evaluation* should be done, to study and verify the benefits of using such tools in the classroom. We hope that it can prove to be more than a homework assistant to the students, that it will help them comprehend the fundamental concepts of the *Theory of Automata* and *Computation Theory*, while additionally serving as a teaching assistant to the professors.
- Exploit the relationship of *finite automata* with *regular languages* by implementing various conversion algorithms from DFA and NFA to the corresponding *regular expressions*. This would further underline the important role that finite automata have in *Natural Language Processing (NLP)*, among others.
- The study of finite automata is the recommended place to start, but in order to delve deeper, one must study more complex structures. Our application therefore should be extended to simulate other types of automata, such as *Turing Machines* and *pushdown automata*. It should also support *multiple, bi-directional tapes*.

From a technical aspect, there is work to be done to improve our application's GUI as well.

- Refactoring it to become more responsive. More detailed profiling should be performed, to track down resource-demanding actions and employ multiple *threads* to divide the workload.

- A better *"exporting to image"* module is needed. One that would export to the popular EPS file format, and would include a *print preview* feature where the user could crop the image to the desired frame.
- Moreover, the third-party library that we currently use to export our graphs to image files *doesn't work* for versions of Java higher than **6**, and in fact, doesn't work for any other Java platform other than Sun/Oracle's JRE. Our application doesn't have such kind of dependencies in general, and it can be run on various platforms and versions (without the export feature of course). We find this to be restrictive, so we should find out a way to resolve these dependencies as future work.
- Employ Java/Swing's *GridBagLayout manager* instead of our current manual layouting of the components. This would add a *dynamic behaviour* to our GUI, since the components would be able to adapt to window resizing.
- Our whole application is written in Java, so it can be run on any platform that bears the appropriate virtual machine. However, it's appearance is optimal under the *GTK Look and Feel*, which can be found in Linux systems. Tests and modifications should be made, so that it could appear on any other platform in an optimal way as well.
- Although Swing is a decent, reliable GUI-library, it no more belongs to the library's that provide *state of the art* graphics. It may suffice when it comes to simple, educational applications, but it isn't on par with other modern GUI building tools. Nonetheless, there are many Swing-compatible, third-party libraries (e.g. *SwingX, JGoodies*), which could be used to upgrade our GUI's appearance in the future.
- Lastly, to take full advantage of using Java as our development language, our future intentions include the deployment of this application as a *JApplet* as well.

To sum up, there is a lot of work to be done and we realise that. We already consider this application to be in *beta* status. We ask our users to bear with us, expect future releases, and provide any suggestions or other forms of feedback they can think of. The source code will be free and available to those who feel like contributing.

7.3 Lessons

The development of this project was an invaluably educative experience, as an undergraduate thesis project should have been. We will attempt only a rough outline of the most important lessons one can extract from this process.

Agile Development

If we were forced to chose the single most important lesson, was learning about the notion of *Agile Development*[1]. Quoting from wikipedia,

> "**Agile software development** is a group of *software development methods* based on iterative and incremental development ... It promotes adaptive planning, evolutionary development and delivery, a time-boxed iterative approach, and encourages rapid and flexible response to change. It is a conceptual framework that promotes foreseen interactions throughout the development cycle."

This introduced an important concept to us. When developing a medium-sized (or larger) project like this, it is difficult to design everything from the start, blindly implementing whatever lies on paper afterwards. Even more so, if the developers lack experience like in our case. Agile development enabled us to engage in a development cycle which kept us in motion, even when stuck: Create a simple prototype, experiment, use feedback to work on design, refactor and remodel the prototype, and so forth. The time period of the iterations was usually *a week*.

Indicatively, we will only mention what our iterations involved in the first few weeks of development.

Week 1: Created a single class called "Machine", that could only simulate DFAs, using a textual input and output at the console.

Week 2: Find an appropriate library to draw graphs, and create a second class called "GUI" to show the results of our Machine. Add a single button to perform forward steps.

Week 3: Add the feature of non-determinism to our Machine.

Week 4: Decompose, design, refactor source.

Week 5: Create a new prototype GUI to accomodate the changes.

... and so on.

In figures 7.1, 7.2 and 7.3 you can see our "Agile Development" story in pictures.

[1] See http://en.wikipedia.org/wiki/Agile_software_development.

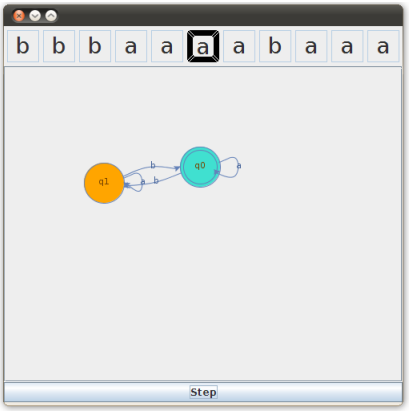

FIGURE 7.1: After two weeks

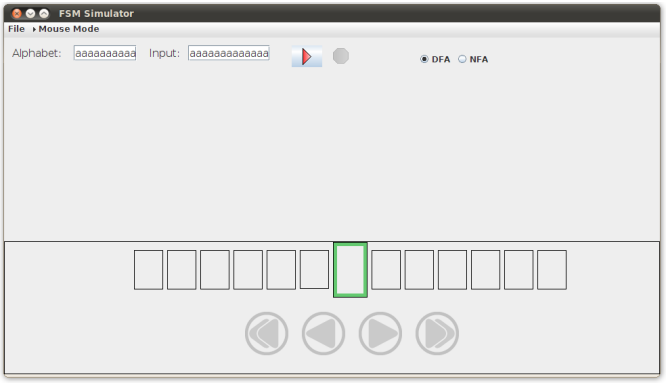

FIGURE 7.2: ... in the second month.

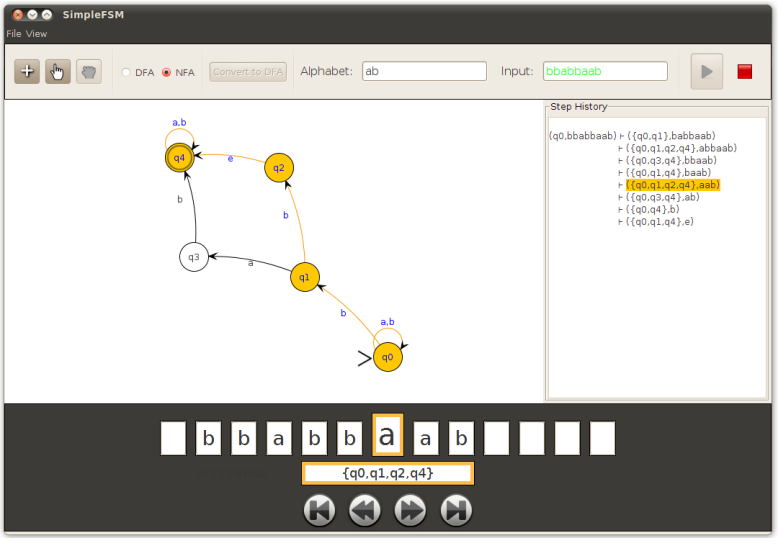

FIGURE 7.3: ... and the current version.

Testing/Debugging

Our second lesson was one taught in the hard way: ***Using JUnit when developing in Java***. JUnit[2] is a *unit testing framework* for the Java language. This tool enables the developers to create small tests, execute them, and check how different parts of the project work, independently.

The combination of *JUnit* with a high-level *debugger* (like one included in most IDEs) is very powerful, and we highly recommend using it. In simple words, if you are a beginner programmer who is just making his/hers first steps in larger-scale projects, getting accustomed with those tools is time worth spending.

Extensive know-how

Finally, this project gave us extensive know-how in many levels. We learnt how to use new IDE's and plugins (NetBeans, JUnit, GUI Builder, e.t.c.), how to program a GUI in Swing (user-generated events and how to handle them, customizing GUI components, e.t.c.), design patterns in software engineering (like MVC) and more. We even learnt how to compose a thesis document in Latex. Additionally, having know-how on JUNG itself (the graph-drawing library that we used) may prove to be a valuable asset in the future in case we need any kind of graph visualization or data representation of some sort, since it is a high-level library with numerous features and scientific applications.

We hope that maintaining and extending this project will help us learn many more things in the future. Our main job - delivering an undergraduate thesis project - is done, but our most important job is ***not***. This is none other, than *delivering a truly educational tool*.

[2] See http://en.wikipedia.org/wiki/JUnit.

Appendix A

User Manual

A.1 Installation

You need Sun's/Oracle's **Java SE Runtime Environment 6** to run this application. You can find it following this link: http://www.oracle.com/technetwork/java/javase/downloads/jre6-downloads-1637595.html.

Running this application in higher Java versions *will work*, although the "Export to image" feature will be broken due to third-party dependencies.

If you are using Linux and you don't desire to install this specific platform in your system, simply download and extract it into a folder and then execute the script *java* (in sub-folder */bin*) with *FASim.jar* as an argument (e.g. */myFolder/bin/java -jar FASim.jar*).

A.2 License

This application is free software, licensed under the **GNU General Public License, version 3 (GLPv3)**. To get the source code, contact:
George Koykoympedakis,
gkoykoy@gmail.com

A.3 General

FA-Sim (Finite Automata Simulator) is a tool created for pedagogical purposes, intended to be used by students and teachers alike. Students can use it as a homework

assistant while studying difficult examples or solving exercises. Teachers, on the other hand, can use it to simplify and visualize complex concepts, such as *nondeterminism* and *the conversion of NFA to DFA*, or as a simple graph editor to create and print examples for their notes and their slides.

If you don't know what *finite automata* are, you can start here: http://en.wikipedia.org/wiki/Deterministic_finite_automaton and http://en.wikipedia.org/wiki/Nondeterministic_finite_automaton.

Here follows a summary of all the features that FA-Sim supports so far:

- Creation of graphs representing ***DFA*** and ***NFA***,
- Defining an ***alphabet*** and an one direction ***input tape***,
- ***Simulating*** the automaton's computation,
- ***Iterating*** through the computational run ***step-by-step*** and with ***backwards iteration***,
- ***Visual*** and ***textual*** representation of the automaton's ***step history***,
- ***Saving*** and ***loading*** graphs from the hard drive,
- ***Importing*** a graph described in the ***JSON***[1] language,
- Exporting the graph in ***SVG, PNG, GIF*** and ***BMP*** file formats.

A.3.1 GUI

The GUI is divided into five main areas (figure A.1):

1. *Menubar*,
2. *Toolbar*,
3. *Canvas (graph display & edit)*,
4. *Step History (log display)*,
5. *Tape Display & Controls*.

[1] See en.wikipedia.org/wiki/JSON for more.

The Tape Display area doesn't become visible, unless an automaton has been simulated *successfully*. Step History on the other hand, can be turned on/off at will, by using the *View -> Step History* menu in the Menubar. The ***Tape Display*** panel consists of three different elements (fig.A.2),

1. *the automaton's input tape contents*,
2. *the current state textfield*,
3. *the tape controls*.

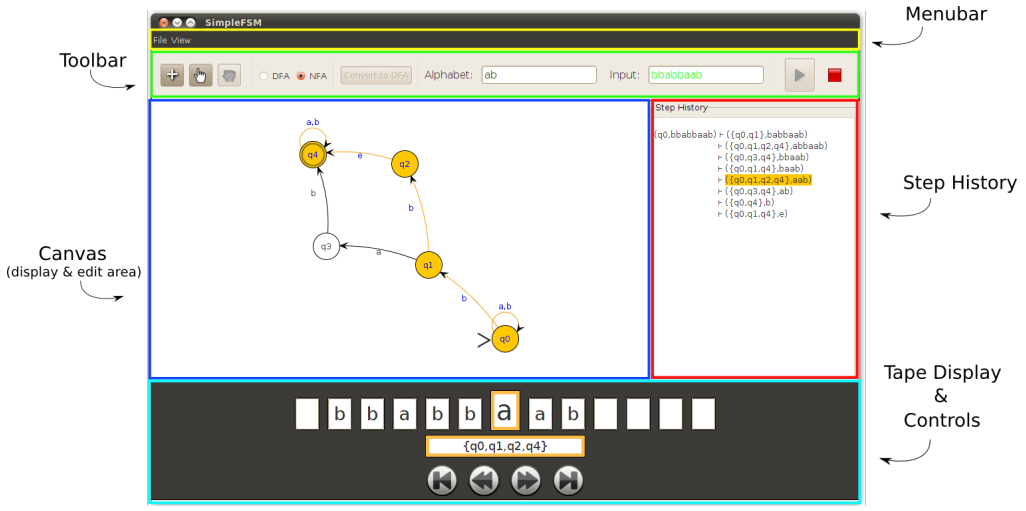

FIGURE A.1: The five main areas

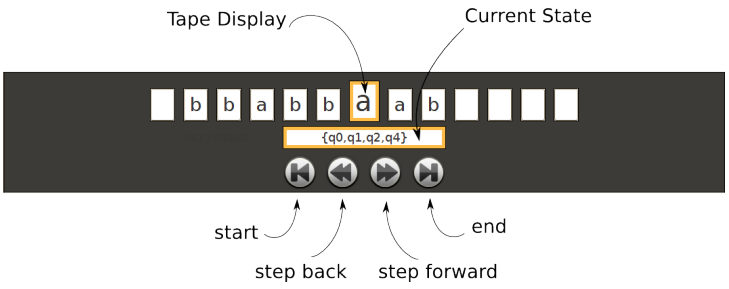

FIGURE A.2: Tape Display and Controls

The last area we need to cover before continuing is the ***Toolbar***. This area along with our ***Canvas*** complement eachother, in terms of describing and simulating an automaton. Its elements are (fig.A.3):

1. *Mouse Mode Selection (buttons)*,
2. *Automaton Type Selection (buttons)*,
3. *Convert NFA to DFA (button)*,
4. *Alphabet & Input (textfields)*,
5. *Run/Stop simulation (buttons)*.

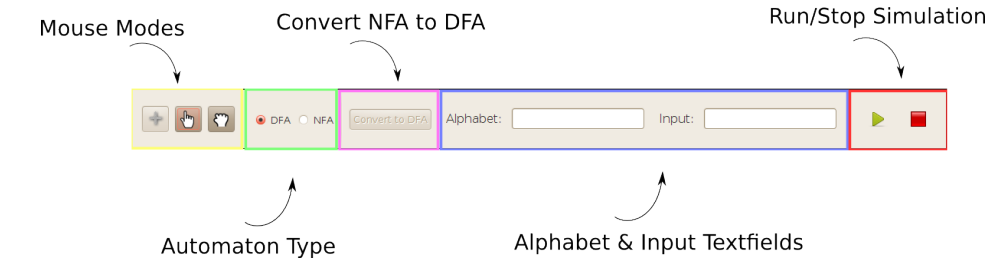

FIGURE A.3: Toolbar

A.3.2 The FA-Sim Mouse

FA-Sim uses a modal mouse, which means that mouse functionality *changes* according to the mode it's in. However, only the ***left-click*** functionality changes, while the ***right-click*** behaves the same throughout all the different modes. These include:

1. *Editing (crosshair icon)*,
2. *Picking (pointing hand icon)*,
3. *Transforming (grabbing hand icon)*.

FIGURE A.4: Mouse Mode Buttons

Here follows a description of the mouse's functionality, sorted by it's different modes:

Editing mode('e'):

- Left-click on empty space to **create a new state**.

- Left-click and hold on a state, drag mouse to another state and release to **create a new transition**.
- Left-click on a single state to **create a looped transition**. This means a transition where the *source state* is the same as the *target state*.

Picking mode('p'):

- Left-click on a state to **pick**, or **select** it.
- Left-click on a state and hold, drag mouse to a new location and release to **move a state**.
- Left-click on different states while holding down the **shift** button, to **select multiple states**.
- Alternatively, left-click on empty space, hold and drag the mouse to form a **box selection** of multiple states.
- Having selected multiple states at once, you can **move them** in the same way as you would move a single state.
- Left-click on a transition to **pick** it. There is no practical reason to do this, only visual.
- Left-click on a state while holding the **ctrl** button to **bring that state to the center of the display**.

Transforming mode('t'):

- Left-click and hold anywhere, drag mouse and release to **pan** (translate) the graph.
- Left-click and hold anywhere while holding the **shift** button, drag mouse and release to **rotate** the graph.
- Left-click and hold anywhere while holding the **ctrl** button, drag mouse and release to **shear** the graph.

Functions that apply to all the modes:

- Use the mouse scroll wheel to **zoom in and out**.
- Right-click on empty space to **delete selected states** (the corresponding transitions are deleted automatically).
- Right-click on a state to access the **state popup menu**. In this menu you can:
  1. **Delete** the state,
  2. Make it a **starting state**,

3. Make it an **accepting state**,
4. **Edit** it's **name**.

- Right-click on a transition to access the **transition popup menu**. In this menu you can:
  1. **Delete** the transition,
  2. **Edit** the transition's symbol. To assign **multiple** transition symbols, separate them by **commas** (e.g. "a,b,c,..." etc). Type in the character ***'e'*** to denote an empty transition.

A.4 Running a Simulation

To run a simulation, first make sure you have selected the appropriate *type* in the toolbar. This will only affect the error messages you'll get, as DFA have additional restrictions to NFA and the application must perform different checks.

Afterwards provide the desired *alphabet* and *input* in the corresponding textfields, *without* separating characters (e.g. commas) between the symbols. Note that the input can be *empty*, while the alphabet *cannot*. Finally, press the ***"run"*** button , at the upper right of the window. If the automaton has errors, you will get a dialog window displaying the appropriate error messages. Resolve them and repeat.

When succeeding in running the automaton the *Tape Display* panel appears. You can now iterate through the automaton's computational steps using the ***control buttons*** (fig.A.2). Optionally, you can enable the *Step History* panel (or *Log Display*), through the **View -> Step History** menu in the menubar (keyboard shortcut **ctrl+H**).

Note that the color of the input string *changes* to indicate the computational result, as it becomes **green** if the input was accepted and **red** if the input was rejected. To further enhance the visual feedback on the computation's result, the *final states* change color as well. All non-accepting states turn **red**, and all accepting ones turn **green**. This means that if there is *at least one green state* among them, the automaton has accepted the input.

Many options become disabled while an automaton is running. Press the ***"stop"*** button to enable them and prepare for your next simulation.

A.5 Converting an NFA to a DFA

The button that corresponds to this action is in the toolbar, at the top of the screen (fig.A.3). This button is *enabled* only if the type of an automaton has been set to *NFA*. When it is clicked, the application checks for errors once again, and informs the user. ***You cannot convert to a DFA, unless you have resolved all the errors.***

A.6 Importing from JSON

JSON is a human-readable XML-like language, widely common in web applications, but we used it as a *Finite Automata Description Language*. You can ***import*** a *.json file in FA-Sim using the ***File -> Import*** menu.

JSON has support for limited types of variables, but we will only list the ones we used:

- ***String*** (double-quoted Unicode)
- ***Array*** (an ordered sequence of values, comma-separated and enclosed in *square brackets*; the values don't need to be of the same type)
- ***Object*** (an ordered collection of key:value pairs with the ':' character separating the key and the value, comma-separated and enclosed in *curly braces*; the keys myst be string and should be distinct from each other)

Here follows a simple example of how we used JSON to describe an NFA:

```
{  "type":"nfa",

   "states":["q0","q1","q2","q3","q4"],
   "transitions": [
                    ["q0","a","q0"],
                    ["q0","b","q0"],
                    ["q0","b","q1"],
                    ["q1","a","q3"],
                    ["q1","b","q2"],
                    ["q2","e","q4"],
                    ["q3","b","q4"],
                    ["q4","a","q4"],
                    ["q4","b","q4"]
                  ],
   "startState":"q0",
   "acceptStates":["q4"],
   "alphabet":"ab",
   "input":"bb"   }
```

Using JSON to describe an NFA

As you can derive from this example, defining more than one transition symbol in the same line ***is not allowed***. Instead, the user should create ***multiple entries*** in the "transitions" array. Also note that we have reserved the character ***'e'*** as a ***special character***, denoting ***empty transitions***. Lastly, non-significant white space may be added freely around the "structural characters" (i.e. brackets " [ ]", colons ":" and commas ",").

In our opinion, any further explanations would be redundant. This is how the described NFA looks like:

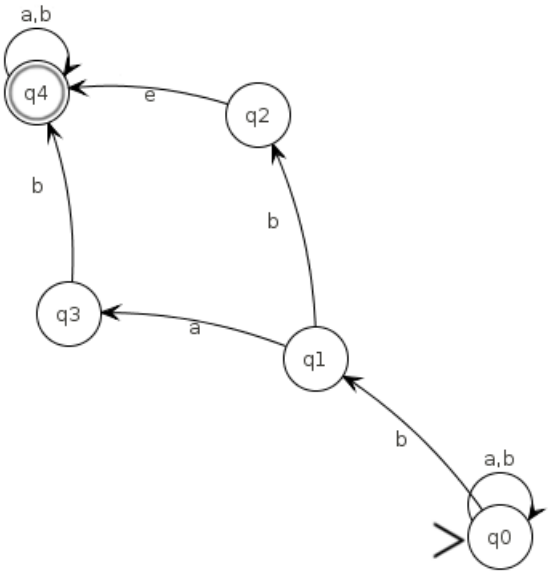

FIGURE A.5: Visualization of the described NFA

The layouting of the graph is being done algorithmically, but the user can reposition all the nodes at will, using the mouse's *picking mode*.

The fact that JSON is a natural, efficient, out-of-the-box solution to describe Finite Automata, is utterly interesting. To our knowledge it has ***never been used in such a way before.***

A.7 Exporting to Image Files

As a last-minute addition, FA-Sim can export to the following image file formats:

- **SVG** (Scalable Vector Graphics)
- **BMP** (Bitmap Image File)

- **PNG** (Portable Network Graphics)
- **GIF** (Graphics Interchange Format)

The exported image has the ***exact*** same appearance as our canvas; and by that we mean that **everything** matters: the location of the graph inside the canvas, any picked states/transitions, ***even the size of the window***. Actually you can use the last one to ***"crop"*** the image. This is a "dirty" solution that we intend to make up for, in the future.

A known bug is that when we export to svg, Java's renderer ***breaks*** and the quality of our graph plummets. The only way to fix it is by restarting the application. Fortunately, this doesn't affect the the ***svg***[2] export feature itself, i.e. even though Java's rendering breaks and the graph appears to be ugly, the exported images look as they were supposed to. This comes in handy when we want to export multiple svg's in succession.

Please bear in mind that we consider this application to be in ***beta*** and will get improved in the future.

A.8 Preferences

This is a simple menu under ***"File"*** in menubar, that gives access to two secondary features.

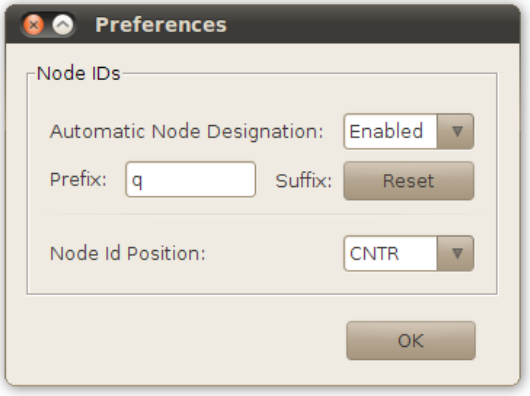

FIGURE A.6: Our *Preferences* dialog

- ***Automatic Node Designation:*** An incremental integer gets initialized at zero. If the respective checkbox (shown in figure A.6) is set to *'Enabled'*, then state ids become *auto-generated* upon creating a new node. The id derives from the combination of the desired *prefix* that has been defined in the correspondent *textfield* and

[2] And only the SVG. Exporting to the other formats (*bitmaps*) breaks too.

the integer's current value as a *suffix*. If the *reset* button is clicked, the increment restarts from zero. This feature is *enabled* by default.

- ***Node Id Position:*** This is a simple *JComboBox*, i.e. a drop-down list of options, which contains the abbreviations of all the posible *relative* positions that a state id can be drawn at. These are abbreviations of directions, such as 'N' for North, 'E' for East, etc, with an additional option to set it in *'AUTO'*. This feature is useful in case of very long state ids that don't fit inside node circles (quite common when *converting* an NFA to a DFA).

A.9 Contact Info

To report bugs, offer suggestions, or help us develop this project in any way, contact:

George Koykoympedakis,
gkoykoy@gmail.com

Bibliography

[1] Michael Sipser. *Introduction to the Theory of Computation*. CENGAGE Learning Custom Publishing, June 2012. ISBN 9781133187790.

[2] Stephen Wolfram. *A new kind of science*. Wolfram Media, 2002.

[3] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, 1998. ISBN 9780132624787.

[4] Formal language - wikipedia, the free encyclopedia. URL http://en.wikipedia.org/wiki/Formal_language.

[5] Automata theory, June 2013. URL http://en.wikipedia.org/w/index.php?title=Automata_theory&oldid=559091717. Page Version ID: 559091717.

[6] Pinaki Chakraborty, P. C. Saxena, and C. P. Katti. Fifty years of automata simulation: a review. *ACM Inroads*, 2(4):59–70, December 2011. ISSN 2153-2184. doi: 10.1145/2038876.2038893. URL http://doi.acm.org/10.1145/2038876.2038893.

[7] Nondeterministic finite automaton, June 2013. URL http://en.wikipedia.org/w/index.php?title=Nondeterministic_finite_automaton&oldid=558772432. Page Version ID: 558772432.

[8] Java (programming language), June 2013. URL https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=558718923. Page Version ID: 558718923.

[9] Bruce Eckel. *Thinking in Java*. Prentice Hall Professional, 2003. ISBN 9780131002876.

[10] History of java technology. URL http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html.

[11] Oracle technology network for java developers. URL http://www.oracle.com/technetwork/java/index.html#943.

[12] Java platform, standard edition, June 2013. URL http://en.wikipedia.org/w/index.php?title=Java_Platform,_Standard_Edition&oldid=555073430. Page Version ID: 555073430.

[13] OpenJDK, June 2013. URL http://en.wikipedia.org/w/index.php?title=OpenJDK&oldid=555523114. Page Version ID: 555523114.

[14] OpenJDK. URL http://openjdk.java.net/.

[15] Swing (java), June 2013. URL http://en.wikipedia.org/w/index.php?title=Swing_(Java)&oldid=557675877. Page Version ID: 557675877.

[16] JUNG - java universal Network/Graph framework. URL http://jung.sourceforge.net/.

[17] JUNG, May 2013. URL http://en.wikipedia.org/w/index.php?title=JUNG&oldid=549582111. Page Version ID: 549582111.

[18] GraphML, May 2013. URL http://en.wikipedia.org/w/index.php?title=GraphML&oldid=556038456. Page Version ID: 556038456.

[19] The GraphML file format. URL http://graphml.graphdrawing.org/.

[20] JSON, . URL http://www.json.org/.

[21] JSON, June 2013. URL http://en.wikipedia.org/w/index.php?title=JSON&oldid=559219644. Page Version ID: 559219644.

[22] json-simple - JSON.simple - a simple java toolkit for JSON - google project hosting, . URL http://code.google.com/p/json-simple/.

[23] FreeHEP, June 2013. URL http://en.wikipedia.org/w/index.php?title=FreeHEP&oldid=552195737. Page Version ID: 552195737.

[24] FreeHEP web site - FreeHEP java libraries. URL http://java.freehep.org/.

[25] Welcome to NetBeans. URL https://netbeans.org/.

[26] NetBeans, June 2013. URL https://en.wikipedia.org/w/index.php?title=NetBeans&oldid=559092227. Page Version ID: 559092227.

[27] M. W. Curtis. A turing machine simulator. *J. ACM*, 12(1):1–13, 1965. URL http://dblp.uni-trier.de/rec/bibtex/journals/jacm/Curtis65.

[28] Pinaki Chakraborty, P. C. Saxena, and C. P. Katti. A compiler-based toolkit to teach and learn finite automata. *Computer Applications in Engineering Education*, page n/a–n/a, 2010. ISSN 1099-0542. doi: 10.1002/cae.20492. URL http://onlinelibrary.wiley.com/doi/10.1002/cae.20492/abstract.

[29] Robert W. Coffin, Harry E. Goheen, and Walter R. Stahl. Simulation of a turing machine on a digital computer. In *Proceedings of the November 12-14, 1963, fall joint computer conference*, AFIPS '63 (Fall), page 35–43, New York, NY, USA, 1963. ACM. doi: 10.1145/1463822.1463827. URL http://doi.acm.org/10.1145/1463822.1463827.

[30] Lawrence L. Rose, Neil D. Jones, and Bruce H. Barnes. Automata: a teaching aid for mathematical machines. *SIGCSE Bull.*, 3(1):12–20, March 1971. ISSN 0097-8418. doi: 10.1145/873674.873676. URL http://doi.acm.org/10.1145/873674.873676.

[31] Romauld Jagielski. Visual simulation of finite state machines. *SIGCSE Bull.*, 20(4):38–40, December 1988. ISSN 0097-8418. doi: 10.1145/54138.54145. URL http://doi.acm.org/10.1145/54138.54145.

[32] Susan H. Rodger and Eric Gramond. JFLAP (poster): an aid to studying theorems in automata theory. In *Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education*, ITiCSE '98, page 302–, New York, NY, USA, 1998. ACM. ISBN 1-58113-000-7. doi: 10.1145/282991.283635. URL http://doi.acm.org/10.1145/282991.283635.

[33] Susan H. Rodger, Bart Bressler, Thomas Finley, and Stephen Reading. Turning automata theory into a hands-on course. *SIGCSE Bull.*, 38(1):379–383, March 2006. ISSN 0097-8418. doi: 10.1145/1124706.1121459. URL http://doi.acm.org/10.1145/1124706.1121459.

[34] Laura A. Sanchis. Computer laboratories for the theory of computing course. In *Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges*, CCSC '01, page 262–269, USA, 2001. Consortium for Computing Sciences in Colleges. URL http://dl.acm.org/citation.cfm?id=378593.378728.

[35] Michael T. Grinder. A preliminary empirical evaluation of the effectiveness of a finite state automaton animator. *SIGCSE Bull.*, 35(1):157–161, January 2003. ISSN 0097-8418. doi: 10.1145/792548.611958. URL http://doi.acm.org/10.1145/792548.611958.

[36] Joshua J. Cogliati, Frances W. Goosey, Michael T. Grinder, Bradley A. Pascoe, Rockford J. ROSS, and Cheston J. Williams. Realizing the promise of visualization in the theory of computing. *J. Educ. Resour. Comput.*, 5(2), June 2005. ISSN 1531-4278. doi: 10.1145/1141904.1141909. URL http://doi.acm.org/10.1145/1141904.1141909.

[37] Mohamed Hamada and Kazuhiko Shiina. A classroom experiment for teaching automata. *SIGCSE Bull.*, 36(3):261–261, June 2004. ISSN 0097-8418. doi: 10.1145/1026487.1008094. URL http://doi.acm.org/10.1145/1026487.1008094.

[38] Carlos I. Chesnevar, Maria L. Cobo, and William Yurcik. Using theoretical computer simulators for formal languages and automata theory. *SIGCSE Bull.*, 35(2):33–37, June 2003. ISSN 0097-8418. doi: 10.1145/782941.782975. URL http://doi.acm.org/10.1145/782941.782975.

[39] Breadth-first search, June 2013. URL https://en.wikipedia.org/w/index.php?title=Breadth-first_search&oldid=559361449. Page Version ID: 559361449.

[40] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959. ISSN 0018-8646. doi: 10.1147/rd.32.0114.

[41] Powerset construction, May 2013. URL http://en.wikipedia.org/w/index.php?title=Powerset_construction&oldid=547783241. Page Version ID: 547783241.

[42] Model–view–controller, June 2013. URL http://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=560711945. Page Version ID: 560711945.