KING'S College LONDON

University of London

6CCS3PRJ:
Push-down Automata Educational Tool

Author: Amin Mansour

Supervisor: Dr Agi Kurucz

Student ID: 1530020

April 14, 2018

Acknowledgements

First, and foremost i would like to thank my supervisor Agi Kurucz, who has helped me through this interesting and at times difficult process. The guidance and support she has shown me has allowed me to complete this project to the best of my ability.

I would also like to thank Michael Sipser, who has compiled his knowledge into the most comprehensive book written on computational theory. Lots of concepts mentioned in this report were taken from "Introduction to the Theory of Computation" [10] by Michael Sipser.

Originality Avowal

I confirm that I am the sole writer of this report, with the exception of explicitly referenced material from other sources.

Amin Mansour
April 14, 2018

Abstract

In the 21st century, there has seen an exponential surge in the number of students taking computer science. The need for educational tools which allow users to explore these ideas has increased. A pushdown-automata is one of those core computational concepts and it is integral to many fields found in computer science. In spite of that, there does not seem to exist an application which is able to simulate a push-down automaton in a clear and concise way. The aim of this project is to design and develop a system which can simulate a pushdown-automata and replicate the procedure of running input on it. This application must be both informative and beneficial to the user.

Contents

âĂŐâĂŐ

Chapter 1

Introduction

.

Push-down automata, or PDA for short, is one of the core technologies deployed in computational theory and computer science in general. The PDA allows us to theorize about machine computation. A PDA in simple terms is a finite automaton with an auxiliary stack known as a push-down. Anthony Oettinger first introduced PDAs as a model in 1961, and since then it has further been explored. It is widely used in many areas, particularly in compiler design and natural language processing [12, p. 210].

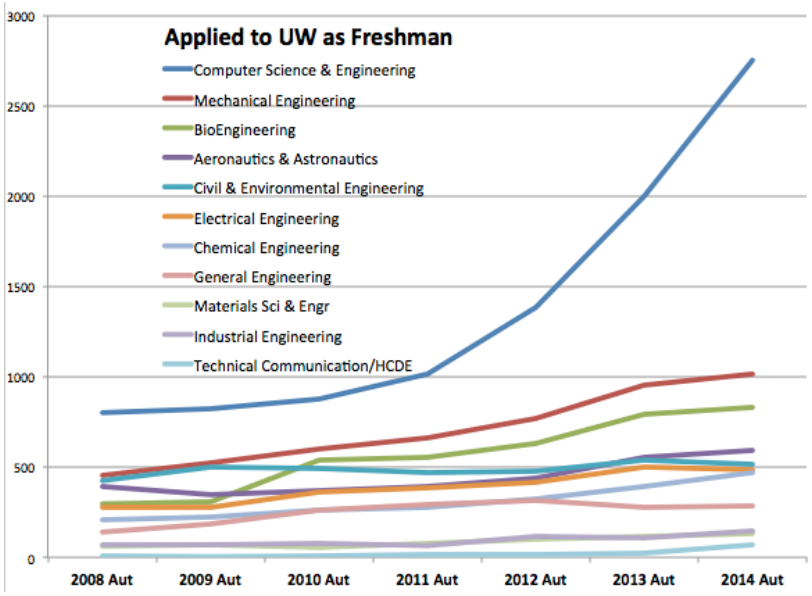


Figure 1.1: Graph showing number of students enrolling in each field at University of Washington per year [2]

The number of students who take computer science has surged each year for the last 10 years Fig 1.1. The need to produce educational simulations to represent these abstract theories has only increased. Such computational tools are an integral foundation to computer science and there is a whole section in Introduction to the Theory of Computation given to PDAs, highlighting their importance [10]. It Is also covered in a wide range of syllabuses from across different universities.

There exists many educational simulations for computational tools such as the Turing machine, or the Finite Automata. However, there does not exist any well-defined applications that can simulate a PDA successfully. There are several which are able to simulate a PDA, but they all come with their flaws and limitations. The objective of this project is to make the first education tool that can simulate both a non-deterministic PDA and a deterministic PDA with clear defined notations. A successful application that can be both used by lecturers, to teach with, and by students to learn from.

1.1 Objectives of project

The desktop application to be produced should be easy and clear to use for both students and lecturers' alike. A program which fulfils this objective should be able to allow the user to visualize what the PDA machine is doing at each transition for any given input and allow the user to go forward and backward in the computation easily. The program should also offer easy PDA manipulation and definition. The application must also conform to common PDA conventions, in cases where it is unanimously agreed upon i.e. using capital letters to represent push-down symbols. This application is primarily aimed towards students, so it must be familiar to them and their understanding of a PDA. There must also be ample amount of additional tools to aid students in their understanding of PDAs. The notations used must be intuitive, easily conveyed, and most importantly, correct.

1.1.1 Main Goals of application

- Very well-defined notations and concepts.
- Easy-to-understand display with clear markers for the user.
- Full flexibility offered, giving the user ability to go back one step, forward one step, or stop the computation all together.

- Allow for user to easily define their PDA machine, using either scripting or graphical representation.
- Add well-known examples that can be quickly loaded by the user.
- Find suitable ways of simulating non-deterministic PDAs .
- Allow for user to save their machines for later use.

1.2 Report Structure

This report will first cover the background behind this project. This first section will comprehensively cover all relevant underlying issues and then go on to look at some of the main alternatives to this project.

Next, the report will detail the requirements, the specification and the design of the proposed system. This chapter comprises of several schematic diagrams which illustrate structure and behaviour.

In the subsequent chapter, it will explain in detail how the proposed system was implemented. It will also go on to address the main problems faced during implementation and how these problems were mitigated.

The report will then examine how correctness was validated and then briefly explore some of the professional and ethical issues found in this project.

The report will then evaluate the outcomes achieved in this project, by looking at what the application did well and identifying the main limitations.

Finally, the concluding chapter will summarize what was achieved, and then explore some possible future works.

1.3 Motivation of Project

Computer science is about identifying a problem from the real world and coming up with a solution to solve that problem. One of the cornerstones of computer science is computational theory, which plays a large part in how we theorize about computational methods. PDA is one of the fundamental tools involved and has played a pivotal role in helping us understand rudimentary concepts. PDAs were first introduced at a time where hardware was still limited, and these computational tools were helpful in allowing us to decouple advance concepts from

high-level hardware. It was not up until the 1980s (The PC Boom) did the hardware start to catch up with the theory. However, the problem was that PDAs and computational tools like it were not able to evolve. They could not be experimented with in systematic ways and proved very tedious to teach to students.

This still plays true with today's hardware and since computers are now established, students find it very difficult to grasp the purpose of a PDA. Students are not able to interact with the mechanism directly, and can only access it through a set of examples. Today, we see several well-rounded educational simulations that attempt to make it easier for students to understand these concepts better. However, there is not a PDA education tool that has a clear and concise representation of the procedure with well-defined user functionality and examples; the key components needed to form a good educational simulator. The objective is to create an open-source PDA educational tool that rectifies these problems seen in other programs.

Chapter 2

Background

There is no certified and official understanding of PDAs; there are many varying explanations, with differences ranging from how the PDA is defined to how the PDA accepts input. This section will delve into those differences and explore the fundamentals of the PDA.

2.1 General Definition of the Push-Down Automaton

A push-down automaton, or PDA abbreviated, is a finite automaton (FA) with an additional memory device called a push-down. The machine can pop and push symbols to and from the stack.

2.1.1 How it works?

PDAs works mostly the same way as FAs, taking a string as input and for each iteration, reading the symbol at head. The difference is a symbol may also be popped or/and pushed from the PDA's stack for each transition. For every transition applied on a PDA, a new configuration occurs. The running of an arbitrary input ω on PDA α produces a sequence of configurations. A single one of those configurations is a formal snapshot of the PDA machine at that moment in the computation.

A PDA's configuration is defined as: (q,s,a)

- q : current state
- s : remaining input
- a : symbols on the stack

2.1.2 Power of a PDA

The additional stack that comes with a PDA gives it unbounded memory; whereas FA's memory is bounded by the number of finite states it has. This additional power allows for the PDA to recognize more languages than just the regular class of languages that FA recognizes. The PDA can also recognize the class of context-free languages (CFL). CFL are languages that can be generated with a context-free grammar. $\{0^k1^k|k \in N\}$ is an example of a context-free language that cannot be recognized by an FA but can easily be recognized by a PDA.

A stack only gives access to its top element at any one time, so the PDA is restricted in how it can access its illimitable memory. The turing machine (TM) is a more powerful alternative to the PDA. For a PDA, you cannot traverse down a stack without forgetting the rest of the stack. But with a TM, you can read and write in both directions on the input tape. This kind of functionally allows for the TM to be able to recognize more languages.

2.1.3 Transitions in a Push-Down Automata

Simply put, a transition represents a move from one configuration to the next; it may involve pushing and popping symbols to and from the stack, reading an input symbol and/or changing the current state. This is determined by the transition function, which maps a configuration, defined by the current state, the head input symbol and the top stack symbol, onto a set of possible actions. An action represents the pairing of the control state and the element found on top of the stack after the transition has occurred. For a non-deterministic PDA (NDPA), the resulting set can contain more than one action, therefore it must be the case that a NPDA must explore every possible action to determine whether a word is either accepted or rejected. In the case of a deterministic PDA, there is only at most one resulting action from the application of the transition function on any configuration.

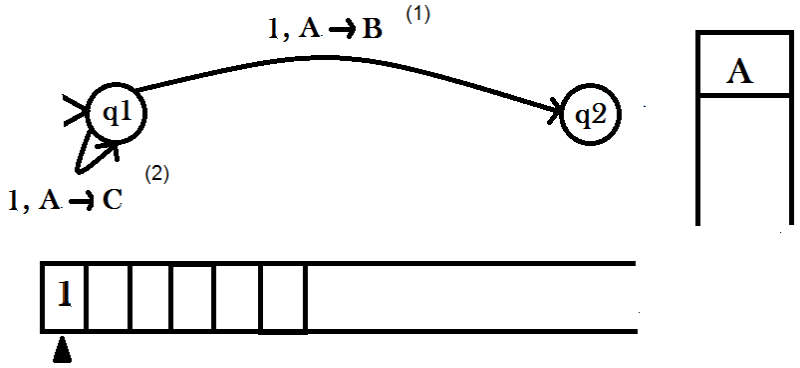


Figure 2.1: PDA example consisting of the two transitions labelled (1) and (2)

More formally, a transition represents the application of the transition function δ on the the current state Q, the current input symbol to be read Σ, and the symbol on top of the push-down Γ; the resulting configuration depends on these 3 variants. In a PDA, a transition is represented as $(s,o \rightarrow i)$: where s is the symbol at head, o is the stack symbol to be popped, and i is the stack symbol to be pushed. Take $(1,A \rightarrow B)$ as an example from 2.1: 1 is the symbol read from the tape, A is popped from the top of stack and B is pushed onto stack.

Non-determinism

Lets take 2.1 as an example to highlight non-determinism. The PDA is non-deterministic since the configuration (*q1,1,A*) results in a set of possible actions {(*q1,C*),(*q2,B*)} when the transition function δ is applied. In other words, the PDA machine has a choice between taking transition (1) and (2) from the current configuration (*q1,1,A*). This property makes the PDA non-deterministic.

Skipping symbols

Transitions can also contain skipping symbols ε; ε mean different things depending on where they come up in the transition:

- $(\varepsilon, \boldsymbol{o} \rightarrow \boldsymbol{i})$ - Machine does not read symbol at head on transition
- $(\boldsymbol{s}, \varepsilon \rightarrow \boldsymbol{i})$ - Machine does not pop the top symbol of stack on transition
- $(\boldsymbol{s}, \boldsymbol{o} \rightarrow \varepsilon)$ - Machine does not push any symbol onto the stack on transition

.

$(\varepsilon, \varepsilon \rightarrow \varepsilon)$ is a jumping transition that is equivalent to the ε-transition found in non-deterministic FAs. It neither reads input nor pushes and pops any symbols.

2.2 Defining a Push-Down Automata

As we discussed before, there exist several different formalizations of a PDA. The one we will use will be the one outlined in Introduction To Theory Of Computation [10, p. 111]. A PDA is a 6-tuple (Q,$\Sigma, \Gamma, \delta, q_0$,F), where Q,$\Sigma, \Gamma, \delta$,F are all finite sets:

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \rho(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the initial state, and
6. F$\subseteq Q$ is the set of accepting states.

For the transition function δ, $\rho(Q \times \Gamma_\varepsilon)$ represents the set of possible (Q $\times\Gamma_\varepsilon$) action pairs that can occur, from the application of δ on the domain $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$, where Q in the pair is the new state and Γ is the symbol pushed onto the stack. Σ_ε equals $\Sigma \cup \{\varepsilon\}$, so it is not a necessary condition for the input symbol to be read for a transition to occur, as was discussed earlier. The same holds for stack symbols being popped and pushed, with Γ_ε equaling $\Gamma \cup \{\varepsilon\}$.

2.2.1 PDA's accepting criteria

There are two different kinds of accepting conditions for PDAs : acceptance by final state and acceptance by empty stack. The two have been shown to be semantically similar. The first of those criteron, accepting by final state, formally put is when a machine terminates on an accepting state and there is no more input to read. In other words, if for a PDA run on word ω, the final terminating configuration contains a control state q, such that q is within the set of accepting states Q, and all symbols from ω have been read, then the ω is accepted by the PDA. If this condition is not met, then it follows, the word is rejected by the PDA.

The second criterion, accepting by empty stack, is defined as when the stack is empty and there is no more input symbols to be read. By the time the final input symbol is read, the push-down is checked to see if the stack is empty . If it is then the word is accepted, and if it is not, the word is rejected. In a accept-by-empty-stack PDA, all states are treated as accepting states; In other words, a PDA can terminate in any state and still accept a word, as long as the stack is empty and all the input has been read.

2.2.2 Are Both Accepting Conditions The Same?

These two offer the same amount of power to a PDA. Its very easy to convert an accept-by-empty-stack PDA to an accept-by-final-state PDA and vice versa. As we will see, diagrams 2.2 and 2.3 represent a easy and systematic way of converting between the two.

Conversion of an Accept-by-empty-stack PDA P_e to an equally powerful Accept-by-final-state PDA P_f

Fig 2.2 represents the process, with the original accept-by-empty-stack PDA P_e nested within the resulting accept-by-final-state PDA P_f. For the conversion, you must introduce a special stack symbol Z_0; this symbol is only used in the interest of representing an empty stack. It is added through the first transition (ϵ,$\epsilon \rightarrow Z_0$) (1) from the initial state q1 to the state e1 (the initial state in P_e). Then from all the states found in P_e, add the transition (ϵ,$Z_0 \rightarrow, \epsilon$) to the new accepting state q2; the transition pops Z_0 from the stack. The machine can now detect when the push-down is empty. By the time the final input symbol is read, the push-down is checked to see if the symbol exists at the top of the stack. If it does then the word is accepted, otherwise it is rejected.

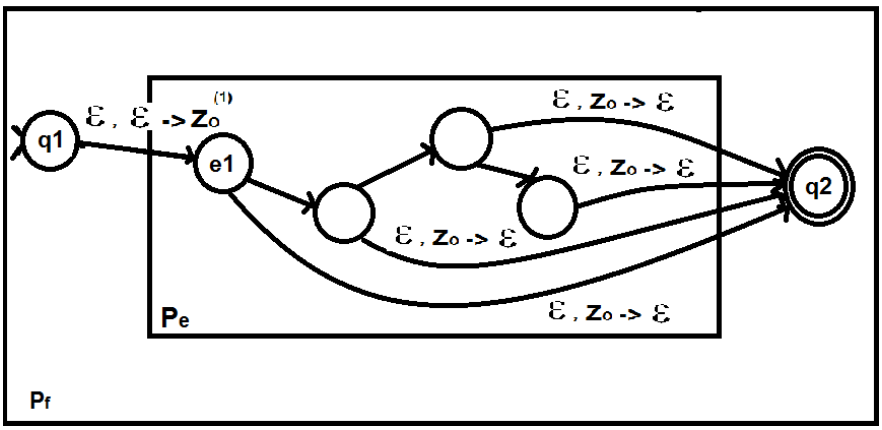


Figure 2.2: A illustration representing how a acceptance-by-empty-stack PDA is converted to an acceptance-by-final-state PDA with equivalent power.

Conversion of an Accept-by-final-state PDA P_f to an equally powerful Accept-by-empty-stack PDA P_e

As seen from 2.3, for each accepting state in the accept-by-final-state PDA P_f, you must add a transition to the state q2. From q2, all the content in the stack can be emptied (2) and the PDA can accept the input. So in a scenario where P_f terminates on an accepting state for a given input, P_e also terminates, but on the state q2 after emptying out the stack and accepting the input. The stack might be empty in a non-accepting state, which might lead to unintended words being accepted by P_e. So to prevent this a special symbol Z_0 can be pushed from the start via a transition from the initial state q1 to the state e1, the initial state in P_f.

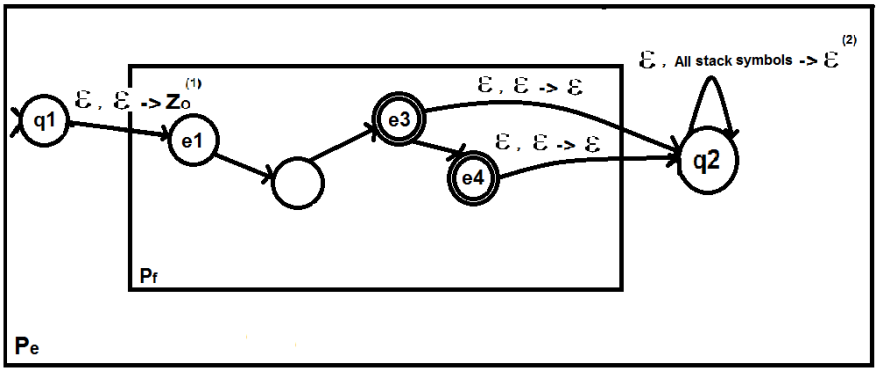


Figure 2.3: A illustration representing how a acceptance-by-final-state PDA is converted to an acceptance-by-empty-stack PDA with equivalent power.

2.3 Differences between NPDA and DPDA

The definition of PDA varies depending on if it is deterministic (DPDA) or non-deterministic (NPDA). The definition described above is the one used to describe NPDAs. The definition of the DPDA below is the same as the one described in Introduction to Automata Theory, Languages, and Computation [6]. With a DPDA, the transition function is described as: $\delta : Q \times \Sigma_\varepsilon \times \Gamma \to Q \times \Gamma_\varepsilon$. So in other words, the application of the transition function on any configuration, will always have at most one possible resulting configuration. For a PDA to be deterministic two conditions must hold:

- For any $q \in Q, a \in \Sigma \cup \{\varepsilon\}, x \in \Gamma \cup \{\varepsilon\}$, $\delta(q, a, x)$, has at most one element.
- For any $q \in Q, x \in \Gamma$, if $\delta(q, \varepsilon, x) \neq \emptyset$, then $\delta(q, a, x) = \emptyset$ for every $a \in \Sigma$.

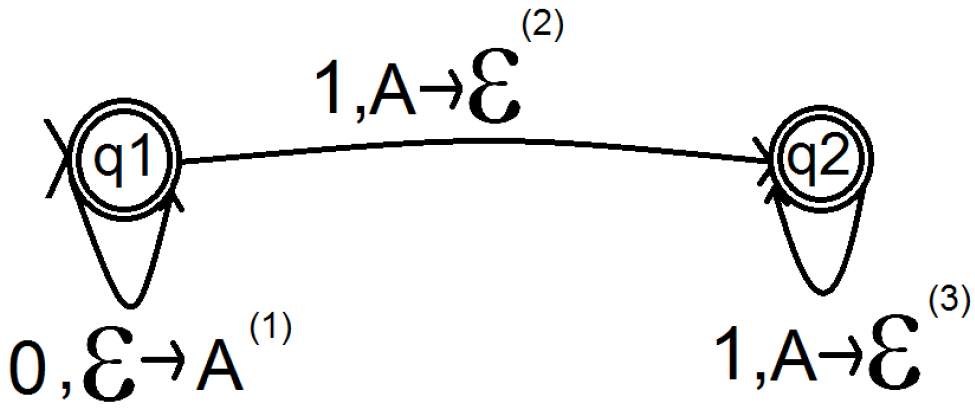
This is as defined by *John E. Hopcroft et al* [6, p. 247]

These added constraints make DPDA less powerful. DPDA can only recognize deterministic context-free languages, a proper subset of the class of context-free languages. A DPDA can easily be converted to an NPDA, since DPDAs are a subset of NDPAs. However, the reduced power of the DPDA means that an NPDA cannot always be transformed into a DPDA.

DPDA are still useful because they can be simulated efficiently with today's hardware, making them a feasible option to be used in parsers. For any computation on a DPDA, a total-order sequence of configurations is always produced. Unlike with an NDPA, which produces a computation tree instead. Thus, to determine whether a word is rejected for an NPDA, all the different paths on the computation tree must be explored. So in order for an NDPA to be simulated efficiently (polynomial time), there must be a feasible way to examine the different paths; however, this problem does not lie within the objectives of this application and has more to do with proving NP = P.

2.4 Walkthrough Example

The example PDA recognizes L = $\{0^k 1^k | k \in N\}$. This PDA accepts input via an empty stack.



Definition :

- Control States Q = {q1,q2}
- Input Alphabet Σ = {1,0}
- Stack Alphabet Γ = {A}
- Transition function:
  - $\delta(q1, 1, A) \rightarrow \{(q2, \epsilon)\}$ (2)
  - $\delta(q1, 0, \epsilon) \rightarrow \{(q1, A)\}$ (1)
  - $\delta(q2, 1, A) \rightarrow \{(q2, \epsilon)\}$ (3)
- Initial Stack Symbol Z_0 = Z
- Initial State q_0 = q1
- Accepting States F = {q1,q2} (accept-by-empty-stack)

Input runs :

In order for a word to be accepted by this machine there must be a sequence of configurations (q,ω,Z) $\vdash^*$ (q,ϵ,Z), where q is a arbitrary state and ω the input word.

- input 0011 - (q1,0011,Z),(q1,011,AZ),(q1,11,AAZ),(q2,1,AZ),(q2,ϵ,Z) : Accept
- input 01 - (q1,01,Z),(q1,1,AZ),(q2,ϵ,Z) : Accept

- input 010 - (q1,010,Z),(q1,10,AZ).(q2,0,Z), stuck! : Reject

Stuck - represents the occurrence where no possible transitions can be made from the current configuration; the word is rejected in this case.

Deterministic?

This PDA is deterministic. The application of the transition function on any configuration only ever results in at most one action. So, the sequence of configurations follow a linear path.

A non-deterministic alternative

A trivial example of a non-deterministic PDA which recognizes the same language is below. We will use it to highlight the non-deterministic property.

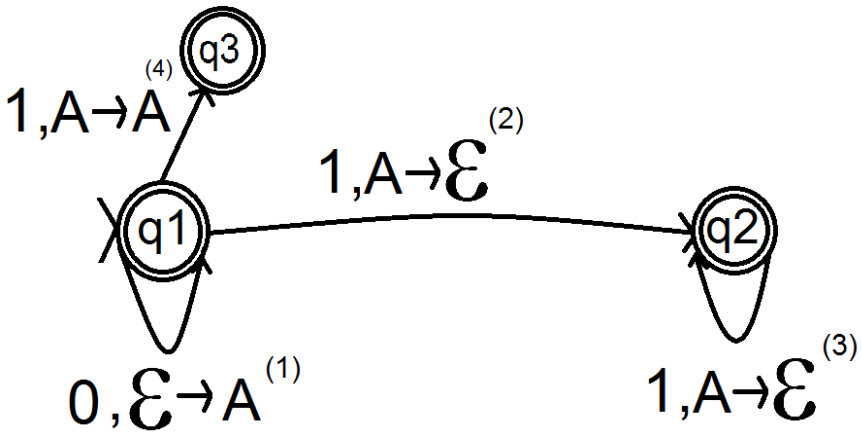


Figure 2.4: A non-deterministic PDA which recognizes the language $\{0^k1^k | k \in N\}$.

The application of the transition function δ on configuration (q1,0,A) results in the action set $\{(q3,A),(q2,\epsilon)\}$; this mapping invalidates the deterministic property and makes the PDA non-deterministic. The computation must explore both actions to determine whether the input is accepted or rejected. In the case of accepting, you keep exploring alternative action branches until you reach a accepting configuration. A clear example of this is shown :

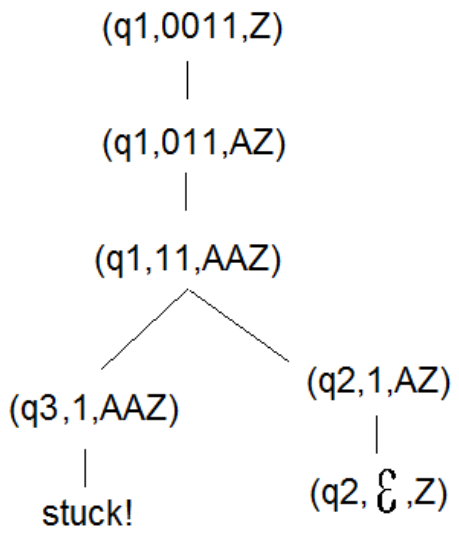


Figure 2.5: The computation path produced from the running of 0011 on the non-determinstic PDA 2.4.

More definitively, for a word to be accepted there must be at least one path which starts from the root/initial configuration and terminates on a accepting configuration. In the computation tree there can exist several different paths which may be taken. In order for a word to be rejected all possible paths must be explored to determine that the terminating configuration is not an accepting one. Take the above as an example, there are two paths which can be taken. The path consisting of the right branch sub-tree contains an accepting configuration as it's final configuration, so the word 0011 is accepted.

2.5 Main Implications of Theory on Developing this System

As was discussed, there are several textbooks that cover PDAs in-depth, all with variations between them; they differ in a range of things from how they define a PDA to the accepting conditions they layout. This project has been heavily influenced by "Introduction to the Theory of Computation" [10], which defines the rudimentary understanding of PDAs. There has to be a decision on what notations to use since the user will require this to produce their own PDAs.

There also must be clever ways of simulating a non-deterministic PDA (NPDA). There must be a mechanism to determine what branch to take for instances where several possible transitions can be taken (a configuration which has a set of possible actions); preferably chosen by the user themselves. The PDA must also be able to backtrack backwards to a previous configuration, so an alternative branch can be explored instead, in cases where the current branch has failed (depth-first search).

As was discussed before, for a word to be rejected each of the possible branches within the computation tree must be explored. For a large machine, this can be a computationally expensive process, so there must be some kind of option for the user to cancel the computation. Also, the system must be able to detect when the PDA is stuck in a loop and take the appropriate action i.e. terminate the machine and reject input.

2.6 Analysis of competitors

There are many PDA simulators that are available online. Some with obvious advantages over others. In this section we will explore some of the main ones.

Key Feature	JFLAP	Cburch Automaton Simulator	Kyle Dickerson's Automaton Simulator
Representation of determinism	Both Non-deterministic and Deterministic PDAs	Only Deterministic PDAs only	Both Non-deterministic and Deterministic PDAs
Ways of defining a PDA	Graphical	Graphical	Graphical
Representation of pushdown stack	True	False	False
Step by step representation of computation	True	True	False
Examples	No	No	Yes

Figure 2.6: Analysis of the three main applications JFLAP [11], Cburch [9], and Kyle Dickerson's Automaton Simulator [5].

JFLAP

The main one of those is JFLAP, an application available to download on the web. It can simulate several other computational tools, alongside the PDA. However as seen, there are major faults with the design of it. This application does many things well, like offering the user the ability to graphically define and manipulate their own PDA machine through the use of visual components. This method proves useful for less experienced students.

This application tries to do a lot, and this can excuse it from some of its minor faults, like the lack of examples offered. However, some of its faults are more consequential and cannot be avoided. The user ca not manually define their PDA using standard notation. It instead forces the user to define it graphically. Not only does this restriction make it tedious to create complicated PDAs, but it also abstracts the machine away from the theory, no longer making

the application a replication of a computational concept. The PDA simulator should allow for the user to define a PDA manually using scripting, specifying each of the required elements in the PDA's definition explicitly. That way the student is more familiarized with each of the components and what role each plays.

Another issue with JFLAP is the ambiguity on how the PDA machine works and the notations used. PDA is a theoretical concept, and must come with a consistent and defined way of working, so the user can reason with the application effectively. Although different computational books vary in definition, they do not assume that the reader is aware with how the machine works before going into examples. In other words, the notations are clearly defined for the reader beforehand. This is not the case with JFLAP, which can prove especially confusing for students who were taught differently.

An example of this comes when Z, the empty-stack symbol, is automatically added to the push-down just before every computation. The Z in this instance represents the initial stack symbol. Although Sipser's definition does not include defining an initial stack symbol, other definitions do [6, p. 221]. However, JFLAP reduces the initial stack symbol to some constant Z, instead of allowing the user to define it themselves. This approach is very alien with respect to the general theory behind PDAs. It is also very difficult for the user to recognize that the initial stack symbol is Z, since the application does not show the stack before computation.

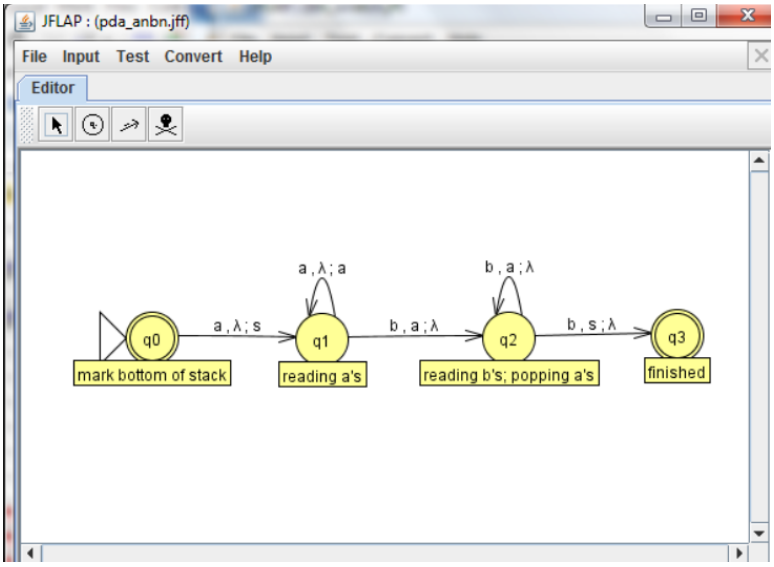


Figure 2.7: A screen shot of JFLAP [11].

Cburch's automaton simulator

Cburch's automaton simulator (CAS) [9] is a Java desktop application. It has some negatives which make running input difficult. The most apparent drawback is in the limitation for a user to create a non-deterministic PDA machine. On top of that, the stack and input alphabets are comprised of only 4 elements (a,b,c,d) and the two alphabets are always equivalent. Both of these constraints contribute to limiting the complexity of PDA that is able to be simulated by the application. It also does not show a visual stack during execution either, which only makes the execution procedure even more confusing. All in all, CAS does not replicate the mechanisms of a PDA very clearly.

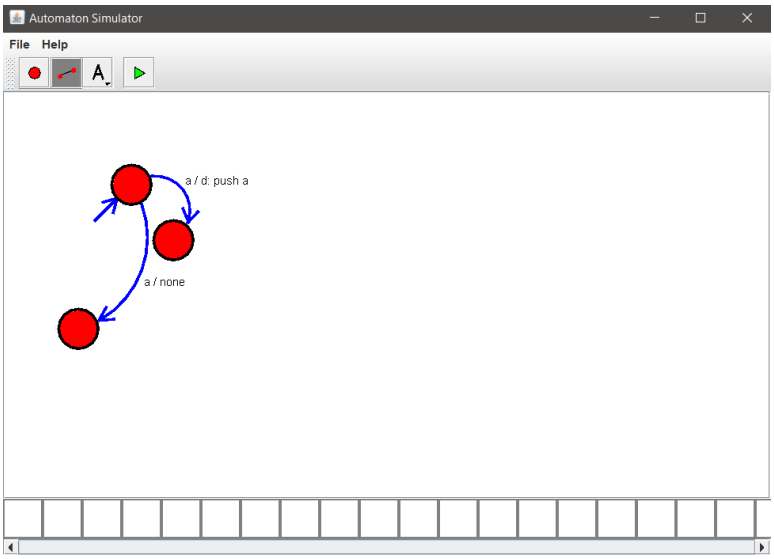


Figure 2.8: A screen shot of Cburch's automaton simulator [9].

Kyle Dickerson's automaton simulator

Kyle Dickerson's automaton simulator [5] is a modern web application. It is very easy for the user to create their own PDA using the drag-and-drop feature. There is also an extensive library of examples to choose from, which is especially useful for students.

However, this application is limited in what it can do. There is no step-by-step walkthrough of the execution procedure. This reduces the application's usefulness. Especially for a student that may be using the system to gain a better understanding of the mechanisms involved. The application does not have to be a stand-alone system that only determines recognizability. It should also show how the output is derived.

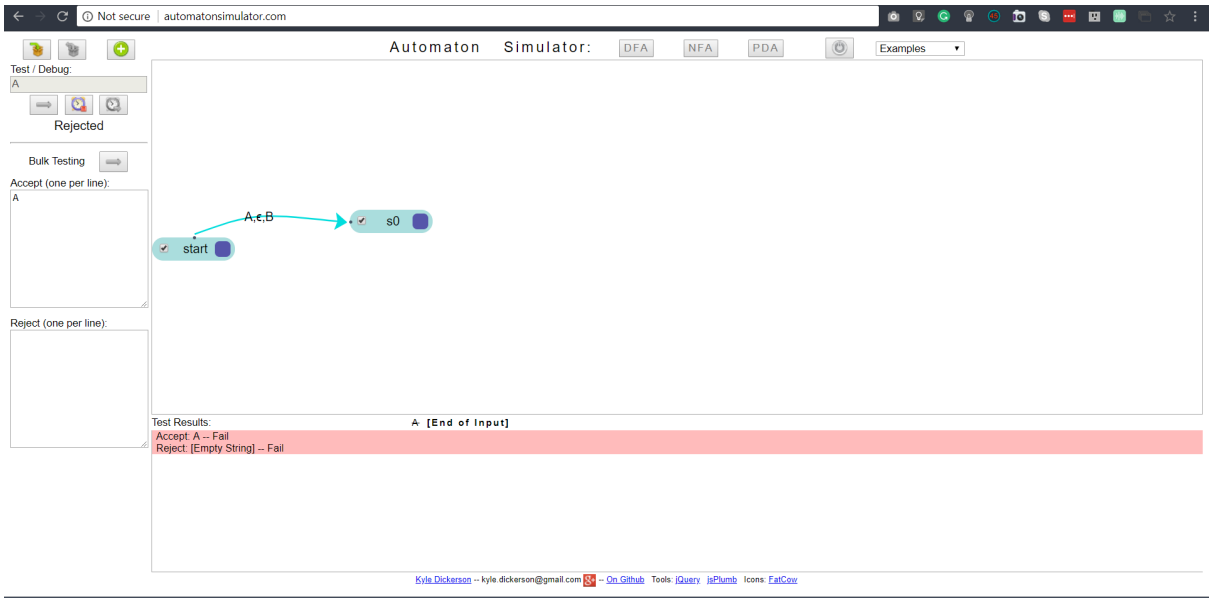


Figure 2.9: A screen shot of Kyle Dickerson's automaton simulator [5].

Chapter 3

Specification and Design

3.0.1 Brief

To design and develop an animated visual of a PDA, which can be used by students to simulate a PDA. The GUI must be easy for beginners, and there must be plenty of examples available for the user to simulate. The application must provide an easy way for the user to define their own PDA machines and run input on them.

3.0.2 Requirement analysis

Requirement analysis identifies the requirements of the application and catalogues them into a list. It is comprised of both functional and non-function requirements. Functional requirements represent the behaviour in a system and non-functional requirements are quantifiable attributes of a system. Identifying all requirements before implementation can help to prevent any development-related problems.

The requirements analysis can be broken down to two parts. The first part is requirement elicitation: identify and gather information on the domain by analysing competitors and reading literature on PDAs, and then convert that information into an exhaustive list of requirements. The second part is evaluating each of those requirements, by identifying conflicts, imprecision or redundancies found in them, to ensure for a correct, complete and consistent list.

This application's objectives are very specific, which makes this application very functional-orientated. There is not any particular focus on any non-functional measures, hence why the list for non-functional requirements is very small.

The functional requirements have been clustered into 6 main groups. Each vaguely represents

a different aspect of the application. For each requirement, it's priority has been categorized as either being high,medium or low. This metric informs the order of implementation.

Functional Requirements (FR)

1. PDA creation/loading
   (a) The user will be able to define their own PDA through visual components. (medium priority)
   (b) The user will be able to select a pre-defined PDA definition from a collection of examples. (high priority)
   (c) The user will be able to define their own PDA through a well-laid out script. (high priority)
   (d) The user will be able to load a previously saved PDA from memory. (high priority)
2. PDA application functionality
   (a) The user will be able to clear the current PDA machine and start again. (high priority)
   (b) The user will be able to save a PDA machine to memory. (high priority)
   (c) The user will be able to access help in the application. (medium priority)
   (d) The user will be able to easily add, remove and modify states and transitions after the PDA has been loaded. (high priority)
   (e) The user will be able to open non-deterministic mode, where they can view all the non-deterministic transitions in the currently loaded PDA machine. (medium priority)
3. PDA graphical manipulation
   (a) The user will able to move the visual control states around a defined region once the PDA instance has loaded (medium priority)
   (b) The user will be able to drag and drop new states (low priority)
   (c) The user will be able to add new transitions by dragging an arrow between two states (low priority)
4. PDA definition

(a) The user will be allowed to specify whether their PDA accepts inputs based on an empty stack or accepting state. (high priority)

(b) The user will be able to define the amount control states the desired machine will have. (high priority)

(c) The user will be able to define the transitions that the desired machine will have. (high priority)

(d) The user will be able to define the input and stack alphabet. (low priority)

(e) The user will be able to define the accepting states. (high priority)

(f) The user will be able to define the initial state. (high priority)

5. PDA running functionality

(a) The user will be able to run input on the current loaded PDA machine and retrieve the correct output immediately. (high priority)

(b) The user will be able to view and traverse step-by-step the computation for any given input. (high priority)

6. PDA simulating functionality

(a) The user will be able to step forward in the computation. (high priority)

(b) The user will be able restore the PDA to the previous configuration. (high priority)

(c) The user will be able to terminate computation in any step. (high priority)

(d) The user will be able to select what transition to execute from a given configuration in cases where there is more than one action to choose from (found in NPDAs). (high priority)

(e) The user will be able to go back to previous computation-branching or go forward to the next computation-branching(case where the transition function maps to several different actions and requires user delegation). (medium priority)

(f) The user will be notified of the current state of the stack and tape, along with the current state of the PDA in every step of the computation. (high priority)

(g) A transition table will be shown throughout run with current configuration and the associated action clearly represented in as a row. (high priority)

Non-functional Requirements (NFR)

1. Help must be offered to user at any location in application.
2. The application must be able to find a solution (an execution path with a terminating accepting configuration) if it actually exists.
3. The representation of the computation must be clear and easy to understand for the user.
4. The application must not be online-dependant nor platform-dependant.
5. The GUI must be user-friendly with friendly colours and descriptive icons.
6. There must not be any unhandled exceptions from any improper user input.
7. The theory of the PDA described in the project must be correctly translated into the application.

3.1 Specification

In this section, we build and describe a UML class diagram and a state machine. Both have their own purposes in representing different concepts with respect to the application.

3.1.1 Approach to Specification

In this section, we will be engineering a new system by representing the structural and behavioural properties through the use of models. It is a systematic and quantifiable approach to the development of software. All understanding of models shown comes from Kevin Lano's "Formal Object-Oriented Development" [8].

3.1.2 Specification of the Requirements

Some requirements are self-explanatory. Only the non-specific requirements have been covered below.

Requirement	Specification	Priority
(1a) The user will be able to define their own PDA through visual components.	This should allow the user to define their desired PDA through graphical manipulation. Features : Allowing the adding of states via drag and drop. Allow the adding of transitions by dragging across visual states.	Low

(1b) The user will be able to select a predefined PDA definition from a collection of examples.	This should offer the user the ability to simulate a set of PDA machines. These example PDAs should be broad-ranging and explore the power that PDAs have to offer.	High
(1c) The user will be able to define their own PDA through a well-laid out script.	This should be a systematic and interactive form which allows the user to construct their desired PDA. This form must be validated before the construction of the PDA is possible.	High
(1d) The user will be able to load a previously saved PDA from memory.	This should offer the user the ability to load their PDAs within memory. More specifically, the Definition instance itself. The Definition instances should be offered to the user in a clear way (i.e. list). This should also allow the user to delete their PDAs if that is what they require.	High
(2a) The user will be able to clear the current PDA machine and start again.	This should allow the user to remove the currently loaded PDA machine and reload another PDA of their choosing. This process should be both easy and fast, to allow the application to be robust	High
(2b) The user will be able to save a PDA machine to memory.	This should allow the user to be able to save their created PDAs to memory. More specifically, the Definition instance itself.	High
(2c) The user will be able to access help in the application.	This should allow the user to be able to access help on how to use the main features in this application i.e. step-run. The help feature should be accessible from all pages.	Medium
(2d) The user will be able to easily add, remove and modify states and transitions after the PDA has been loaded.	This should allow the user to continue modification of a PDA after it has been created. These modifications include adding and changing transitions.	High

(2e) The user will be able to open non-deterministic mode, where they can view all the non-deterministic transitions in the currently loaded PDA machine.	This should allow the user to view all non-deterministic transitions in the currently loaded PDA. In the case where there are no non-deterministic transitions (deterministic PDA), then there must be a clear way of conveying that to the user.	Medium
(3a) The user will able to move the visual control states around a defined region.	allow the user to move visual control states of the PDA around the visual display region, to get a better visual arrangement of states.	Medium
(3b) The user will be able to drag and drop new states.	This should allow the user to add new states to the currently loaded PDA via the drag and drop feature, as seen in JFLAP.	Low
(3c) The user will be able to add new transitions by dragging an arrow between two states.	This should allow the user to add transitions to the currently loaded PDA via the dragging of arrows across visual control states.	Low
(4a) The user will be allowed to specify whether their PDA accepts inputs based on an empty stack or accepting state.	This should be offered to the user in the form of a radio button since it is a binary choice that needs to be made by the user.	High
(5a) The user will be able to run input on the current loaded PDA machine and retrieve the correct output immediately.	This should allow the running of input specified by the user on a PDA and successfully be able to determine whether that PDA accepts of rejects the word. This requirement specifically pertains to being able to evaluate input, in contrast to 5b.	High

(5b) The user will be able to view and traverse step-by-step the computation for any given input.	This should allow the user to go through each configuration in the execution while being able to view the components of the PDA. The user should have complete control of the execution process deciding whether to move forward or backward. The requirements 6a, 6b, 6c, 6d , 6e and 6f outline the functionality that should be offered to the user.	High

3.1.3 Class Diagram

Class diagrams are conceptual models that are used to represent structure. They define entities of the system and the relationships between them.

The class diagram 3.1 represents the structure of the core application. Only model entities are represented in 3.1 since all business logic will be contained within the model. The view is not important at this stage and more generally this application, so, has been excluded from the class diagram. Also, several getters and setters have been removed from entities in cases where they are not important, to keep the class diagram concise and simple.

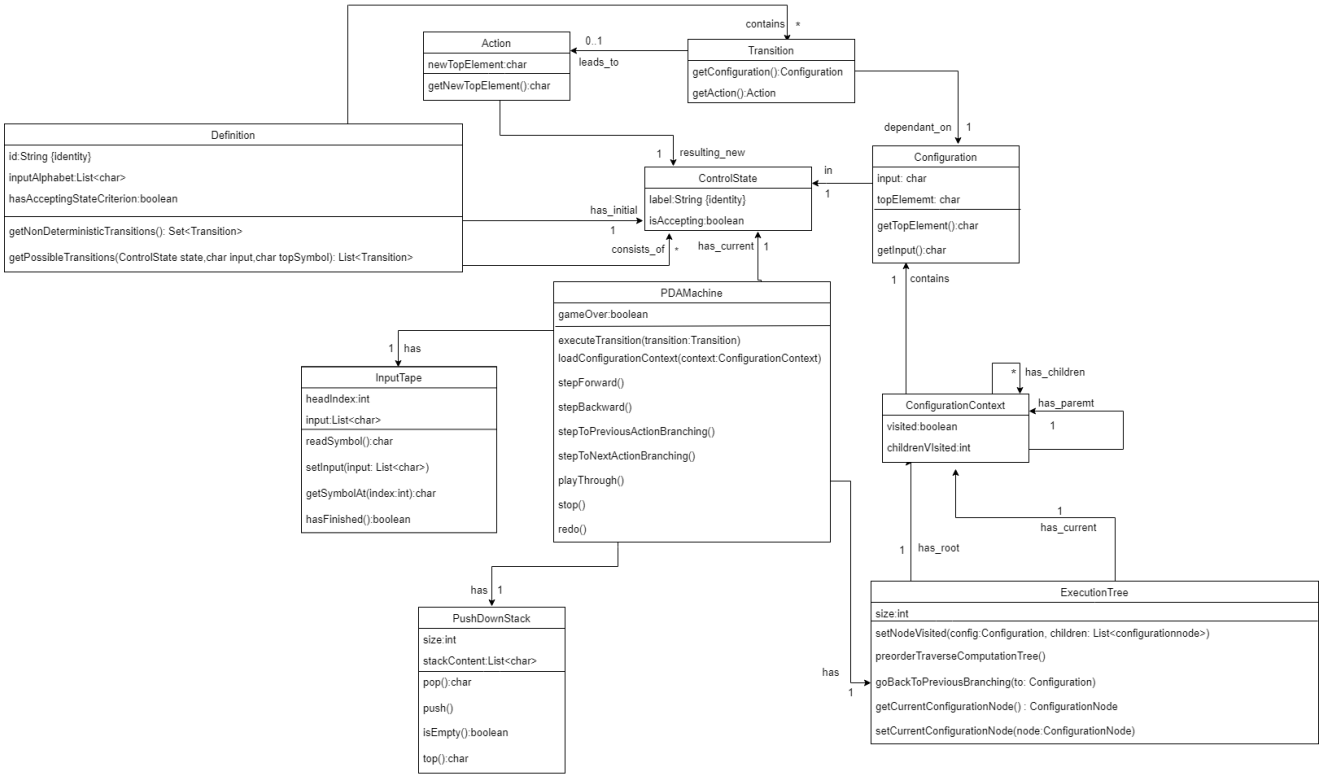


Figure 3.1: A class diagram to represent the classes that will be produced and how each of those interact with each other.

3.1.4 Explanation

The ExecutionTree entity is responsible for keeping a track of all previous configurations in the current execution. Its purpose is to enable backtracking within the execution, and also detect already visited configurations in the execution path. It is a tree data structure which accounts for several ConfigurationContext objects.

Each ConfigurationContext holds a configuration snapshot of the PDAMachine and this snapshot can arbitrarily be loaded into the PDAMachine. It also has a pointer to its parent ConfigurationContext instance and its many children.

Transition entity is equivalent to the one found in PDA theory. It represents a one-to-one mapping from a Configuration object to an Action object. The Definition object defines the transitions. A Definition object represents the structure of a PDAMachine instance. The PDAMachine object holds no more than one Definition object at any one time. PDAMachine is in charge of simulating the Definition instance with input. It encapsulates the additional components required to do this (i.e. PushDownStack and InputTape).

The unique id field in the Definition entity allows for instances of definitions to be distinguished from each other, important for when the user is loading, saving or modifying definitions.

PDAController

This application might possibly require a global controller to arbitrarily load PDA machines. The PDAController represents the functional PDA machine and acts as the gateway between the user and the PDAMachine, facilitating the interaction between the two. It is also responsible for controlling the visual components found in a PDA, such as the visual stack, and keeping them in sync with the model components. The controller has a set of functions which will allow the user to interact with the PDA machine, such as next(), which moves the PDA to the next configuration in the computation.

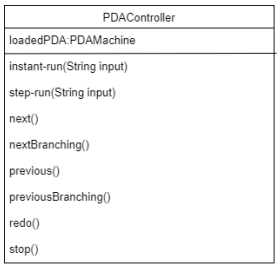


Figure 3.2: The PDAController entity.

3.1.5 State Machine

State machines are used to define object behaviour, through the representation of life cycles; it consists of states and transitions. State machine 3.3 represents the life cycle of a PDA instance, from where it is first defined to when it is used to simulate a input word. This state machine address the two main objectives: the process of creating the PDA and the process of running input.

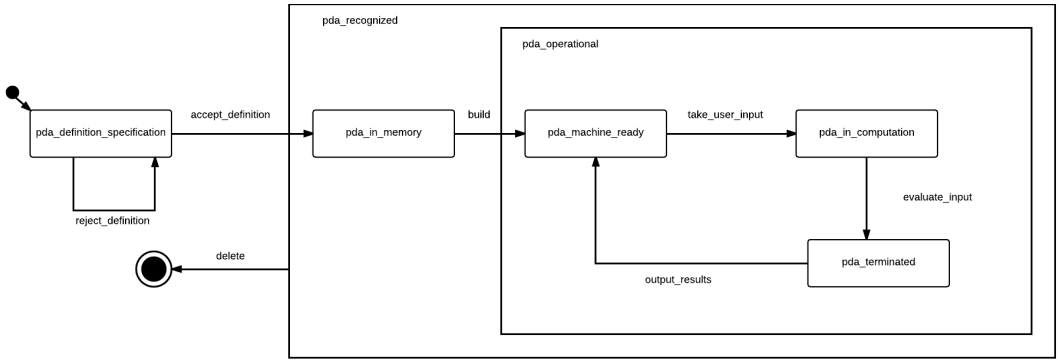


Figure 3.3: A state machine to represent the life-cycle of a PDA

Explanation

This is a walk-through explanation of the state machine 3.3. Initially, the user defines their desired PDA. The application then assesses the correctness of this definition, either accepting or rejecting it. In the case where the system accepts, the PDA's definition is saved to memory. Its now recognized by the system and can be loaded. When the PDA is requested, its definition is retrieved from memory and loaded onto the system. The definition is what is required to build the PDA. Once the PDA has fully been built, it becomes fully operational. At this state, the PDA is able to be simulated with input the user specifies. Once the PDA finishes evaluating an input, the results are outputted to the user and the PDA machine terminates. As long as the PDA is operational, the user can continue to input words into the PDA (represented from the loop found in the pda_operation state). Once saved, it can be deleted from memory at any time.

3.2 Design and Architecture

3.2.1 3-tier Architecture Diagram

Architecture diagram specify the key components in a system and define how they interact. Figure 1 shows the architecture diagram for this Application. The application is divided into 3 main components: the application tier, which defines the user-interface; the logic tier, which defines the functional core; and the data tier, which defines the mechanisms for which data can be retrieved and manipulated. The diagram has only one entry point, since there is only one actor that uses this system, the user. There are 2 essential groups of operations (modules) in this application. They can be categorized as the PDA's general operations, which pertains to anything involving PDA manipulation and creation, and the PDAs run operations, which covers the functionality that is offered to the user when running a input. These two modules form the business logic of the application.

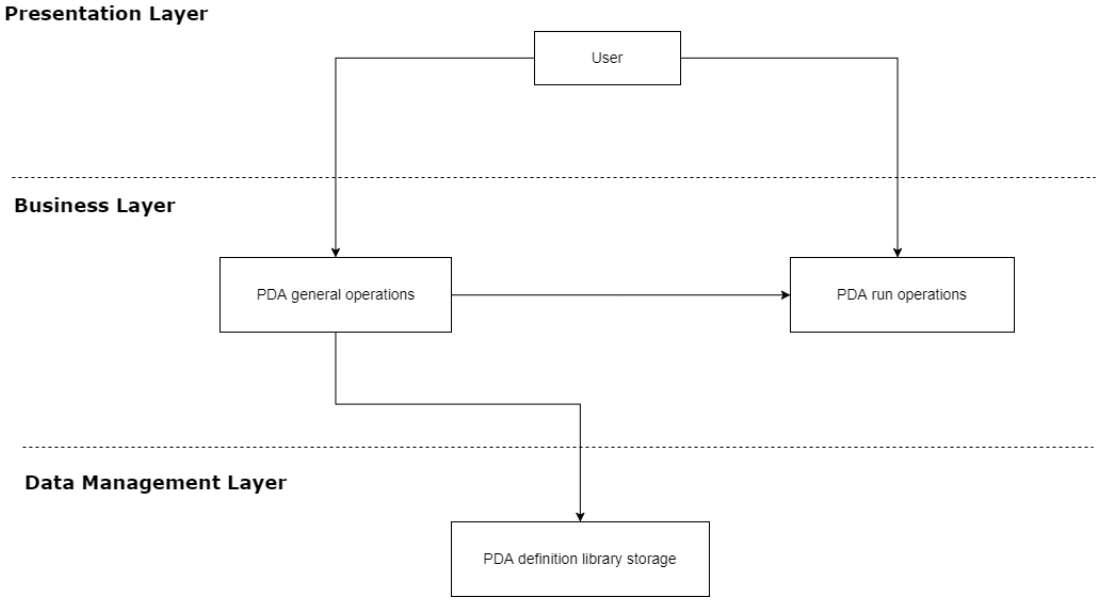


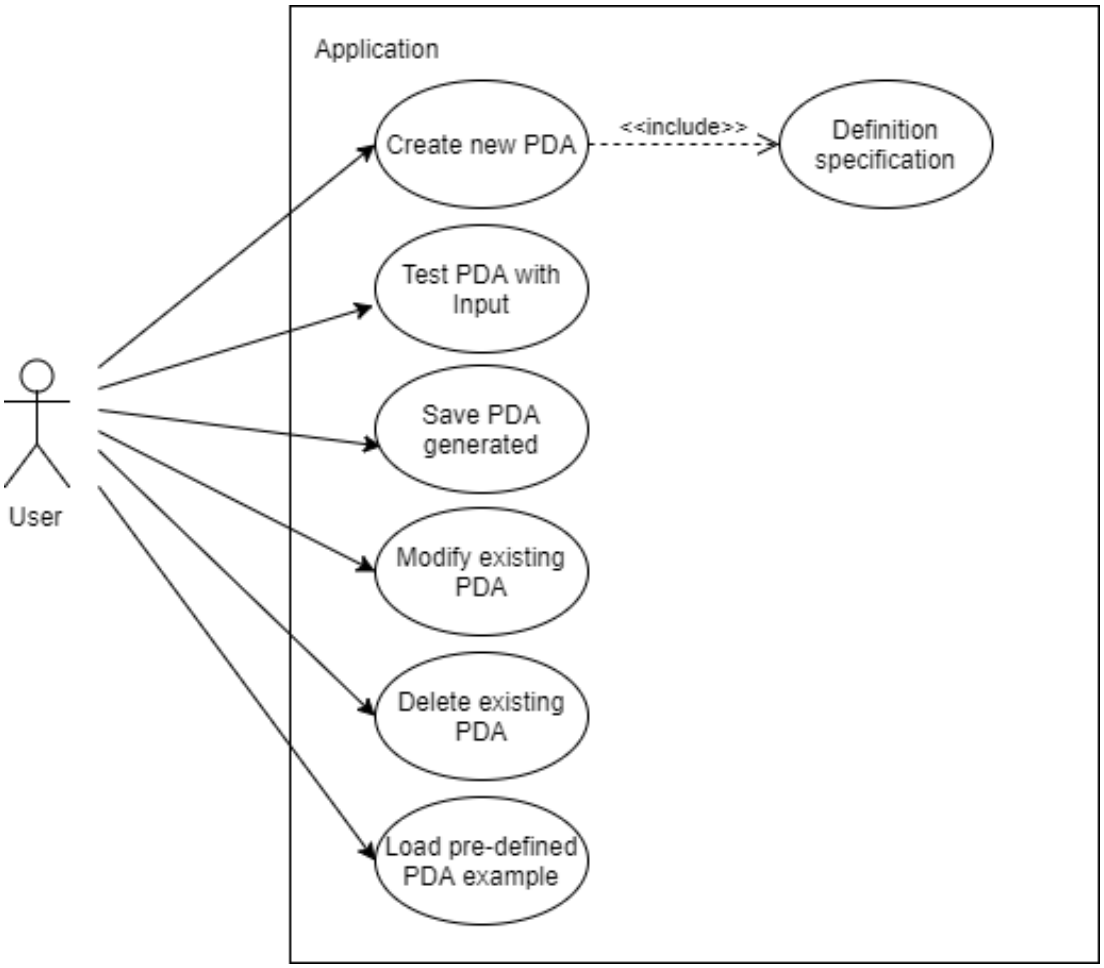
Figure 3.4: A 3-tier architecture diagram

3.2.2 Use Cases

As stated in the architecture diagram above, there are two main modules in the logic tier: PDA general operations (covers creation,modification and general interaction) and PDA run operations (the functionality offered to the user when running input). The use cases below highlight the functionality of each.

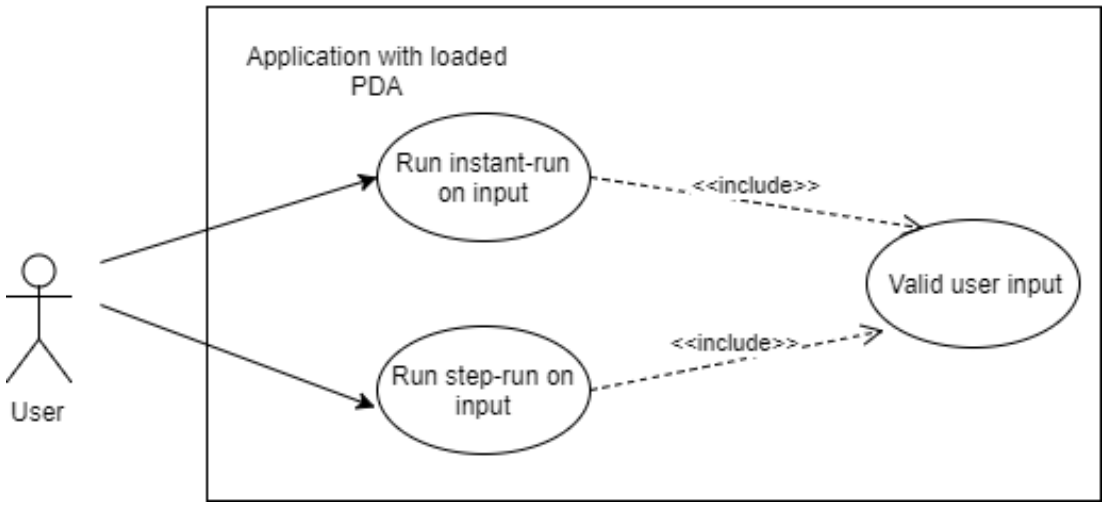
PDA General Operations Use Case

A use case that covers the creation,modification and the general interaction with the PDA, as seen by the user.



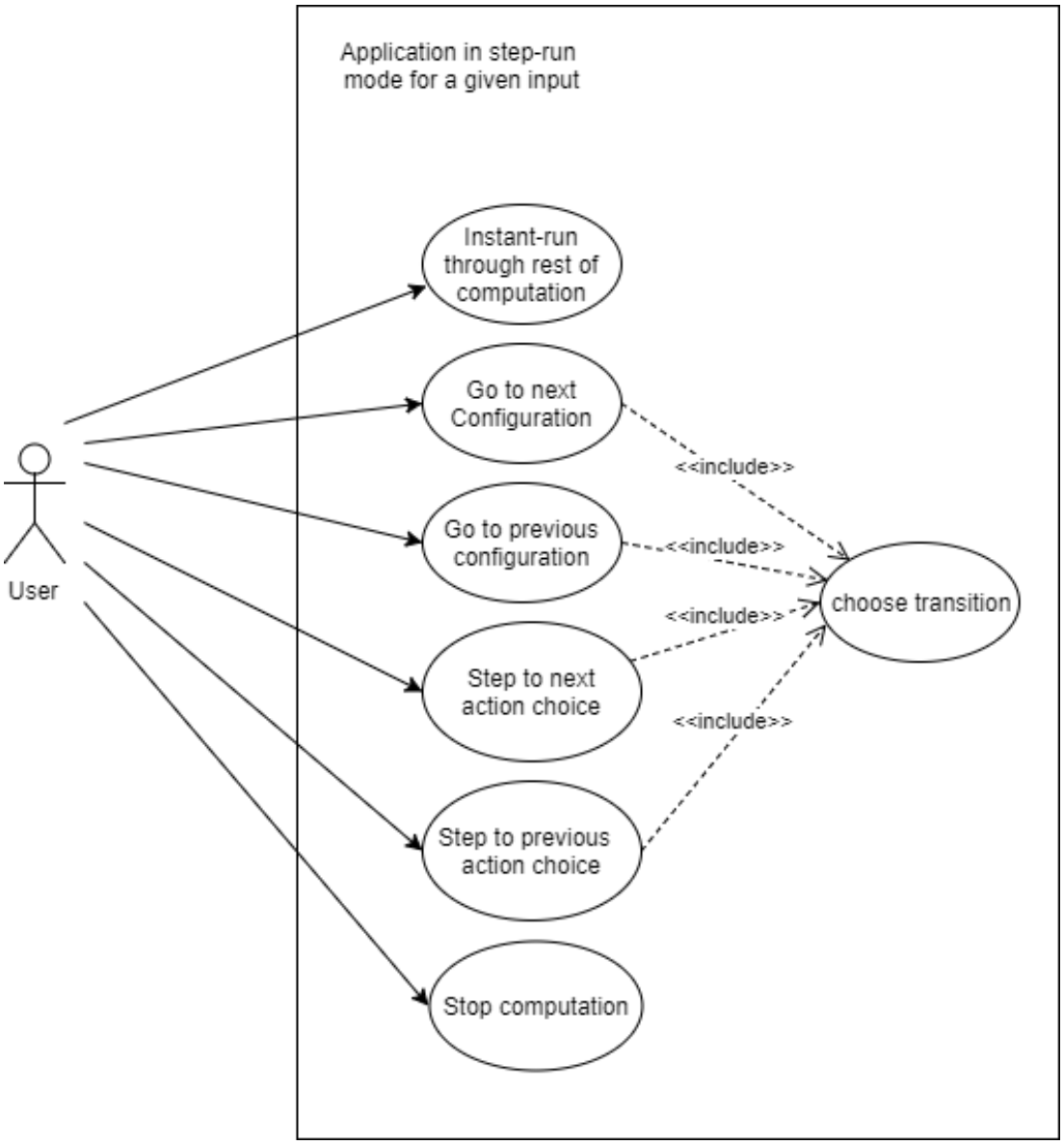
PDA Run Operations Use Case

There are two modes in which a user can run input in. The first of which is instant-run. Instant-run looks for a solution by exhaustively searching all possible branches. When a solution has been found it is returned to the user. If all branches have been searched and no solution is found then the results is shown to the user. The other is step-run which is a guided step-by-step walk-through of a run. They both require valid user input. The main difference between the two is instant-run is immediate and computer-controlled, and step-run is spread out and requires user interaction for progression. This use case specifically addresses the requirements *5a* and *5b*.



PDA Step-Run Use Case

Step-run is a step-by-step walk-through of a input run, where the user can go forward or backward in the computation, and also be able to select what configuration to explore next in cases where there are several choices(non-determinism).



3.2.3 Design Patterns

A design pattern is a reusable solution that can be applied in development to solve a reoccurring problem. The application has been modelled around the Model-View-Controller (MVC) design pattern, which separates the controller, the user interface and the data classes from each other. There are strict protocols on how these different components communicate. The model can not access the view directly and vice versa. However, the controller can access both the model and view and must mediate between the two; the controller provides the functionality to the actor of the system. Diagram 3.5 perfectly highlights this process.

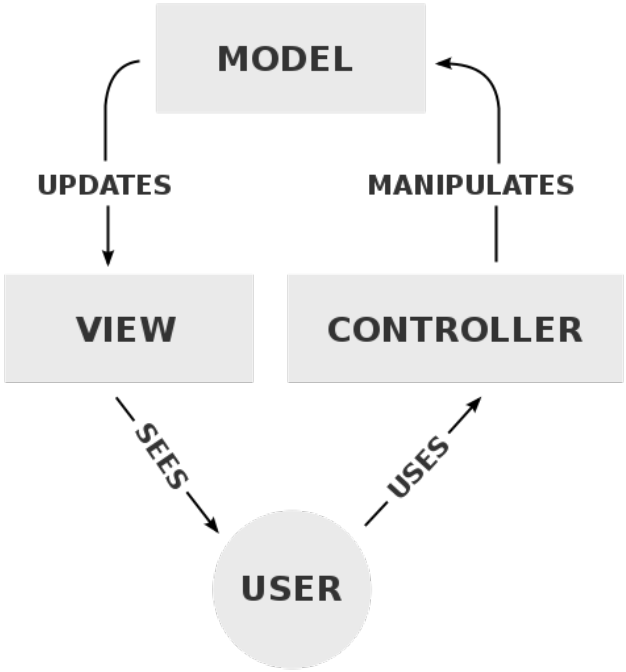


Figure 3.5: A diagram representing the interactions in the Model-View-Controller pattern [1].

MVC separates code nicely into their respected components, resulting in much-improved code quality. From the class diagram outlined in 3.1, MVC is ideal since we have a central PDAController which gives functionality to the user and several model classes which the PDAController manipulates and retrieves data from.

3.2.4 Data Structures

Lists and hashmaps, the standard data structures, are used across this application. As specified in the class diagram, there will also be a new custom data structure ExecutionTree, which will extend from the standard tree data structure. It will hold a root ConfiguationContext and also

keep a track of the current ConfigurationContext in the execution. Each ConfigurationContext holds a single configuration object, and a pointer to its parent ConfigurationNode and its many children (a list of ConfigurationContext instances). This data structure is used to keep a track of the previous computations in the run.

The PDA itself can also be seen as a data structure. It holds and organizes data in a specific way, allowing us to extrapolate meaningful information from it.

3.2.5 Interface Design

Green, white and grey are the three main colours across the application. Grey and white to give the application its formal look. Green was principally chosen because it is a colour commonly associated with education and learning. The UI is made to be as appealing as possible. The features are kept spruce to make the application purposely simple for the user. These are early stage mock-ups which demonstrate the three main screens across the application. The first is the home screen, which defines ways in which a user can load a PDA into the application. The second is the PDA definition screen, which will allow for users to create their own PDA. The third is the PDA graphic screen, which is where the user can interact with the PDA. These mock-ups were used to inform the implementation phase of the application.

Figure 3.6: Home Screen

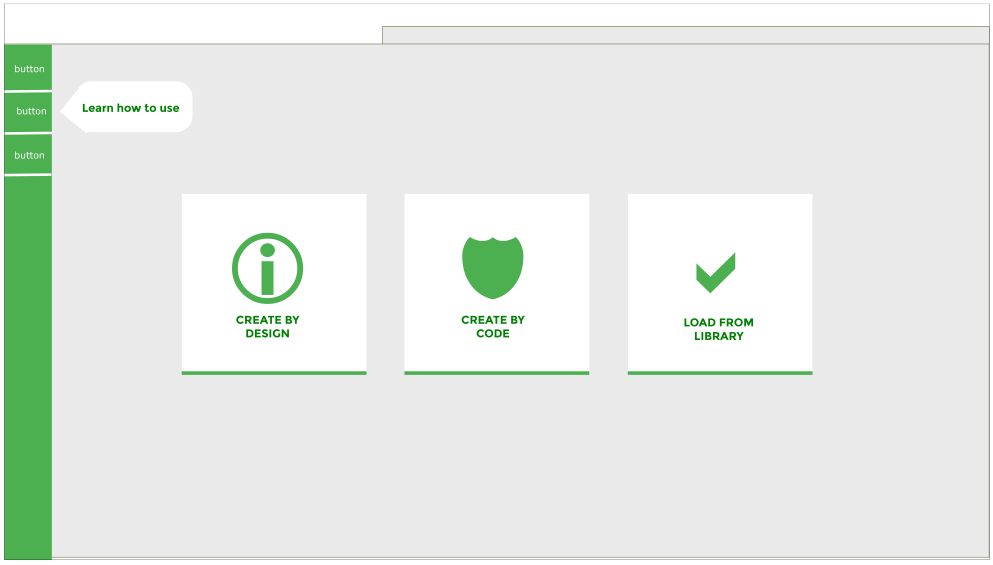


Figure 3.7: Definition Screen

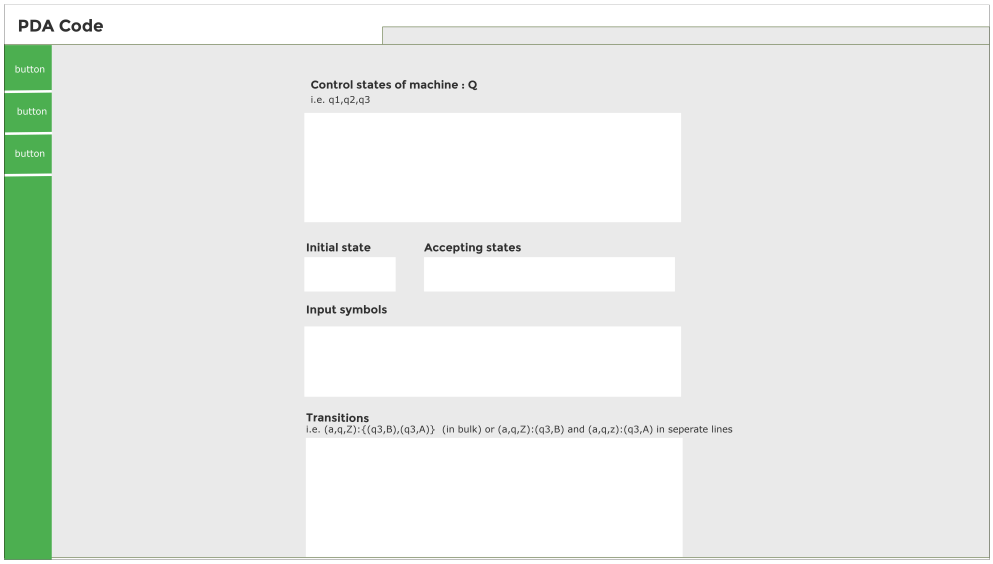
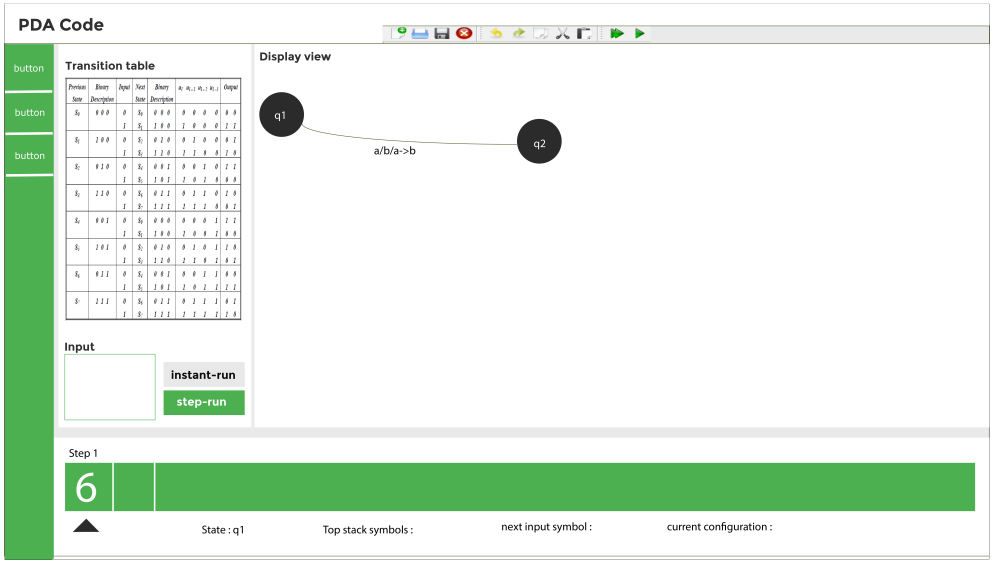


Figure 3.8: Graphic Screen



3.3 Technologies/Languages that will be used

Out of the possible choices, It was determined that the desktop application was the most ideal. A mobile application was not seen as a practical option, because students rarely use their phones to learn with, mainly due to the informality that comes with mobile phones. A web application option would restrict the tool by requiring internet connection. This would prevent any offline use and since the application itself does not require internet, this would hamper the accessibility of the application.

Java is used to develop this desktop application. Java is a powerful language that comes with an extensive number of well-written libraries to do many of the algorithm requirements found in this application and works across multiple different operating systems. JavaFX is also being deployed in this application, which is a powerful Java API used to make high-level interfaces. Each PDA generated needs to be represented with its own states and transitions; this requires a powerful drawing mechanism which JavaFx provides [3]. Gradle, an automation build tool, has also been used, to allow for easy inclusion of any third-party libraries.

Chapter 4

Implementation

This section will delve into how the system was implemented. It will cover the different aspects of the system and how they work in unison. In contrast to the previous chapter, this section will focus on *how* the system was implemented. It will also explore some of the key implementation decisions that were made, as well as the rationale behind those decisions.

4.0.1 Development approach

This application was exclusively implemented by a single actor, which meant that there would be inevitable problems that arose during the development of this application, due to the limited foresight and capacity a single individual has. This problem was anticipated before the implementation phase, and to mitigate this problem a hybrid development approach was taken. This hybrid mix comprised of both agile and waterfall principles. Both of these methodologies had features which made working independently more effective. The agile principles embraced the welcoming of changing requirements and encouraged keeping the implementation simple and maximizing the amount of work not done. Waterfall method was used because it was ideal for small groups and defined a systematic approach to engineering this application. A rigorous specification and design phase was carried out. This allowed for a lot of the obstacles to be foreseen before the implementation phase took place, which in turn allowed for a more successful implementation.

4.0.2 Implementation approach

In the *specification* chapter, the requirements were categorized into clusters. Each represented a different aspect of the system. Before implementation, each requirement was qualified with

a score which represented its importance with respect to the main objectives. This metric was the basis for deciding what features to implement next in the system. Requirements were semi-completed on first run-through and later revisited for a more vigorous implementation attempt. This ensured that the most important features were implemented first and accommodated for changing requirements.

4.0.3 Overview

As was mentioned in the *specification* chapter, the main functionality is defined in the model of the application. For each key component (i.e. the tape) found in the PDA, there is a model class which simulates it. The PDAMachine is a model class which simulates the PDA and is in charge of overseeing the communication between the components found in the model. The view is a "stupid" interface; it is notified of any changes that have occurred in the model and updates its content accordingly. The controller classes facilitate user interactions with the tool, triggering certain model functionalities and notifying the view when a change occurs. This approach is the standard model-view-controller approach.

4.1 Model

The model contains the business logic of the application. It comprises of several different classes, where each replicates either a concept (i.e. ExecutionTree) or a component (i.e. PushDown-Stack). The main classes are :

PushDownStack: A class responsible for simulating the push-down stack in a PDA. It is principally a wrapper class that encapsulates a list of characters. Access to that list is restricted in the same way a stack is. All the regular methods of the stack data structure are supported i.e. top(), pop() and push(char characterToPush). It also defines a method loadState(List<Character> state), which is responsible for allowing the PushDownStack to load immediate state, important for when the PDA needs to go back to a previous configuration. It also defines a clear() method which empties out the content of the pushdown stack. The PushDownStack class has a corresponding view class which uses the data it encapsulates to represent it in a visual way to the user. Any time the stack's content changes, the stack_partial view is automatically updated.

InputTape: A class which replicates the input tape in a PDA. It is principally a wrapper class that encapsulates an input list of characters. It contains a head index (a field) which points to the current head in the list of input characters. One of the main methods is readSymbol() which reads the input symbol at head and increments the head pointer to the next index. It is also accompanied with setHeadPosition(int headIndex) which sets the head index. This method is important for when the PDA needs to go back to a previous configuration. setInput(List<Character> input) is a key method which is responsible for loading new input into the tape. The class also defines a clear() method which removes the current input from the tape. The InputTape class also has a corresponding view class which is exclusively used to represent the data it encapsulates in a visual way to the user. When the tape's content or head changes, the tape_display_partial view is automatically updated to reflect that change.

ExecutionTree: A tree data structure responsible for keeping track of the ConfigurationContext instances that have already been created in the execution. It holds the root ConfigurationContext instance, which represents the initial PDAMachine configuration, and the current ConfigurationContext instance. ExecutionTree facilitates the procedure for restoring the PDA to a previous configuration and also allows for the system to be able to detect already visited configurations. A record must be kept of the previously executed transitions, to prevent the depth-first search revisiting the same path every time.

ConfigurationContext: The instance represents a data snapshot of the PDAMachine at a specific point in the execution. More specifically, it is a store of the stack's content, tape's head position, current state and the step in the computation. It is principally used to allow for a record of previous configurations in the execution to be kept. Instances can be loaded directly into the PDAMachine via the loadContext(ConfigurationContext context) method. For each transition executed, a ConfigurationContext must be recorded in the ExecutionTree. Each ConfigurationContext has a parent instance (predecessor ConfigurationContext instance) and zero or more children (resulting ConfigurationContext instances).

Definition: A class that provides the formal description of a PDA instance. It defines the Transition instances, the ControlState instances, the accepting ControlState instances, the initial ControlState and the acceptance criterion of the PDA.

Configuration: A class which replicates a regular configuration found in PDA theory. A Configuration instance contains the current state, the input character at head and the top

character of stack.

Action: A class which replicates a regular action found in PDA theory. It contains the resulting control state and the resulting top character of stack.

Transition: A class which replicates a regular transition found in PDA theory. It links a Configuration instance with a Action instance.

PDAMachine: A class which simulates the PDA. It's primary role is to oversee the communication between the different components in the PDA. The PDAMachine encapsulates a Definition instance, which defines the PDA's structure. All the other components like InputTape and PushDownStack are encapsulated within the PDAMachine instance. It also keeps a track of the current ControlState of the PDA.

Key methods:

- **executeTransition(Transition transitionToBeExecuted)** – A method which applies the Action part of the Transition on the PDAMachine. As was mentioned before, a ConfigurationContext instance is created to represent the resulting change and this instance is stored within the current ExecutionTree.
- **loadConfigurationContext(ConfigurationContext context)** – A method which takes the ConfigurationContext instance and loads its data back into the PDAMachine. This method is especially important for allowing the PDAMachine to go back to a previously visited configuration. It utilizes the current ExecutionTree which stores all the ConfigurationContext instances up to that point in the execution.
- **previous()** – A method which utilizes the loadContext(ConfigurationContext context) method. It retrieves the parent ConfigurationContext of the current ConfigurationContext instance and then loads it into the PDA machine. If the current ConfigurationContext is the root instance of the current ExecutionTree, then this is not possible, since this instance has no predecessor.
- **stop()** – A method which removes input from the current InputTape, brings the head back to the start and empties the stack's content.
- **isAccepted()** – A method which determines whether the current PDA has accepted input or not. It is evaluated by examining the PDA's components with respect to the

acceptance criterion of the Definition instance.

- **redo()** − A method which brings the head index back to the start and sets the current state to the initial ControlState. Also, creates a new ExecutionTree instance for the new execution.

- **getNonDeterministicTransitions()** − A method which returns the set of Transition instances that are non-deterministic in the PDAMachine.

Other model classes The rest of the classes found in the model are method factories. Method factories are classes that define static methods which are reused several times across the application.

MemoryFactory - contains methods that pertain to saving and loading to and from memory.

ModelFactory - contains useful manipulation methods i.e. stateLookup(List<ControlState> list, String stateLabel) which looks for the ControlState with the specified label in the list.

4.2 Controller

Controller facilitates user functionality with the application and updates the view with model changes. Each view component has a corresponding controller class which links to it. That controller class is responsible for updating the view's content.

StackController: The controller responsible for controlling the stack_display_partial view. It is notified when a change to the PushdownStack model occurs. The StackController calls the requestUpdate() method which gets the content from the stack model and then loads it into the visual stack.

TapeController: The controller responsible for controlling the tape_display_partial view. It is responsible for being notified when a change to the current InputTape model occurs. The TapeController calls the requestUpdate() method which gets the data from the tape model and loads it into the visual tape.

TableTransitionController: The controller which oversees the transition_table_partial view. It facilities the ability for the user to be able to modify a transition row and add a new transition row. When a transition is added to the system, the new TransitionTableEntry is passed to the controller where it is added to the row collection, which automatically

adds it into the TableView. The TableTransitionController is also responsible for highlighting certain rows by transition via the select(Transition transition, boolean isMulti) method. This can be used to highlight the transitions that are being executed in the current execution or alternatively be used to highlight the non-deterministic transitions in the PDAMachine.

MachineDisplayController: The controller responsible for maintaining the visual PDA display shown to the user. Initially, when loaded, it takes the current Definition instance and generates a visual representation within a given region. For each ControlState, a VisualControlState instance is created and added to the Pane found in the pda_display_partial view. It mathematically makes calculations on where to arrange each VisualControlState object on the screen. Then the individual VisualTransition instances are added. This controller is also responsible for highlighting VisualControlState and VisualTransition instances during the process of a step-run execution.

UserActionController: The controller responsible for controlling the user_action_partial view.

UserInputController: The controller responsible for controlling the user_input_partial view. This controller initiates all new executions requested by the user.

PDARunnerController: This controller encapsulates all the controllers listed above and is responsible for facilitating the communication between these controllers. It is also responsible for manipulating the main pda_runner_page view. The PDARunnerController also contains the currently loaded PDAMachine model. This is the only controller that can execute functionality belonging to the model. This class acts as an extra layer above the PDAMachine class which facilitates user interaction.

Key methods:

- **loadPDA(PDAMachine model)** – A method which loads a PDAMachine model instance. The transition table and PDA visual are updated to represent the new model.
- **next()** – The method retrieves all the possible transitions that can be executed from the current configuration. Each is opened to the user for selection. The PDAMachine method executeTransition is called on the chosen Transition and the appropriate changes are made to the model. The visual components are notified of the change and are updated.

If there is only one transition that can be executed, then that transition is automatically chosen to be executed.

- **previous()** − A method which takes the PDAMachine back to the previous configuration, by calling previous() in the model. The visual components are notified of the change and are updated.
- **stop()** − A method which restarts the model from scratch and clears the content from the visual components.
- **nextBranching()** - a method that goes to the next computational branching in the step-run. It works by continually calling next() until a branching occurs or alternatively the end configuration is reached (the PDA terminates).
- **previousBranching()** − A method that goes back to the previous computation branching in the step-run. It works by continuously calling previous() until a branching occurs or alternatively the initial configuration is reached.
- **redo()** − A method which restarts the model from scratch and loads the input into the PDAMachine again. The visual components are notified of the change and are updated.
- **stepRun()** - A method which initiates step-run on a given input.
- **instantRun()** - A method which initiates instant-run on a given input. A depth-first search is carried out on the PDAMachine to find a solution. When a dead-end configuration is reached, the PDAMachine backtracks to the previous branching and an alternative configuration is chosen.

The PDARunnerController also facilitates the opening of popup dialogues:

- openStepRunOutputDialog(boolean isAccepted)
- openNewTransitionDialog()
- openInstantRunResultsOutputDialog(boolean isAccepted, boolean hasSingleSolution)
- openSaveDialog()
- openTransitionOptionDialog(List¡Transition¿ transitions)
- openConfirmationDialog()

Page Controllers

There are also controllers that in charge of overseeing individual pages

- **QuickDefinitionController**: Controller for the quick_definition_page view. In charge of validating user-input.
- **HomeController**: Controller for the home_page view.
- **InfoController**: Controller for the info_page view.
- **LibraryController**: Controller for the library_page view. Initially, loads all Definition instances from memory to a ListView. Each Definition is available to load.
- **HelpController**: Controller for the help_page view.
- **ExamplesController**: Controller for the examples_page view.

4.3 View

Views in this application are principle written in xml. The view itself can either be a stand-alone view or a component that is reused across several views i.e. save_dialog_partial. Controllers are responsible for making the views dynamic. Every view has a linked controller which does this job. The approach was to keep the UI as minimalistic as possible. Too much clutter on the screen would only distract the user from the main message trying to be delivered.

4.3.1 PDARunnerController

The runner page is the main view of this application. It is incharge of loading a PDA and representing the procedure of execution to the user. This page can be found in the appendix.

main components :

- **Visual stack (stack_partial)** - A visual stack which updates its content throughout the computation.
- **Visual PDA display (display_partial)** - A visual representation of the PDA's Definition instance. Control states are represented with circles and transitions with arrows. It is redrawn every time a change is made to the current Definition instance. The visual display is also used to represent procedure of execution. The current state is highlighted

in the visual. Also, the transition arrow is highlighted when the corresponding transition is executed on the PDAMachine.

- **Transition table (transition_table_partial)** - A table which contains all the transitions found in the PDA. The current transition being executed on the PDAMachine is highlighted.
- **Visual Tape (tape_partial)** - A graphical tape which updates throughout the computation.

4.4 Main features

4.4.1 Step-run Mode

As was discussed in the specification, step-run is one of the key features found in this application, so it was imperative for this feature to be both robust and user-friendly. The user can enter step-run at any time after a PDAMachine instance has been loaded. For every stage in any execution, the current configuration of the PDA is made apparent to the user. The current state is also highlighted in the visual PDA display. There is also a visual stack and tape which updates throughout the execution. Once the user enters step-run, the input from the input box is taken and loaded into the PDAMachine and the corresponding views are updated. An Input word is not obligatory for step-run to be entered, as the input itself can be empty. Once the input has been loaded into the PDA, the user is offered several actions to manipulate the current execution. The main one of those is ”next”, which progresses the current execution forward by one configuration. In cases where there is more than one possible transition to chose from, the option is offered to the user to select the transition of preference. ”previous” takes the PDA instance back to the previous configuration, truncating the previously applied transition. ”next branch” takes the PDA instance to the next instance where there is more than one possible transition to select from. For ”next branch”, if there are no instances, then the execution just progresses until the PDA terminates. ”previous branch” works much the same way but going back to the previous instance where there is more than one transition to chose from. With ”previous branch”, if there are no previous instances of branching, then the current configuration goes all the way back to the initial configuration. ”stop” terminates the execution, clears the tape and empties the stack. ”redo” resets the current PDA instance to the initial configuration, moves the head back to start of tape and starts a new execution instance.

”next”, ”next branch” and ”previous branch” trigger animations on the visual PDA display. The transition being executed is highlighted alongside the resulting state. This allows the user to follow the procedure of execution. The transition chosen to be executed is also highlighted in the transition table view.

Problems and Observations

Initially, large PDAs that contained high amounts of transitions produced visual representations that were over-complicated. This was mainly due to the sheer number of (transition) arrows that were required to be drawn and the transition labels which overlapped as a result. This problem was mitigated by only rendering one visual arrow for transitions that had the same source and target. The application also was made to partially prevent labels from overlapping on screen. An example would be If there were 6 transitions going from the same source and target, then the application would be able to stack 3 transition labels above the arrow and three below. Essentially, the application became more spatially aware when rendering visual components onto the screen. But, this problem becomes exponentially more difficult to handle when there are diagonal transitions involved. The application can at times move labels too far from their corresponding transition arrow which can lead to even more confusion. The application is not meant to simulate unnecessarily large PDAs. This is made clear to the user by certain restriction put in place i.e. 50 states limit.

Also, when a PDA has a large number of states, it was discovered that for a lot of those states, they would be rendered off-screen and there would be no way for the user to view them. There was only a certain number of pixels given to rendering the visual PDA. This problem was initially alleviated by limiting the user to only be able to create PDAs involving at most 20 states. But this restriction was later relaxed to 50 and a zoom-and-pan feature was added. This gave the user the ability to zoom out of the visual PDA and see all the VisualControlState instances involved regardless of size. This effectively solved the initial problem identified by giving the user more control.

It must also be noted that for deterministic PDAs, both ”next branch” and ”previous branch” become ineffective as there will be no occurrences in the computation where more than two transitions can be executed at the same time.

4.4.2 Instant-run Mode

Instant-run was also one of the main features. It is a feature responsible for evaluating whether a PDA accepts an input or not. It does this by examining the different branches of execution. More precisely, it runs a depth-first search (dfs) on the execution. It starts from the initial configuration of the PDA and retrieves the possible transitions that can be executed from that position. It arbitrarily selects a transition to execute. It keeps progressing through the execution in the same way until it reaches either a configuration where the accepting conditions are met or where there are no more possible transitions to execute. In the first case, the dfs is cancelled and the PDA accepts the input word. In the second case, the PDA backtracks to the previous branch and reselects another transition which has not yet been explored. It explores this branch the same way. If all transitions in the branch have been explored, then the PDA is restored to the previous branch before that. If there are no previous branches, then the PDA is restored back to the initial configuration and the PDA rejects the input. It is enough to conclude that the PDA does not accept the word. This idea was covered in detail in the *background* chapter, where all branches must be explored before a PDA can reject an input and only a single sequence of configurations is required for a PDA to accept an input. When the input word has been evaluated, the results are shown to the user via an output dialogue box.

4.4.3 Problems and Observations

As was discussed in the *background* chapter, some non-deterministic PDA machines never terminate. More specifically, non-deterministic PDAs that contain cycles of jumping transitions. When this occurs, the simple depth-first search algorithm outlined above is no longer a viable solution that can evaluate input, since it is no longer a tree that is being explored for a solution. There must be a way of detecting that the current execution is in a n-cycle loop (where $n \geq 1$) and to be able to backtrack to a configuration which preceded the loop and pick an alternative branch. This problem was partially addressed initially by improving the dfs algorithm to be able to detect cycles of size $n \leq 2$ but this approach only made the computation exponentially more expensive at run-time. This was later removed, and an alternative approach was taken, where at the end of every 20th configuration in the current execution, the user is offered a dialogue box, outlining the choice of whether to continue the search or whether to enter step-run from the current configuration. That way in the case of the PDA being stuck in a loop, the user will be given the opportunity to visualize the problem and cancel the execution. This is a better approach because it makes the procedure transparent to the user and does not drain

CPU usage.

Instant-run sometimes must explore all possible branches to evaluate an input. The worst case running time of instant-run is O(n + m), where n represents the number of configurations and m represents the number of edges (backtracks). This can be computationally expensive for very large non-deterministic PDAs. The solution specified above was also used as a way of mitigating this problem. For every 20 configurations, the user is offered the ability to enter step-run or continue with the search.

It must also be noted that for deterministic PDAs, this algorithm becomes exponentially easier to evaluate since there will always be at most one transition that can be executed from any configuration. The worst case running time becomes O(n) where n is the number of possible configurations found in the PDA.

4.4.4 Storage Feature

As defined by requirement *1d* in the *requirement analysis*, the ability for the user to save and load custom PDAs is a feature of great importance. The Definition instance in JSON format is what is saved into memory. The Definition instance as mentioned above represents the internal structure of the PDAMachine.

Saving

The user can either save their custom definition on the quick definition page directly or after the PDA has been loaded into the PDA runner page. It is never enforced on the user to save every Definition instance that is created. This choice is left to the user themselves.

When the user requests to save a definition instance, the user must specify an id to attribute to the definition instance. This id is what identifies the Definition instance from all other instances when being loaded back by the user, so it must be a unique id for it to be distinguishable. A unique id is a prerequisite for a successful save. If a Definition instance with the given id already exists, then the application will prompt the user to re-enter a new id. Once the id is provided, the Definition instance is taken and passed to *MemoryFactory.save(Definition definitionToSave)*. This function looks for a library.json file in the same directory as the jar. If found then the new Definition instance is added to the library collection, and the collection is serialised into a JSON object and stored in the file. Alternatively, If it does not exist then the library.json is created and the JSON content is stored on it.

loading

The library_view is the view in charge of showing Definition instances found in memory to the user. It is primarily controlled by the LibraryController. This controller must first look for the library.json file in memory. If found, it converts its content into java code, creating an ArrayList of Definition instances. The collection of saved Definition instances is offered to the user in the form of ids in a list view contained within the library_view, which can either be loaded or deleted by the user. Once the user has decided what definition to load, the chosen Definition is encapsulated within a PDAMachine instance and then loaded into the PDARunnerController via the *loadPDA(PDAMachine machine)* method.

Problems

Java uses security protocols which ensure that the jar file cannot be modified. Jar files are signed and changing a single bit from the file will invalidate it. This meant that the library.json file, which is constantly changing in content, could not be stored within the jar file itself. It was decided instead for the library.json file to be stored in the same directory as the jar. If the jar file at any time is moved from this directory, the application would lose scope of the library.json file and thus no longer be able to load the definition instances from memory; it would mean that the library store would be empty on application load. Additionally, in the case of someone modifying the library.json file, the application would crash because it would not be able to parse the content of the file back into Java objects.

4.4.5 Quick-Definition Feature

This feature provides a way for the user to define their own push-down automata through simple description. It provides the minimum number of fields required to generate the desired PDA. It restricts the user by forcing the systematic definition of a PDA. The user must first specify the number of control states that the desired PDA will have and the system will generate those. From those generated states, the user can choose their initial state and their accepting states. Only after they have decided on these 3 fields can the user progress to the next section where they can define the transitions of the system. There is a section responsible for adding transitions. There is also a terminal displaying the transitions already added, where the user is able to modify and delete those added transitions. The user must also specify the type of accepting criterion the PDA will have. Once all fields have been successfully validated, the user can generate the PDA. It validates the fields and then creates the Definition instance based on

the description provided. The Definition is then encapsulated within a PDAMachine instance and loaded into the PDARunnerController via the *loadPDA(PDAMachine machine)* method.

Problems and Observations

During implementation, it became very difficult to find a satisfactory way of presenting this form to the user. During the design phase a method was put forward that was too formal and was not interactive enough. This form was initially implemented but later dropped for an alternative approach. The approach defined above was taken instead, which introduced a more systematic and interactive approach to defining PDAs.

Also, it was decided on during implementation to not include fields asking for the input alphabet or the stack alphabet. These two criterions were abrogated from the Definition instance altogether. They were seen as a way of restricting what the user could do with their PDAs. It also required a level of intuition from the user to know for certain what their required stack alphabets and input alphabets had to look like to simulate their PDAs as required.

4.4.6 Non-Deterministic Mode

This is a feature which identifies the non-deterministic transitions in a PDA. The mode can be entered once the PDAMachine has fully been loaded. Entering the mode will highlight all transitions in the visual PDA display which are non-deterministic. It will also highlight the transition table rows that correspond to the non-deterministic transitions. If there are no non-deterministic transitions in the PDAMachine (a deterministic PDA), then no transition is highlighted and the user will be alerted.

4.4.7 Examples Feature

This feature offers the user a collection of pre-defined Definition instances to load. The pre-defined Definition instances are stored within a JSON file contained in the jar file itself. This prevents it from being written to. The Definition instances come from the examples outlined in "Introduction to the theory of Computation" [10]. Together, they give a broad overview of the type of functionality offered by a PDA.

4.4.8 Help Feature

There are two categories of help a user may request for. The first of which is on the main features and how to use the system and the second is on the understanding of PDAs as a concept. An application which provides for both will ensure that the user gets the most from the application.

User Manuals

This feature is one of the most important features. The success of this application is hinged on how easily the user can use this tool. The user manuals offer a systematic step-by-step explanation of the main features in this application. It covers in detail how a user can define their own PDA and how input can be simulated. The user manuals themselves are PDF files that can be read from the application itself. The user manuals are accessible from anywhere in the application and can be opened from the side-bar.

Information Feature

There is also an information section which is responsible for conveying the essentials of the PDA. Enough of the fundamentals are covered in order for the user to use this application successfully. Some students are less experienced than others, so it is important as an educational tool to cater for the ranging skill-levels. This feature is facilitated by the InfoController which loads a PDF into the info_page view and can be accessed from the home_page.

4.5 Requirements Omitted

There were requirements that were left out due to the time restrictions enforced on this project. As was mentioned before, each requirement was classified with a score and this metric was used to determine the order of implementation.

The requirements 3a, 3b, 3c, and 1a have all been omitted from the implementation. They have medium to low priority scores. These group of requirements relates to the graphical and visual specification of PDAs. It was decided that a descriptive approach would have a higher priority than a graphical one since a descriptive approach would be more informative. Initially, the aim was to allow for both of these approaches to feature in the system and to allow the user to choose their desired approach. But, this was ultimately dropped because of the estimated time it would take to implement. Its usefulness was not considered great enough to justify the

limited time that would be spent in implementing this feature.

The rest of the requirements have been accounted for in this application. Another indicator that this application is successful in reaching its objectives.

4.6 Plugins

There were several plugins which were used to implement this system, all of which were open source and free to use. Gradle was used as the library manager for this application. A copy of the gradle.build file can be found in the appendix, where it gives references to all the plugins used.

The plugins :

- Junit: Facilitated unit and integration testing. Junit is an open-source testing suite for Java.
- Google Guava: Provided ListMultiMap data structure, which was used to store a state to transition mapping in the PDAMachine class. It allowed for the quick retrieval of transitions by source state.
- Google GSON: Allowed for serialization of Java objects. More specifically, it facilitated a way of converting the user's collection of Definition instances from JSON to Java code and vice versa.
- ControlFX: Provided the CheckComboBox view element, which allowed for multiple selections to be made by the user (specifically used for defining accepting states).

Chapter 5

Testing

Testing was used throughout this project to validate individual components and more specifically examine how these components work in unison. This application is principally a tool, so most of the testing was focused on proving the tool's correctness, instead of validating how the user interacts with the tool.

5.0.1 Approach to Testing

The testing was solely focused on examining the model in the model-view-controller application. Ensuring that the model is fully tested is enough to ensure that the application works as intended.

The tool can be divided into several sub-components. The PDAMachine oversees the communication between these sub-components. For each individual sub-component, testing was used to examine correctness.

5.0.2 Method of Testing

The Junit plugin was used to carry out the testing on the application. As was discussed before, Junit is an open-source plugin which allows for both unit tests and integration tests to be written in the Java language; it is specifically endorsed by Java as their principle testing suite.

5.1 Unit Testing

Unit testing was especially important for this project. It was carried out extensively on each of the different components found in the system. It was used to ensure all the components work

as intended, and this, in turn, contributed to proving the system's correctness.

All in all, there are around 40 implemented unit tests. Getters and setters have not been included for testing since they are assumed to work as intended. The plugins used in this application have been tested by the publishers, so testing of them have been skipped.

Main unit tests of PushDownStack component

- Testing the construction of a PushDownStack instance (examines the initial fields of the created instance)
- Testing the pushing of an element
- Testing the popping of an element (covering empty stacks as well)
- Testing the retrieval of the top element of the stack (covering empty stacks as well)
- Testing the clearing method of the stack
- Testing the isEmpty method of the stack
- Testing the loadState method (A method which allows for quick loading of state into the stack)

Main unit tests of InputTape component

- Testing the construction of an InputTape instance (examines the initial fields of the created instance)
- Testing the setting of tape input
- Testing the reading of an element
- Testing the previous method (A method which moves head to the left)
- Testing the clearing method of the tape
- Testing the isFinished method
- Testing the getSymbolAtHead method of the tape

Main unit tests of both the ConfigurationContext and ExecutionTree component

- Testing the construction of a ConfigurationContext and ExecutionTree instance (examines the initial fields of the created instance)
- Testing the adding of a new child ConfigurationContext to a ConfigurationContext instance
- Testing of the hasChild method of a ConfigurationContext instance

Main unit tests of the PDAMachine

- Testing the correct identification of the non-deterministic transitions
- Testing the correct retrieval of possible transitions from the current configuration
- Testing the loadConfigurationContext method

As was mentioned at 4.1, the ExecutionTree is only a container for the ConfigurationContext instances. It is only responsible for holding the root ConfigurationContext and the current ConfigurationContext. This is why there is only 1 unit case examining the ExecutionTree.

The unit testing for factory classes has also been carried out. For both the ModelFactory and MemoryFactory, all methods have been tested.

5.2 Integrated Testing

Integration testing was carried out within the PDAMachineTest class. These integrated tests focused on examining how the different components within the PDAMachine came together to determine the transitions to execute and what is the effect of those transitions being executed on the individual components. The parts which were tested previously individually were combined and tested collectively as a group. The reason for such testing is to expose faults in the interaction between integrated units. This is especially important for this type of application which has many distinct components working in unison.

- Testing the execution of a transition
  - Moves tape by one
  - Updates stack

  - Change of current control state
  - Record ConfigurationContext in current ExecutionTree

- Testing the stopping of an execution
  - Set current state to NULL
  - Clear tape
  - Clear stack
  - Set current ExecutionTree to null

- Testing the loading of input into a PDA machine
  - Empty the stack
  - Set input into the tape
  - Set head to start of the tape
  - Set current state to initial state
  - Create a new ExecutionTree and create a root ConfigurationContext instance

- Testing the redo of an execution
  - Set current state to initial state
  - Set head to start of the tape
  - Clear stack
  - Create a new ExecutionTree and create a root ConfigurationContext instance

- Testing the restoring of the PDAMachine to the previous configuration (previous() method)
  - Mark current ConfigurationContext instance as not in the path
  - Retrieve the previous ConfigurationContext
  - Moves tape back by one by calling previous()
  - Load stack state found in ConfigurationContext
  - Change current control state to ConfigurationContext's state

5.3 System Testing

The ExamplesRunTest validates the running of the PDA examples. 5 vastly different examples can be found in this application. It is impossible to guarantee that a PDA correctly recognizes a language without examining all possible inputs. For each of the examples, three inputs which the PDA should and should not accept are tested.

The purpose of system testing is to evaluate the system's compliance with the requirements. More specifically, to examine the correctness and reliability of both the outcome and procedure. The tests focus on examining the non-functional requirements :

- The application must be able to find a solution (an execution path with a terminating accepting configuration) if it actually exists.
- The theory of the PDA described in the project must be correctly translated into the application.

5.4 Alpha Testing

Alpha testing was carried out on a small sample of university students. It was carried out only after the main bulk of features were implemented. Students with varying skillsets were purposefully chosen to give a broad range of perspectives. It was used to identify any bugs and recognize how effective the application was in fulfilling its main objectives. The only main contention identified during alpha testing was around the feature responsible for allowing users to define their own PDAs. It was ubiquitously described as being too confusing and very limited by the students in the sample. This process recognized that an alternative approach was needed. The approach outline at 4.4.5 was instead taken.

Chapter 6

Professional and Ethical Issues

6.1 Ethical implications

6.1.1 Security

This application is uniquely an offline one, which eliminates many of the security risks that might arise. It also does not ask for any user-sensitive information, so privacy is not a major factor either. The application's security concerns are inconsequential for them to be considered.

6.1.2 Availability

The intention was to make this application accessible to everyone. This is not always possible due to the limitations in software and hardware. The prerequisites to run this program are purposefully kept at a minimum to allow for the majority of users to be able to run this application successfully.

6.1.3 Education Standard

This application is principally an educational tool, so there is a certain obligation for correctness. The information conveyed by the program must be substantiated and correct. Any less would see this application as ineffective and might mislead users. Rigorous testing and planning were put in place to ensure accuracy.

6.2 Code of Conduct and Code of Good Practice

The Code of Conduct and the Code of Good Practice summaries the practices and responsibilities demanded from actors within the IT profession. The guidelines that pertain specifically to this project have been followed throughout. Precisely, rules relating to correctness and planning standards. A lot of the rules outlined in the document are not appropriate for this type of project.

Chapter 7

Evaluation

This section will go on to explore some of the different aspects of the project and some of the limitations that were discovered during implementation.

7.1 Limitations

7.1.1 Limitation in software

One of the most apparent limitations was the lack of support given to legacy computers that had older versions of Java. The application itself is a JavaFX application and makes use of lambdas. Both features were first introduced in Java 8. The application itself exclusively runs on Java 8 and above; while a lot of legacy computers do not support Java 8 and are not able to open it. This reduces the overall accessibility of the application.

Another limitation comes in the representation of the epsilon jumping symbol. It is not recognized by all operating systems. It is also not represented on the English keyboard, which makes it difficult for the user to define jumping transitions. The decision was made to instead use the '/' character to represent the jumping symbol. This essential adjustment caused a minor split between the notations seen in the textbooks and the application itself. This was problematic since it was initially specified in the requirements that the application must abide by the theoretical notations since the user (the student) would be able to more easily familiarize themselves with the system. Although not avoidable, these changes were made clear to the user.

7.1.2 Limitation in time

It was imperative to keep the implementation time-frame constrained to 2 months. This came with its fair share of complications. The set of requirements that defined how the user could view and interact with the PDA visually was drastically simplified, and this was not initially the aim. The functional requirements 3a, 3b, 3c and 1a were all ignored.

The application does well in that it is able to represent a PDA machine visually. However, in its current state, it only supports a static representation of the PDA machine. When a machine has an excessive amount of transitions, it becomes exceptionally tedious understanding the process of execution. Given more time, the adding of a more dynamic display would solve this problem. A possible feature would offer the user the ability to move the control states around a given region, in order to get a better arrangement of transitions. A feature seen in Kyle Dickerson's automaton simulator [5].

7.1.3 Limitation in theory

As was discussed in the background and theory chapter, some PDA machines never terminate. More specifically, non-deterministic PDAs which contain cycles of jumping transitions. The problem of detecting cycles of n-length (where $n \geq 1$) is a non-polynomial problem. The most optimal solution would be one which could identify when the execution is stuck in a loop and be able to backtrack to the last configuration before the loop started. But given today's hardware, there is no efficient solution for this problem. The application could initially detect cycles involving one or two states, but this was later removed.

The main objective of the application is not to derive output for every word inputted. Its main objective is to be a visualization tool that represents the procedure of execution clearly and transparently. So instead, after 20 steps, the user is offered the choice to continue the search or go into step-run to see what's going on.

7.2 Project evaluation

The success of this project was reliant upon the amount of understanding shown and the extensity of the planning and specification. A systematic approach was taken in this project and as was mentioned before, a strict schedule was followed to ensure that the project was fully completed in time.

For this project, the theory was mainly taken from "Introduction to the Theory of Computation" [10] . In cases where divergence was necessary, either because of the lack of information offered, or the difficulty in representation or implementation, a different approach was taken. An example of this is where '/' is used as the jumping symbol instead of ϵ (epsilon symbol), because of the lack of support shown by legacy operating systems. This is the same symbol used to represent a jumping transition in JFLAP.

Sipser's book also does not cover determinism or deterministic transitions in any kind of detail. Instead, "Introduction to Automata Theory, Languages and Computation" [6] was referred to for this information.

An important resource in this project was the set of pre-existing applications that could already simulate a PDA. All in all, there was a total of 3. Each was rigorously examined for both their positives and negatives. In section 2.6, a brief analysis of each is given. A large section of the *requirement analysis* was directly informed by the applications examined. This was to ensure that the same mistakes were not made.

As was said, the implementation was heavily reliant on the *specification*. The requirements were implemented in order of priority score. Requirements were semi-completed on first run-through and later revisited for a more vigorous implementation attempt. This ensured that the most important features were implemented first and also accommodated for changing requirements. One example is the zoom and pan feature which allowed users to focus on certain points in the visual PDA. This requirement was not initially identified, but this robust approach allowed for it to be implemented.

The application has been very successful in achieving the main objectives of this project. The system created is easy and clear to use for both students and lecturers. It allows the user to visualize the PDA machine's procedure of execution for any given input and allows them to effortlessly adjust the current configuration for that execution. The application also offers the user the ability to define their own PDA machines without any difficulties. The application conforms with common PDA conventions in cases where it is unanimously agreed upon and takes the good features found in other simulators and builds upon them. There is also an ample amount of additional tools to aid students in their understanding of PDAs. The notations used are intuitive, easily conveyed, and most importantly, correct.

Chapter 8

Conclusion

The main objective of this project was to produce an educational application that was able to simulate a PDA machine correctly and clearly to the extent where the user could easily visualize and learn. By these metrics, the application is successful. It takes what's good in other applications and builds upon them.

Even though the PDA is a stationary concept in computing and very little has been done in the way of further exploring it in the last 20 years, the requirement for visualization tools that represent these concepts still does remain. It covers the fundamentals and prerequisites found in modern-day computing, so the need for these tools will only surge as the number of students taking computer science increases. So the goal moving forward is to make effective applications that are able to replicate computational concepts in more interesting and innovative ways.

8.1 Future work

A non-functional metric of how good this system is can be measured as: the amount of user interaction offered to the user. Since there is no finite bound for this metric, there will always be additional features that can be added to make this system even better. The improvements offered below are some of the possible solutions to the limitations identified throughout this report, and more specifically acknowledged in the evaluation.

8.1.1 Possible improvements

The application should be made to be compatible with older versions of Java and thus be able to run on more computers. Java has stopped it's support for the Windows XP os, stating "we can no longer provide complete guarantees for Java on Windows XP since the OS is no longer being updated by Microsoft" [4]. A way of catering for older systems is to migrate the application to a web-based system. This would improve the availability of the application, by migrating the storage and hardware requirements to a remote web-server. This is only a problem for more poorer countries that have less access to technological resources.

Another improvement that can be made is to make this application online-compatible. The online side of the application would involve a community of users discussing and reviewing different PDAs. Each user would be able to publish their own PDA and be offered feedback. This approach encourages dialogue and learning amongst students and is symptomatic of a modern learning application. Discussions welcome reflection and deeper thinking.

As was touched on in the *evaluation*, this application should offer the opportunity for users to visually define their own PDA machines. The user should be able to move states around, add visual states via drag and drop and add transitions by dragging arrows between the different states. It will provide an alternative way of definition and might prove a better learning technique for some students who prefer visualization over description. There have been studies shown to support the claim that visualization does indeed help improve recall and retention. Hundhausen, Douglas, and Stasko did a meta-analysis on the 20+ studies carried out on educational visualization effectiveness [7]. Around 8 of those studies concluded no substantial effect on performance. But for around half, there was a noteworthy change measured.

A possible extension to this application could involve incorporating more computational tools like the Finite Automata together into one centralized system, as seen in JFLAP [11]. This will allow the application to explore deeper ideas about computational power and convey those ideas in interactive ways to the user. Another enhancement could involve allowing a feature which could translate any accept-by-accepting-state PDA into the equivalent accept-by-empty-stack PDA and vice versa. Since the two accepting criterion can be shown to be equivalent, there will always be a conversion either way. This was previously covered in detail in section 2.2.2 of the report.

Less obvious improvements may include:

- Making the application compatible with mobile and tablets
- Making it more disability-friendly by adding voice narration for the visually impaired
- Adding a more in-depth and visual walkthrough of the application

All these improvements contribute to making the application more user-friendly.

References

[1] Model-view-controller wikipedia. https://en.wikipedia.org/wiki/Model-view-controller. Accessed: 04-12-2017.

[2] "the exploding demand for computer science education, and why america needs to keep up". https://www.geekwire.com/2014/analysis-examining-computer-science-education-explosion/. Accessed: 29-10-2017.

[3] "what is javafx?". `https://www.geekwire.com/2014/analysis-examining-computer-science-education-explosion/`. Accessed: 29-10-2017.

[4] "windows xp and java". `https://www.java.com/en/download/faq/winxp.xml/`. Accessed: 3-4-2018.

[5] Kyle Dickerson. Automaton simulator. http://automatonsimulator.com/. Web application.

[6] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[7] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A Meta-Study of AlgorithmVisualization E Âą ectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.

[8] Kevin Lano. *Formal Object-Oriented Development.* Springer, 1995.

[9] College of Saint Benedict and Saint John's University. Cburch automaton simulator. http://www.cburch.com/, September 2001. Java Desktop Application.

[10] Michael Sipser. *Introduction to the theory of computation.* PWS Publishing, 1997.

[11] Duke University Susan H. Rodger. Jflap. http://www.jflap.org/, August 2009. Java Desktop Application.

[12] Mariko Funada Yoshihide Igarashi, Tom Altman. *Computing: A Historical and Technical Perspective*. CRC press, 2014.