Imperial College
London

An Educational Tool for Modelling and Simulating Automata Machines

Masters Project | Amin Mansour

Supervisor: Anandha Gopalan

April 13, 2022

Contents

Acknowledgements

Thanks to Dr Anandha Gopalan, my supervisor, for providing assistance and feedback throughout this project. Discussions were essential to motivate me to think creatively and to develop this application in a thorough and pragmatic way.

Abstract

Technology has advanced considerably in a short period. The number of students deciding to adopt a career in technology has increased with the number of new enrollments in a technology-related course in the US estimated to eclipse 50000 by 2020 (CRA [2018]). E-learning applications are becoming increasingly necessary. Automata theory introduces a collection of machines that help to illustrate fundamental computational concepts that are important for students to grasp. These machines are taught to students in abstract ways, and often students are not provided with opportunities to interact with these conceptual devices. In this paper, we introduce an e-learning application that enables students to interact with these tools meaningfully. The application itself tries to utilise well-established pedagogy principles and accurately incorporate and represent the main ideas covered in the automata literature. We carried out a systematic review of the application and its effectiveness at achieving the central objectives illustrated in this paper. The tool definitively outperforms alternative applications. Most participants in our study strongly agreed that the application provides the necessary working features to support the student. They also noted that there is an effective use of illustrations and animations throughout the application. Unanimously, students believed that learners would benefit from integrating this tool into a study program. In this paper, we will discuss the implications of our findings and outline the process of constructing this pedagogical tool.

Chapter 1

Introduction

The need to create informative applications to portray abstract computational theories has increased. The number of students who take computer science has risen each year (Figure 1.1). Such computational tools are a necessary component for any technology-related course. These devices are covered in a number of university syllabuses and reappear in many areas of literature.

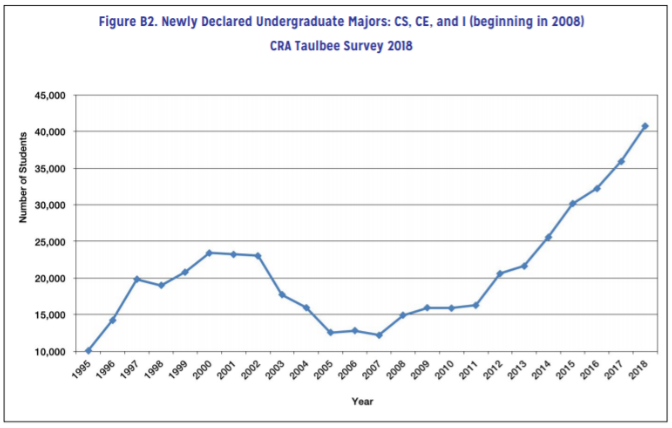


Figure 1.1: The trend of new enrollments into computer science (or related) fields in the United States. CRA [2018]

Automata theory is the study of abstract machines and the computational problems that can be recognised by employing them. Before the 1990s, automata theory was considered as an active area of scientific research. It was targeted towards graduate students and taught as a graduate-level subject. Today, the volume of activity has significantly decreased, and little motivation is shown to the possibility of new research being carried out within the field. It has evolved to become a static component of the undergraduate curriculum. Computer science, in its current iteration, has become more vocational-orientated. However, there are tools conveyed in automata theory that are still relevant today, with many newer technologies incorporating them.

For instance, a well-established approach to implementing a standard compiler is to combine several automata machines. Finite automata (FA) is ideal for lexical analysis - dividing input into recognisable independent entities. The pushdown automata (PDA) is suitable for syntax analysis when examining the relationship amongst entities. Due to the pushdown stack, the pushdown automaton is able to retain state - a necessary property required for verifying relationships amongst entities. These machines can be used in combination to execute the different compilation stages. The alternative approach would be to implement the compiler from scratch. However, if a language has a wide-ranging lexicon, this approach is not fast or scalable. Using these tools in combination to solve specific use cases is an effective technique. These tools can provide both reliable (i.e. easily testable) and efficient strategies for solving decision problems.

Automata theory introduces a collection of machines that help to illustrate fundamental computational concepts that are important for students to grasp. These machines are taught to students in abstract ways, and often students are not presented with opportunities to interact with these conceptual devices. In this paper, we introduce an e-learning application that will enable students to interact with these mechanisms directly and meaningfully.

As we will discuss, several applications propose to simulate the same conceptual machines. They each carry flaws that negatively impact the educational effectiveness and user engagement of the application. The motivation from this project comes in trying to produce a tool that effectively utilises well-established educational and HCI principles. A design that fully encapsulates the automata theory concepts and enables students to interact meaningfully with the system. In this paper, we will formalise the different stages of development to show how features were chosen and integrated into the system.

Chapter 2

Background

In this section, we will comprehensively cover all relevant underlying issues that went into developing this educational application and then go on to look at some of the alternatives that currently exist.

2.1 E-Learning Applications

It remains important that fundamental educational principles are considered when designing this system. A number of works have been written on the subject of educational technology. Mental development has long believed to be improved through the use of games (Rieber [1996]). This application will be presented as a type of game that the user can directly interact with. Such systems must provide learners with sufficient motivation to seek knowledge themselves. One of the educational theories articulates that learning should be both self-motivated and rewarding Malone [1981]. This overlaps with fundamental beliefs in game theory. Thomas W. Malone analysed some of the main factors that contributed to making games engaging. Challenges in the form of levels, unpredictable outcomes and fantasy were found to be the main determinants. Fantasy being "mental images of things not present to the senses or within the experience of the person involved" Malone [1981]. Engagement is achieved from two main functions in this application. The first comes from offering the user challenges. These challenges will analyse the user's ability to design machine instances and try to engage the user on a mental level. Mastery goals (i.e. challenges that have concrete objectives) have been showed to foster engagement and confidence amongst students compared to performance goals (i.e. challenges that utilise a scoring scheme) Ames and Archer [1988]. The other comes from allowing the user unrestricted access to these abstract tools. The freedom and ability to customise and configure machines allow the user to more effectively reason with these concepts.

Another key goal of an e-learning application is to communicate effectively the importance of the topics being conveyed. An example of achieving this would be to include real-world examples that build on these abstract ideas. There must exist a motivational justification for using this tool. Quinn argued that constructing a motivation is important for any educational application Quinn [1994]. This project's primary goal is to propose an application that can be incorporated and used in conjunction with the teaching of these concepts at university. The theories demonstrated must be well-established and general in scope. The notations must be natural, easy to express and, above all, accurate.

2.2 Objectives

The central objective is to engineer a system that is invisible to the user's experience. Too many buttons or obscure features will have negative consequences on user experience. In the paper *A Study of Educational Simulations Part I - Engagement and Learning*, several characteristics have been identified to encourage student learning. We will adjust this list only to contain components specific to this project.

An Engaging Approach: The application will encourage engagement through both interaction and exploration. As was discussed before, the more interactive an education application is, the easier it is to learn with (D. Bransford [2000]). The application will offer a series of challenges with varying difficulty. They will test the user's crafting ability. Neal [1990] mentions that goal formation is a motivating factor in any educational game. The user will be able to create their own machines or modify pre-existing machines and then simulate input on those machines. This will include the ability to visualise and control the execution of an input. For instance, the user will be able to go back or forward in the execution. For non-deterministic machines, where two or more transitions can be explored from a configuration, the user should be able to choose. This could be in the form of a dialogue box that allows the user to determine the next transition.

Constructivist In Nature: Von Glasersfeld [2012] stated that to solve a puzzle, an individual must contextualise the puzzle's internal structure first. They must be familiar with the obstacles that interfere with the progress towards a goal. Once a problem can be visualised, the learner has the capacity to come to a solution. The fact that a solution can continually be tweaked and tested serves as an excellent exercise for the learner. The student can comprehend how a solution was derived. If the user can be made to appreciate the process of designing machines, then the user will be more engaged and have an easier time understanding the underlying ideas. There is a problem-solving aspect of developing machines to solve certain challenges. In order to create machines confidently, the user must be comfortable with the ideas presented. In instances where concepts and notions are accepted universally, e.g. using the epsilon symbol to depict jumping transitions, the specification must adhere to those standards. The application must build on established knowledge of these theories so that the student can establish a strong familiarity with the tool. There must also be a number of additional resources to help students understand these concepts more quickly. For instance, a system that includes a set of well-known example machines with which the user can load and interact would be beneficial to the student.

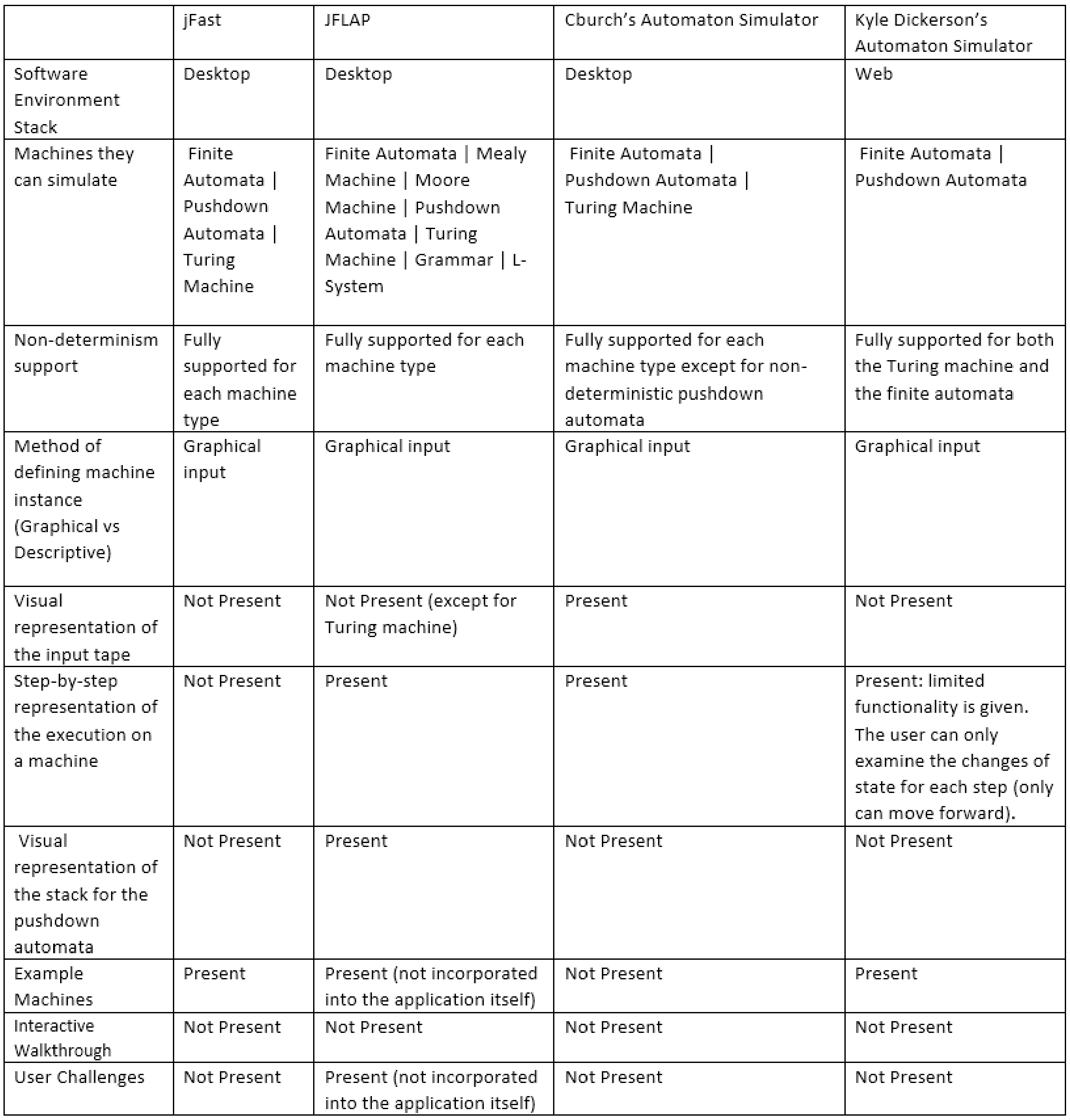
The utilisation of visualisation: Through animation and practical illustration, we will simplify these conceptual ideas into easy-to-understand representations. The theoretical machines will be portrayed as graphical objects that can change dynamically. This encourages students to understand these machines as if they were physical devices operating in the real world. The components specific to each automata machine must be visually represented, e.g. the push-down stack for a push-down automaton must be represented as a standard visual stack. The user must be able to visualise what the machine is doing at each step of the execution. The state of the device must dynamically change to match the current configuration during execution. The user must have control and be allowed to go forwards and backwards in the execution. The user should be presented with effective manipulation and definition capabilities. Exploration will be motivated through the use of visualisation and animation.

Usability is the most important aspect of the system from the viewpoint of the intended actor who needs to use the tool. The user must be able to reason effectively with the processes contained within it. Rigorous planning must be taken to avoid a counterproductive overload of the sensory channels.

An elegant interface which utilises well-known HCI principles is additionally important. The application must be intuitive to use. Failure in achieving this would nullify the entire system.

Also, additional features that make the system more versatile should be introduced. Only features that would significantly improve the user experience should be taken into account. In this regard, we make a distinction between useful features and gimmicks. A useful feature is one that works to increase the effectiveness of the core features found in this application. For instance, a feature that allows users to save machines to memory would be characterised as useful. It provides an incentive for students to return to the application by allowing them to start from a previous session. The interest comes in trying to limit the number of gimmicks. We want to keep the tool simple enough to allow learners to recognise the main ideas effectively.

2.3 Competitors



	jFast	JFLAP	Cburch's Automaton Simulator	Kyle Dickerson's Automaton Simulator
Software Environment Stack	Desktop	Desktop	Desktop	Web
Machines they can simulate	Finite Automata | Pushdown Automata | Turing Machine	Finite Automata | Mealy Machine | Moore Machine | Pushdown Automata | Turing Machine | Grammar | L-System	Finite Automata | Pushdown Automata | Turing Machine	Finite Automata | Pushdown Automata
Non-determinism support	Fully supported for each machine type	Fully supported for each machine type	Fully supported for each machine type except for non-deterministic pushdown automata	Fully supported for both the Turing machine and the finite automata
Method of defining machine instance (Graphical vs Descriptive)	Graphical input	Graphical input	Graphical input	Graphical input
Visual representation of the input tape	Not Present	Not Present (except for Turing machine)	Present	Not Present
Step-by-step representation of the execution on a machine	Not Present	Present	Present	Present: limited functionality is given. The user can only examine the changes of state for each step (only can move forward).
Visual representation of the stack for the pushdown automata	Not Present	Present	Not Present	Not Present
Example Machines	Present	Present (not incorporated into the application itself)	Not Present	Present
Interactive Walkthrough	Not Present	Not Present	Not Present	Not Present
User Challenges	Not Present	Present (not incorporated into the application itself)	Not Present	Not Present

Figure 2.1: The alternative applications [Patel, White, Burch, Dickerson]

There are a handful of pre-existing tools (Figure 2.1) that perform comparable features to the one being proposed. We will explore the main ones and examine their performance against our learning objectives.

jFast (Figure 2.2) exhibits the capabilities to simulate the primary devices. However, the execution process is not represented. There is no step-by-step demonstration of the execution process. The application should not be a system that only can determine recognisability. It should be one that also illustrates how the output is reached. This will include showing the state of each component at different stages of the execution. As previously stated a lot of the ideas around computation are demonstrated through the execution process, so it remains essential for this procedure to be represented.

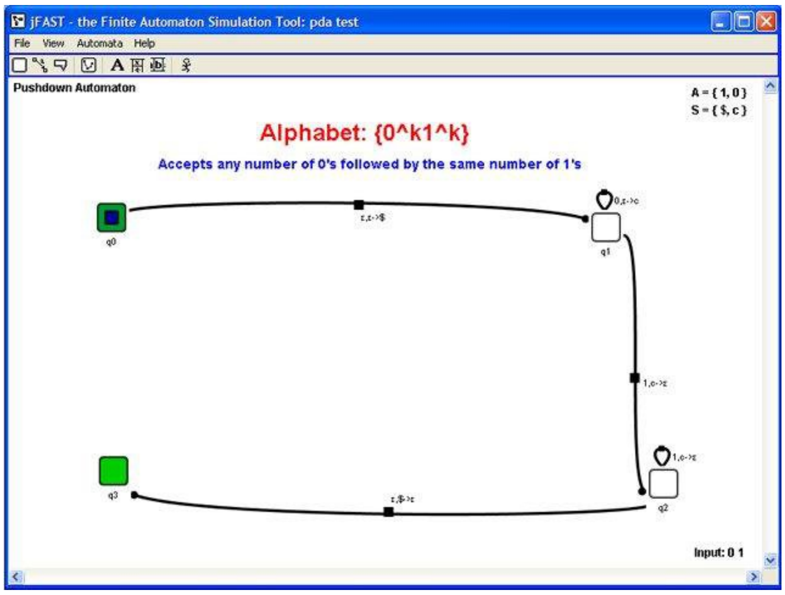


Figure 2.2: An FA machine instance modelled using jFast (Patel [2018])

Kyle Dickerson's automaton simulator in its current iteration can only simulate an FA or PDA. The exclusion of the Turing machine (TM) is not ideal. TMs can be utilised to communicate computational ideas that might not be apparent in other machines — for instance, illustrating the impacts of its bi-directional head on computation can be used to introduce the concept of memory-based computation to the user.

For Cburch's automaton simulator, only a deterministic pushdown automaton can be represented. For all machines that can be constructed using the application, the input alphabet (and stack alphabet) is automatically configured as having four elements (a,b,c,d). An illustration of this is seen in Figure 2.3. This restriction limits the user's ability to build complicated machines. The tools deployed must be versatile to encourage user exploration.

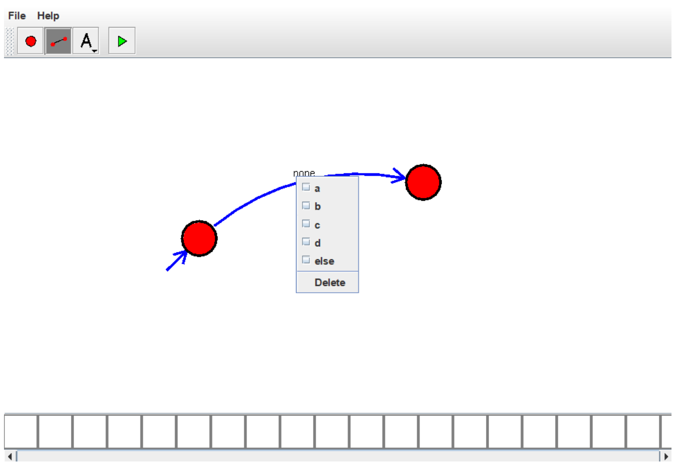


Figure 2.3: Setting the input element of an FA transition in Cburch Automata Simulator. (Burch [2001])

The graphical definition procedure for each application is complicated and unintuitive. The user should be able to create and modify machines easily. This is a requirement for this application to be useful. Additionally, there is an insufficient number of features that exist to promote learning. As a result, these systems do not feel like traditional e-learning tools.

2.3.1 A closer look at JFLAP

JFLAP (Figure 2.4) is the most established alternative. It is an open-source application that can simulate many of the abstract tools found in computational theory. These include L-systems and Grammars. For intensive purposes, we only need to consider the features that will overlap with the application being proposed. The user is able to simulate a number of different machines. The scope of JFLAP is broad, and there is no particular focus given to any tool. JFLAP provides the necessary functionality to create machines and simulate input on those machines.

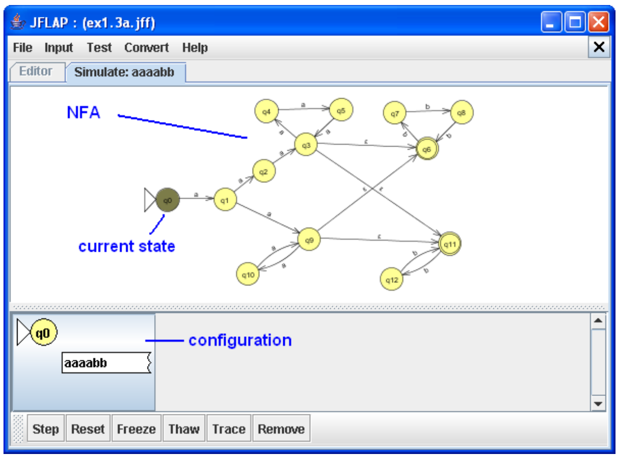


Figure 2.4: An NFA machine instance modelled using Jflap (Patel [2018])

JFLAP's lack of auxiliary features is problematic. There are no incorporated example machines that a user can load and interact with. Examples are an effective way for students to familiarise themselves with the system.

An interactive walkthrough of the simulation tool would be the best way of introducing a system to a new user. Without adequate instructions, many of the proposed features would overwhelm the user and limit the learning effectiveness of the overall system. The JFLAP website provides external documentation that describes how to utilise these tools. The documentation is not adequate and exists independently from the application.

The abstract machines must have a coherent and established style of operating to encourage user exploration. A significant part of this comes from establishing well-defined notations. There is no theoretical justification for many of the notations featured in JFLAP, and a lot of the symbols are not automatically evident to the user. For instance, in JFLAP, the epsilon transition is represented with the lambda symbol rather than an epsilon symbol.

A useful e-learning tool must enable the user to visualise each step of the execution. JFLAP does not represent the execution in any intuitive way. For example, the input tape, a component shared across all machines, is not represented as a visual component during execution. Additionally, It is not clear what transition is chosen for a particular step in the execution. The user can infer from the results, but this becomes challenging for complicated machines. The dynamic representation of each configuration in the execution and the changes that occur must be illustrated through expressive animation and precise illustration. A significant element of this e-learning system is visualisation, and it has been identified as being essential to helping convey abstract ideas.

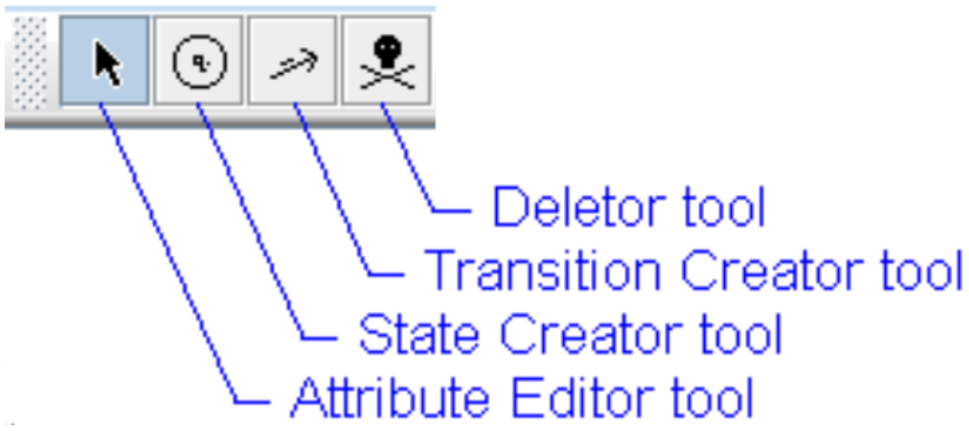


Figure 2.5: JFLAP's FA construction toolbar

JFLAP constrains the user to define machines graphically (Figure 2.5), i.e. dragging and dropping components (i.e. states and transitions) on a region. This process is tedious for the user. This is true for machines that operate with a high number of transitions. Another issue with this approach is that it distances the user from well-established ideas that persist in the automata literature. There already exists a formalisation procedure to define machines. Every automata machine has a prescribed definition that describes the components contained within them. A better approach would see the student explicitly define their machines, describing each component individually. This approach helps the user to become more familiarised with the components involved, and the role each plays in the machine.

2.3.2 Summary

In summary, every application provides the essential ability to simulate abstract machines. However, each fails at being a coherent system that a user can benefit from. The ambiguous representations, the lack of auxiliary features and the disconnect from key theoretical ideas are all limitations that play a significant role in making these applications ineffective. They are presented as operational tools rather than educational programs. There are few opportunities for freely analyzing, assessing and hypothesizing computational ideas. JFLAP is the most established alternative with a comprehensive list of features. The other options fail by excluding basic features or imposing unnecessary restrictions on the user. However, JFLAP does not feel like a regular educational program. The representations are confusing or lacking, and the system's overall educational effectiveness has been reduced by the inclusion of too many features - the effect of which leads to cognitive overload. There should be a strict focus on only representing a tiny handful of machines. They must be expressed effectively, demonstrating the underlying ideas for each.

Chapter 3

Design and Implementation

To produce an automata simulator which can be incorporated within a university context and successfully be used by students to learn with, the user interface must not be imposing, and certain features must be included.

In this section, we will outline these features. We will also examine the possible uses of this application.

The theories integrated into this system come primarily from the ideas covered in the textbook *Introduction To Theory Of Computation* Sipser [1996]. It is a well-established book that has been referenced in many educational programs and cited over 3800 times. The paper that inspired a lot of the design decisions is *A Study of Educational Simulations Part I - Engagement and Learning* Adams et al. [2008]. We additionally used the data study Adams et al. [2008] to inform the design process. The study involved 200 individual interviews with 89 participating students. It reviewed a set of tools (PHET) that were targeted to students in a science-related field. These tools are similar in nature to the application being proposed. The underlying ideas are just as abstract.

3.1 Machine Selection

As was discussed, automata theory is a vast subject and covers a range of computational devices. Several machines can be simulated in JFLAP. JFLAP, in including too many machines, has reduced its overall effectiveness as an e-learning platform. A design choice has to be made as to what machines should be simulated. There must be an inherent and obvious relationship to warrant devices being included together. The user must be able to extrapolate connections efficiently across them.

The computational machines to incorporate:

- Finite Automata (FA)
- Pushdown Automata (PDA)
- Turing Machine (TM)

Automata theory is the study of abstract machines and the computational problems that can be recognised by employing them. Within this context, a problem is categorised as a language, a collection of words collectively defined by a distinct pattern. A machine recognises a language when there is a machine instance that can identify the words that are contained within that language. From Figure 3.1, the different devices are represented within a Venn diagram. The nesting represents the variation of power. In this context, power represents the number of languages that an automata machine can recognise. The power differential that exists between the various devices serves to highlight the fundamental computational ideas they present. For instance, why introducing a head that can move in both directions, a feature only found in Turing machines, can allow for a machine to have more power compared to one where the head can only move in one direction. An effective demonstration of these machines can encourage students to discover ideas independently. Automata machines themselves are tools that are used to illustrate computational concepts. That is the primary reason for their inclusion in a majority of undergraduate curriculums.

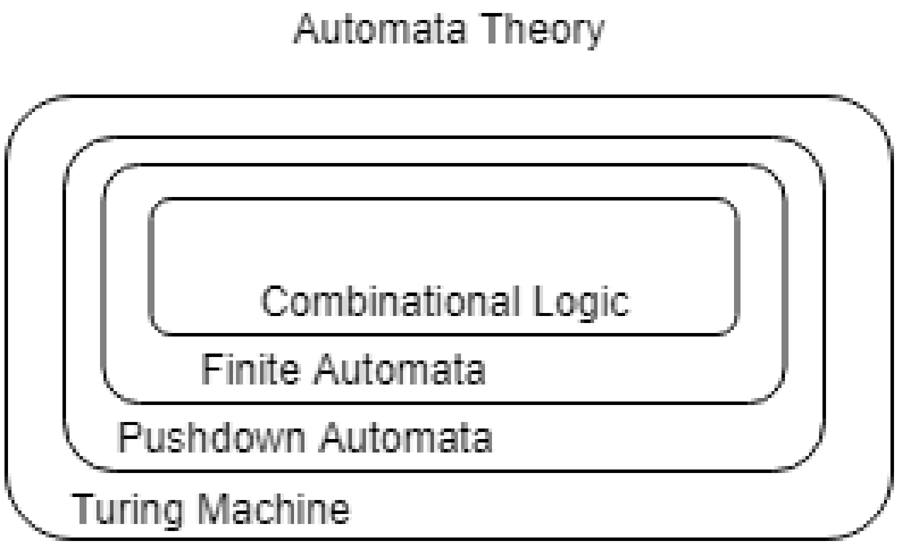


Figure 3.1: The class of automata machines.

3.2 Technology Selection

The choice of the environment plays a key role in deciding how this system is used. A desktop solution was identified as being the most optimal. It will enable the application to operate within a school context without any restriction.

Advantages
+ Does not require internet
+ The application can be ported easily across devices
+ Much faster than the equivalent web application

We also analysed mobile and web solutions and examined their limitations.

Mobile Native Approach

Advantages
+ Open application market (readily accessible)
+ Much faster than an equivalent web application
+ Only requires internet at installation-time (offline use)
+ More people have mobile phones then computers or laptops (pew [2019])

The mobile native approach was identified as not being practical. Many of the ideas communicated in this application are done through visualisation. Smartphone display sizes are often inadequate, and representing these simulations on small layouts will reduce the overall effectiveness of the visuals.

Additionally, a lot of the features in this application require user input. For example, building a machine requires the user to explicitly define the states and transitions (i.e. via keyboard input). Presenting this type of feature within a mobile environment would come with several challenges. The mechanism for providing user input on smartphones is not ideal (Page [2013]).

Another challenge comes in there being no agreed way of compiling apps on smartphones. The process entirely depends on the vendor of the operating system. The Android operating system requires apps to be written in Java and IOS requires them to be written in Swift. The application would need to be implemented on both.

Web Approach

Advantages
+ No hardware or software prerequisites
+ Only one version of the application is available
+ Can provide support for mobiles devices
+ No downloads required

An internet connection is necessary to launch this application on a browser. This would stop offline use of the program, and since the application itself requires no internet, the availability of the system would be hindered.

Additionally, a web application is slower than an equivalent desktop application. This becomes primarily true for systems that comprise of many components that operate on communication. As a result, the development process would become more challenging.

APIs and Libraries

In developing this application, Java was an ideal choice. It contains a range of well-written libraries to satisfy the requirements found in this project. A program written in Java can also be made to run on multiple different operating systems.

JavaFX is deployed in this project as well. JavaFX is an API which consolidates both CSS and XML with Java to allow developers to create high-level GUIs. It introduces a collection of form elements, that provide the necessary form-handing capabilities needed to define machines. It also provides the vital drawing mechanisms needed to represent machines graphically.

We did not incorporate many libraries. This was done deliberately to keep the application lightweight and portable across devices. Gson has been used to facilitate object serialisation within Java. It provided the necessary procedures to save and load machines. We also incorporated Junit to carry out our unit and integration testing.

This application is presented as a JAR file. The JAR file combines the dependencies, the class files and resources into a bundle that can easily be run. The user will only require the JAR file to run the application.

3.3 Development Procedure

The agile model has been identified in the past to be effective in developing e-learning systems Chun [2004]. The agile approach encourages working closely with the customer. The customer in this context is the student. This frequent interaction during development allows us to find ways of making the application more effective as an educational tool.

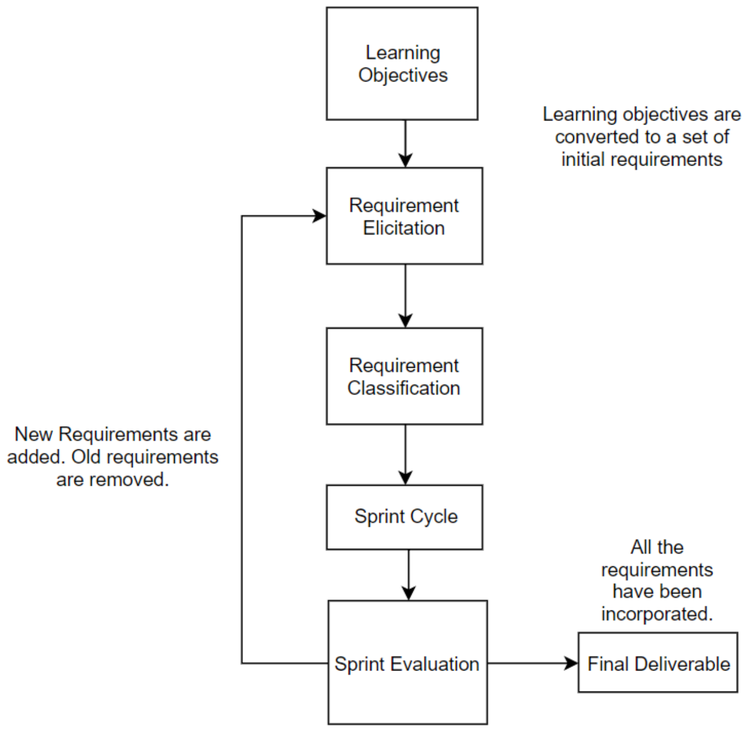

Learning Objectives
Learning objectives are converted to a set of initial requirements
Requirement Elicitation
Requirement Classification
New Requirements are added. Old requirements are removed.
Sprint Cycle
All the requirements have been incorporated.
Sprint Evaluation
Final Deliverable

Figure 3.2: The design process.

We start by codifying the requirements into a list. As discussed, the pedagogical objectives take precedence in this application. A priority score is assigned to each requirement. Each requirement is examined against a set of learning objectives. Once all requirements have been finalised, they are grouped into ordered sets, where each set is allocated to a sprint cycle. At the end of each cycle, we evaluate the success of a sprint.

3.4 Requirement Specification

In this section, we codify the requirements of the application into a concise table. This construction involved gathering information on the domain by analysing previous related works. Each requirement is awarded a priority score. The priority score represents how closely a feature aligns with the main aims of the project. An informed choice on the pedagogical functions to include had to be made. Overwhelming an educational tool with too many features would only make the application less effective.

The learning outcomes a feature can be classified under:

- Engaging: a feature that requires the user to think and interact actively with the tool
- Explorative: a feature which gives the user the freedom to experience an aspect of the tool however they wish
- Visual: a feature that illustrates to the user an aspect of the program
- Descriptive: a feature which serves to covey the ideas covered in the literature
- Constructivist: a feature which actively allows the user to construct their own learning experiences
- Useful: a feature which supports the core functions in the program

	Requirement	Specification	Learning Outcomes Achieved	Priority
R1	The user can graphically define their machines	This feature would allow the user to start with a blank machine (no states) and iteratively modify it by dynamically adding states and transitions. The user is allowed to experience the impact of their modifications by running input and examining the results.	Explorative, Visual, Constructivist, Engaging	High

R2	The user can descriptively define their machines in a systematic fashion	This feature would allow the user to construct machines using a form which lays out the various components in a systematic fashion. This would match the automata machine definition format found in the literature. Form validation would also be incorporated at this stage to identify any possible confusions the user might have.	Descriptive, Engaging	High
R3	The user should be able to both save and restore machine instances from memory	This feature would allow the user to save their definitions to memory. The user could then be able to load these machines instantly. This feature would give the user an incentive to return to the application by allowing the user to start from a previous session.	Useful	Medium
R4	The user should be provided with information on how to use the system most optimally	The user can access theoretical and instructional information concerning the individual components involved in creating an automata machine. This information would be provided to the user during the definition process.	Descriptive, Useful	High
R5	The user should be able to create their TM, FA and PDA instances	The application would allow the user to define their machines. Each definition page and procedure is unique. The simulator would be able to represent each machine type. The relationship of power that persists across the different machines will be used to convey computational ideas.	Explorative, Engaging, Visual, Descriptive	High

R6	All the important components belonging to each machine type must be represented to the user visually. The representations must be utilised during execution	The simulator would be able to represent all machine elements visually. These will include the distinct components only found on certain devices, e.g. the stack for the PDA is represented as a visual stack. The components must dynamically change during the execution to convey the state of the machine.	Visual, Descriptive	High
R7	The user should be able to run input on a machine and receive the output	This feature would allow the user to simulate input on a machine instance. The application would prompt the user with the results. The solutions found during the execution are collected and shown to the user.	Descriptive, Useful	High
R8	The user should be able to visualise the machine's execution process	This feature would allow the user to view a step-by-step representation of the execution process. It would illustrate how the output is reached. This includes showing the state of each component at different stages of the execution. The transition that was chosen is highlighted for the user, and an information panel is dynamically updated to report the state of the machine. It must be obvious to the user what the current configuration of the machine is.	Visual, Descriptive, Constructivist, Engaging	High
R9	The user should be able to control the execution process for a particular input	This feature would allow the user to control the execution process. They can go back to a previous configuration to replay part of the execution, progress to the next configuration or stop the execution. The user is given the freedom to explore the execution procedure.	Explorative, Descriptive, Constructivist, Engaging	High

R10	The user should be able to load example machines and run input of them	This feature would allow the user to load pre-defined machines into the simulator instantly. The set would consist of a series of well-known models (i.e. ones that appear in the automata literature). There would be examples available for each machine type. Once an example is loaded, the user can modify or/and save the machine to memory.	Useful, Descriptive, Visual	High
R11	The user should be able to modify existing machines and save those to memory	The user would be able to reconfigure machines by adding/removing states and transitions. This will encourage learning through interaction and exploration.	Explorative, Engaging, Constructivist	Medium
R12	When the machine is in a loop for a run, the application should intervene and alert the user.	Whenever the machine is in a loop for a quick run procedure, after a defined number of configurations, the application should pause the execution and prompt the user on what to do next. The user can then either continue the run or can enter step-run interactive mode to view the cause of the loop. This would load the state of the current configuration into the simulator, and the user could then manually continue the execution.	Engaging, Visual	Medium

R13	In the case of non-determinism, the user should be able to choose the next transition	In the step run mode, whenever there are two or more possible transitions (i.e. non-deterministic transitions) that can be taken, the choice should be left to the user. The user should then be able to replay that decision when returning to that configuration. A track of what transitions have been visited must be maintained to encourage the user to explore alternative unvisited paths.	Explorative, Constructivist, Engaging	Medium
R14	The user should be able to view all non-deterministic transitions present in a machine	The visual transitions that satisfy the property of being non-deterministic are highlighted to the user whenever the user prompts the application for this feature.	Visual, Descriptive	Medium
R15	The user should be able to view a graphical representation of the machine that is currently loaded	The application would generate visual illustrations for each component. It would also dynamically change to match the further modifications made to the machine.	Visual	High
R16	The user should be able to partake in solving challenges (design automata machines of a certain type based on description)	This feature would provide pre-defined challenges that the user can partake in. These challenges analyse the user's ability to design machine instances and would try to engage the user on a mental level. The user would construct machines and submit them. Only when a submission passes a set of words will the submission be accepted as a solution. The challenges become progressively harder.	Engaging, Explorative, Constructivist	high

R17	The user should be able to save their challenge progress (amount of challenges they completed and their last attempt) and their challenge solutions.	This feature would allow users to save their current attempts. Whenever the user completes a challenge, the submission used to solve it is automatically saved, and a record is kept of the progress. This feature would give the user an incentive to return and continue their challenges.	Useful	High

3.5 Overview of Application

3.5.1 Lecturer use

The application is packaged to be used by two primary principals. The first is that of a lecturer teaching automata theory in a classroom setting. This scenario would require the student to be given an overview of the automata concepts first. The lecturer can then incorporate the simulator as part of the lesson plan. The program in this context would primarily be used to reinforce taught ideas and produce opportunities to develop more complex ones. The application offers the lecturer the ability to teach these conceptual processes of operation interactively. The lecturer takes on a primary function in the learning process and conveys the information directly to the pupils. The application would be used in combination with the teaching to achieve this.

One of the ways a lecturer could incorporate this tool is by utilising the examples found in the application. The models are prepackaged and tested machines that the user (i.e. the lecturer) can load and run. They are devices that are emphasised in the automata literature. They are simple enough not to overwhelm and challenging enough to engage. The lecturer can load an example machine into the simulator and run input on that machine. The lecturer is able to visualise the execution procedure and partake in a step-by-step walkthrough of the execution. The lecturer can construct a lesson plan around analysing the application's execution process. The user (i.e. the lecturer) is given full control of the execution process and can go forward and backwards in the execution. A visual illustration of each element in the machine is represented, and each is dynamically changed to match the state of the execution. This approach would be ideal for communicating across when a machine accepts an input versus when it does not. By demonstrating both scenarios on an example machine, you can establish a clear distinction in the student's head. This application is a tool that primarily aids the lecturer in explaining abstract concepts that are difficult to teach using conventional methods. The pictorial representation of computation within the application enables the lecturer to demonstrate first-hand, the key computational ideas.

Another application could be to construct group-based exercises around the use of it. The students could be encouraged to assemble a machine collectively. Working in groups has been shown to engage students and improve communication and decision-making skills. The students could participate in solving progressively harder challenges within the application. The challenge feature would allow for the user (i.e. the lecturer) to start with a blank machine (no states) and iteratively modify it by dynamically adding states and transitions. The graphical representation of the current machine is updated dynamically. The impact of modifications can be immediately experienced by running input and examining the results. The group of learners actively cooperate in their learning experience. This is a constructivist approach to teaching. The lecturer as before could incorporate this approach into a standard lesson plan. It would serve as a test that could evaluate the students level of understanding for taught concepts. It would also serve as an effective way of reinforcing constructs inside the student's head. Black and Wiliam [2010] demonstrate that the utilisation of formative assessments within teaching leads to students retaining twice as much.

A combination of both approaches would be most optimal. The lecturer would start by demonstrating how to construct machines (i.e. instructivist approach), and then the students would be able to implement what was communicated by constructing machines and solving challenges collectively (i.e. constructivist approach).

The introduction of the save feature allows the user (i.e. the lecturer) to construct and package examples that can be loaded up at a later date. These examples can be abstractions of computational ideas. For instance, to convey non-determinism, the lecturer might construct a machine with non-deterministic transitions and utilise it to teach with. The visual execution process would serve as an effective way to show non-determinism to students.

3.5.2 Student use

The second principle is of a student with a basic knowledge in automata theory. This application does not intend to be a replacement for the curriculum taught at universities. It primarily reinforces concepts by giving the user the ability to expand on their current understanding. It accomplishes this by introducing a collection of features that motivate and encourage learning.

One of the main features of the application is the ability for the user to create or modify automata machines and simulate input on those machines. It will provide the opportunity for the learner to hypothesise, examine, evaluate the computational ideas that are represented within. Constructivism makes the point that learners create their own understanding through active interaction. The user would have the freedom to develop an idea (i.e. the machine) and evaluate its effectiveness immediately after (i.e. simulate input on that machine). The user continuously learns from the experience of modifying a machine and assessing the outcome. This constant cycle of results empowers the learner to establish causal links between how a device is constructed and how a device operates.

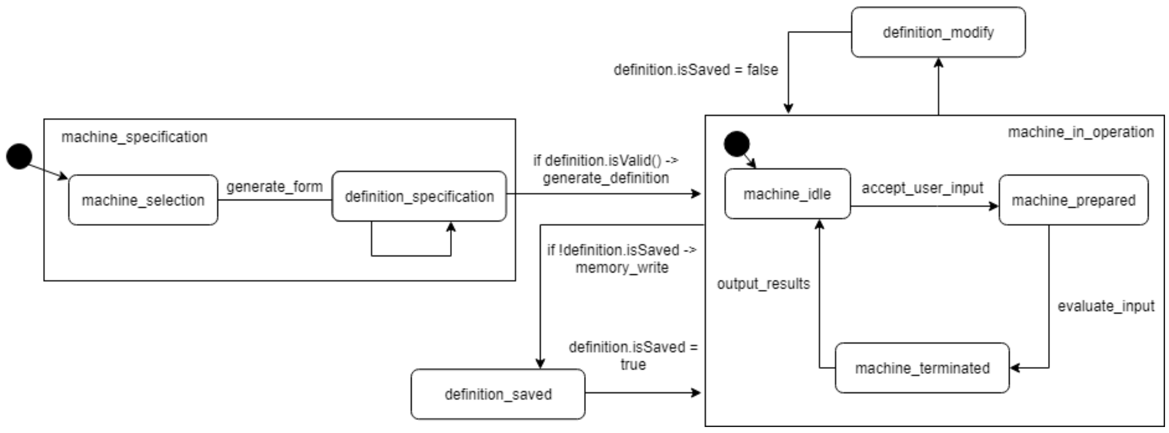


Figure 3.3: A general state machine that covers construction and simulation

Figure 3.3 illustrates the process of creating and operating machines. The Definition instance is what is generated when the user defines their machine. The simulator uses the Definition instance to construct the machine.

The user begins at the machine_list view. The user chooses the machine of their choice. Once the device is chosen, the required form is generated for the user. This view contains the necessary components that are needed for that machine type. The form walks the user through constructing a machine. The form includes both theoretical and instructional information concerning the individual components associated with creating that automata machine. Error handling is also incorporated to ensure that the user has the correct understanding when constructing these machine.

Once the user is satisfied with their machine, the application verifies the Definition instance. In the case of an error, the user is prompted to correct it. The user has an opportunity to reflect and learn from their misunderstanding. If the construction is valid, then the Definition instance is generated and loaded. From there, the user is free to simulate and evaluate execution. While the Definition instance remains active on the simulator page (i.e. on the screen), it can be saved to memory. Once saved, it can be restored at a later time.

Additionally, a user can modify their machine after it has been generated, e.g. change the states or transitions of the Definition instance. Each change triggers the simulator to rerender the visual representation of the automata machine. The user would need to save their device to persevere the changes made.

Another way for the students to experience automata machines and the ideas contained within them are through examples. The example feature allows the user to load prepackaged and tested machines without the need to define mechanisms first. The definition procedure is a vital aspect of the system. The hands-on construction of machines can allow students to identify the role of the device’s constituent parts more effectively. Examples are ideal for learners that want to focus on the computational aspects of the application. The models are obtained from the automata literature. They are used specifically to represent and express how automata machines work. It is crucial for there to be a comprehensive collection of examples. This is so not to encourage learners to link a particular instance with the meaning of a concept.

The user can also modify example machines to behave differently. This function encourages experimental learning. It is achieved by establishing a foundational groundwork of understanding (i.e. the example machine) and then giving the user the freedom to build off that foundation. If the user can first experience how an example machine operates, they can make their own modifications and evaluate the changes by simulating input. Examples also provide a good introduction for beginners. Examples give exposure to the central features found in this application.

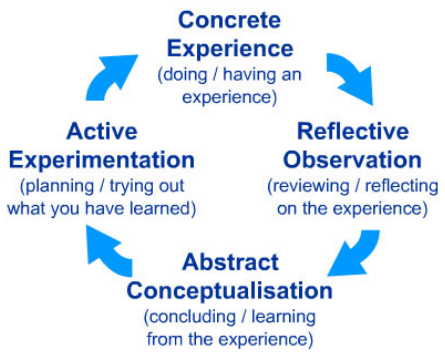

Concrete Experience
(doing / having an experience)
Reflective Observation
(reviewing / reflecting on the experience)
Abstract Conceptualisation
(concluding / learning from the experience)
Active Experimentation
(planning / trying out what you have learned)

Figure 3.4: The learning cycle was first proposed by Kolb [1984]

A different way for a student to experience automata machines and the ideas contained within are through the challenges. The user (i.e. the student) starts with a blank machine (no states) and iteratively modifies it by dynamically adding states and transitions. The graphical representation of the current machine is continuously updated to match these changes. The impact of modifications can be instantly experienced by running input and examining the results. As before, the user develops an idea (i.e. the machine) and evaluate its effectiveness immediately after (i.e. simulate input on that machine). The user continuously learns from the experience of modifying a machine and assessing the outcome. The challenge component incorporates the opportunity to evaluate. The solution at submission is simulated against a series of input words. The inputs that deliver an incorrect output are shown to the user. From there, the user is able to simulate each unsuccessful input on the machine and examine the execution procedure. Once the problem is identified, the user can modify their solution accordingly and try again. The user has the opportunity to analyze and revise their solutions if necessary. Only when all outcomes are correct can the solution be accepted as valid. This procedure follows from David Kolb's learning cycle (Figure 3.4). Learners develop their existing knowledge to reach deeper levels of understanding.

Automata theory is self-contained, and a lot of the computational ideas are called upon when constructing these machines. The challenge feature can be used as a mechanism to reinforce previously taught concepts or establish newer, more complex ideas.

3.6 General Structure

In this section, we will give a brief overview of the system's structure. Specifically, we will cover how the system's components combine to form the application.

The objective was to build a robust application which took into account the complexities of the system. It was identified from the beginning that a compact design structure had to be constructed to prevent poor system design.

This was achieved primarily with the use of the Model-View-Controller (MVC) design pattern. A design pattern is a refined and reusable strategy that can be employed during development to solve a particular problem. MVC (Figure 3.5) provided the necessary abstractions between classes. It established a clear separation between the view classes (i.e. what the user sees), the model classes (i.e. the business logic) and the controller classes (i.e. the classes responsible for handling user interaction and overseeing component communication).

For each distinct machine component, there was a corresponding FXML view class, a controller class which handled the user interaction and communicated the model's state to the view, and a model class which represented the internal state of that component. An example representation of this can be seen in Figure 3.6. The apparent decoupling between the view and model improves the maintainability of the code. The abstraction allows for a machine to run input in the background without needing to invoke the foreground (i.e. the visual simulator itself) — a necessary property for Quick-run.

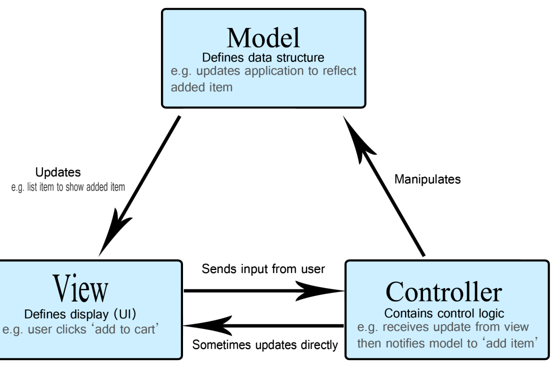

Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

Figure 3.5: A basic model-view-controller representation. MVC [2018]

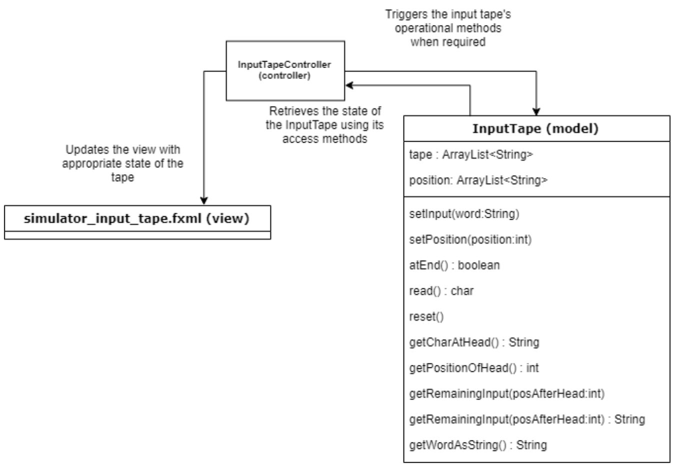

Triggers the input tape's operational methods when required
InputTapeController (controller)
Retrieves the state of the InputTape using its access methods
Updates the view with appropriate state of the tape
simulator_input_tape.fxml (view)
InputTape (model)
tape : ArrayList<String>
position: ArrayList<String>
setInput(word:String)
setPosition(position:int)
atEnd() : boolean
read() : char
reset()
getCharAtHead() : String
getPositionOfHead() : int
getRemainingInput(posAfterHead:int)
getRemainingInput(posAfterHead:int) : String
getWordAsString() : String

Figure 3.6: A model-view-controller representation of an individual component of the system.

The practical application of code reuse allowed us to save time and decrease redundancy by isolating reoccurring functionality across classes. Due to the similar nature of the machines, MVC was an ideal approach to accomplishing this. For example, the InputTape model class is used in several different locations in the application. By encapsulating both the functionality and state of an input tape within a model class, it can easily be accessed multiple times.

We also introduced the use of utility classes. HelperFactory and UIFactory are classes that contain functions that are required in several locations in the code. Instead of defining functions in classes that need to access them, they can be implemented as static methods in the utility classes. Since these methods are static, all classes have access to them. Whenever a utility method needs to change, the body only needs to be modified in a single location. Utility classes are essential in removing redundant code and improving maintainability.

We will only consider documenting model classes. The controller, for the most part, only triggers self-contained processes within the model.

3.6.1 Model

As was discussed before, the model contains the business logic for this application. It encapsulates the functionality of the different machines and their elements. Each machine has a corresponding class which is responsible for simulating it. For instance, the FiniteAutomata class provides the state and behaviour required to simulate a finite automaton. This machine instance has access to several structures and a definition instance that describe its internal construction. It is the machine's responsibility to organize the various elements during execution. This is primarily done by establishing a channel of communication amongst them. Below we outline the essential model classes involved in this process and the role each plays.

InputTape

This class is responsible for simulating an input tape component. An InputTape object is created with each machine instance. It is a wrapper class that encapsulates an ArrayList of characters (i.e. the state of the tape) and a head position that represents the head. It defines a collection of access methods which are important for documenting the input tape's state and relaying information to the view. It operates like a standard input tape. The *loadInput()* method loads an input word onto the tape ready for processing. It defines a *read()* method which reads the symbol at the tape's head and increments the head pointer by one. It also defines *setPosition(newPos:integer)* which sets the current head position. This method is employed when a machine instance needs to backtrack to a previous configuration in the computation. We keep track of the head positions of older configurations and can load them dynamically. This InputTape also incorporates methods that are only invoked in the TuringMachine class. For a Turing machine, the tape can move in both directions and can be modified (i.e. written to). To simulate this, we incorporate null values in the *tape* to denote empty cells. We dynamically modify the content by shifting the ArrayList. *empty()* is a method which is used by a Turing machine object to evaluate when a tape is empty

ControlState

This class is responsible for representing a control state component. It is a wrapper class which encapsulates the properties of a control state. It holds a *name* field and two boolean fields, *isAccepting* and *isInitial*. The ControlState class is used by Machine instances to carry out computation. They are also used to represent the control states within the Definition class.

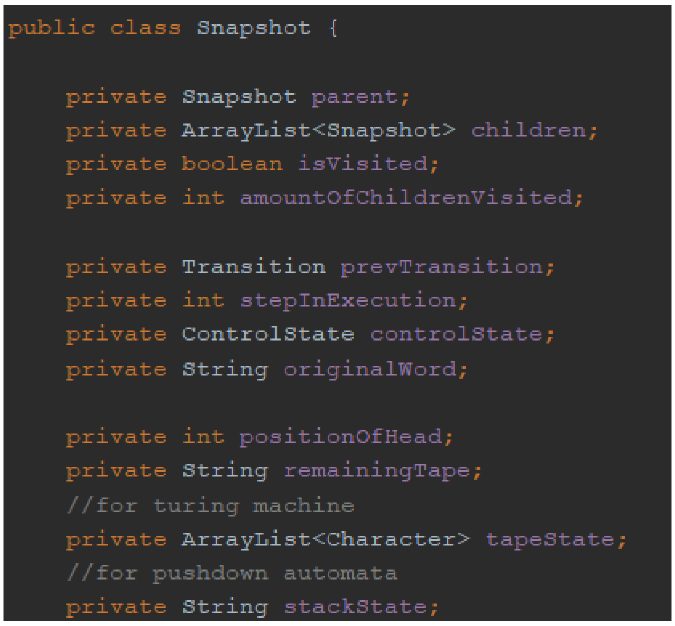
Transition

This class is responsible for representing a transition component. It is a wrapper class that encapsulates the properties of a transition. Each automata machine defines transitions differently. For instance, the PDA defines a transition as including the element to pop and the element to push. Since the other devices do not incorporate a stack, the transition formats for a TA and FA do not require these. It was decided to have a single Transition class to represent all of the different formats. A Transition instance would be identified by the *type* field which would define the kind of transition, e.g. a transition instance for a FADefinition would be represented with the type 0. The Transition class defines a separate constructor for each machine type.

Snapshot

This class encapsulates the state of a machine at a particular configuration in the execution. It is a wrapper class that encapsulates the tape's head position, the current ControlState instance, the step in the computation and other necessary values (Figure 3.7). A Snapshot instance for a

PushdownAutomata machine would also contain the stack's state. From a current configuration (i.e. a Snapshot instance), a collection of possible configurations are generated for each transition that can be taken.



```
public class Snapshot {

    private Snapshot parent;
    private ArrayList<Snapshot> children;
    private boolean isVisited;
    private int amountOfChildrenVisited;

    private Transition prevTransition;
    private int stepInExecution;
    private ControlState controlState;
    private String originalWord;

    private int positionOfHead;
    private String remainingTape;
    //for turing machine
    private ArrayList<Character> tapeState;
    //for pushdown automata
    private String stackState;
```

Figure 3.7: The Snapshot class

The Snapshot instance operates as a node within an execution tree. The Snapshot maintains a list of Snapshot children nodes and a Snapshot parent node. In the case of non-determinism, several possible branches can be taken. The criteria for rejecting input in a non-deterministic machine is that all possible branches must fail. The tree is used to keep track of the execution. A Snapshot instance also maintains an isVisited field to mark when a configuration has been visited. Backtracking is necessary when the current path does not result in an accepting configuration. On backtrack of the tree, the machine instance must be able to keep a memory of what branches have already been explored.

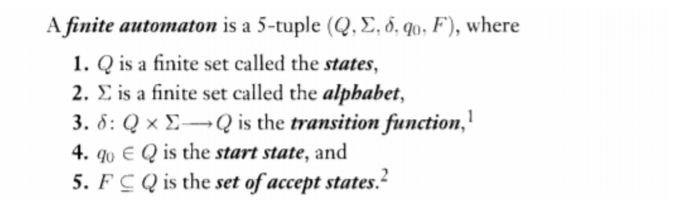
PushdownStack

This class is responsible for simulating a stack component. It is a wrapper class which encapsulates an ArrayList of string characters. It defines a collection of access methods which are important for both documenting the stack's state and relaying information to the view. *top()* is a method which returns the top element of the stack. The *setStackState(String stackstate)* method loads the stack with state. It works by breaking the input down from left to right into an array and loading each onto the empty stack. The PushdownStack is not a conventional stack data structure. It does not define *pop()* or *push(String charToPush)*. The stack's state is mainly controlled by the PushdownMachine class via the *setStackState(String stackState)* method. The Snapshot stores the stack's state. When a configuration is entered, the machine retrieves the *stackState* from the Snapshot and loads it into the PushdownStack. *push()* and *pop()* are not necessary since we generate the Snapshot instances before we need to explore them (i.e. in the previous configuration). The PushdownStack instance is only used in the interest of representing the current stack's state.

Definition

The Definition class is a blueprint of a machine. It represents the internal structure of a Machine instance. There is a separate definition class for each machine type. FADefinition for finite automata, PDADefinition for the pushdown automata and TMDefinition for the Turing machine. The Definition is what is generated when the user defines their machine. The simulator uses the Definition instance to construct the machine onto the view.

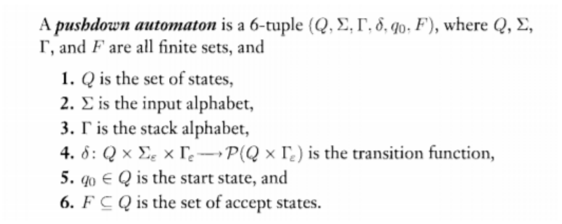
FADefinition - This class encapsulates the formal description of a finite automaton. It is a wrapper class that contains the Transition instances, the ControlState instances and the initial ControlState instance. It utilises the literature's definition of the FA (Figure 3.8).


A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta \colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,[1]
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.[2]

Figure 3.8: The finite automata definition as defined in *The Introduction To The Theory Of Computation* Sipser [1996]

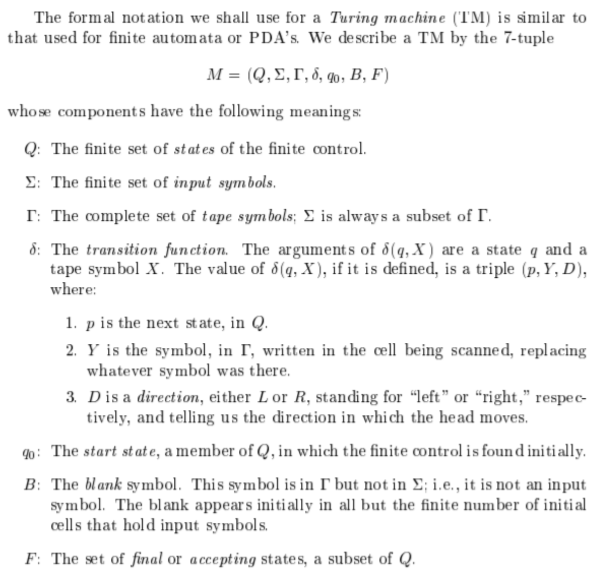
PDADefinition - This class encapsulates the formal description of a pushdown automaton. It utilises the literature's definition of the PDA (Figure 3.9). It also includes an *isAcceptByFinalState* field. This field is in charge of determining the machine's acceptance criterion. Since there is only two possible acceptance criterion for the PDA, a boolean field is enough to represent both.


A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Figure 3.9: The pushdown automata definition as defined in *The Introduction To The Theory Of Computation* Sipser [1996]

TMDefinition - This class encapsulates the formal description of a Turing machine. It is a wrapper class that contains the Transition instances, the ControlState instances and the initial ControlState instance. It utilises the literature's definition of the TM (Figure 3.10).



The formal notation we shall use for a *Turing machine* (TM) is similar to that used for finite automata or PDA's. We describe a TM by the 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

whose components have the following meanings:

Q: The finite set of *states* of the finite control.

Σ: The finite set of *input symbols*.

Γ: The complete set of *tape symbols*; Σ is always a subset of Γ.

δ: The *transition function*. The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:

1. p is the next state, in Q.
2. Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.
3. D is a *direction*, either L or R, standing for "left" or "right," respectively, and telling us the direction in which the head moves.

q_0: The *start state*, a member of Q, in which the finite control is found initially.

B: The *blank* symbol. This symbol is in Γ but not in Σ; i.e., it is not an input symbol. The blank appears initially in all but the finite number of initial cells that hold input symbols.

F: The set of *final* or *accepting* states, a subset of Q.

Figure 3.10: The Turing machine definition as defined in *Introduction to Automata Theory, Languages, and Computation* Hopcroft et al. [2006]

Machine

The Machine interface declares the essential methods that an automata machine must contain. All individual machine classes must implement this interface. This relationship is represented in Figure 3.11.

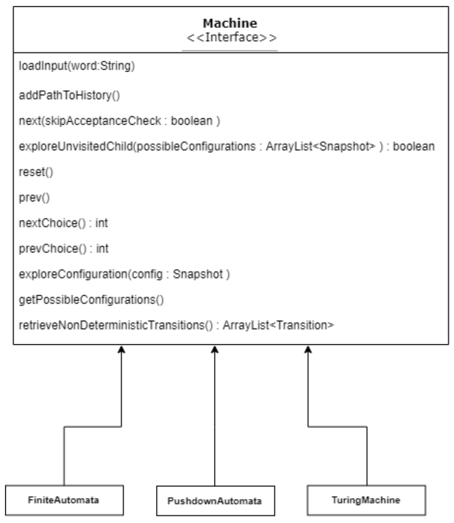

Machine
<<Interface>>
loadInput(word:String)
addPathToHistory()
next(skipAcceptanceCheck : boolean)
exploreUnvisitedChild(possibleConfigurations : ArrayList<Snapshot>) : boolean
reset()
prev()
nextChoice() : int
prevChoice() : int
exploreConfiguration(config : Snapshot)
getPossibleConfigurations()
retrieveNonDeterministicTransitions() : ArrayList<Transition>
FiniteAutomata
PushdownAutomata
TuringMachine

Figure 3.11: The Machine interface's relationship with the other automata machine classes

The introduction of a universal interface enables us to decouple what the automata machine is from how it does it. There are several automata machines that are featured in this application. They share a lot of the same operations. Much of the code that can be implemented in one machine can be reused in the same way for the others. The interface establishes a general reference type that we utilise to build generic methods.

Fields

Each instance has access to its constituent elements and a definition instance which describe its internal construction.
Additional fields to keep track of the execution :

- **Snapshot currentConfig** - This object encapsulates the state of a machine (i.e. any machine class) at a particular configuration in the execution. It is a wrapper class that encapsulates the tape's head position, the current ControlState instance, the step in the computation and other necessary values. When the machine is not in operation, it is set to null.
- **ObservableList<ArrayList<Snapshot» history** - A list which keeps a track of all the recognised solutions in the current session. An execution might have more than one possible solution due to non-determinism. *history* is used as a means to keep a track of what solutions have already been discovered and ultimately allow for newer solutions to be found.

Methods

- **loadInput(String word)** - A method which loads input into the machine instance. It starts the execution off by creating an initial Snapshot to represent the starting configuration. This Snapshot holds the initial ControlState instance of the machine.
- **getPossibleConfigurations()** - A method which returns the possible configurations that can be explored from the current Snapshot. It is defined in the form ArrayList<Snapshot>. This list is generated by applying each possible transition to the current configuration and generating for each a resulting configuration Snapshot. Each transition consists of two parts: the configuration (i.e. the prerequisite of the machine) and the action (i.e. the impact on the machine). The possible transitions that are retrieved from the Definition instance must satisfy the current configuration of the machine.

- **exploreConfiguration(Snapshot config)** - A method which takes a configuration Snapshot and loads it into the current machine instance. This Snapshot encapsulates the state of a machine at a particular configuratioan in the execution. It is a wrapper class that encapsulates the tape's head position, the current ControlState instance, the step in the computation and other necessary values. This method is essential for exploring newer configurations and prior ones that occur as a result of backtracking.
- **exploreUnvisitedChild(ArrayList<Snapshot> possibleConfigurations)** - A method which takes a list of possible configuration snapshots and explores the first one that has not been visited.
- **addPathToHistory()** - This is a method which adds the current sequence of Snapshot instances to the *history* field. The execution sequence can be extracted from the machine by exploring the current Snapshot's ancestors via the parent field. *addPathToHistory()* checks that the sequence of Snapshot instances is not present in *history* before adding it.

- **next(boolean skipAcceptanceCheck)**- A method which is responsible for retrieving the possible configurations via the *getPossibleConfigurations()* method and setting it to the current Snapshot's children set. An illustration of this is represented in Figure 3.12. If there are no possible configurations (i.e. no transitions can be taken), then the method returns the result code *zero* to indicate this. For each next() call, the acceptance criterion is checked. If the condition is satisfied, then *one* is returned. In all other cases, the method returns *two*.

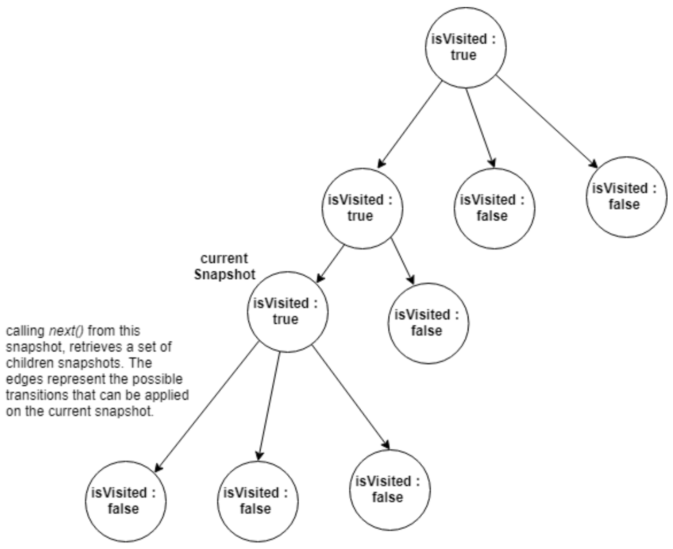


Figure 3.12: A visual representation of *next()*.

- **prev()** - A method which loads the previous configuration. It does this by retrieving the current Snapshot's parent and loading it into the machine via *exploreConfiguration(Snapshot config)* . An illustration of this is represented in Figure 3.13.

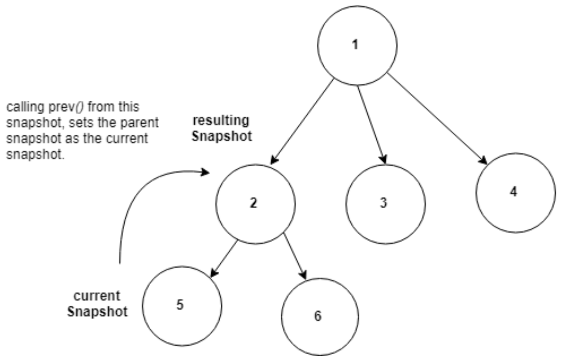

calling prev() from this snapshot, sets the parent snapshot as the current snapshot.
resulting Snapshot
current Snapshot
1
2
3
4
5
6

Figure 3.13: A visual representation of *prev()*.

- **nextChoice()** - A method which explores each subsequent configuration from the current until a configuration is found where its children size exceeds one. This occurrence represents the case of non-determinism where more than one possible configuration can be produced from the current configuration. *nextChoice()* repeatedly calls *next()* and *exploreUnvisitedChild(ArrayList<Snapshot> possibleConfigurations)* in a loop. An illustration of this procedure is represented in Figure 3.14.

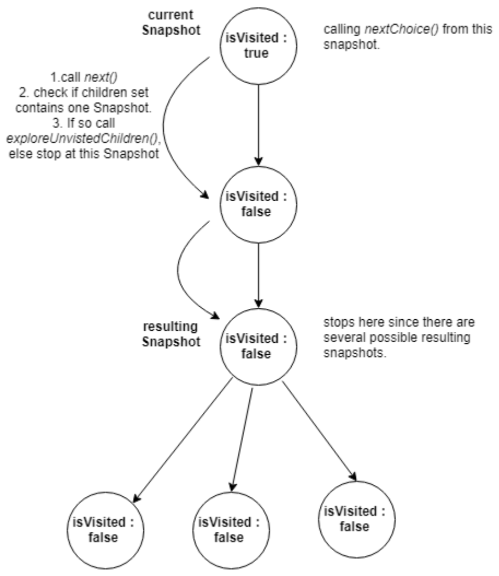

current Snapshot
isVisited : true
calling nextChoice() from this snapshot.
1.call next()
2. check if children set contains one Snapshot.
3. If so call exploreUnvistedChildren(),
else stop at this Snapshot
isVisited : false
resulting Snapshot
isVisited : false
stops here since there are several possible resulting snapshots.
isVisited : false
isVisited : false
isVisited : false

Figure 3.14: A visual representation of *nextChoice()*.

- **prevChoice()** - A method which explores each previous configuration from the current until a configuration is found where its children size exceeds one. This occurrence represents the case of non-determinism where several different configurations can be produced from a single configuration. *prevChoice()* works by repeatedly calling the method *prev()* and examining how many children each node has. It works the same as *nextChoice()*, however instead of going down the execution tree, it goes up.

- **quickRunFromCurrentConfig()** - This method incorporates all the above methods to carry out a depth-first search of the execution tree starting from the current configuration. It explores each branch until a Snapshot is found which, when loaded satisfies the acceptance criterion of the machine. In the case where a branch results in a configuration that is not an accepting one, the algorithm leverages *prev()* to go back up the tree and *exploreUnvistedChildren()* to explore an alternative Snapshot which has not been marked as visited. This process is repeated until the entire execution tree has been explored, and no accepting configuration has been identified. At this point, the algorithm outputs the result *zero* to represent this outcome. In the case where a valid sequence of Snapshot instances are found, then the solution is appended to *history* via the *addPathToHistory()* method. The repeated calling of *quickRunFromCurrentConfig()* would produce a different solution. The *history* field is responsible for keeping track of the solutions during the execution. The method utilises the *history* field to locate different solutions. If a solution is found in the process, which already exists in *history*, then the machine backtracks and explores a different path instead. An illustration of the *quickRunFromCurrentConfig()* procedure is represented in Figure 3.15.

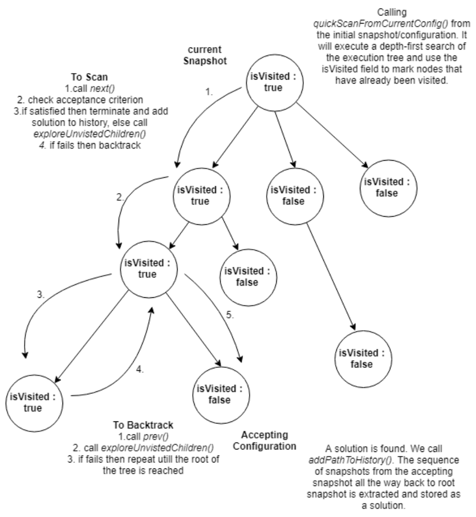

Calling
quickScanFromCurrentConfig() from the initial snapshot/configuration. It will execute a depth-first search of the execution tree and use the isVisited field to mark nodes that have already been visited.
current Snapshot
To Scan
1.call next()
2. check acceptance criterion
3.if satisfied then terminate and add solution to history, else call exploreUnvistedChildren()
4. if fails then backtrack
1.
2.
3.
4.
5.
isVisited : true
isVisited : false
To Backtrack
1.call prev()
2. call exploreUnvistedChildren()
3. if fails then repeat utill the root of the tree is reached
Accepting Configuration
A solution is found. We call addPathToHistory(). The sequence of snapshots from the accepting snapshot all the way back to root snapshot is extracted and stored as a solution.

Figure 3.15: A visual representation of *quickrunFromCurrentConfig()*.

PushdownAutomata - This class is responsible for representing a pushdown automaton. The PushdownAutomata class implements the Machine interface. It introduces a stack which can be utilised during execution. The PushdownAutomata instance dynamically loads state into the PushdownStack during execution.

TuringMachine - This class is responsible for representing a Turing machine. For a Turing machine, the tape is bi-direction (i.e. can move in both directions), and it can be modified during execution. To keep the same InputTape class, the TuringMachine must manage the content of the InputTape explicitly. To simulate empty cells, the TuringMachine incorporates null objects on the *tape*. This is important for when two characters are separated by several empty cells. To simulate going left on an input tape, the TuringMachine decrements the head position of the tape by one. In the case where the head position is at *0* already, then it shifts the array forward and includes a null object at the start of the tape. We keep the head at the same location.

3.7 Main Features

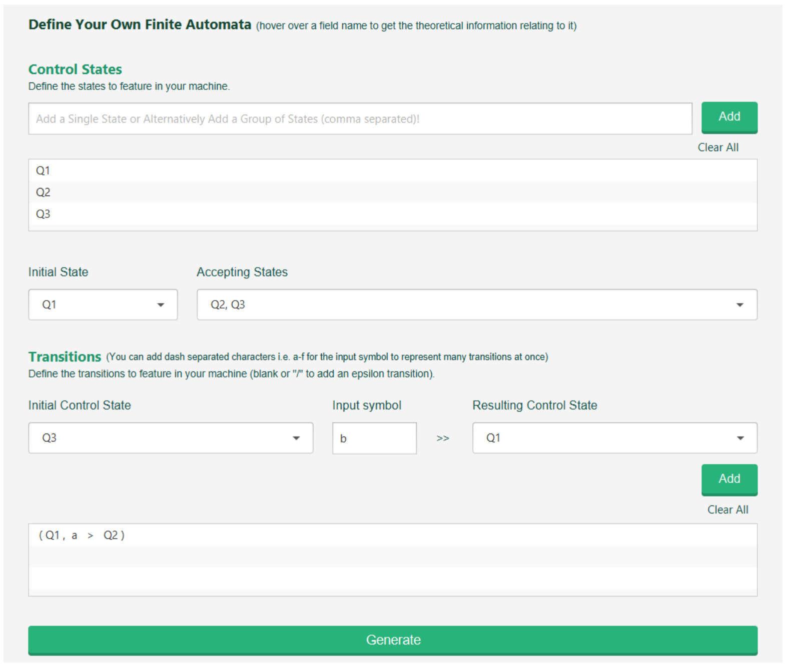
In this section, we look at the primary features and their more general role in this application. We will explore how each came about and outline some of the challenges that arose in constructing them.

3.7.1 Definition Procedure

This features enables the user to construct devices. It is achieved through the use of detailed and descriptive forms. In the machine_list view, the user can choose their desired machine type. This page contains the main devices. For each, a concise description is given. This description covers the computational advantages of each. The information presented tries to communicate across why creating these machines are important.

Once the device is chosen, the form is generated. This view contains the necessary components that are required for the machine type. As was mentioned already, each machine form mirrors the corresponding formal definition described in the literature. The form walks the user through assembling a machine. The FA form can be seen in Figure 3.16. First, the user starts with defining their control states. Then the user establishes their initial state (i.e. the source) and accepting states (i.e. the targets). The user then defines their transitions. The user is effectively generating possible routes between the control states. The ordering of the fields is significant in allowing the user to appreciate the process of designing a machine.

The form also includes theoretical and instructional information relating to the individual components that are associated with creating the automata machine. Compared to the alternatives, this application incorporates technical information wherever applicable in order to aid the user's understanding. Some users are less familiar with these concepts than others. It is therefore essential to accommodate for the different skill levels as an educational apparatus.


Define Your Own Finite Automata (hover over a field name to get the theoretical information relating to it)

Control States
Define the states to feature in your machine.

Add a Single State or Alternatively Add a Group of States (comma separated)!

Q1
Q2
Q3

Initial State: Q1
Accepting States: Q2, Q3

Transitions (You can add dash separated characters i.e. a-f for the input symbol to represent many transitions at once)
Define the transitions to feature in your machine (blank or "/" to add an epsilon transition).

Initial Control State: Q3
Input symbol: b
Resulting Control State: Q1

(Q1 , a > Q2)

Figure 3.16: The form for constructing the finite automata.

The alternative graphical definition approach (i.e. drag and drop) adopted by JFLAP is complicated and ineffective. The user should be able to create and modify machines easily. The descriptive approach enables users to construct their machines explicitly. It also mirrors many ideas represented in the automata literature.

Obstacles and Considerations

definition simplification - During development, unnecessary fields were removed from each form. Fields such as the input alphabet only limit the ability for the user to construct machines and simulate input. These types of fields were recognised as restricting the learning effectiveness of the overall system.

error handling - A prototype of this application was given to students to demo. The results found that many students were constructing machines incorrectly, and as a result, could not simulate them successfully. From this, we decided to incorporate error handling within the definition view that forced the user to define each component correctly. This is a means of ensuring that the user will always generate valid machines.

wildcard transitions - Some languages require a machine to have many transitions in order to recognise them. For instance, the language of English words that end with 'ing'. There are 26 alphabetic letters that we need to express for this example. This would require the user to define 26 individual transitions. This process would be exceedingly long for the user. Instead, we implemented wildcards to allow the user to represent groups of transitions with minimal effort. An illustration of this is represented in Figure 3.17. The simplification enables the user to create complicated machines more easily.

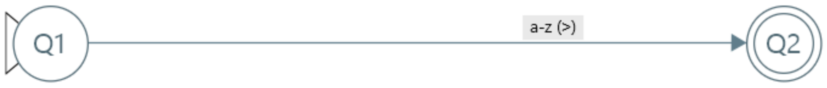


Figure 3.17: A transition wildcard representing 26 different transitions.

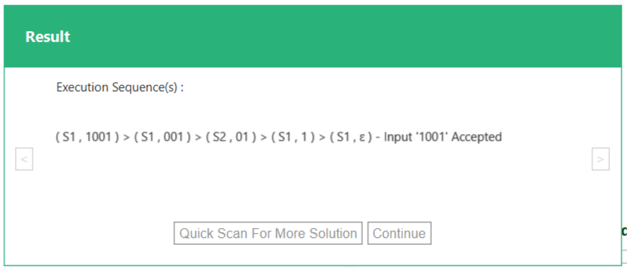
3.7.2 Execution of a Machine

The system must be able to simulate the execution of each machine to represent the ideas they communicate effectively. The motivation of this project comes in wanting to convey this process in a meaningful way to students by using effective visualisation and interaction. Many of the key ideas are represented within the execution process, so it remains an essential part of the system.

This application introduces two modes of operation. Both allow the user to simulate input.

Quick-Run

This feature allows the user to run input and instantly review the results of the execution. It works by running a depth-first search in the background to find an individual path that results in an accepting configuration. The decoupling achieved through MVC allows us to simulate a machine without having to interact with the visual components. In the case where input is accepted, the user will be prompted with the sequence of configurations which lead to the machine accepting the input (Figure 3.18). This is a basic feature which the user can use to quickly evaluate their machine's effectiveness.

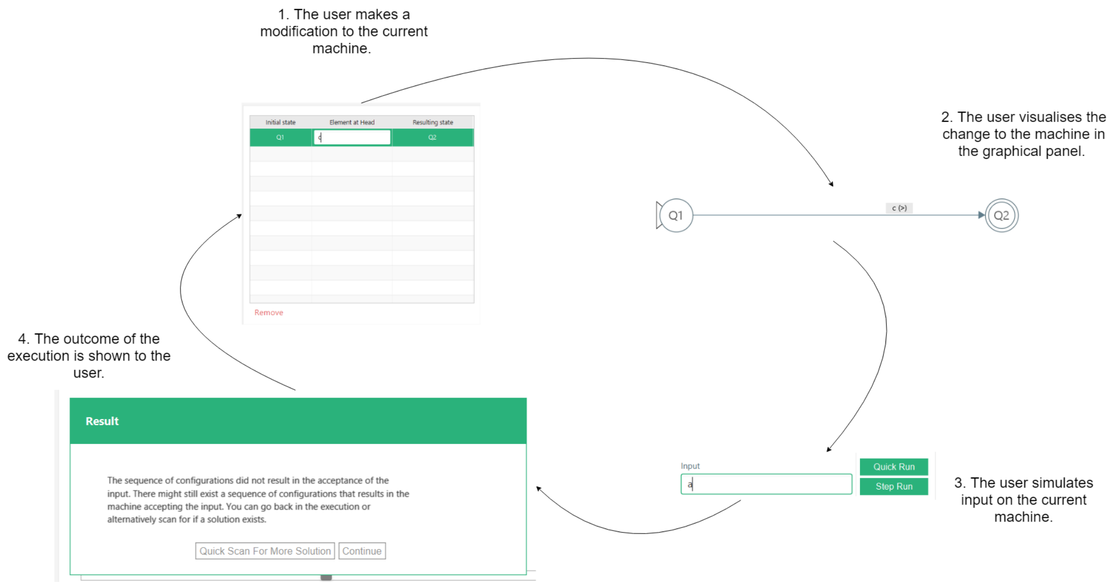


Execution Sequence(s) :

(S1 , 1001) > (S1 , 001) > (S2 , 01) > (S1 , 1) > (S1 , ε) - Input '1001' Accepted

Figure 3.18: A result pane.

This feature is essential for the user's learning process. The user continuously learns from the experience of modifying a machine and assessing the outcome. This constant cycle of results empowers the learner to establish causal links between how a device is constructed and how a device operates. An illustration of this process can be seen below in Figure 3.19.


1. The user makes a modification to the current machine.
Initial state
Element at Head
Resulting state
Q1
c
Q2
Remove
2. The user visualises the change to the machine in the graphical panel.
Q1
c (>)
Q2
Input
a
Quick Run
Step Run
3. The user simulates input on the current machine.
4. The outcome of the execution is shown to the user.
Result
Quick Scan For More Solution
Continue

The sequence of configurations did not result in the acceptance of the input. There might still exist a sequence of configurations that results in the machine accepting the input. You can go back in the execution or alternatively scan for if a solution exists.

Figure 3.19: The learning process of continually modifying a machine and assessing the outcome.

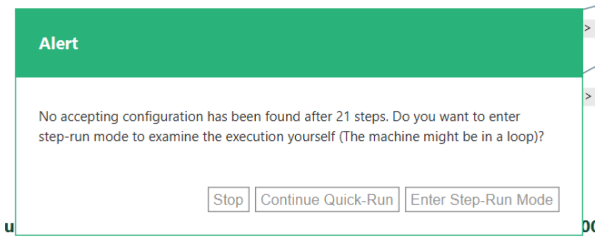
Obstacles and Considerations

simulating non-determinism - In order to simulate these machines entirely, non-determinism must be implemented. Non-determinism occurs when two or more transitions can be taken in the execution. To tiebreak in quick-run, we utilise the original order that transitions were defined in and choose the first in that arrangement that results in a configuration that has not been visited yet. The user can later examine the transitions that were explored in the results pane.

finding multiple solutions - It was observed that for a non-deterministic machine, there could exist several accepting sequences. The system must serve the user with these alternative solutions. The user can request the application to look for more solutions. We incorporate the *history* field in the Machine instance to keep a track of the solutions that have been found during the quick-run process. The Machine instance looks for newer solutions in the same execution tree.

non-terminating machines - The quick-run operates by exhaustively searching the execution tree until a unique sequence of configurations is found. There are some machines which never terminate during execution due to them continually entering newer configurations. When this occurs, the machine can never backtrack in the execution tree. The introduction of an epsilon transition makes this possible. The input tape is finite since the input word is finite as well. However, the introduction of the epsilon transition means that the reading of the symbol at the head is not a necessary step. The user can construct a cycle of transitions that never modify the input tape. JFLAP does not address this problem, and such an event will cause it to crash.

For this application, whenever the machine is in a constant loop for a quick-run procedure, after 20 configurations, the application will pause the execution and prompt the user on what to do next. An illustration of this is seen in Figure 3.20. The user then can either continue the run or can enter step-run mode to view the cause of the loop. This would load the state of the current configuration into the simulator, and the user could then manually continue the execution, allowing the user to review the process themselves.


Alert

No accepting configuration has been found after 21 steps. Do you want to enter step-run mode to examine the execution yourself (The machine might be in a loop)?

Figure 3.20: An alert box which is shown to the user when a quick-run process exceeds 20 configurations.

Step-Run

This feature allows the user to visualise a step-by-step representation of the execution process. It illustrates how the output is reached for a particular input. This includes showing the state of each component at different stages of the execution. This feature will promote learning through effective interaction and exploration. As was stated previously, a lot of the ideas around computation are demonstrated through execution, so it remains essential that this process is represented correctly.

Graphical representation

Alternative applications do not represent the procedure of execution in any valuable way. Unlike JFLAP, the input tape will be integrated into the execution with other important elements. These components are intentionally made simple. Overcomplicating the visuals can have an adverse effect on the user's experience.

- **Input Tape** - This view (Figure 3.21) is a representation of an input tape. It is dynamically modified to match the state of the InputTape model class.

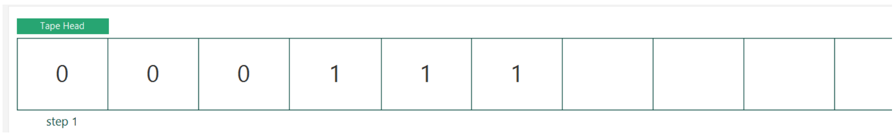
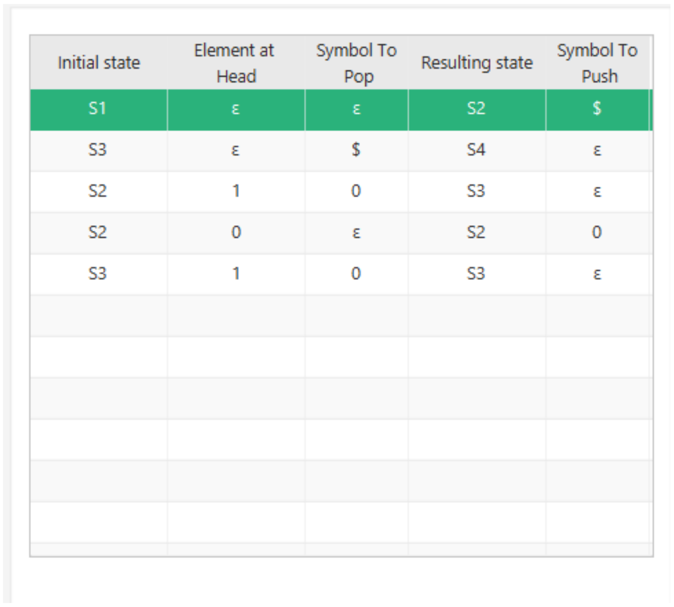


Figure 3.21: An input tape.

- **Transition Table** - This view (Figure 3.22) encapsulates the current set of transitions within a table. When a transition is explored in the execution, the corresponding row is highlighted.



Initial state	Element at Head	Symbol To Pop	Resulting state	Symbol To Push
S1	ε	ε	S2	$
S3	ε	$	S4	ε
S2	1	0	S3	ε
S2	0	ε	S2	0
S3	1	0	S3	ε

Figure 3.22: A transition table.

- **Pushdown Stack** - This view (Figure 3.23) is a representation of the pushdown stack, a feature found in the PDA. It is dynamically modified to match the state of the PushdownStack model class.

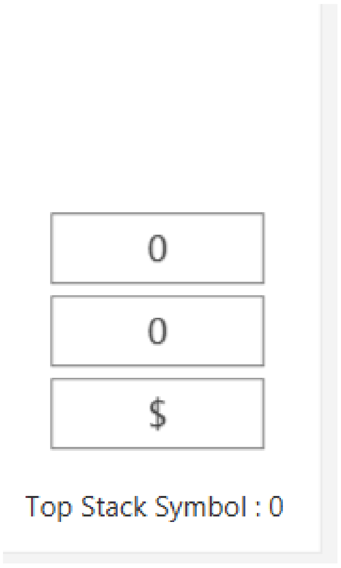
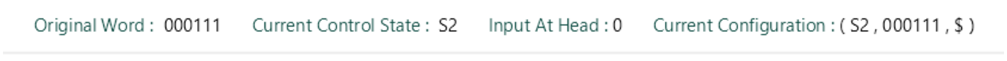


Figure 3.23: A pushdown stack.

- **Information Panel** - This view (Figure 3.24) expresses a set of labels that are vital in describing the state of the current machine.



Original Word : 000111 Current Control State : S2 Input At Head : 0 Current Configuration : (S2 , 000111 , $)

Figure 3.24: An information panel.

- **Graphical Panel** - This view (Figure 3.25) is responsible for representing the automata machine visually. All the contained states and transitions are illustrated as visual components. On execution, the current state is highlighted. When a transition is explored, it triggers an animation that highlights the visual transition for a duration of two seconds.

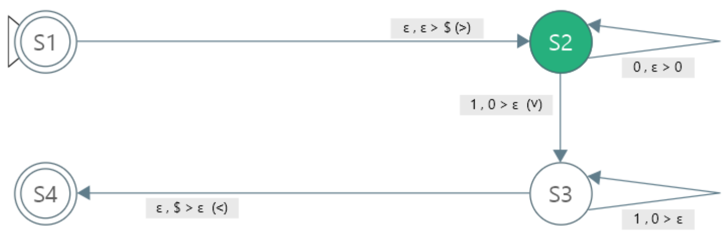
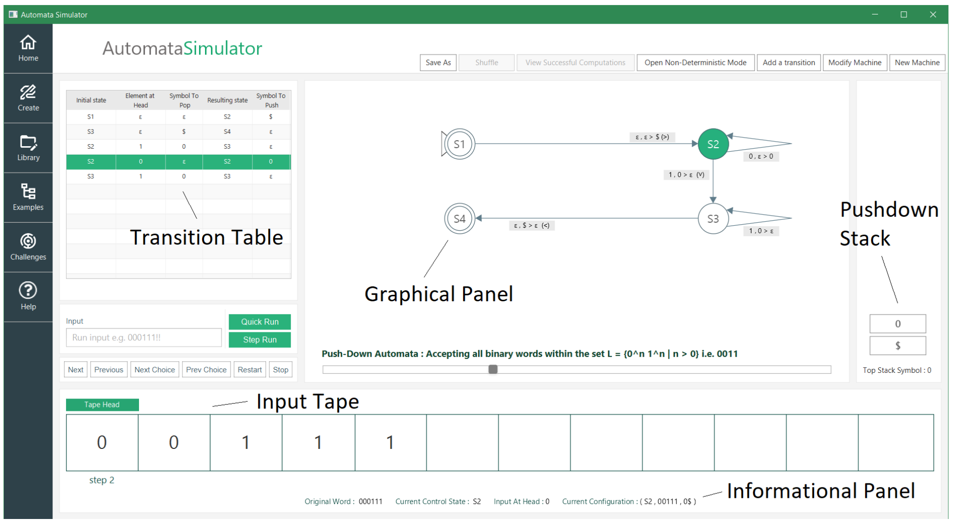


Figure 3.25: A graphical panel.

All these components are used in combination to represent a procedure. Through animation and practical design, we simplify these conceptual ideas into easy-to-understand representations. The theoretical machines are portrayed as graphical objects that change dynamically. This encourages students to interpret these machines as if they were physical devices operating in the real world.



Initial state	Element at Head	Symbol To Pop	Resulting state	Symbol To Push
S1	ε	ε	S2	$
S3	ε	$	S4	ε
S2	1	0	S3	ε
S2	0	ε	S2	0
S3	1	0	S3	ε

Push-Down Automata : Accepting all binary words within the set L = {0^n 1^n | n > 0} i.e. 0011

0
$

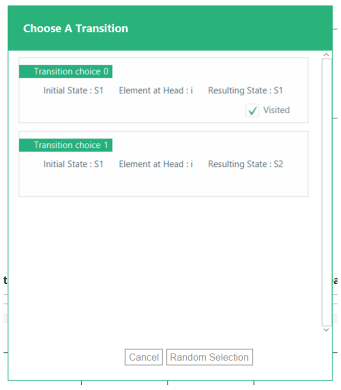
Top Stack Symbol : 0

0	0	1	1	1							

Original Word : 000111 Current Control State : S2 Input At Head : 0 Current Configuration : (S2 , 00111 , 0$)

Non-Determinism

Non-determinism occurs when two or more transitions can be taken in the execution. To tiebreak for step-run, we prompt the user with a transition selection box (Figure 3.26). The user has the freedom to determine the execution path. The application allows the user to pick the transition that will be explored. This function encourages experimental learning. When the user visits a configuration, it is marked as visited in the execution tree. The next time the user visits that configuration, the transition that was previously selected is indicated to the user as previously being visited. The user in these instances are encouraged to explore alternative paths.


Choose A Transition

Transition choice 0
Initial State : S1 Element at Head : i Resulting State : S1
✓ Visited

Transition choice 1
Initial State : S1 Element at Head : i Resulting State : S2

Cancel | Random Selection

Figure 3.26: The transitional selection panel.

For JFLAP, non-determinism is not really addressed as clearly. An illustration of this is below in Figure 3.27. JFLAP explores several different branches in parallel when non-determinism occurs and repeatedly does this for each new occurrence. When a machine has a lot of non-deterministic transitions, the execution process becomes challenging to trace. The user becomes overwhelmed with the information that is being shown to them.

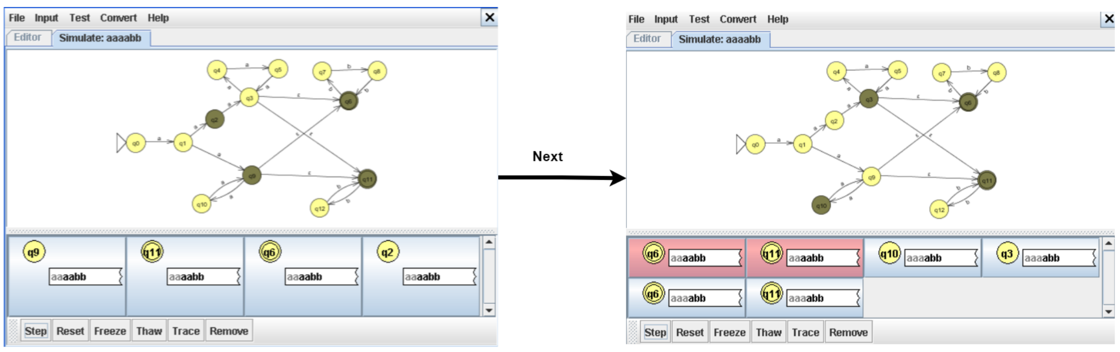


Figure 3.27: The left image represents a point in the execution where four non-deterministic transitions can be taken. The right image shows the resulting effect of this when *Step i.e. Next* is called. The JFLAP application splits the execution into four. The first two paths fail, and the second two continue to split further.

User Interaction

The other aspect of the step-run procedure is the interaction it offers the user during the walk-through. The application allows the user to control the execution process. Step-run allows the user to traverse the execution tree for a particular input. They can go back to a previous configuration to replay part of the execution or progress to the next configuration. The user is given the freedom to explore. It goes further than JFLAP by giving the user more control. For instance, in JFLAP, the user cannot go back in the execution.

- **Next** - allows the user to go forward in the execution by one configuration. If a subsequent configuration does not exist in the execution tree, then the machine can evaluate the results of the current execution path. For each next call, the transition that was chosen is highlighted in both the transition table and the graphical panel. Also, the new visual control state is highlighted. The input tape and the information panel are refreshed to represent the new state of the machine.
- **Previous** - allows the user to go back to the previous configuration to replay part of the execution.
- **Next Choice** - Explores each following configuration from the current until a configuration is found where the number of transitions which can subsequently be applied exceeds one. This represents the case of non-determinism where more than one possible configuration can be generated from the current configuration. The user is prompted to choose the transition to explore when this occurs. If there are no cases where this occurs (i.e. a deterministic machine), then it will progress till the machine reaches an output.
- **Previous Choice** - Explores each previous configuration from the current until a configuration is found where the number of transitions that can be subsequently applied exceeds one. This represents the case of non-determinism where more than one possible configuration can be produced from the current configuration. The user is prompted to choose the transition to explore when this occurs. If there are no cases where this occurs, then it goes all the way back to the initial configuration.
- **Stop** - allows the user to cancel the execution and reset the simulator
- **Reset** - allows the user to start back from the initial configuration. The execution tree is preserved, so all visited configurations will be marked as such to the user.

Obstacles and Considerations

finding a solution - It was found that it would take considerable effort on the user's part to find a solution for a machine that contained a high number of non-deterministic transitions. It would involve the user manually exploring every possible branch in the execution in the hope of finding an accepting configuration. To remedy this event, we incorporated Quick-run as part of the Step-run function. The user can initiate it at any point, leaving the system to search for a solution instead.

3.7.3 Save Feature

The save feature allows the user to save their machine instances to memory and reload them back after closing the program. It provides an incentive for students to return to the application by allowing them to start from a previous session. As was mentioned earlier, It also allows teachers to package machines which can then be used within a classroom context to teach ideas related to automata theory.

In this application, we utilise a library called GSON, which allows us to serialise Java objects into JSON. Whenever the user requests for their machine to be saved, the Definition instance, which represents the internal structure of the machine, is extracted and serialised into JSON. The user additionally assigns a description to the machine to allow them to identify it later. We maintain a list of saved Definition instances in the GlobalStore class. When a definition is saved, it is added to that list. The list is then serialised into JSON and stored as a single file in memory. We maintain a store file for each Definition type. For example, FADefinition instances are stored in finite-automata-store.json.

The user can view their machines in the machine_store_view. They can load or remove machines from here. When the user needs to load a machine from their library, the store file is retrieved from memory and serialised back into a Java object (i.e. a collection of definition instances). The particular Definition instance is found and loaded into the simulator.

The user can also modify a machine after saving. The changes made can be written to memory. This is important for incrementally allowing the user to improve their solution.

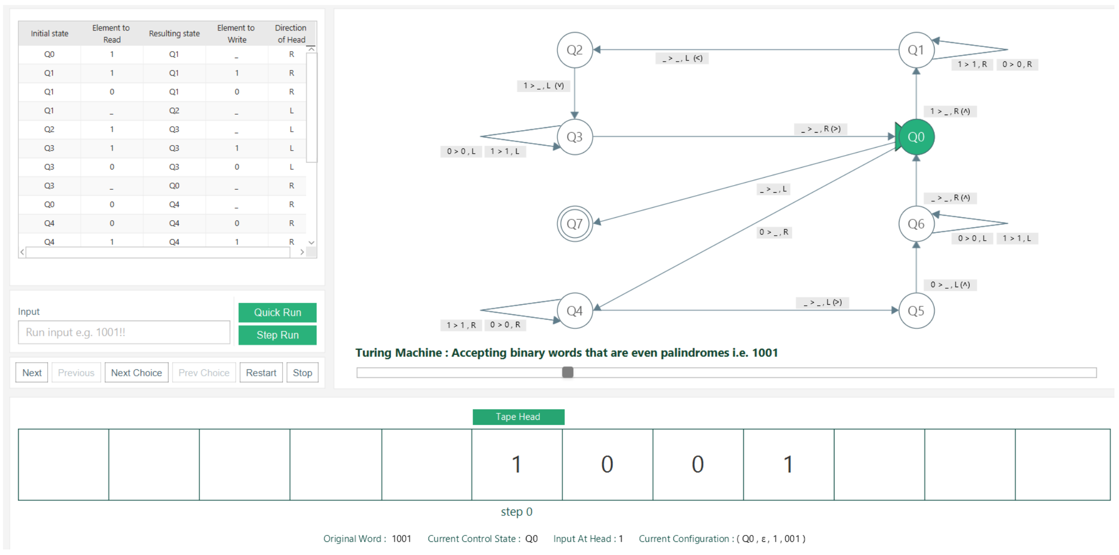
Obstacles and Considerations

This application is presented as a JAR file. Jar files are read-only. They package all dependencies within a compressed file (i.e. in ZIP format). This meant that we were not able to include the storage files inside the JAR. These files are not static and need to be continuously changed. The application instead saves the files in the same directory as the JAR file. On loading a machine, It first checks that these files exist in that location, and if they do not, they are created. This approach presents another problem. The JAR file on being relocated will result in the user losing their saved machines.

3.7.4 Examples Feature

The feature allows the user to load pre-defined machines into the simulator instantly. The models are prepackaged and tested machines that the user can load and run. They are obtained from the automata literature, and are used specifically to represent and express how automata machines work. They are simple enough not to overwhelm and challenging enough to engage, and can be deployed to allow the user to familiarise with key concepts quickly.

Kyle Dickerson's automata simulator is the only alternative that incorporates integrated examples. There is a total of three examples on the platform, all of which are basic. It is crucial for there to be a comprehensive collection of examples. This is so not to encourage learners to link a particular instance with the meaning of a concept.



Initial state	Element to Read	Resulting state	Element to Write	Direction of Head
Q0	1	Q1	_	R
Q1	1	Q1	1	R
Q1	0	Q1	0	R
Q1	_	Q2	_	L
Q2	1	Q3	_	L
Q3	1	Q3	1	L
Q3	0	Q3	0	L
Q3	_	Q0	_	R
Q0	0	Q4	_	R
Q4	0	Q4	0	R
Q4	1	Q4	1	R

Turing Machine : Accepting binary words that are even palindromes i.e. 1001

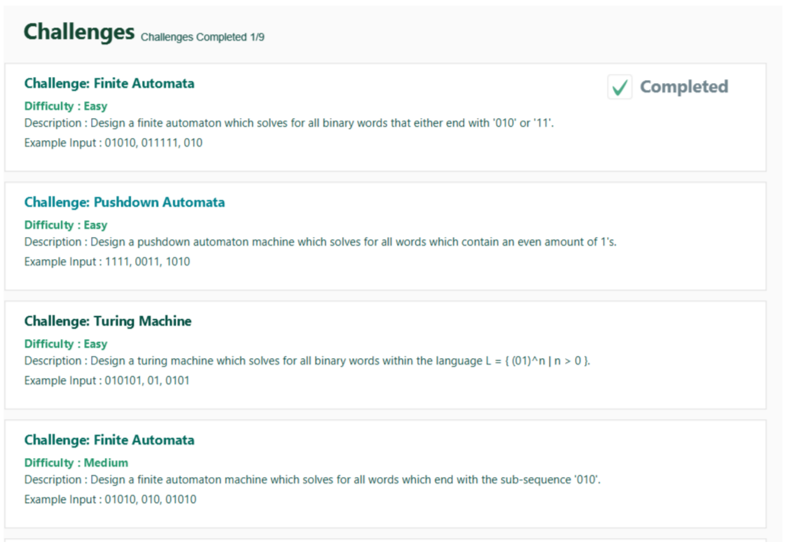
step 0

Original Word : 1001 Current Control State : Q0 Input At Head : 1 Current Configuration : (Q0 , ε , 1 , 001)

Figure 3.28: An example Turing machine that is available to the user to load. The machine recognises the language of even palindromes.

3.7.5 Challenge Feature

The feature allows the user to participate in solving progressively harder challenges within the application. The challenge dashboard provides a list of challenges (Figure 3.29). Each challenge comes with a difficulty level, a machine type, and a description of the language to solve. The challenge feature allows for the user to start with a blank machine and iteratively modify it by dynamically adding states and transitions. The graphical representation of the current machine is updated dynamically. The user can examine their machines by running input on them during this process.



Challenges Challenges Completed 1/9

Challenge: Finite Automata ✓ Completed
Difficulty : Easy
Description : Design a finite automaton which solves for all binary words that either end with '010' or '11'.
Example Input : 01010, 011111, 010

Challenge: Pushdown Automata
Difficulty : Easy
Description : Design a pushdown automaton machine which solves for all words which contain an even amount of 1's.
Example Input : 1111, 0011, 1010

Challenge: Turing Machine
Difficulty : Easy
Description : Design a turing machine which solves for all binary words within the language L = { (01)^n | n > 0 }.
Example Input : 010101, 01, 0101

Challenge: Finite Automata
Difficulty : Medium
Description : Design a finite automaton machine which solves for all words which end with the sub-sequence '010'.
Example Input : 01010, 010, 01010

Figure 3.29: The user's challenge list.

When a user has constructed a solution, they can submit it. This will open a results box that will run the machine against a set of pre-defined inputs. Only when the machine successfully interprets all input words will the application accept the machine as a solution. When a user completes a challenge, the solution is saved, and the challenge is marked as complete. The state machine documents this process (Figure 3.30). As was mentioned before, this feature will encourage exploration through engagement. The user will be given an opportunity to experience the impact of their modifications by running input and examining the results. These challenges try to analyse the user's ability to design machine instances and also engage the user on a mental level.

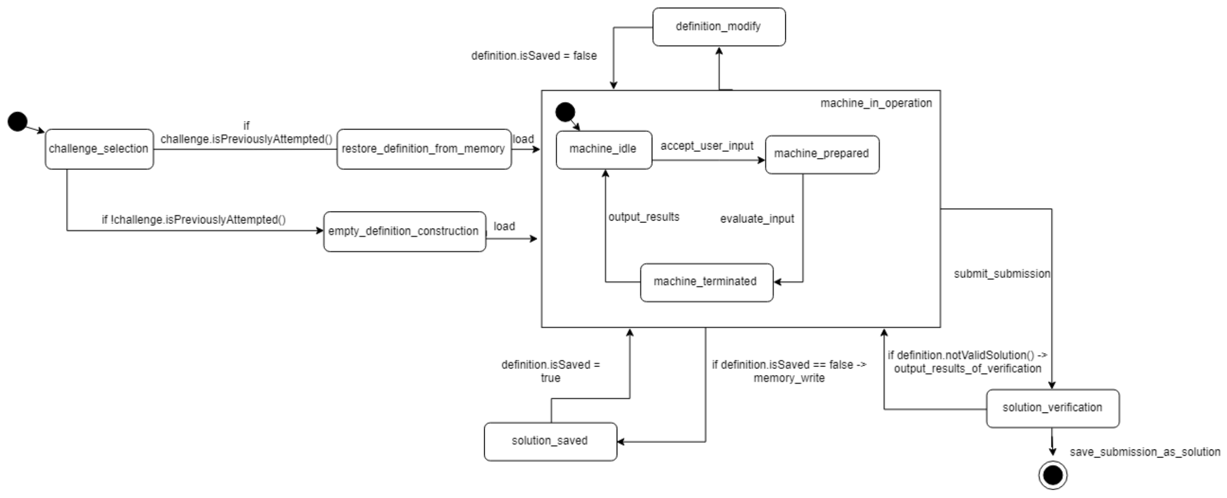

definition_modify
definition.isSaved = false
machine_in_operation
challenge_selection
if challenge.isPreviouslyAttempted()
restore_definition_from_memory
load
machine_idle
accept_user_input
machine_prepared
if !challenge.isPreviouslyAttempted()
empty_definition_construction
load
output_results
evaluate_input
machine_terminated
submit_submission
definition.isSaved = true
if definition.isSaved == false -> memory_write
if definition.notValidSolution() -> output_results_of_verification
solution_verification
solution_saved
save_submission_as_solution

Figure 3.30: A state machine documenting how a user solves a challenge within the application.

3.7.6 Other important features

The application introduces *non-deterministic mode*, which allows the user to view the transitions that are non-deterministic in the present machine. It does this by highlighting the visual transitions that are non-deterministic. It is useful for the students to recognise non-determinism as a concept because it can help them to understand determinism and notably what is possible during execution. The introduction of non-determinism as a theoretical concept makes it easier for students to reason with these computational procedures.

It was identified that the visual representation of these machines had to be clear and concise. The graphical panel of the application is responsible for generating a visual image of a machine instance. The graphical panel initially represented the states in the order in which they were created. From this arrangement, it was often the case that a lot of the visual transitions would intersect, and as a result, It became difficult for the user to follow the execution. An algorithm had to be constructed, which analysed the best arrangement (i.e. how easy is it for the user to comprehend). The algorithm works by computing a score for each possible arrangement. This score is calculated by counting the number of transitions that intersect. The combination with the lowest score is chosen to be represented.

Chapter 4

Evaluation

Students and lecturers are the primary users for this tool. We have previously laid out the main functions of this system concerning the kind of user operating it. We can evaluate the system's effectiveness by examining how a primary user employs this tool to carry a defined function.

4.1 Testing

An educational tool must be correct. The user must be able to extrapolate accurate information from the application. The failure in achieving this would make the application void. Both an extensive research process and a comprehensive collection of tests can help to ensure accuracy.

Junit, an open-source Java testing framework, was employed to carry out the testing requirements for this application. We implemented a series of wide-ranging unit tests that examined the correctness of individual elements within this application. We focused the model classes that were responsible for simulating the different components within the automata machine. For instance, we constructed a series of unit tests to evaluate the InputTape class. It analysed the functionality of the tape, and specifically how it operates (i.e. how a symbol is read on the tape). We created tests for each of the main model classes within the system.

We also implemented integration tests for each automata machine class. These tests examined how the individual elements combined to carry out functionality. For us to verify that the system could correctly simulate these machines, integration testing was a necessary step. We constructed tests for several key edge-cases and reviewed primary functions within the Machine class. These tests looked to examine components at different parts of the execution and analyse how they interact with the machine as a whole. We also generated tests to examine the correctness of the examples that are available in the application. It involved running a group of input words on each model and reviewing the generated outcomes.

For this project, we adopted an agile approach. The early addition of testing enabled us to take a test-driven approach when implementing features. The codebase became significantly large and as a result, meant that code adjustments were likely to have adverse outcomes on functionality. The introduction of tests assured us that no code was being broken during development. For the limited amount of time that was available, testing was an effective way for us to integrate new features into the system.

4.2 User Survey

Evaluation objectively analyses a program. It involves gathering and interpreting information regarding a program's activities, features, and results. Its intention is to make determinations about a program, strengthen its usefulness, and advise choices on programming. Given the depth of the application and the time allocated to implementing it, the evaluation is not as exhaustive as it should be. An entire paper could be given to examining the effectiveness of this application.

In this part of the evaluation, we analyse the application from the perspective of a student using it to learn. We began by establishing a sample size of students that could participate in each experiment. The sample consisted predominantly of third-year computer science students. We examined the eligibility of each participant. The students were required to have some prior knowledge of automata theory. This would enable them to use the features of the application more effectively. We interviewed each candidate to examine their levels of understanding. We asked simple questions like "what are transitions?". We wanted to ensure that participants were at a basic level before we began. A total of 10 students were chosen.

4.2.1 Blinded Experiment

The first experiment involved the participants demoing a set of the automata machine simulators. We excluded jFast from the experiment because it could not simulate the execution procedure.

The applications that were examined:

- Automata Simulator (i.e. the proposed application)
- JFLAP (Patel)
- Cburch Automata Simulator (Burch)
- Kyle Dickerson's Automata Simulator (Dickerson)

Each participant was given the applications in random order with no information as to what system was ours. We did not want the results to be skewed. The information that may have affected the participants was maintained until after the end of the experiment. Each participant had 20 minutes to demo each application. They were given a list of aspects to consider during the demo:

- The creation/defining process
- The automata machine that can be simulated (FA, PDA or TM)
- The visualisation of the execution process
- The auxiliary features
- The user interactivity
- The engagement and enjoyment level

This was done to focus the participants ' attention on evaluating the application's usefulness instead of on how it looked. Students were told to review and rank the applications from best to worst. The outcome of the study is defined below.

Name of Application	first choice allocations	second choice allocations	third choice allocations	fourth choice allocations
JFLAP	0	5	5	0
Kyle Dickerson's Automata Simulator	2	3	5	0
Automata Simulator	8	2	0	0
Cburch Automata Simulator	0	0	0	10

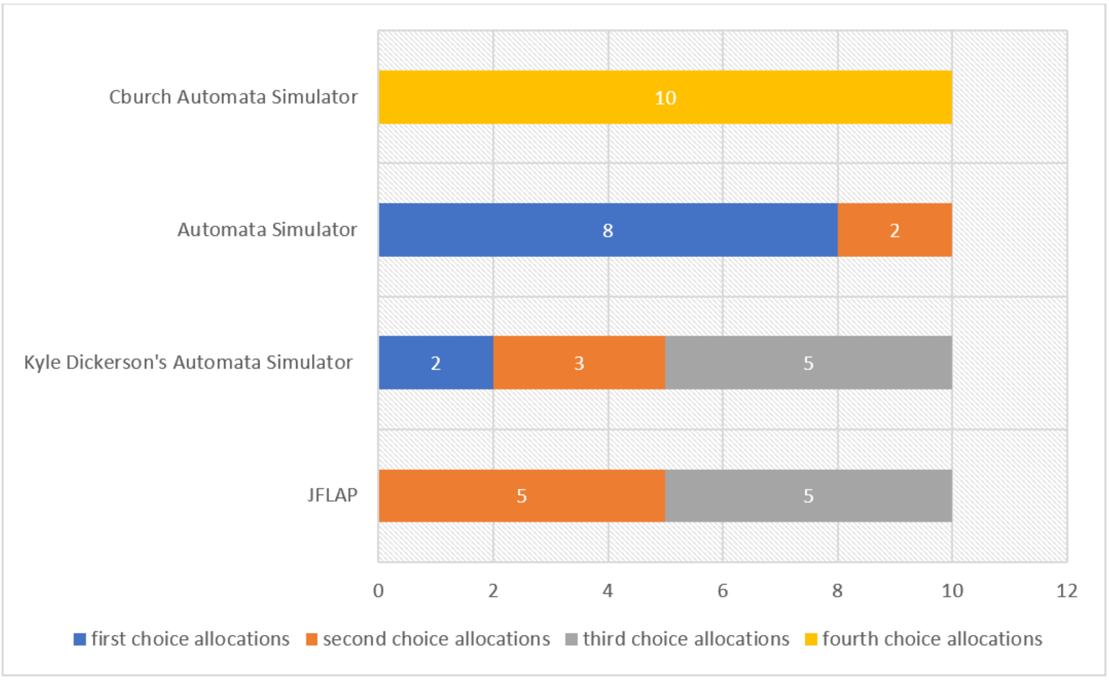


Figure 4.1: A graph representing the participant rankings of each application.

For eight of the ten participants, our application was identified as being the best tool. One of the original aims was to produce an educational tool that was more effective than the alternatives. A lot of the participants preferred the simple mechanics and illustrations that were incorporated. The students appreciated the inclusion of challenges. They identified challenges as a self-motivating part of the application that could be used to engage them.

JFLAP came third amongst the collection. This was a surprising outcome, as JFLAP was identified as the most comprehensive solution during research. Many of the criticism came from it being too hard to understand. The users had a challenging time understanding what was being conveyed. This continued even after the user had access to the JFLAP documentation. This confirms our initial assessment that the inclusion of too many features by JFLAP made the application less effective as an educational tool. However, many students did like the definition procedure. They liked interactively modelling their machines using visual mechanisms (e.g. drag and drop).

Kyle Dickerson's automata simulator came in second. Its modern appearance was recognised as the main reason. It incorporated examples that the students appreciated. They especially liked how simple the system was. The application concentrated on a single task (i.e. simulating the execution procedure). Kyle Dickerson's automata simulator had several flaws. For instance, the application was not able to simulate a Turing machine. This was the main constraint noted by participants. Yet it is still perceived to be better than JFLAP.

Cburch automata simulator was last placed. Many participants described the tool as being incomplete. The finite automata and pushdown automata functions were not working as intended. The students were unable to enter input. From amongst the set of machines, only the Turing machine could be simulated. Students were having a challenging time constructing and simulating these machines. They did not know when a machine accepted or rejected an input as there was no indication of either event. From the perspective of the student who uses this system, usability is the most necessary criteria. The Cburch automata simulator is not a user-friendly solution.

Summary

In summary, it was identified that students did not necessarily want an e-learning application that could do many tasks. They wanted a tool that could do a single task really well. JFLAP's complex operations demotivated the students to use their system. An excellent user interface was also identified as being critical to an e-learning system. It allows an application to appear better than it is in reality. This experiment provided us with valuable data concerning the effectiveness of each application.

4.2.2 Focused Interviews

The second part of the user study is more focused. We evaluate the application itself and in particular explore the main aspects of the system. It consisted of a series of questionnaire-based interviews. For each interview, the participant interacts with the application. The user will be prompted with a series of statements that look to examine the different parts of the application and will be asked to answer on whether they agree or disagree. These require a fixed response by the participant. The classification of answers are defined below.

- Strongly Agree = 2
- Agree = 1
- Neither Agree or Disagree = 0
- Disagree = -1
- Strongly Disagree = -2

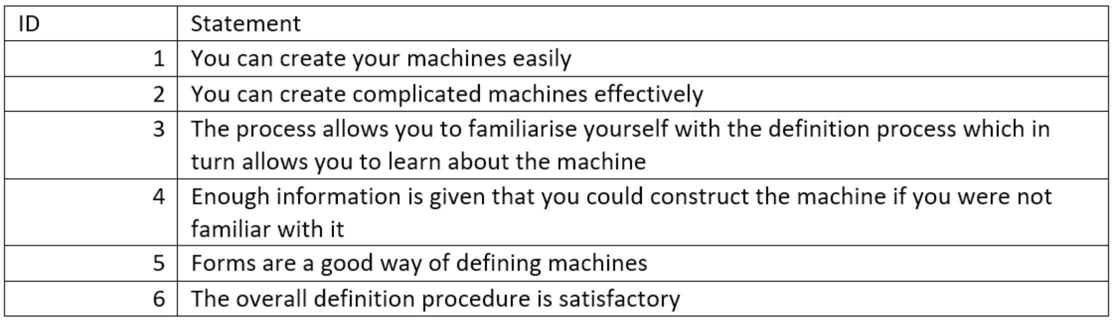
In this section, the conductor actively participates in the user's journey through the application. We also incorporated a think-aloud strategy during the demoing of the application. The participants could elaborate on any ideas or critiques they might have during the demo regarding the various statements.

The features of the system to examine are:

- Machine Definition Procedure
- Execution Procedure
- Auxiliary Feature (e.g. examples, challenges)

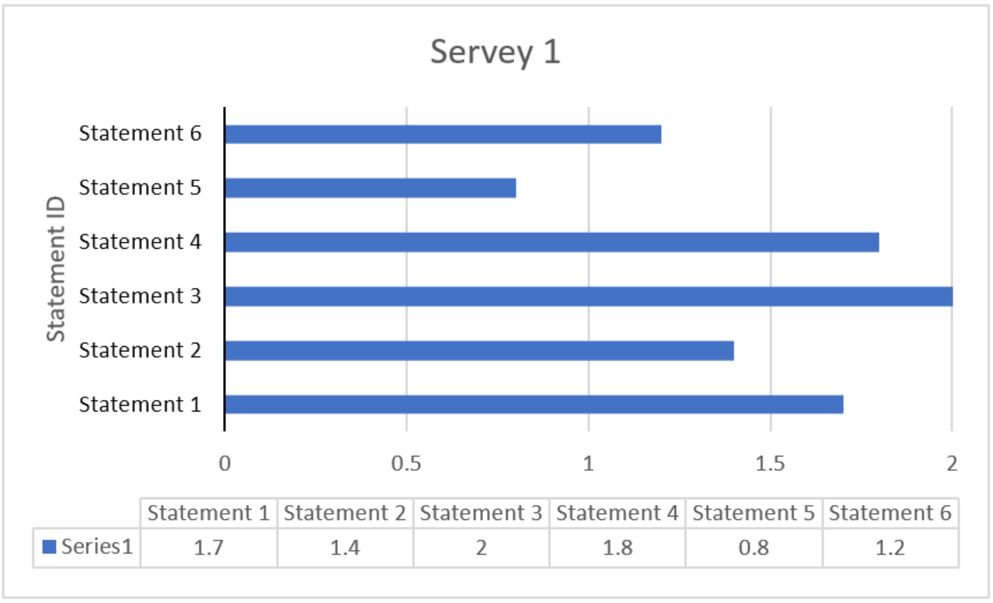
4.2.3 Machine Definition Analysis

In this section, we asked a number of focused questions to examine the effectiveness of the definition procedure. The demo part of this analysis involved each participant constructing their machine. The entire survey for this section is available in the appendix. A summary of its findings is found below.



ID	Statement
1	You can create your machines easily
2	You can create complicated machines effectively
3	The process allows you to familiarise yourself with the definition process which in turn allows you to learn about the machine
4	Enough information is given that you could construct the machine if you were not familiar with it
5	Forms are a good way of defining machines
6	The overall definition procedure is satisfactory

Figure 4.2: A table of statements concerning the definition procedure of the application.



	Statement 1	Statement 2	Statement 3	Statement 4	Statement 5	Statement 6
■ Series1	1.7	1.4	2	1.8	0.8	1.2

Figure 4.3: A graph representing the average response for each statement outlined in Figure 4.2.

In summary, the average score was 1.48 across all statements. Every participant entirely agreed with statement 3. The participants liked the additional information that was presented for each machine type. They also liked how the element fields featured in the form were clear and precise. The process of defining the machine explicitly forced the users to reason with the elements involved, and this gave them a greater appreciation for the individual components.

Statement 5 and 6 achieved an average of 0.8 and 1.2. These were the lowest scores from amongst the statements. For statement 5, four participants believed that the form was not the best way for users to define machines. The participants felt that the process was not engaging enough. All participants had experience with the interactive mechanisms defined in JFLAP. They appreciated the ability to manipulate machines directly. A few of the students' described the definition procedure as good; however, mentioned that the inclusion of an interactive procedure that enables the user to manipulate their machine physically would make it better. This was a commonly-held opinion across all statements.

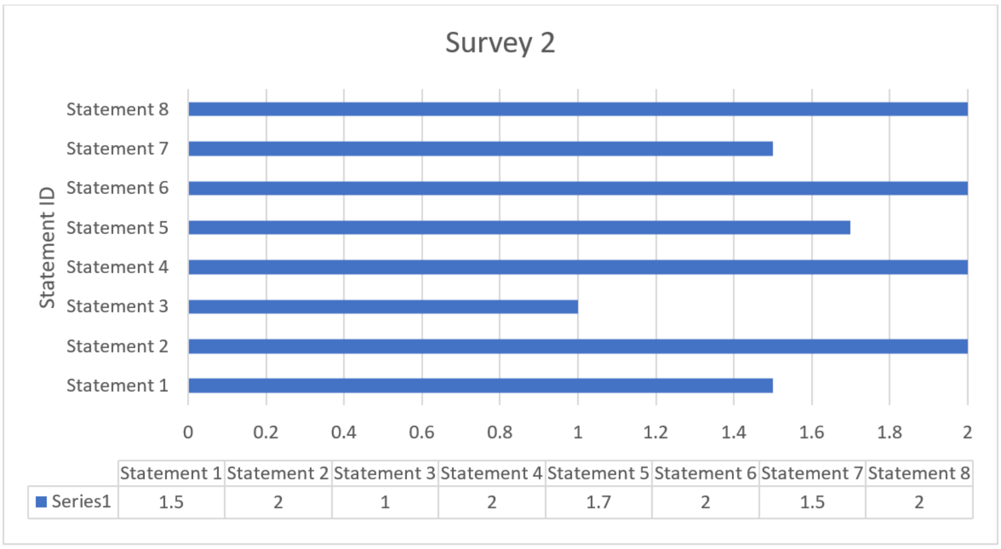
In general, the results are very positive. The survey confirms that the definition procedure is adequate for the user. The results verify that the user can effectively create machines of any complexity. It was additionally noted that participants wanted an interactive way of defining their machines. However, as we recognised before, incorporating such a feature will abstract the mechanisms of the system away from the theoretical concepts they represent. The feedback may not be characteristic of each strategies ' usefulness. Each participant had no basis for using the system, and their analysis was principally constructed on their experiences at the time. In order to evaluate both methods within an educational context, a more comprehensive analysis has to be done.

4.2.4 Execution Analysis

In this section, we asked a number of focused questions to examine the effectiveness of the execution procedure. The demo part of this analysis involved each participant simulating input on an example machine and tracing the execution process. The entire survey for this section is available in the appendix. A summary of its findings is found below.

ID	Statements
1	The representation of the execution procedure is useful in understanding these machines
2	It is easy to understand what the role of each visual component is concerning the machine being simulated
3	It is easy to understand what each button in the user control panel does
4	You can recognise why a machine is rejecting or accepting an input
5	Non-determinism is demonstrated clearly to you
6	You know the state of the machine for each step of the execution
7	You could infer how a machine that you never experienced before worked by undergoing the execution process
8	Both the illustrations and animations incorporated are useful and easy to understand

Figure 4.4: A table of statements concerning the execution process of the application. These are put to the students.



	Statement 1	Statement 2	Statement 3	Statement 4	Statement 5	Statement 6	Statement 7	Statement 8
■ Series1	1.5	2	1	2	1.7	2	1.5	2

Figure 4.5: A graph representing the average response for each statement outlined in Figure 4.4.

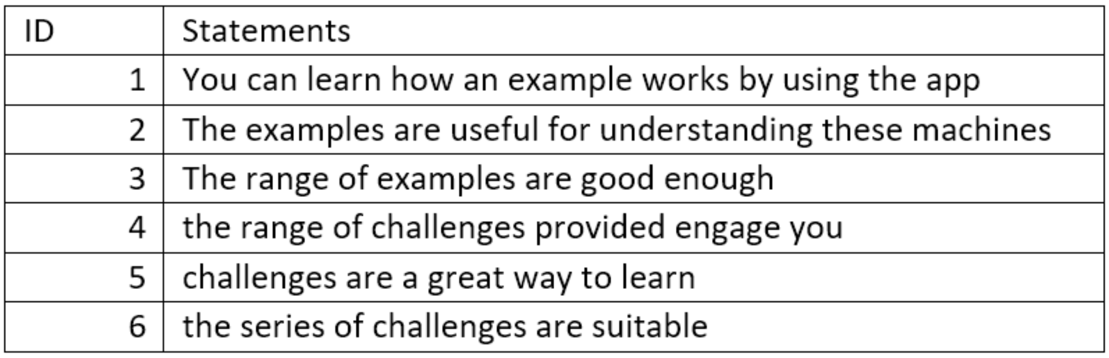
The average score was 1.71 across all statements. The students liked the execution mode and notably the amount of control it gave in allowing them to decide which branch to explore in the execution. They also liked the representations. The illustrations and animations were recognised as being used effectively within the system. The users found the chained events of execution easy to follow. As a result, the participants were able to reason with the application effectively.

Some participants did not believe the execution features were intrinsically obvious. It was noted that there should be a guide feature to introduce users to the individual controls of the system.

It was also noted that these features were not enough to convey new ideas to students. Most students had no prior experience of the pushdown automata. In experiencing the machine for the first time, they were able to pick-up the functionality easily, but they could not extrapolate the ideas relating to what those additional elements meant in terms of power. An application that could more actively compare the different machines together would encourage the user to identify the differences and comprehend the underlying ideas more easily.

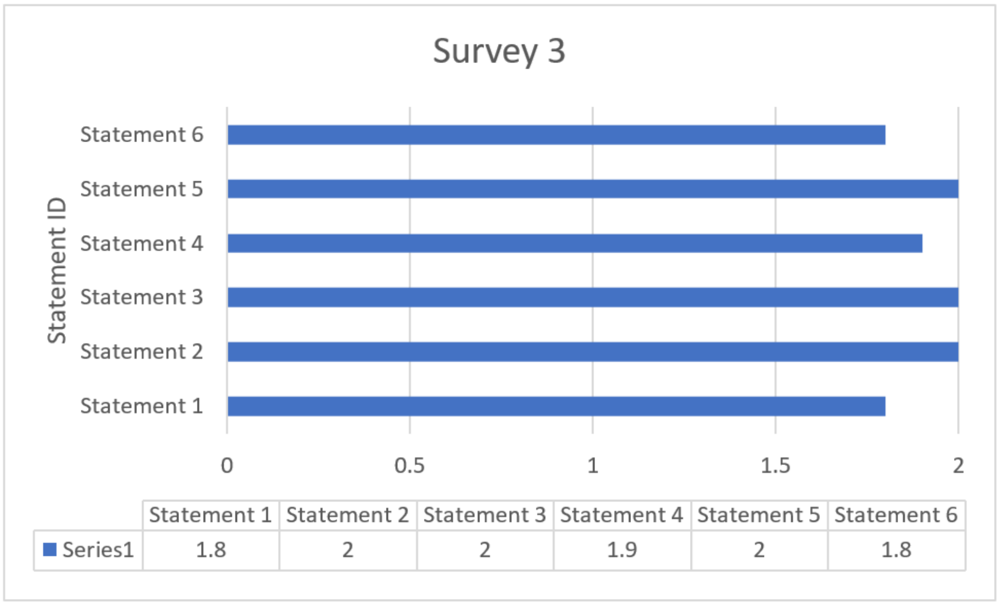
4.2.5 Auxiliary Features Analysis

In this section, we asked a number of focused questions to examine the effectiveness of the challenges and examples available within the application. The demo part of this analysis involved each participant interacting with a series of example machines and attempting a collection of challenges. Due to the constraint on time, it was not exhaustive and did not cover all aspects of these features. The entire survey for this section is available in the appendix. A summary of its findings is found below.



ID	Statements
1	You can learn how an example works by using the app
2	The examples are useful for understanding these machines
3	The range of examples are good enough
4	the range of challenges provided engage you
5	challenges are a great way to learn
6	the series of challenges are suitable

Figure 4.6: A table of statements concerning the auxiliary features of the application. These are put to the students.



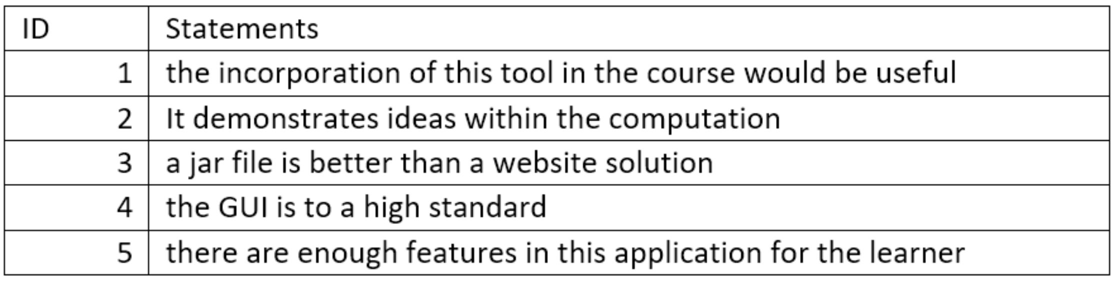
	Statement 1	Statement 2	Statement 3	Statement 4	Statement 5	Statement 6
■ Series1	1.8	2	2	1.9	2	1.8

Figure 4.7: A graph representing the average response for each statement outlined in Figure 4.6.

The average score was 1.92 across all statements. In general, the students liked the auxiliary features. They all agreed that challenges and examples were a necessary component of the application. They liked the diverse collection of examples and challenges that were made available to them. A participant noted the challenges themselves should be more elaborate. They should not just examine one thing but many things. An example of this could be to incorporate challenges which involve the user transforming a deterministic finite automaton to an equivalent pushdown automaton. This would also address our previous concern that too little comparison is made between machines.

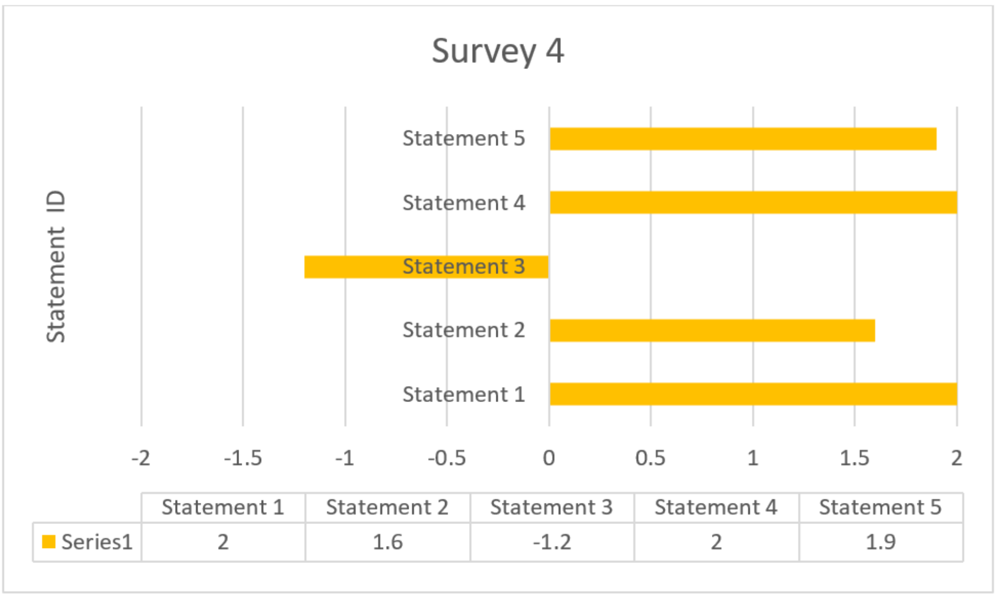
4.2.6 General Analysis

In this section, we asked a set of generic questions. These looked to explore the broader role of the application from the perspective of the student. The entire survey for this section is available in the appendix. A summary of its findings is found below.



ID	Statements
1	the incorporation of this tool in the course would be useful
2	It demonstrates ideas within the computation
3	a jar file is better than a website solution
4	the GUI is to a high standard
5	there are enough features in this application for the learner

Figure 4.8: A table of general statements relating to the application. These are put to the students.



	Statement 1	Statement 2	Statement 3	Statement 4	Statement 5
🟨 Series1	2	1.6	-1.2	2	1.9

Figure 4.9: A graph representing the average response for each statement outlined in Figure 4.8.

The users all agreed that the integration of this tool within a university course would be beneficial for the user. All participants had prior exposure to automata concepts. Participants recognised that this system represents the process of execution effectively. The respondents appreciated the graphical interface and how simple it was. They also liked the number of added features that were made available to the user.

The majority of participants believed that a web solution was the best approach for this application. They noted that a web application's ease of access would make the application more convenient for the user.

4.2.7 Summary

In summation, the participants responded well to the application. A more extensive analysis would need to take place that considers the long-term impacts. These informal experiments were useful for measuring the engagement levels of the application.

We identified from the surveys that a help feature is necessary to ensure that a user is familiar with the mechanisms of the application. As a result, we included a help section within the application that outlined the primary procedures to the user.

Chapter 5

Conclusion

We have developed a tool in this project that builds on existing alternatives. It consolidates fundamental learning strategies through its design. The tool tries to simplify abstract devices to mechanisms that the user can effectively reason with. It does this by using accurate representations and expressive functionality. The primary goal of this project was to introduce an automata simulation tool that could be integrated and used in conjunction with the curriculum at universities. We carried out a systematic review of the application and its effectiveness at achieving the central objectives illustrated in this paper. The tool definitively outperforms alternative applications. Most participants in our study strongly agreed that the application provides the necessary working features to support the student. They also noted that there is an effective use of illustrations and animations throughout the application. Unanimously, students believed that learners would benefit from integrating this tool into a study program. A more extensive analysis would need to take place that considers the long-term effects of the system.

In concluding, technology provides the capabilities to represent and teach abstract concepts in computational theory effectively. The broader aim of this project is to help realise this potential. We have outlined a process for constructing a pedagogical system that achieves this.

5.1 Future Works

The effectiveness of an educational platform can be measured on how well it communicates an idea (or a collection of related ideas) to the user. In this regard, an educational application can continually be improved by integrating more functions. The challenge arises in not bloating a system with too many features. As seen with JFLAP, the introduction of too many functions overwhelms the user.

5.1.1 Extensions

Community Integration

The application could build a community of users that actively participate. This would be in the hope of encouraging collaborative learning, where students give their knowledge of what is to be understood, collaborate with each other, give guidance, and engage in appropriate and important processes that support them — a more active learning approach. Two possible strategies have been identified below.

The first is an online repository of user-defined machines. For this feature, the user is encouraged to publish their own machines to allow others to run them. This offers the users a chance to share their conclusions with the broader community. Users can also discuss and review published machines on the platform. This will permit clarification of ideas through discussion. The second is an online repository of user-defined challenges. For this feature, the user is encouraged to publish their challenges to allow others to solve them. Users can post their solutions online. An active community would generate a broad range of challenges that would engage the user.

Migration To Web

It might be more useful if the tool is rewritten as a web application. As was identified during our evaluation, a lot of students preferred a web solution over a desktop one. Web applications have conventionally been slower than desktop programs. The recent emergence of frameworks like React has allowed web applications to be more similar to conventional desktop applications. For the current solution, we are unable to enforce updates on the user. The application at a certain point may contain bugs or lack necessary features. Such situations would require a mandatory update to fix.

A web application is more accessible to students and will enable the application to operate without any constraint within a school environment. Oracle, as of January 2019, stopped supporting Java 8 (Ora [2019]). The application needs to be continually updated to operate on modern operating systems. This will have the reverse effect of depreciating the tool for older systems that present no support for later Java versions. This application is written in Java 12.

Bibliography

Mvc - mdn web docs glossary: Definitions of web-related terms | mdn. https://developer.mozilla.org/en-US/docs/Glossary/MVC, 03 2018. (Accessed on 08/26/2019).

Oracle java se support. https://www.oracle.com/technetwork/java/java-se-support-roadmap.html, 04 2019. (Accessed on 09/06/2019).

Demographics of mobile device ownership and adoption in the united states, Jun 2019. URL http://www.pewinternet.org/fact-sheet/mobile/. (Accessed on 08/20/2019).

W. Adams, S. Reid, R. LeMaster, S. Mckagan, K. K. Perkins, M. Dubson, and C. Wieman. A study of educational simulations part i - engagement and learning. *Journal of Interactive Learning Research*, 19, 01 2008.

C. Ames and J. Archer. Achievement goals in the classroom: Students' learning strategies and motivation processes. *Journal of Educational Psychology*, 80:260–267, 09 1988. doi: 10.1037/0022-0663.80.3.260.

P. Black and D. Wiliam. Inside the black box: Raising standards through classroom assessment. *Phi Delta Kappan*, 92(1):81–90, 2010. doi: 10.1177/003172171009200119. URL https://doi.org/10.1177/003172171009200119.

C. Burch. Cburch automaton simulator. http://www.cburch.com/proj/autosim/, 2001.

A. Chun. The agile teaching/learning methodology and its e-learning platform. volume 3143, pages 11–18, 08 2004. doi: 10.1007/978-3-540-27859-7_2.

CRA. 2018 taulbee survey. pages 24–25, 2018. URL https://cra.org/wp-content/uploads/2019/05/2018_Taulbee_Survey.pdf.

J. D. Bransford. How people learn: Brain, mind, experience, and school: Expanded edition (2000). 01 2000.

K. Dickerson. Automaton simulator. www.automatonsimulator.com/, 2019. (Accessed on 08/26/2019).

J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

D. Kolb. *Experiential Learning: Experience As The Source Of Learning And Development*, volume 1. 01 1984. ISBN 0132952610.

T. W. Malone. Toward a theory of intrinsically motivating instruction*. *Cognitive Science*, 5 (4):333–369, 10 1981. ISSN 1551-6709. doi: 10.1207/s15516709cog0504_2. URL https://doi.org/10.1207/s15516709cog0504_2.

L. Neal. Implications of computer games for system design. In *Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction*, INTERACT '90, pages 93–99, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co. ISBN 0-444-88817-9. URL http://dl.acm.org/citation.cfm?id=647402.725289.

T. Page. Usability of text input interfaces in smartphones. *J. of Design Research*, 11:39 – 56, 01 2013. doi: 10.1504/JDR.2013.054065.

J. Patel. Jflap 7.1. www.jflap.org/, 2018. (Accessed on 08/26/2019).

C. N. Quinn. Designing educational computer games. In *Proceedings of the IFIP TC3/WG3.2 Working Conference on the Seign, Implementation and Evaluation of Interactive Multimedia in University Settings: Designing for Change in Teaching and Learning*, pages 45–57, New York, NY, USA, 1994. Elsevier Science Inc. ISBN 0-444-82077-9. URL http://dl.acm.org/citation.cfm?id=647111.716300.

L. P. Rieber. Seriously considering play: Designing interactive learning environments based on the blending of microworlds, simulations, and games. *Educational Technology Research and Development*, Jun 1996. ISSN 1556-6501. doi: 10.1007/BF02300540. URL https://doi.org/10.1007/BF02300540.

M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996. ISBN 053494728X.

E. Von Glasersfeld. A constructivist approach to teaching. In *Constructivism in education*, pages 21–34. Routledge, 2012.

T. White. jfast. http://jfast-fsm-sim.sourceforge.net/, 2006. (Accessed on 08/26/2019).