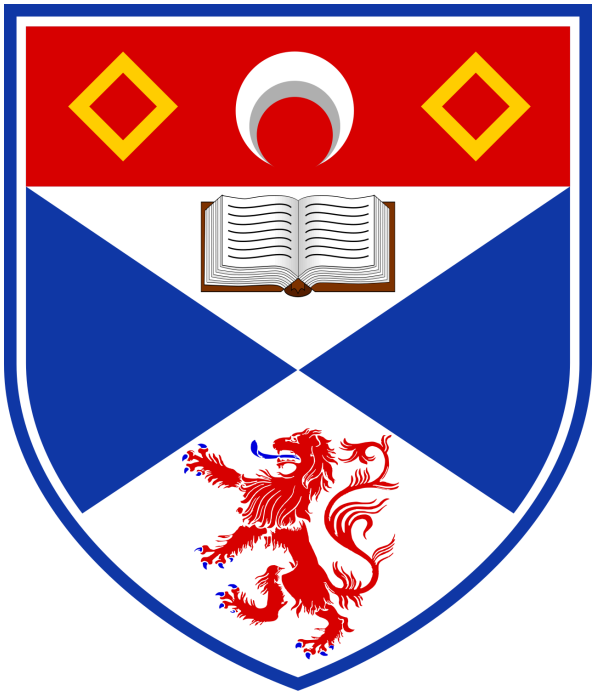
Creating a Finite Automation Simulator and Visualiser

James Hart
210010341



Project Supervisor: Dr Michael Young

School of Computer Science
University of St Andrews

4th of April 2025

Abstract

There are no highly usable applications for simulating and visualising finite state automata, leading to staff and students of the School of Computer Science to rely on publicly available unverified tools, which are cumbersome. Despite this, some quality applications exist, but most are outdated and involve unnecessary steps to use. Following the Interaction Design Process, I have developed a simulator and visualiser for finite automata with the intent for staff to use it to create finite automata simulations and diagrams for educational material, and for students to submit a created automaton for assignments. Users may graphically create a finite automaton, and run an input word on it, receiving its output. The format of the diagram may be edited, and a button to automatically organise the layout to be presentable for academic material is provided. Users may export their automaton as an image file or JSON file. The application was developed using the React library and Next.js framework to develop a dynamic, seamless user interface and easy creation of finite automata. An anonymous survey was released with a prototype of the application to gain direct end-user feedback, which has been used to improve the application's control schemes, creating a moderately usable application. Automated user interface testing was performed at each stage of the project for continuous integration, achieving 84.11% lines of code covered by automated unit tests. The results of this project are key insights into the needs of staff and students of Computer Science for finite automata simulator and visualiser applications, and the creation of an application that the department may use to simplify the creation of and support the education of finite automata.

Acknoledgements

I would like to extend a great thanks to my supervisor, Dr Michael Young, for providing key guidance, feedback, and understanding during my external difficulties in semester 1. Our collaborative talks provided direct feedback from an intended end user, which highly elevated the quality of my artefact.

I would additionally like to state an overwhelming thank you to the University of St Andrews for approving my discretionary fund and paying a significant contribution to my tuition fees, words cannot convey the immense gratitude I have for being allowed to complete this academic year.

I would briefly like to thank my girlfriend, who has been a pillar of support and security throughout my academic career.

Finally, I want to state the biggest thank you to both of my parents for their immense sacrifice, their great hardships and their support, which allowed me to attend this university and not only provided me with unforgettable experiences but also the ability to pursue my passion at the highest available level, a luxury that I am honoured to have received.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 17,222 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Disruption

Unfortunately, in October 2024, near the beginning of the project, my family experienced extreme financial hardship and as a result, were unable to afford my university fees as an international student for the remainder of the year. Due to my desire to complete my degree this academic year, much of my time during the first semester was spent researching, contacting and applying to numerous organisations for funding, as any job I would be eligible for would not cover my remaining debt to the university. Thankfully, this time spent allowed me to receive funding and complete the academic year.

As my other three modules in semester 1, totalling 45 credits, did not exceed one semester, I prioritised those modules with the remainder of my time and as a result, had very limited time to work on this project. Within the first semester, I completed my research, initial design plan, project setup and an incomplete implementation of my first task. Despite difficulties, I ensured to keep my supervisor informed throughout the disruption and still attended weekly meetings discussing the project. I properly began development during the winter break and worked consistently on the project until the final two weeks of submission, in which I prioritised the report. These time constraints were the primary factor in many of my secondary goals being uncompleted.

Contents Page

Introduction

The staff and students of the School of Computer Science do not have a highly usable application for simulating and visualising finite state automata for educational purposes. To resolve this, I have been successful in creating a dedicated application that allows staff and students to create a finite automaton, graphically display and export the automaton's state diagram, and run any valid input word on it, providing them with a usable tool for simulating and visualising finite state automata.

Background

To introduce the project, the following section will introduce the terminology used to outline the key concepts of the project, briefly explaining what a finite automaton is as well as what is meant by a simulator and visualiser.

Finite State Automata

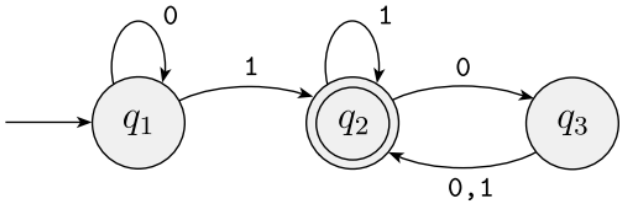


Figure 1: Example of a Finite Automaton [12]

A finite automaton represents the simplest model of computation; it contains multiple states, represented as circles, in Figure 1, these would be q1, q2 and q3. An automaton can read a word containing multiple characters as a String, and transitions from state to state based on which character is currently being read. These transitions are specified with the arrows; in Figure 1, we can see that if the automaton is in state q1 and the '1' character is read, the automaton will transition into state q2. The state with an arrow with no character on the left is called the start state, which every automaton has. Finally, any state with an internal ring is an accepting state; if the computation of the input string ends on an accepting state, then the automaton accepts the word, and if not, rejects it. For example, the word '101' would be accepted. A transition originating and ending on the same state is called a self-pointing transition and does not change the state. An example input is '001', which would be accepted as any number of 0s may be entered in state q1 without transitioning.

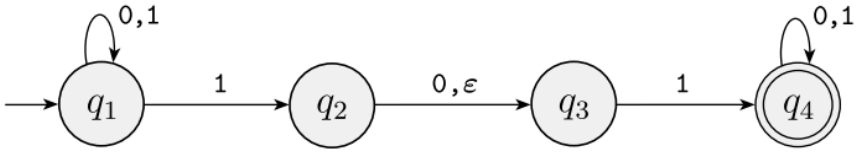


Figure 2: Example of a Nondeterministic Finite Automaton [12]

There are two types of finite state automata; Figure 1 shows a deterministic automaton, where each state specifies exactly the same number of transitions for each valid input character. A deterministic machine does not have to make any choice when running input, all inputs result in the same output. A nondeterministic finite automaton, which can be seen in Figure 2, contains multiple options for input characters at different points. An example is state q1, where if the character '1' is read, it can either transition to q1 or q2. The ability to represent multiple options in an automaton is key in representing many real-world systems where the algorithm is forced to make a choice between two states. The empty word, commonly represented as 'ε', may only exist in nondeterministic finite automata as it does not count as a character and allows the automaton to transition to another state without ticking over the current character. For example, an input for Figure 2 is '10' and could finish in either q2 or q3, as the empty word represents a blank character. The role of nondeterminism allows us to model computational algorithms and explore solutions to problems simultaneously by representing multiple different paths, proving to be a significant tool in branch-based algorithms.

Report Terminology

For the sake of this project, I will define a 'finite automaton simulator' as a device which may accurately reflect the behaviour of an automaton by returning the correct result upon running any input word for the specified automaton. I will define a 'finite automaton visualiser' as a device which may graphically display an automaton's state diagram.

Within this report, common abbreviations may be used, such as:

- Finite State Automaton (FSA), or referring to this as a Finite State Machine (Machine)
- Deterministic Finite Automaton (DFA)
- Nondeterministic Finite Automataon (NFA)

Problem Statement

Currently, there are no highly usable applications for visualising and simulating finite automata for the staff and students of the School of Computer Science for educational purposes. Because of this, most lecturers design educational material such as lectures and tutorial sheets by using online publicly available tools that lack certain features which would make automaton creation easier. Additionally, when students are assigned practical work to create automata diagrams, the

school lacks a robust, well-tested in-house application for students to use, forcing them to use a variety of unknown tools, which may produce diagrams that vary.

Objectives

The following are the objectives for this project requested by my supervisor ot create a useful application for educational purposes, prioritised as primary and secondary goals. Further discussion of these can be found in the Requirements Specification, with a summary of my achievement may be found in the Requirements section of the Evaluation.

Primary

The application should:

- Allow users to create finite automata with their diagrams graphically displayed
- Allow users to choose and manipulate the exact positions of nodes
- Verify that the automaton is valid
- Allow for both deterministic and non-deterministic finite state automata
- Provide users with a way to automatically reconfigure the layout of the diagram to be in an organised and presentable manner, suitable for lectures
- Allow users to export the finite automata diagram in at least one known image format
- Allow the user to simulate an input word on the finite automata by displaying its path, output and whether the word was accepted or rejected

Secondary

The application may:

- Allow users to control the exact angle of edges between nodes
- Implement additional exporting formats such as SVG, PNG and LaTeX
- Utilise animations when simulating an input word to clearly show the path taken
- Allow users to save a video file of the simulation
- Allow users to create their finite automata configuration through text entry, which uses a specific language to communicate the automaton's formal definition
- Have a high level of usability by staff and students of Computer Science
- Perform automated testing of all parts of the program, including the user interface

Report Structure

The report begins with my research into other similar simulators and visualisers, detailing my key takeaways from the most prominent ones. Then, the process of my project is discussed, detailing a breakdown of my requirements, my software engineering process, and ethical information. The main part of the report is my Design and Implementation sections, which explain my design process, choices and evolution of the application as well as key algorithms

and how the application was implemented. After this, there is a summary of how testing was conducted and then an evaluation and critical appraisal of the artefact produced, with respect to my original objectives and other works. Finally, my project is summarised in the Conclusion along with detailing what future work I would conduct if this project were to be continued or extended.

Note that all the following state diagrams were made using my application.

Additionally, the final application is currently hosted on the school servers and is accessible by following this hyperlink on the school network: https://jeh27.teaching.cs.st-andrews.ac.uk/

Context Survey

This chapter aims to summarise the most significant publicly available applications that can simulate and visualise finite automata. The summary will start with a broad overview of the landscape of applications and then detail relevant examples, which I intend to consider when designing my application and later evaluate my application against.

There is a vast number of publicly available visualisers and simulators for finite automata in various forms, each with varying levels of research behind them. While some applications of good quality are usually the first ones to appear upon a Google search, these are mostly independent projects without any known research behind them or a clear end-user. As such, I will only be discussing the most notable applications created as a part of academic research and intended to support Computer Science education, as these are the most credible and align with the objectives of my application.

'Fifty Years of Automata Simulation: A Review' [24] was published in an academic journal in 2011 and reports on numerous finite automata visualisers created to support university education. Each application varies in what exactly is being simulated, whether it is DFAs, NFAs or Turing Machines, however, all allow a user to create some state machine and run an input word on it. Many are reported to have 'high-quality graphics' [24], use animations when simulating an input for easy interpretation and have conducted surveys or collected direct feedback from students on the application's usability, reporting it has facilitated their learning of state machines. As time progresses, the quality of the user interface across applications improves as each simulator is 'highly influenced by the software development tools available at the time of their development' [24]. As most simulators were developed prior to 2015, the need for a new application reflecting current user interface trends seems appropriate to appeal to incoming and future students.

finsm.io

In October 2024, Cornell University published finsm.io [19], a web application for building, simulating and exporting Finite State Automata to improve learning and teaching. DFAs and NFAs may be built and exported to LaTeX code, and when prototyped for students, have successfully improved learning. While the application is highly usable, the modular approach to building, simulating and exporting slows down automaton creation as the three are separated into three modes, which the user must toggle through. Additionally, some features require opening another window, such as simulation and documentation access, which can slow development. Despite this, the application maintains a minimalist design and stands as the most usable and current application for simulating and visualising finite state automata.

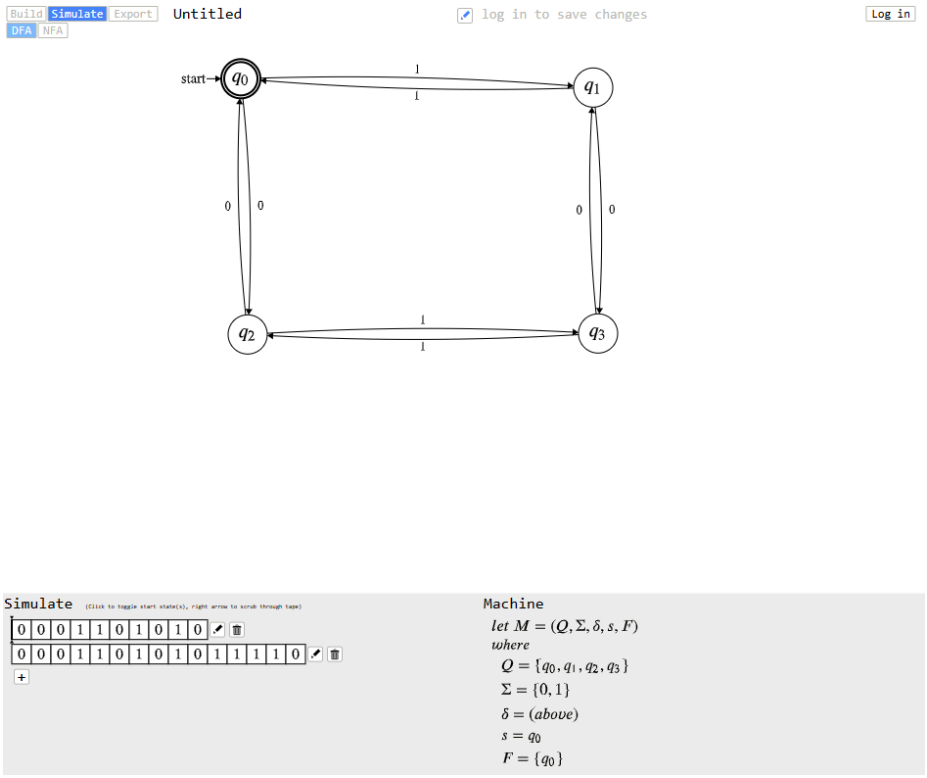


Figure 3: Example DFA in finsm.io

JFLAP

Developed in 1990 and consistently updated until 2018, JFLAP (Java Formal and Automata Package) [22] is regarded as the most sophisticated application for visualising state machines. The application can visualise and simulate: DFAs, NFAs, pushdown automata, Turing Machines, regular expressions, pumping lemmas, context-free pumping lemmas and more. It contains extensive documentation on its website and frequently provides patches based on user feedback. In contrast to finsm.io, the application was developed in Java and must be downloaded. While

finsm.io operates using hotkeys and clicking on the machine itself, JFLAP operates through a toolbar.

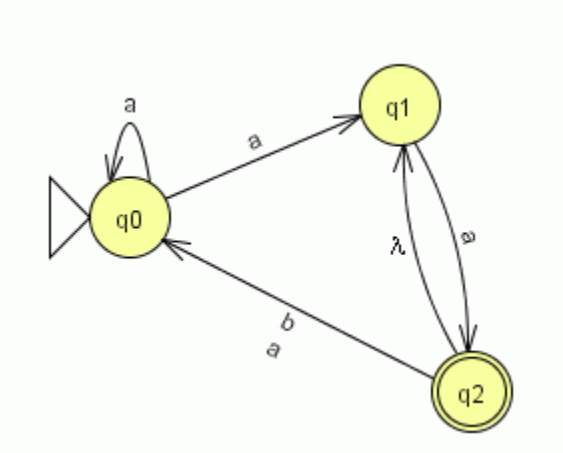


Figure 4: Example NFA in JFLAP

jFAST

Finally, jFAST [20] is an extensively researched application made in 2006, using applications like JFLAP to improve upon. DFAs, NFAs, pushdown automata and Turing Machines may be created and simulated, however, the display is not reflective of common finite state machine notation and has a UI mimicking Windows XP developed in the early 00s. Notable features are the ability to zoom in and out for larger diagrams and utilising drop-down menus upon right-clicking for most operations. A survey was conducted on students, with the majority finding jFAST 'easy to learn and use for creating FSMs' [20].

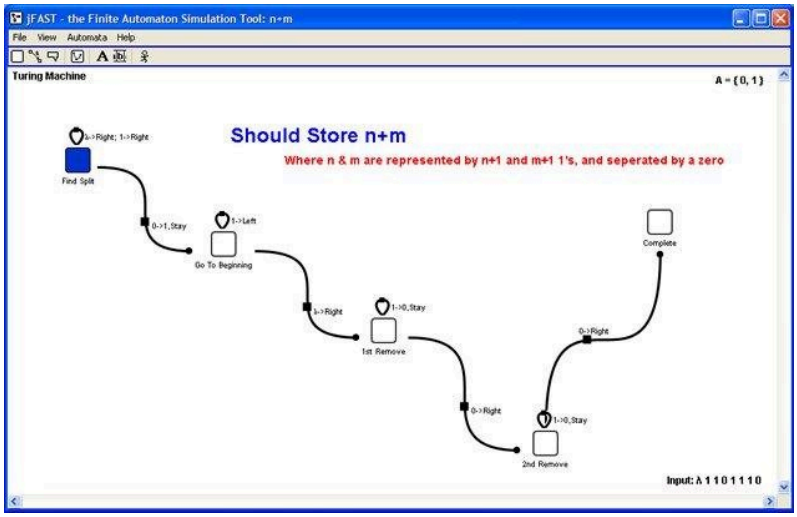


Figure 5: Example Turing Machine in jFAST

Key Takeaways

To summarise, numerous publicly available applications exist for visualising and simulating not only deterministic and nondeterministic finite automata, but pushdown automata, Turing Machines and more. Many of these are backed by academic research of previous iterations, each with varying methods of interactivity such as hotkeys, toolbars or popups, reflecting the current trend of user interface prevalent at the time of development. Despite the success of these prior applications, there is no application designed to reduce the time and complexity taken to build finite automata. Most applications currently use a design which hinders FSA development, such as the toolbar requiring interaction with another tool to change the automata instead of editing it directly, or the separation of build and simulate modes in finsm.io, given users are likely to develop and test their FSA simultaneously I find this to be unintuitive, leading to constant swapping between modes. Given that staff and students already have busy schedules and limited time, I aim to design an application which allows them to build, simulate and export finite automata with as little effort and time taken as possible, so the focus may be on the creation of the automata instead of navigating or using the application provided. Additionally, many of the applications seen do not represent state diagrams with universal minimalist notations, as seen in Sipser [12], given one purpose of the application is to showcase finite automata to students, it seems appropriate to design the state diagrams with a minimalist design that most computer scientists will be familiar with.

Requirements Specification

This project aims to develop a highly usable standalone application for creating, visualising and simulating finite state automata. The application is intended to assist students and staff for teaching purposes, such as creating finite state diagrams for lectures or allowing students to design finite state machines for practical work.

The goals for this application were prioritised into primary and secondary, with the primary focusing on core functionality and the secondary focusing on enhancing usability and additional features. At its core, the application must allow users to design a finite state machine, be able to run various input words on it (showcasing the result) and export the diagram as an image.

Primary

The application should:

- Allow users to create finite automata with their diagrams graphically displayed
- Allow users to choose and manipulate the exact positions of nodes
- Verify that the automaton is valid
- Allow for both deterministic and non-deterministic finite state automata

- Provide users with a way to automatically reconfigure the layout of the diagram to be in an organised and presentable manner, suitable for lectures
- Allow users to export the finite automata diagram in at least one known image format
- Allow the user to simulate an input word on the finite automata by displaying its path, output and whether the word was accepted or rejected

Secondary

The application may:

- Allow users to control the exact angle of edges between nodes
- Implement additional exporting formats such as SVG, PNG and LaTeX
- Utilise animations when simulating an input word to clearly show the path taken
- Allow users to save a video file of the simulation
- Allow users to create their finite automata configuration through text entry, which uses a specific language to communicate the automation's formal definition

Additional Objectives (Secondary)

An additional objective refers to an objective that my supervisor and I agreed on in the middle of development that would improve the quality of the project and was added to the list of requirements for the project.

The following additional secondary objectives were added:

- Have a high level of usability by staff and students of Computer Science
- Perform automated testing of all parts of the program, including the user interface

Originally, my supervisor and I did not consider usability as a goal, focusing more on the functionality of the application; however, given my research into the current landscape of applications and the project's intended use, we felt it appropriate to make usability for staff and students a secondary objective. With my supervisor and I agreeing to add this objective, the application will not only be functional, but additionally be an improvement upon publicly available applications and make it easier to produce finite automata for teaching purposes.

Furthermore, my supervisor and I decided to add a secondary objective to perform automated testing on all parts of the application to ensure it was well-tested and developed with continuous integration. We chose to emphasise the testing of the user interface to support the objective of achieving a high level of usability and to ensure the interface contains no faulty edge cases.

Software Engineering Process

I developed the application using an Agile [16] approach and Scrum [53] methodologies to achieve a flexible, consistent and user-focused development process. Agile is an umbrella term for frameworks focused on working flexibly on a long-term project and adapting to change, containing core principles that target production of short, frequent tasks for routine evaluation and adaptation. Scrum is a methodology focused on delivering consistent results in a controlled, collaborative manner, which frames tasks in the form of user stories, promoting a user-focused development cycle. Both Scrum and Agile are intended for collaborative teamwork however, I have adapted them for a single developer project by taking on small portions of each role in a team such as a Developer, Quality Assurance, Scrum Master and Product Manager, treating my supervisor meetings as weekly Scrum meetings to reasses the quality of deliverables.

These methodologies were used in both my junior honours project and summer internship in a software development team, in which I found creating user stories, working in sprints and using a Kanban board very effective in task management and productivity. Agile methodology felt most appropriate due to its focus on adaptability, as I was handling multiple deadlines simultaneously and receiving weekly feedback from my supervisor, which allowed me to adapt and shift priorities throughout both semesters as needed. Moreover, Agile's user-focused design principles aligned with the goals of my project.

At my internship, we were taught how to effectively write user stories and track their progress on a Kanban board. I used the website Trello [17] to adopt this by breaking up all of my primary and secondary goals into manageable user stories and acceptance criteria, ensuring development was focused on the user and original objectives (Figure 6). The Kanban board was very effective in tracking my progress and encouraged me to only focus on one user story at a time and avoid splitting my focus between multiple tasks.

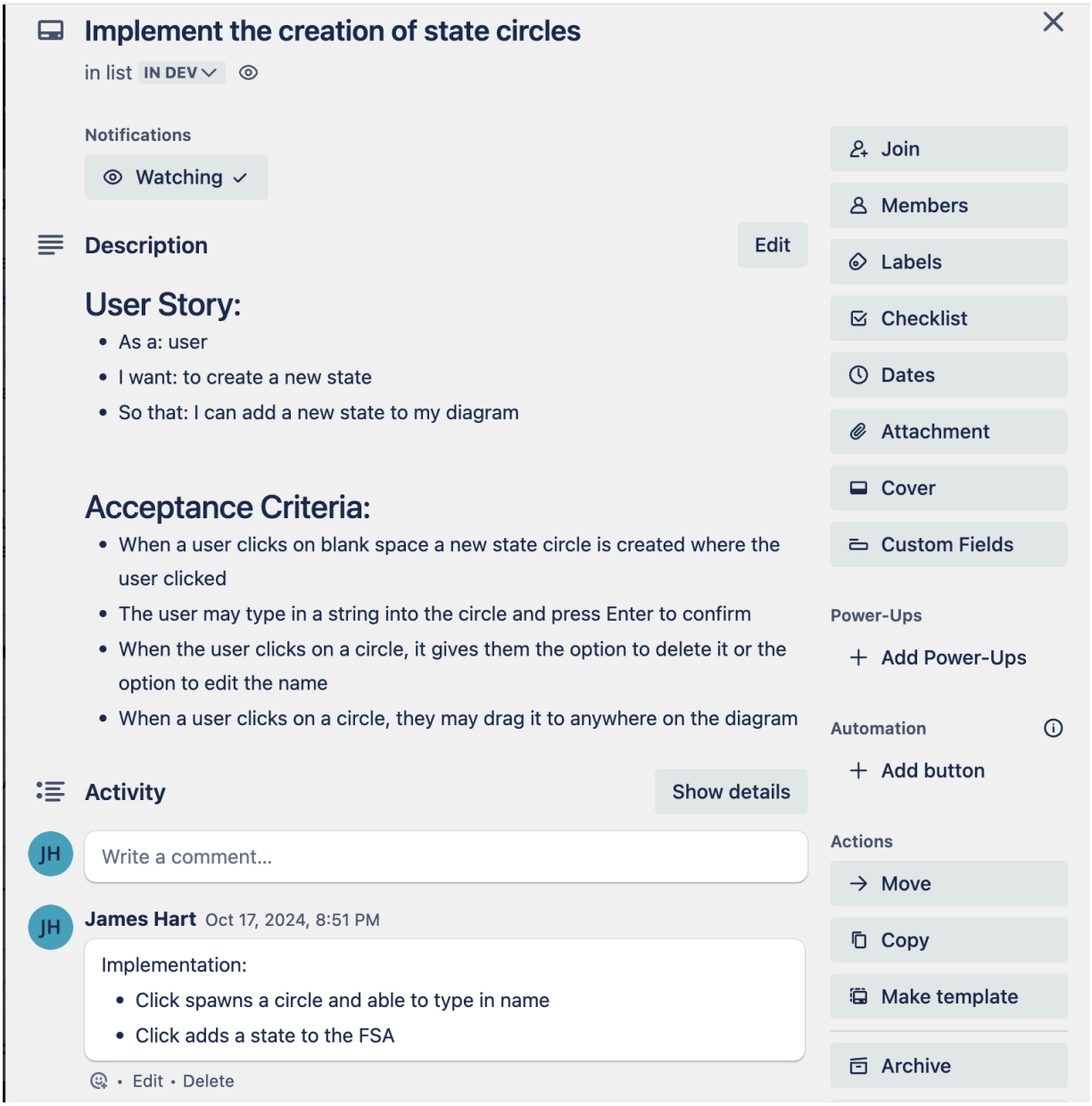


Figure 6: User Story - Task 1

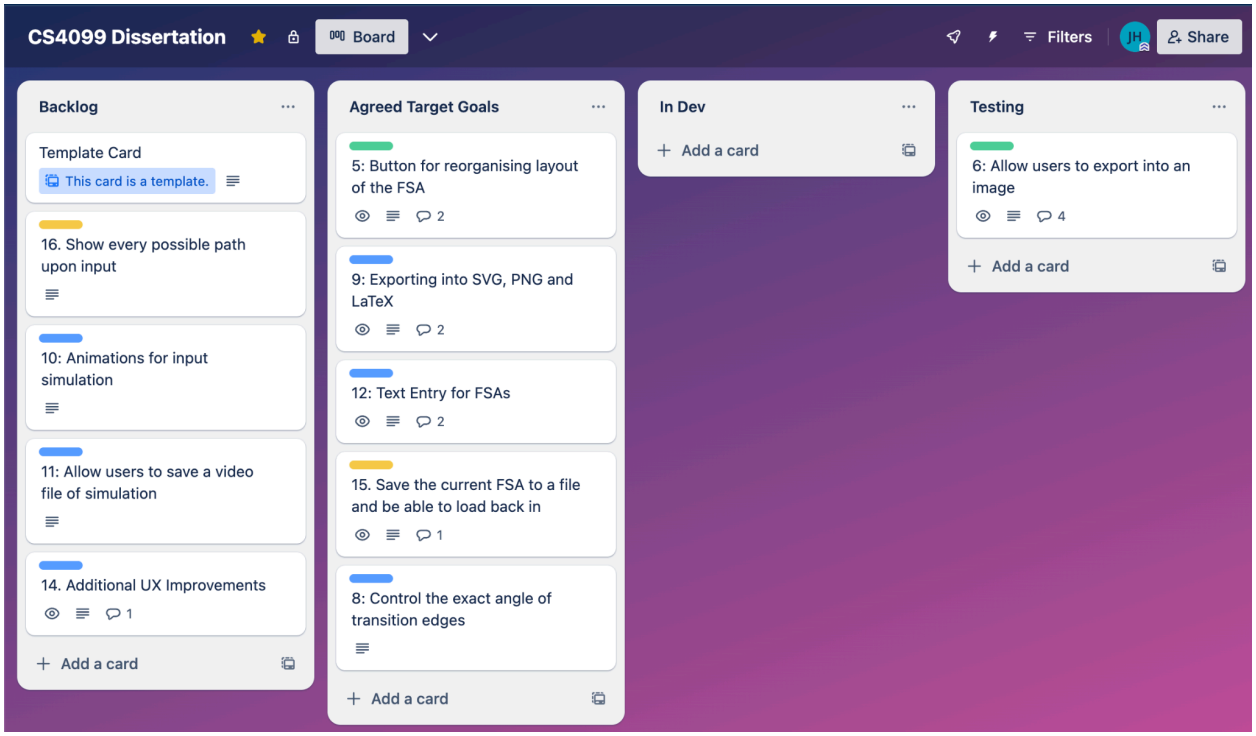


Figure 7: Portion of Kanban Board using Trello

My Kanban board maintained the following lists:

- **Backlog:**
  - Stories for future development
- **Agreed Target Goals:**
  - Stories to be completed within the current sprint
- **In Dev:**
  - Being developed
- **Testing:**
  - Being tested with Jest and React Testing Library
- **Commenting:**
  - Comments being added
- **Documentation Notes:**
  - Writing up the design and implementation for this story
- **Waiting For Approval:**
  - Story is completed and waiting to be showcased at the next supervisor meeting
- **Done:**
  - Supervisor approved and story branch has been merged to main
- **Bugs Remaining:**
  - Bugs found during development/testing
- **Bugs Done:**
  - Resolved bugs

Each time a user story was moved to ‘In Dev’, I created a new branch for that story to isolate development. Once the story was in ‘Done’, its branch was then merged into main to add its feature to the project after it had been tested and approved. The ‘Bugs’ lists were very useful in bug fixing as they gave me a place to note down bugs during development, which I usually came back to fix near the end of feature development, ensuring I stayed focused on the current task.

Due to my experience in automated testing for GUIs, my supervisor and I decided to make automated testing a secondary goal. I made sure to practice continuous integration using an automated testing framework throughout the project. When a story was in ‘Testing’, I ran my testing framework to check that all current tests were successful. After this, I checked the code coverage report and aimed to add more unit tests until the report confirmed that 100% of my source code lines were covered by tests. This process was difficult to achieved and resulted in a very robustly tested application, more is covered in the Testing section. Once all tests were successful, only then would I move the story to ‘Commenting’. After commenting was done, I moved the story into ‘Documentation Notes’, which is where I wrote up notes for both the design and implementation sections of the report for that specific feature, which proved very useful upon writing the report.

While Scrum is more applicable to a team of developers, I used aspects of Scrum, such as weekly sprints, taking on a portion of each role in a development team. Using my weekly supervisor meetings as a checkpoint, I prepared a record of each week using the following columns to discuss each meeting in OneNote [21]:

Date	**What has been done**	**Plan for next week**	**Things to Discuss**	**Notes from meetings**

Table 1: Weekly Sprint Record
(Full table may be found in the Appendix)

Working in weekly sprints kept me accountable and ensured that each week I focused on tasks with the highest priority, having discussed them with my supervisor.

Ethics

As usability is a priority for the application, my supervisor and I agreed to advertise an anonymous survey to all staff and students in the School of Computer Science to gather feedback on the usability of the application after most of the core functionality had been completed. Given that the questionnaire collected no information about the participant, whose identity remained anonymous and only asked questions about the artefact, the project was covered by ethical

application CS15727. A signed copy of the Artefact Form may be found in the Appendix, along with a copy of the survey sent out and the attached participant information sheet. The survey was conducted using Qualtrics [18] and completely abides by ethical application CS15727, which collects no participant information and deletes partial responses and all collected data after the completion of this project. A PDF containing the survey's questions and all participant responses is located in the appendix.

Design

This section details and explains the design decisions of the application throughout the project, including my design process, design of required features and interaction method for users. It begins with the initial design I created upon starting the project, given my key takeaways from other similar applications. Then the details of the anonymous survey that was sent out with the prototype are presented, including my analysis of participant responses. This section concludes with the design of my final artefact, detailing the changes I made based on participant feedback and new features to complete more of my objectives and increase usability for the target end users.

Initial Design

Interaction Design Process

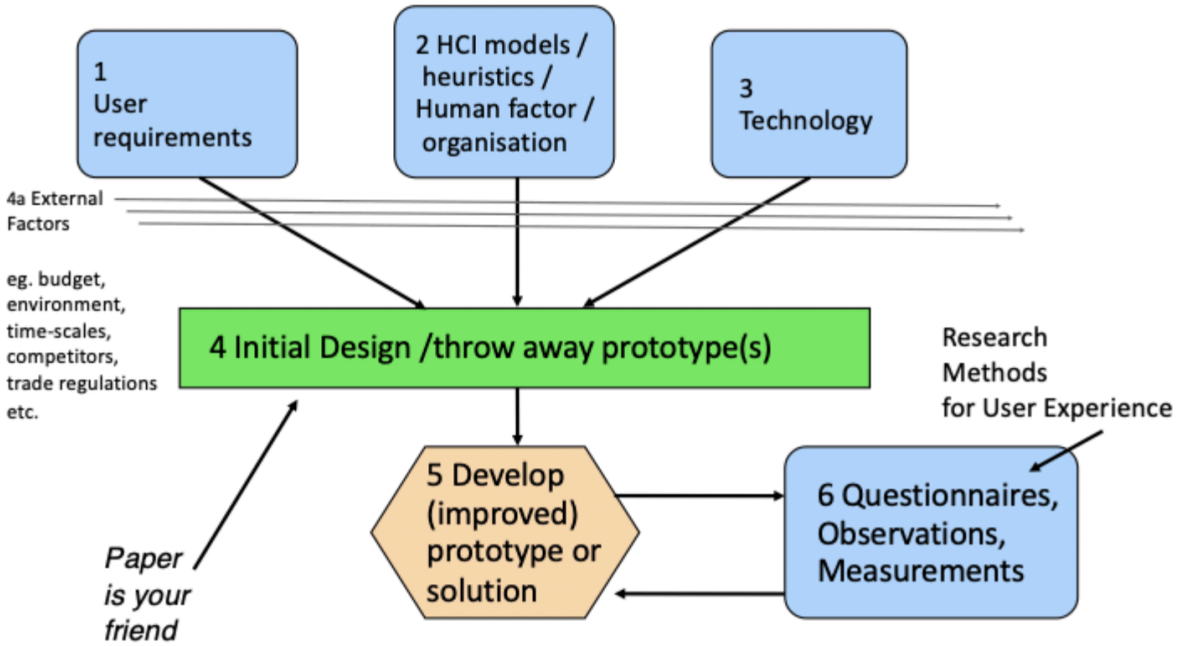


Figure 8: Interaction Design Process taken from CS3106 Lectures 2023 [36]

Having taken CS3106 (Human Computer Interaction) in the previous year, we were taught about the Interaction Design process for developing user-focused applications. I used this to allow me to create an initial version of an application based on my implementation-focused primary goals and key takeaways from my research, before focusing on enhancing the application's usability. After this, I would send out a questionnaire to staff and students of Computer Science to gather their personal preferences for how the application should interact with its users, allowing me to develop the majority of my usability with the preferences of its end users in mind. I chose this design process over others, which begin with user inquiry (such as the Rapid Contextual Design Method [36]), as it allowed me to start developing immediately and focus on achieving the implementation goals of my application, which were of a much higher priority than my goal of usability.

I followed Jakob's 10 Usability Heuristics [34], which I studied in CS3106 and found highly effective. A poster explaining each heuristic may be found in the appendix. A summary of how my design has followed these heuristics will be found in the evaluation. After this, I chose my technologies (discussed in Implementation) and created an initial mockup.

Initial Mockup

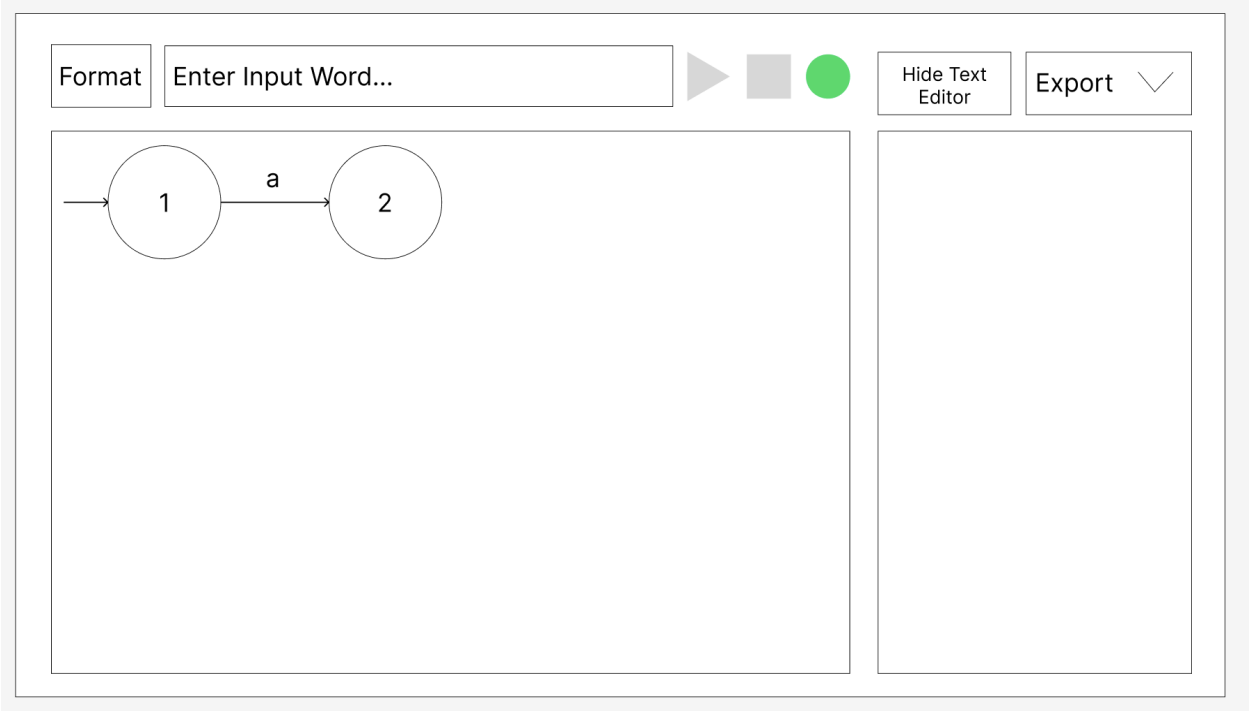


Figure 9: Initial Mockup using Figma [35]

Based on my research of current automata builders, my initial design focused on simplicity and speed of use. I wanted the application to be a very quick tool that would not impede development through a difficult or intrusive user experience. I chose to develop a web application, as these would be easily accessible by URL without requiring any installation. As such, the button or tool for every requirement may be found on one page, avoiding any navigation or switching between modes for simultaneous simulating and visualising. Users would build their automata using the viewport (bottom left box), reformat it using the top left button, run any input on it using the input bar and press play.

To better showcase the flow of an input on the user's machine, each arrow of the input's path would grow and shrink to highlight the input word's path. As this would not be instant, the stop button may be pressed to halt input. The green circle would remain green if the machine was valid, or turn red if it was not. The right side of the screen would contain an editable textual version of the machine, which would update the machine's diagram dynamically. This could be hidden to hide the window and extend the viewport across the entire screen if the user just wanted to edit visually.

Finally, the export drop-down allows users to choose which form to export the machine in. I aimed to include all of these features without it feeling cluttered, and I aimed to implement this mockup with fully working functionality before surveying for usability preferences. To achieve top development speed, any changes made to the FSA would be dynamic, with both the textual and graphical representation matching at all times.

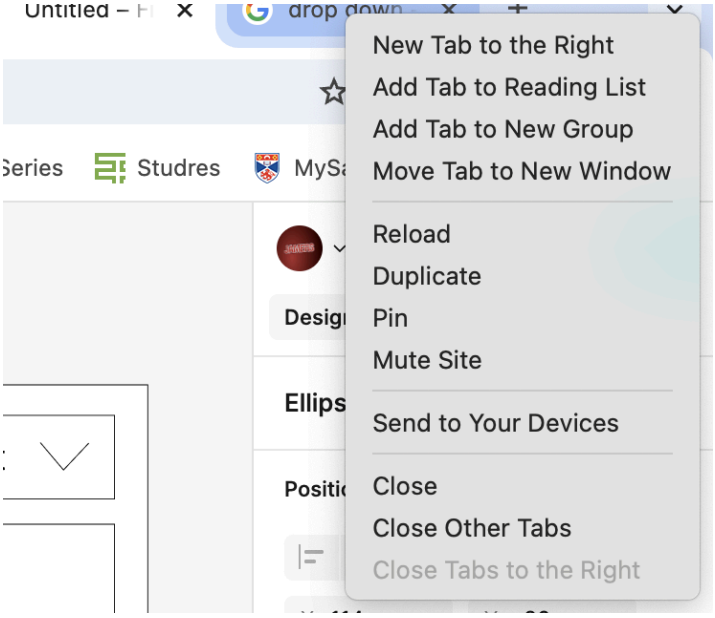


Figure 10: Right-click drop-down on Google Chrome [37]

My initial design for the controls to edit/build the automaton was for users to right-click on the viewport, states and transition arrows to present a list of options which they would click on, similar to right-clicking on most web browsers. However, I felt this might be intrusive, blocking some of the diagram and given most browsers already use the right click, my assumption was that I could not develop using right clicks as the key was already being used. An alternative control scheme I found was hotkeys (combinations of clicks and keys), a lot of current visualisers utilise hotkeys to create their diagrams, and while taking some time to learn, I found these to work suitably well for development, given their speed.

Hotkey controls for the initial prototype application:

- Click on a blank space: Creates a new state at the mouse's position.
- Clicking on a state: Allows the user to change the state's name.
- Dragging a state: Updates the state's position.
- Shift + Click (Two States) Creates a transition between the two states.
- Shift + Click (One State) Creates a self-pointing transition.
- Alt + Click (State) Deletes the state and all connected transitions.
- Alt + Click (Transition) Deletes the transition.
- Double Click (State) Toggles whether the state is an accept state or not.
- Shift + Alt + Click (State) Turns the selected state into the start state.

Most hotkey and functionality pairings mimicked the controls for other automata builders and changed through what I found most comfortable during development. A final note is that all connected transitions are deleted when a state is deleted to avoid hanging transitions.

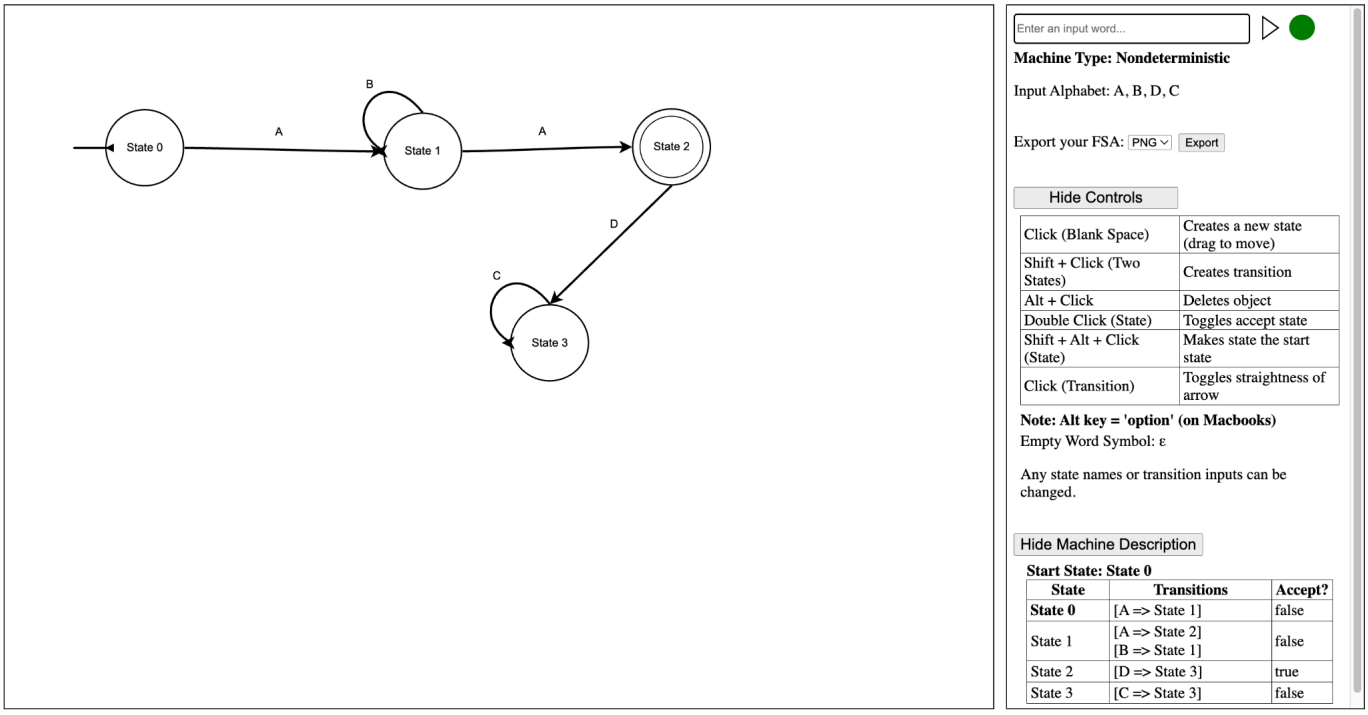


Figure 11: Example FSA within the released prototype

Figure 11 showcases a screenshot of the initial version of the application, which was hosted on the lab machines and used to gain participant feedback in the anonymous survey. The biggest change from my mockup is that the viewport takes up more of the screen space to allow for the development of more complex automatons with larger diagrams, and the creation of the interaction window on the right.

Having two clear sections keeps the viewport clear of clutter and allows user to create a consistent workflow between editing their diagram and testing it by running input and viewing changes to the FSA dynamically, with the interaction window holding all the tools needed for my specification requirements. Much of the right window lacked styling as I prioritised functionality at this time. The second half of the interaction window contains two toggleable tables, one for controls so users may quickly have these shown at all times when learning the tool and a description of the machine to verify that changes to the diagram reflect those changes to the stored FSA in real-time.

Looking above the controls table near the input bar, I removed the stop button as I hadn't implemented animations, so running an input was instant. Running input words on the FSA results in a pop containing either stating whether the machine accepts/rejects the word or an error if either the word or machine is invalid, I chose to consistently use a popup for this to not have to move any components on the screen and to ensure the message was clearly seen. At this time,

only the path for a DFA is shown due to the complexity of path finding for NFAs. The machine type (Deterministic, Nondeterministic or Invalid) and output of the machine's input alphabet were to aid FSA development. The green circle on the side turns red when the machine is invalid to clearly indicate to the user the validity of the machine. When invalid, the machine cannot run input, and the app displays a warning when exporting the machine to prevent errors. Finally, I chose the FSA design to match the notation used by Sipser [12] shown in the Introduction, given that this textbook is favoured by current lecturers and familiar to students, having been used for teaching in CS3052. Unfortunately, the arrows for the start state and self-pointing transitions were not properly formatted at the time of survey release due to time constraints.

Anonymous Survey

Following the Interaction Design process, now that an initial version of the application had been completed with 6/7 primary goals implemented, I created and released an anonymous survey to staff and students of Computer Science to provide feedback on the usability and intuitiveness of the application. I created the survey using Qualtrics [18], including all required sections of the Artefact Form and questions approved by my supervisor. The questions, responses and participant information sheet are in the appendix.

Survey Questions

1. **Overall, how would you rate the artefact as a tool for building finite state automata?**
   Given that the participants are the end users of the application, gaining measurable qualitative data about how useful they found it would allow me to gather a consensus on the usability of the application. This consensus would allow me to evaluate the effectiveness of my application, and relate their scoring to their thoughts in the following question to identify how prevalent each thought was in the scoring, to know what most needs to be improved.

2. **Please elaborate, indicating what the artefact does well and what may be improved.**
   This was chosen to gauge participants' immediate thoughts on the application's strengths and weaknesses and collect thoughts that may not be directly prompted by the following questions.
3. **How intuitive did you find the artefact to use?**
   Given that the more intuitive an application is, the more usable it is for new users, this question aimed to gauge how effective the controls were in being intuitive to make the application more usable.

4. **Was there anything that you found slowed down FSA development?**
   Given my design focused on speed and ease of use for the user, this question aimed to gather information on what hindered users' experience of the application, as if something slowed them down, it must have prevented them from using it at a comfortable speed.

5. **How did you find the design of the user interface? Would you prefer your FSA to be displayed in any alternative way?**
   This question aimed to establish trends or a consensus on what users liked and disliked about the user interface, to assess what to improve and what to change. The follow-up question directly asks their opinion on the notation of the machine. I added this as many other automata visualisers use different notation, and I wanted to tailor the display to benefit users the most.

6. **Do you feel that there are any features missing from the application? Or anything useful that could be added to make it easier to use?**
   This question was designed to be very open-ended and to allow staff and students to comment on what features they would like to be added, which I could then add to make the application more usable for them.

7. **Did you encounter any errors or unexpected behaviours while using the tool? If so, how did the tool handle them?**
   This question allowed me to use the participants of the survey to test the application and report on any edge cases they found when using it. Given that they were all new users, this would be great for debugging to make the application more secure.

8. **If you used similar FSA-building tools before (either online or university-provided), how does this artefact compare in terms of usability?**
   I asked this final question to again assess the quality of the application, but additionally, learn what features of other similar applications participants liked to incorporate into my application to increase the usability or capabilities.

Analysis of Survey Responses

Participant Ratings of the Application Out of 10 (Sorted - Ascending)										
Overall	3	3	5	6	6	6	6	6	7	8
Intuitiveness	3	3	4	4	4	5	6	6	7	7

Questions 1 and 3

Both questions 1 and 3, which ask participants to provide a rating from 1 to 10, with 1 being the worst and 10 being the best, produced very mixed results about the overall quality of the application. The average rating of the overall quality of the application was 5.6 out of 10, a minimum rating of 3 and a maximum of 8. Despite this variety, responses were mostly consistent in what they found worked well and what they struggled with, with their level of ease upon using the hotkey controls being a primary factor towards a reduction in rating. Looking at the ratings, most participants found the application just above average as an application for building finite automata, with most finding the application unintuitive to use.

What Participants Liked

All participants found the user interface and format of state diagrams desirable, with some stating the application had a ‘smooth interface’ and ‘clean design’, stating they favoured a simple design clear of distractions. Additionally, some participants stated they preferred this tool over previous visualisers for finite automata due to the additional features it offers, such as image exporting and displaying machine type and input responses. Participants favoured the ‘seamless’ nature of the application, with one stating, ‘It is easier to use than a previous tool I used in CS3052, as that required use to manually describe the FSA using a specific text format. Therefore, this was quicker and simpler to use’, and another praised the ‘seamless creation of transitions’. Finally, all participants report no errors with their finite automata, stating everything is ‘functionally valid’, only reporting visual bugs with the diagram. My takeaway from this is that staff and students appreciate the simple display and unintrusive creation of their diagram, and that all functionality operates as intended, with no major bugs being reported.

Majority Criticism of Control Scheme

Alternatively, while all were able to successfully create and run an input on their automata, most struggled with the hotkey controls, with all reporting some form of issue when using them, such as being difficult to use or accidental actions being performed. Participants reported that the ‘control scheme is very poor’ and that ‘remembering the hotkeys is annoying’, overall being the primary factor for many of the lower ratings for usability and overall quality of the application. Accidental actions due to similar or unintuitive controls were: creating unwanted states when clicking on the background, deleting a critical state when attempting to make it the start state and incorrect transitions being created when using shift clicks. My deduction from this feedback is that the controls are too similar and should not use the same keys, for instance as to create a transition uses a shift click, deleting a state uses alt click and setting a start state uses shift alt click, this lead to numerous accounts of misaction as users forgot to press one of the tree keys needed to assign a start state. Additionally, as a transition is created by selecting the origin state and then the destination state, users would not hold shift and click on a transition, believing they had set the origin state for a transition, then going to shift click on a destination state when infact they were setting the origin state, confused when no transition was created, they attempt to create

the transition again which selects the origin state as the destination, created an incorrect transition. Finally, accidentally creating a new state by clicking on the viewport is due to how exact the pointer must be; users clicking just past the edge of a state circle would create a state that overlaps their intended one. Moreover, for those with fewer issues with the controls, they still found them 'unintuitive' and 'finicky', reporting they slowed down development and proved difficult to utilise. One participant expressed that the combination of mouse and keyboard felt difficult, having to utilise two control schemes and suggested that the application use only mouse or keyboard controls. I suspect this will help usability, as users will only need to focus on one area of their computer's controls. In contrast to the consensus, one participant states, 'although the controls take a little time to become comfortable with, they eventually become second nature', finding the controls seamless. Coincidentally, this is the only participant to rate the application's ability to create finite machines an 8 out of 10, with the majority rating the application 6 or below.

Criticism of Transition Arrows

While most criticism of the application was about the controls, many participants commented on the incorrect arrow notation when building their finite state machine; the prototype released with the survey contained improperly formatted arrow heads for both the start state and self-pointing transitions. I was aware of this at the time of release, and so this was expected. Further criticism of the transitions was that the root of them only binds to one of four anchors on the origin state, restricting design as states needed to be moved to format arrows correctly. Finally, two participants believed that NFAs could not be created, I suspect because you cannot draw multiple transitions between two states using the hotkeys, but instead may only do this by adding a comma and another word to the label of the connecting transition, and this was not communicated to users.

Suggested Features for the Application

Question 6 asks participants if there were any functionalities they felt were missing from or would enhance the application. Below is a list of all requested features:

- Ability to convert an NFA to a DFA
- Utilising a regex generator
- Including an undo button (x4)
- Ability to drag a transition arrow from a state (Visual queue that the node has been selected)
- Ability to edit FSA table data directly
- Allowing users to zoom in and out within the viewport
- Ability to save machines for future use
- Ability to export the Machine Description table
- Including a button to clear the viewport (x2)

- Using buttons to create different types of components (normal state, start state, accept state)
- Including a toolbar with drag and drop functionality
- Ability to save a set of input tests and use them for continuous integration
- Transitions initialised by prompting the user for input, instead of defaulting to ‘A’
- A horizontal Interaction window
- Ability to select an object (state or transition) and edit it through a side window
- Visually display input running through an automaton
- Utilising right clicks and drop-downs to edit the diagram
- Ability to generate a general example machine

Given the results of the survey, there are many aspects of the application I aim to change before the final submission of the project to increase usability for staff and students of Computer Science. The most apparent changes to be made are the inclusion of an alternative method of building the user’s FSA for users who do not wish to use hotkeys and the changing of some conflicting hotkey controls for users who do want to use them. This would provide users with the option to use the hotkey controls or not, tailoring it to their needs and retaining hotkey controls for more advanced users who are willing to invest the time to learn how to use them efficiently. Another main fix would be to ensure the arrow heads match common FSA notation so the machine may be presentable for educational material. Finally, including a clear method of creating multiple transitions between two states or indicating this to the user would resolve any confusion. After these changes have been made, I feel the application will reach a moderate level of usability; however, due to time constraints, I will not be able to perform another survey and will be evaluating the final design without end-user feedback.

According to my judgment of the requested features, the most useful feature to add would be a button that reverses the most recent action, as this would allow users to correct any mistake that occurs, removing the need to have to redesign any part of the automaton. Another highly useful feature would be the ability to save a machine description to a file and load it for future use or distribution. Interestingly, I considered this previously and added it to my Kanban board for possible implementation and aimed to add this after discussion with my supervisor. Finally, the ability to textually design the FSA through the machine description table is already a solution to a secondary goal of my requirements specification to allow users to create their automaton using text entry. This would prove highly useful, providing users with a less graphical method of design, which would already be fairly intuitive given its simplicity.

Final Design

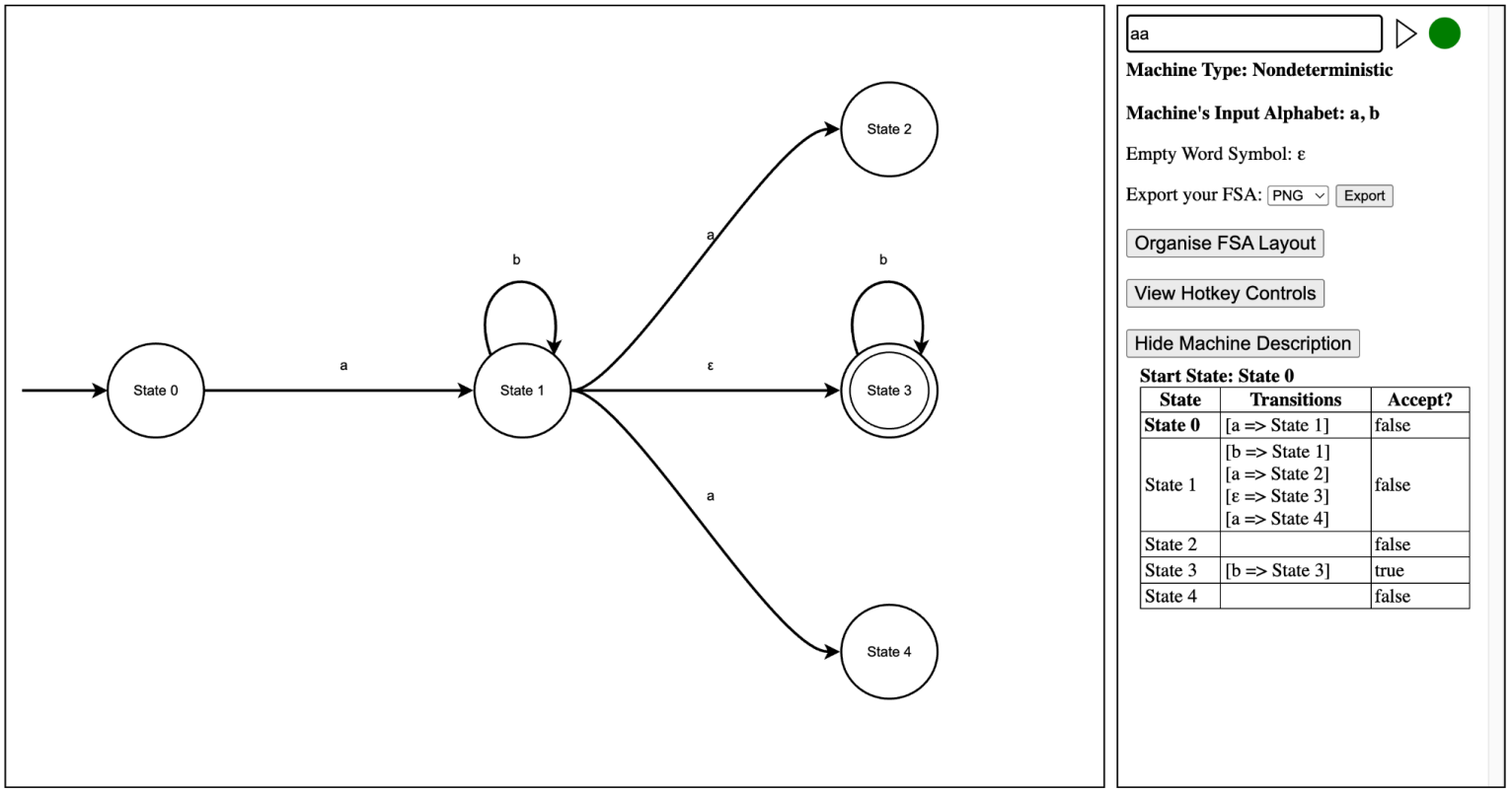


Figure 12: Final Design of Application

Figure 12 shows the final layout of the application, with correctly formatted transition arrows for both the start state and self-pointing states and a slightly reformatted interaction window. The only change of note for the interaction window is that I removed the empty word symbol from the controls tab for easier access and visibility. Furthermore, I implemented path tracking for nondeterministic machines to ensure that the path of an accepted input for NFAs is always shown to the user. Finally, user may now export their machine as a JSON file, allowing an alternative method of verifying a machine built with the application.

Additionally, my final primary goal of allowing the user to automatically configure the layout of their machine diagram has been completed. By clicking ‘Organise FSA Layout’ with any machine marked as ‘Deterministic’ or ‘Nondeterministic’, the application will evenly move the states in the viewport to create a neat and visible diagram for any valid automaton. The algorithm is designed to treat the machine’s layout like a tree, and evenly spaces each level of the tree (determined by the number of parents) across the viewport. For example, all nodes with 0 parents, then 1, then 2, etc, are all aligned vertically, with each level evenly spaced across the viewport. For vertical spacing, this is calculated in slices and will be expanded upon in the next

section; each child of a node is spaced evenly within the allocated space of that node, as the example above shows 3 children, each child is placed in the centre of each third of the viewport's y axis. I ensured the calculated layout maximised the viewport space as exporting an image will capture the entire viewport, with this design, it mitigates white space and makes the layout very uniform, which I felt was the easiest form to interpret the machine's flow with sibling transitions being symmetrical.

In response to the survey, the most significant design change I've made is to the controls of the application by not only reconfiguring the hotkey bindings but also allowing users to alternatively solely use right clicks and selection from a menu for machine building. The new hotkey control can be seen in Figure 13, with a redesigned controls description for clarity and a more precise explanation of the controls, given participant confusion.

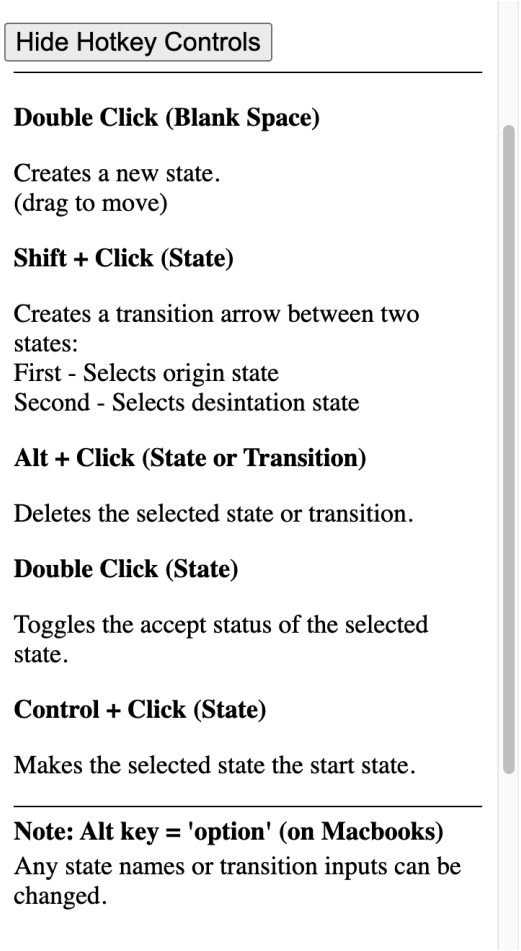


Figure 13: New Design for Hotkey Controls

My main goal with the redesign was to avoid the accidental errors caused by overlapping controls, which would perform the functionalities of both a desired and undesired command. Users must now double-click on a blank space to create a state due to the mistakes that would

occur with a single click, as a new state is accidentally created as soon as the user lets go of a key, such as shift or alt. The double-click makes it slightly harder and therefore more intentional when creating a state, and users may now click on the viewport to exit inputting a transition label without performing unwanted changes to the machine. The other main change was to change the control to set an existing state as the start state from 'Shift + Alt + Click' to 'Control + Click', avoiding accidental creations of transitions or deletions of objects due to users letting go of shift or alt before clicking. These changes would not have been made without the feedback from the survey, as they rely on accounting for new users using the application for the first time and are more prone to human error, which resulted in creating a more intentional application.

Figure 14: Context Menu for State Circles

On either the background, a state circle or a transition input, users may right click to view a context menu at the top of the viewport, allowing them to perform all of the same functionality as the hotkeys, providing them with a completely pointer-based interaction method. Figure 14 shows the menu when right-clicking on a state circle; users can click on the first option to create a transition by assigning the origin state of the transition and then again right-clicking and selecting create transition for the destination state. One additional functionality is the option to clear the viewport when opening the menu for the viewport, as this was a simple requested feature, allowing users to create a new machine without needing to delete each state individually. I chose context menus as an alternative control scheme, as it meant that the user could fully develop a finite machine without needing to use the keyboard and mouse together, which many participants in the survey found strenuous. Additionally, context menus are a standard control scheme which many users would be familiar with, being the interaction method for GUIs. By offering both control schemes, it allows users to choose which one they prefer using or allows them to be used in tandem. While there are numerous features I would have liked to include in the final design, I prioritised those that finished the core of the application and strengthened my primary goals to create an extensible application of higher quality and usability for staff and students of the School of Computer Science.

Implementation

This chapter describes and explains how each area of the application's functionality was achieved, beginning with what technologies were chosen to create it based on my design and then how those technologies were used to create the final artefact by outlining the project as a whole and then breaking down each region of functionality.

Technologies

To begin implementation, I needed to choose appropriate technologies and languages for development. The following will introduce each core technology used and justify why it was suitable for the project.

JavaScript and React

React [27] is a JavaScript library for building dynamic user interfaces for web applications. In React, JSX statements, acting similarly to HTML tags, may be used within JavaScript files to render a page dynamically. In addition, states are a type of persistent variable in React that may be set or passed to various components as props, changing the values used to render each component. I chose React and JavaScript as the primary technologies for developing this application due to my extensive familiarity with React development, my intention to create a dynamically rendering web application and the vast number of open-source React libraries available.

For my junior honours project last year, my supergroup and group developed a web application using React, Next.js and JavaScript that visualised Paleontological data geographically, and for my summer internship, another intern and I developed a standalone application for customising and exporting colour and image configurations for ATM screens using the same technologies. While I could have chosen languages such as Python or Java to develop applications similar to jFLAG contained in a runnable script, I knew that using React would allow me to shift all my focus on the application, since it had been two years since I developed using those languages.

Some primary requirements could have been developed in any usable language, as they simply require the simulation of a finite state machine, which any Turing-complete language would be able to do. However, React's combination of HTML and JavaScript provided the interaction between computation and display that the app required and its ability to update the page instantly after a user's interaction aligned with my intended design after using other automata visualisers. I felt that the setup of prior applications could be simplified, as many involved downloading and installing them, along with any required languages like Java. To reduce this, I felt a web application that anyone could instantly navigate to without any pre-requisite installations would make the app more accessible. Moreover, React's dynamic rendering reduced the number of clicks and time taken to view changes made to the automata built. React's extensive open-source libraries heavily simplified development, as most complex functionality, such as draggable divs or converting HTML to an image format, would be packaged and provided to me, allowing my focus to be on the application's functionality and user experience. Given their open-source nature, I ensured to use only well-tested and documented libraries, all of which are appropriately referenced. Finally, I appreciated JavaScript's modularity in having different visual components held under a designated file, making it easy to navigate code during development and its minimal syntax reducing overhead when coding.

Many alternatives to React could have been chosen; however, given my extensive background in React development and vast library of visual components in comparison with other technologies, it felt most suitable for a robust application developed in a short time frame. Alternative user interface-focused JavaScript libraries include Angular [43], which is TypeScript-based, and while it does provide more secure 'typing and improved error detection' [42], I felt my unfamiliarity with TypeScript would slow down development and lead to fewer goals being met. Vue.js [44] could have been used given its simple code structure, leading to faster development and allowing me to spend more time developing as many features as possible, however, it lacks the extensive number of public libraries that React has, which proved to be foundational to my application. In contrast to this, the application could have either been developed without a dynamic graphical user interface and as such, could have been developed using Vanilla JavaScript, Java Swing [49] or Web Assembly [50] with users developing their automaton textually and having the application simply render the diagram and simulate input. While the method and behaviour of building finite automata were not specified in my goals, I felt that dynamic rendering would enhance the usability and, therefore, the quality of the application overall for its intended end users.

Next.js

Next.js [28] is the most widely used React framework, focused on enhancing the capabilities and experience of React-based applications, providing many additional features and maintaining up-to-date documentation. The primary benefit of Next.js is server-side rendering, a feature which reduces dynamic page rendering times as the HTML content is fully rendered on the server and sent to the client instead of sending the client unrendered HTML, which will be rendered on the client side, achieving the fast user interaction in my design. Another benefit is serverless functioning, which allows an application's server management (provisioning and maintenance) to be handled by a third party, allowing the user to solely focus on the application's functionality and not the handling of a server. Further advantages include: allowing CSS to be written within JavaScript files (allowing for dynamic CSS), hot module replacement (rendered code will automatically update upon code changes), caching (faster load times) and more specific React-based error messages. Initially, I felt that using a framework designed for much larger-scale applications may cause unnecessary overhead, and it would be more straightforward to use a framework more suited for lightweight applications or no framework at all. However, having used it before, I did not need to learn anything new, and as such, this overhead did not feel great enough to outweigh the support provided by Next.js.

Alternative React frameworks include Gatsby [46], Remix [47] and Eleventy [48]; however, most do not have Next.js's level of optimisation or focus too much on data handling and lack the flexibility of Next.js. Gatsby suits the project quite well and could have been chosen as it focuses on increasing performance for static sites with dynamic elements and holds 'over 2,500' [45]

plugins to ease development. However, this is more tailored to sites with backend and data fetching, which my application does not utilise, with Next.js supporting both server-side rendering and incremental static regeneration to prioritise seamless page re-renders as opposed to Gatsby's optimised data fetching. This is similar to Remix, which holds similar functionality to Gatsby and was not chosen due to much of its functionality being built for content-heavy websites. Finally, Eleventy is designed to handle lightweight frontend-focused applications, which suits my project well and lacks client-side rendering, improving render times. Despite these benefits, I chose Next.js due to its optimisations and streamlining of the React pipeline, which would greatly streamline development despite being less focused on supporting smaller projects.

Jest and Babel

Jest [30] is the most commonly used automated testing framework to test React applications, integrating and interpreting React code very well. Jest also provides detailed code coverage reports, identifying which specific functions or lines of code have not been covered in its unit tests. With all unit tests and code coverage reports being runnable from the terminal, it eased development and ensured code was consistently functional. Having used Jest during my summer internship, I chose to use Jest to test each aspect of functionality, aiming to achieve 100% code coverage upon final submission. Babel [31] is a JavaScript compiler and is required when using Jest with React code, as it compiles the ES6 and React JSX tags to ES5 JavaScript (previous version), which may be interpreted by Jest.

Git

Git [33] is a version control system which I've used extensively throughout university and my internship. It provides feature branching to maintain different versions of code. It proved useful in development when operating in Scrum sprints, as I would be able to create an isolated version of the project on a new branch for each feature developed and only merge that code into the main secure branch of the project once that feature had been properly tested.

Project Outline

To introduce the implementation of my Next.js application, I will briefly explain the application language and data structures used to create it. The project was initialised as a Next.js project, which contains a *config* folder holding the framework's preset configurations, which I have slightly altered for compatibility with Jest. Both *package.json* and *package-lock.json* were created with Next.js and maintain the version, dependencies and scripts to ease development by condensing common commands such as running 'npm install' to install all dependencies of the project instead of having to specify each one on the command line. *src* contains the source code for the project, all of which I have written, apart from app/*layout.js* which was created within Next.js to attach the JSX from *app/page.js* to the root of the host server when hosting the project,

ensuring it is shown. *app/page.js* is the root page of the application, with all other files in *src* being components or classes that are called within *page.js* to build the application. Next.js is the framework for running and configuring the application, which is developed using JavaScript and statements from the React library. Finally, the README.md file contains terminal commands to run the application and the automated testing framework.

React utilises a type of variable called a state, which holds a value persistently across re-renders of the application and is coupled with a 'setter' that may update the state. An example is *machine* and *setMachine* in *page.js*, as the application only requires one finite state machine to be built at one time and therefore stored; this state persistently stores the current design of the application's finite state machine across renders of the page and may only be updated using the *setMachine* setter. React operates by re-rendering the page when any state setters have been called, with the page re-rendering with the new state values. The page consists of function components, seen in the return statement of *page.js,* which act as custom JSX tags that return their own user interface tags and can contain internal functions. All dedicated function components for the application can be found in *components;* parent components may pass states to their children, which are called 'props' within the child function component and are used to change how the child component is rendered.

Simulating Finite State Automata

FSA.js

The user's finite state automaton is stored within the *FSA.js* JavaScript class, containing all the attributes and methods needed to simulate it and is created as an object stored as a state that is passed to all components within the application. In order to update the FSA, any component which has access to *setMachine* may pass a new FSA object to set it globally, ensuring all components maintain the same updated FSA. Originally, I chose to use the React library xstate [40] as the library had a pre-built implementation for creating and editing finite state machines. However, upon using xstate to create an automaton, I realised that much of the underlying functionality was abstracted and required me to learn the complex code of another developer, which would take some time. I realised that an automaton could easily be stored as a JavaScript object and felt this approach would be far simpler to implement, given my familiarity with JavaScript and primarily the ability to tailor the implementation of the FSA to the needs of my project. By having FSA operations be done through the editing of an object containing attributes, it provided me with a very straightforward and entirely customisable FSA implementation. Throughout development, the class was extended as I developed each feature of the application and given that any component with the machine state object could call any new method, it greatly streamlined the development of new features. One area of note is the constructor, which either instantiates a new empty FSA object when entering 0 or creates a copy of the machine given to it. The constructor creates a copy of an FSA object due to React code being unable to

re-render when the same variable reference is passed to its setter. For instance, if you altered *machine* and passed this to *setMachine,* React would not re-render its components as it has the same object reference. By creating a copy of the current *machine* and then altering the copy and passing that to *setMachine,* the machine object passed would be correct and cause all components of the application to re-render with the updated machine.

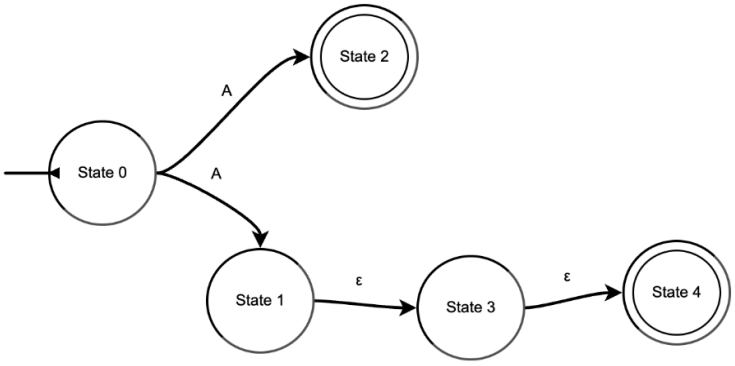
All methods that involve adding, deleting or changing an aspect of the machine are done through the Viewport to manipulate the user's machine object and involve the simple editing of arrays containing states or transitions. To ensure that all state IDs are unique across the use of the application, the FSA object tracks the total number of states ever created and assigns the total as the ID of each new state, ensuring uniqueness even when states are deleted. Aside from these methods, some are more complex and warrant further discussion, which I will detail below.

status()

My supervisor and I agreed that the application should be able to output to the user whether their machine is deterministic, nondeterministic or invalid, which this method does by returning the machine's type as a String. First, the method checks if the machine is invalid by ensuring it has a start state, at least one accept state, at least one character in its input alphabet and an input letter for every transition. If valid, the algorithm checks if every state has one transition for every letter in the input alphabet or if any state contains a transition with the empty word. If so, the machine is returned as nondeterministic, however, if it passes these checks, it is returned as deterministic. I ordered the checks this way as I could not run any determinism checks on an invalid machine, and reduced time complexity by returning 'nondeterministic' as soon as the algorithm finds a state that is not deterministic.

runInput()

When simulating the running of an input word on the machine, checking the status of the machine is key, as it impacts which approach to take for tracing input. Once both the machine and input word are verified to be valid, with the input word consisting of only letters in the machine's input alphabet, tracing is then performed differently depending on whether the machine is deterministic or nondeterministic. If deterministic, the method simply loops through the input word and, for each letter, iterates through the machine, storing the current state at each letter and outputs the path for both accepted and rejected words.



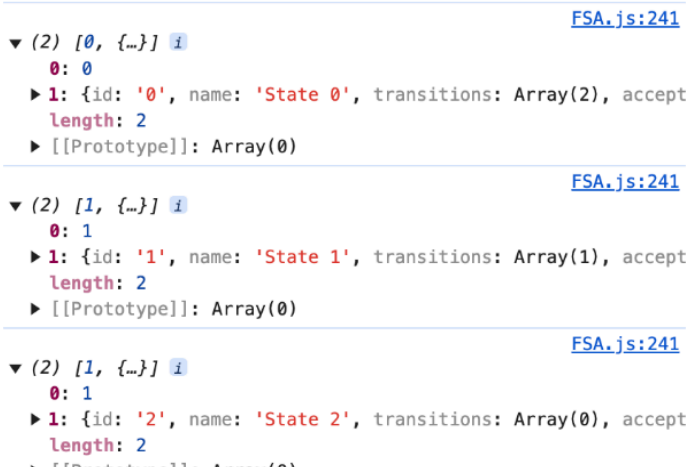


Figure 15: Console Log Output When Running Input Word ‘A’ on Machine

For nondeterministic finite automata, however, the method performs a breadth-first search when tracing the input to find the shortest path that ends on an accepting state. At each state where a choice must be made, both branches are explored in order of level and checked if the current state is an accepting state when the end of the word has been reached. Figure 15 demonstrates this by showing the order in which nodes are searched, given an input ‘A’, the algorithm only iterates once, but due to the choice explores both State 1 and 2. Note that the algorithm first explores State 1 and does not then traverse to State 3, as this would be on the next level of the machine, instead, it then checks State 2, which is on the same level as State 1 and accepts the word, as State 2 is accepting and the word has a length of 1. I chose to implement a breadth-first search of the machine's branching choices as this mitigates infinite loops and prioritises finding an accepting state as early as possible. I originally implemented a depth-first search to explore the branching transitions, however, I soon realised that given a state with a self-pointing transition with the empty word, my depth-first search ran infinitely and did not explore other branches.

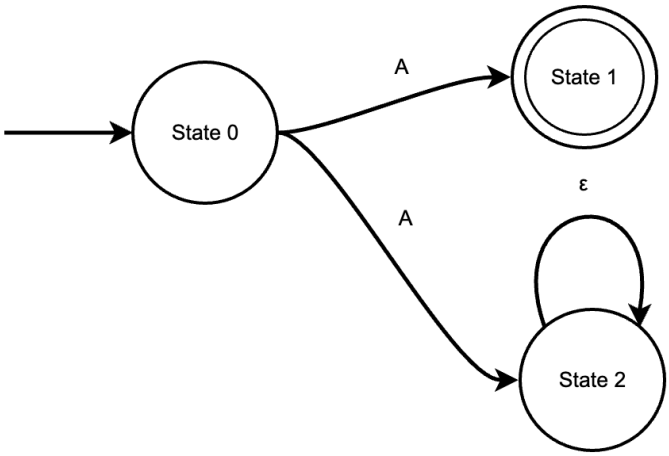


Figure 16: Machine That Runs Infinitely on a Depth-First Search

The traversal works by iterating through the word, treating each state that contains multiple options for the current letter as a branch. An option could be either a transition containing the next letter of the word or the empty word, in which the current letter would remain the same. Branches are tracked using an array of nodes called the node store. A node stores the index of the current letter at that node and the corresponding state, which may be returned to at any point. If there are no valid branches, the machine rejects the word. If the word is at the final state of a branch and it is an accepting state, the machine accepts the word. However, upon testing, I found an edge case where the final state can contain an empty word transition to an accepting state. To

accommodate this, if the final state is not accepting but contains a transition with the empty word, it will traverse it and check again.

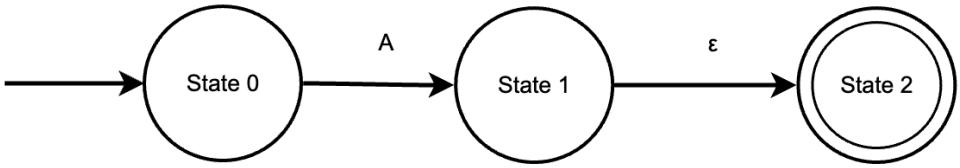


Figure 17: Machine Edge Case

Within the final version of the application, this method was extended to output the path of an accepting input on a nondeterministic machine and does so by storing the ID of all parent states in each node.

retrieveDepth()

This method is used to calculate the maximum depth of the machine for spacing calculations when organising the layout of the diagram and involves recursively building a nested array representing the machine's topology and then using a recursive algorithm to calculate the depth. The constructed array represents the machine by representing each state as an array of 2 elements, first the ID and then an array of all its child states, each element of this array of child states will also consist of an array with the ID of the state and an array of its children with leaf nodes storing null as their array of children. This is built by recursively calling the *findChildren* method on each child node starting with the start state, with each leaf node acting as the base case, returning null. This approach accounts for branching for an unknown length and works for any finite automaton, while an algorithm similar to the nodestore would work, which primarily utilises the input word to iterate through the automaton. Once the nested array is built, I adapted an algorithm I found online [38] which counts how many times it takes to merge each layer of the array until the structure is no longer nested, thereby counting how deep the nesting is and therefore the automaton.

Visualising Finite State Automata

Viewport

This section focuses on how the viewport was implemented, including how users may interact with it to edit their automaton and how the application then renders those changes and updates the global *machine* state.

Control Schemes

To design their automaton, users may either utilise the hotkey controls by pressing and clicking using the combinations specified or open each component's context menu through right-clicking on a specific component and selecting one of the options displayed. The hotkeys are handled through the *handleClick* and *handleDoubleClick* functions when the mouse is pressed, which use conditional statements to check which combination of keys has been pressed and perform the corresponding functionality. Originally, I did not use the control key for the hotkey controls as it causes a right-click on Mac OS, causing the context menu of browsers to open. Additionally, Mac OS did not have an Alt key; as such, I placed a note on the app controls stating to use 'Option' instead of 'Alt' on Mac. Every element is given an ID to perform different functionality when clicking on different objects, the viewport uses 'Viewport', state circles uses their respective ID number as a string and transition arrows use the ID of their origin state, then '=>', then the ID of their destination state which is are the same if it is a self pointing transition and the start state uses 'start' but has no functionality. By allowing the ID of states and transitions to be passed upon click, it allows functions to be called on those specific state IDs and prevents overhead. Initially, I did not pass state circle IDs upon click and used a function called *getCircleOverlap,* which calculated the state circle the mouse was currently within the boundaries of by iteratively checking the position of each circle that was routinely updated and stored in an array containing all current positions. Additionally, I tracked the mouse's position through the 'mouse-position' [4] React library, which caused many errors with my testing framework. As such, storing the ID values within the state circle components functionality was now bound to the ID of the element clicked on instead of routinely checking the mouse's current position, avoiding much unnecessary computation and removing errors with the testing framework, as testing could be performed more computationally instead of through simulated mouse tracking.

To enable the use of the context menus, I overrode the default context menu, which opens upon right-clicking on most web browsers and passed the *handleRightClick* function. By utilising a state containing JSX, it meant that I could update it at any point to render different context menus, which the mentioned function does. Depending on the component, different lists of buttons are rendered, which call the same functions as the hotkeys, all end with setting the context menu to null, which renders nothing in the return statement to hide the context menu after selection.

State Circles

To add a state to the automation, within *Viewport.js,* state circles are added to *machine* with their default name being 'State ID_VALUE' to easy tracking and then rendered by adding an object containing their ID value and default position to the *circleArray* state which creates an instance of the StateCircle component for each circle in the array set at the mouse's position when clicking. The StateCircle function component returns the JSX for a circular input box that is

made into a draggable div through the react-draggable [15] library, as it enables any div contained inside the ‘Draggable’ tag to be draggable. The circular input box was chosen to allow users to type in the state name at any point, with an input layered on top of a circular div seeming redundant. To ensure that circles cannot be dragged outside of the Viewport, the Draggable div usefully has a prop which sets the bounds to the immediate parent div. The *positionState* state tracks the circle’s current position and is updated on drag, the *fixedPosition* prop passed from the Viewport and only set when the ‘Organise FSA Layout’ button is pressed which updates the circle’s position to format the automata diagram, calculations for this will be covered in the Organise FSA Layout section, Prior to this feature, the circle’s position was managed internally by the Draggable div, however, I needed to set the circle’s positions manually to format the diagram. Initially, this was done through react-draggable props and a state which quickly set and unset the circles as draggable; however, this either caused too many re-renders due to the constant state changes or made the circles undraggable after organising their layout by rendering them before they were set back to draggable. As such, deciding to handle circle position updates manually instead of trying to manipulate the externally provided Draggable div proved tidier and more direct.

Toggling the accept status of each state is performed by applying styling to the input that adds an outline to the circle that is moved inside using a negative offset. Toggling was previously handled within each StateCircle component using a state; however, upon implementing the context menus, I needed a way of toggling the accept status from the Viewport’s menu instead of double-clicking on the circle. This is now done through maintaining an array of the accepting states within the Viewport and passing a boolean based on whether each state is within this.

Transition Arrows

Transitions arrow components are stored within *transitionArray* in *Viewport.js,* which contains an array of JSX tags returned from the TransitionArrow component, which are all rendered within the Viewport’s return statement. A transition arrow may be added or removed from the Viewport by updating this array; to create a transition between two states, the user must first select the origin state and then the destination state. Arrows are drawn using the ‘react-xarrows’ [10] library, which was chosen due to its dynamic rendering and simple binding of anchor points. An Xarrow is a component that renders an SVG arrow between two divs by passing their ID values as props, as each state circle’s HTML ID value corresponds to its ID within the machine, this anchors each arrow’s start and end points to the correct state circle divs. As Xarrows are built to work in tandem with draggable divs, any components placed inside an Xwrapper that call *useXarrow* will cause all Xarrows in that Xwrapper to re-render. By calling this function upon any circle being dragged, all transition arrows are re-rendered upon the movement of any state circle, ensuring that arrows consistently point to the correct state circle divs when being dragged or repositioned. If a circle is deleted, all connected transitions are also deleted to avoid any hanging transitions.

There are three types of transition arrows within my application: transitions between two different states, a self-pointing transition and the start arrow pointing to the start state. The start arrow is in fact a self-pointing arrow, both of which use their state's id as both the start and end props and involved much difficult styling to format correctly, as I was working with components from 'react-xarrow's library with far less documentation and online help. To release the prototype in the survey, I knew this issue was fixable, but mistakenly felt participants would not pay it much attention in the survey and aimed to fix the styling issue at a later date.

After the survey, this was my primary change, and it was resolved by passing specific styling numbers to affect the anchor points and curvature of the self-pointing arrow, which were calculated through experimenting with the Xarrows props. These changes can be seen between figures 11 and 12. Given that these were only between one state, I considered creating my own SVG arrows and conditionally adding them to the StateCircle component, however, I would need to handle dynamic re-rendering myself, and this felt like a very long workaround. Xarrows can have labels that can be set to specific JSX tags, which I used to allow both regular and self-pointing arrows to have typable labels that automatically update the machine when changed. I chose this approach as it meant users could directly type to change any transition's input word and felt very direct from a programming side, instead of displaying the transition input text and updating it externally. Some participants in the survey did not know that you could simply add a comma to the transition input's label to allow for multiple input words to transition from one state to another, as this was not made explicit. To avoid this, I wanted to add highlighting to the transition's input label upon creation of the transition, or if a transition already exists between two states, and the user adds another. To do this, I set the input div to highlight on initial render, and if a user adds a transition that already exists, it is highlighted and a comma is automatically added to the input's value through the *getElementByID* function in Vanilla JavaScript.

Interaction Window

The interaction window was developed using common HTML tags such as button, table, p and b with many components created within functions that are called to simplify the code structure. In order to present information about the machine, the *machine* state is passed to the Interaction Window so that all information dynamically updates whenever the machine is updated within the application. To access information about the machine, FSA methods are called through the *machine* state. I chose to create the Input Bar component as it felt like a separate section to the rest of the Interaction Window; this could have been kept within *InteractionWindow.js,* however, given the extra machine computation needed for this specific section, it felt more modular and organised to separate the two. For both the controls and machine description tables, a state is used to track the user's setting of each being hidden or shown, as these values will persist between renders.

Areas of Note

Areas of note include the machine description table and exporting the automaton. The JSX for the machine description table is returned by *printMachine,* which builds the tags for a common HTML table by pushing a row of the table for each state in the machine. This demonstrates the benefits of React code as these JSX tags will print the corresponding HTML tags when returned, but can be dynamically calculated and applied within JavaScript, allowing for the machine description table to dynamically add or remove rows or update state values as the table is generated every render.

The dropdown component used to select which option for exporting was taken from an online source [7] as it provided a simple implementation for a dropdown in React code, as using HTML would require turning a portion of the Interaction Window into a form, which I felt was disruptive to the code structure. Users may export the machine as a PNG, SVG or JSON file, providing both graphical and textual output of their machine. To output both image formats, I used a library called 'html-to-image'[13], which easily converts the HTML of an element into an image file that may be downloaded. The only change I needed to make was to reposition the Viewport using transform, as the function did not accommodate for margins and captured the top leftmost point of where the div would be, cutting off the bottom right corner. To convert the machine into a JSON file, I needed to attach the data to a file and then create a link for that file. I turned *machine* into JSON data and converted it to blob data to represent the JSON data; using the URL class, I made this into a URL for *downloadExport* to use. After this, I chose to remove the created URL in case it persisted and caused issues for users. Now that all types of export were converted to a URL, I created the internal function *downloadExport* to convert the link into an 'a' tag which may be forcefully clicked on to download the file for the user.

Organise FSA Layout

This section explains how the new positions of state circles are calculated and applied when a user clicks the 'Organise FSA Layout' button in relation to the viewport, detailing the recursive algorithm and graphical calculations used.

Overview

As the button is located in the Interaction Window, and the functionality needs to operate within the Viewport, I created a state called *organiseLayout* that the button sets to true, which is passed to the Viewport by passing through their shared parent component *page.js* as props can only be passed to children and not adjacent components. *useState,* which is used to create the various states, is a React hook; another type of React hook is *useEffect,* which runs a block of code either on every render or every time a prop within that component has been changed, this is used to detect if *organiseLayout* has been changed upon the user pressing the 'Organise FSA Layout' button. The *positions* state stores an array of every circle's ID and its position values, storing null if the position is not currently being set. Within the *useEffect* hook, if *organiseLayout* has been

changed, then the *organiseCircles* function is called, which calculates the organised positions and updates the *positions* state. Once the positions have been set, *organiseLayout* is set to false so that on the next render of the page, in which the circles would have been positioned using *positions*, their fixed positions can be set back to null, which makes them draggable again.

newPositions is an array containing an object of each state's ID number and new position coordinates returned by the *organiseChildren* function. As I felt it easier to treat the state diagram like a tree, I will be referring to state circles as nodes and use common tree terminology like parents and child nodes to refer to states that transition to or from other states, however this should translate easily. *organiseChildren* is a recursive function that treats an FSA diagram like a tree structure, and is called upon each node in the diagram to calculate its new position and then call itself upon each of its children. As each branch reaches its base case, which is a leaf node, the function simply returns the leaf node's position object, and each callback's nodes are concatenated into a single array. Upon the final call, this outputs an array of objects containing each position object, having explored each branch of the diagram.

As the Viewport's dimensions are dynamic and change based on the window, all position calculations are done by fetching the Viewport's current height and width. The position node of each state stores the proportions of the Viewport the state should be displayed on each axis, with X storing the level of the tree the state is on, and Y storing the height percentage of the Viewport for it to be placed.

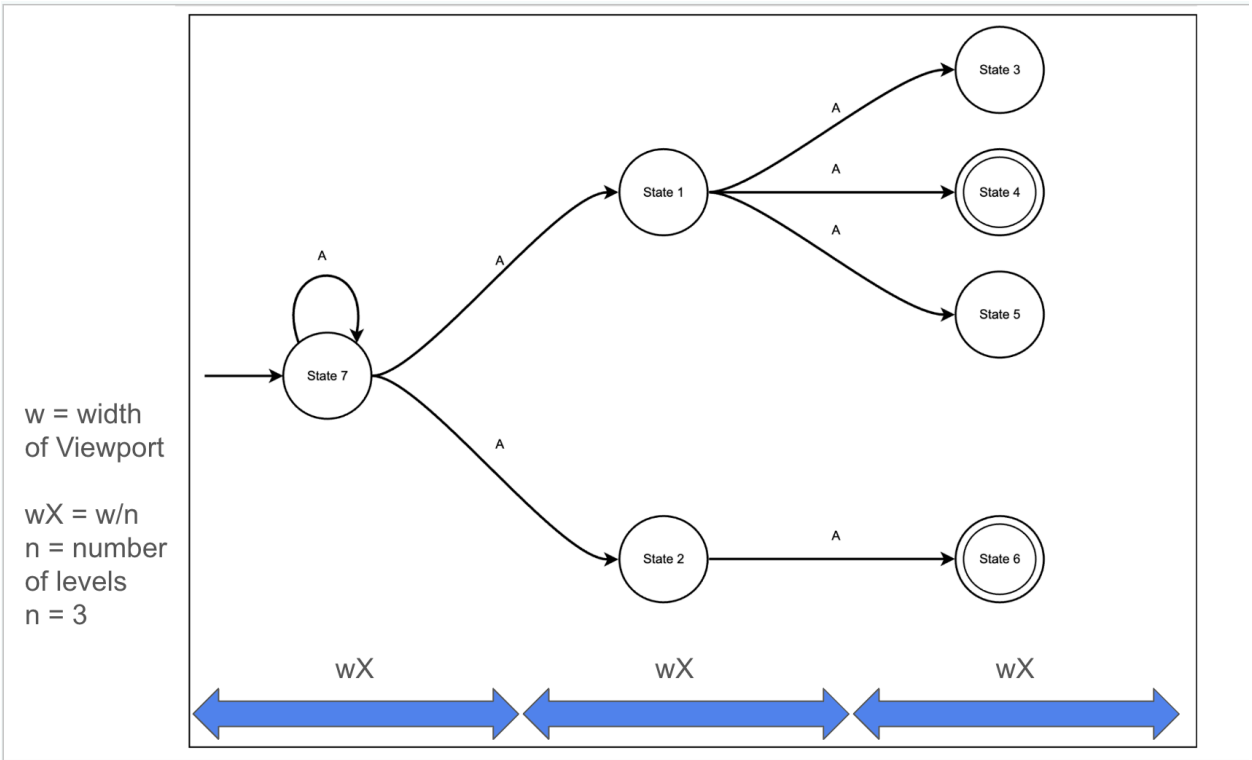


Figure 16: Example FSA after clicking 'Organise FSA Layout'

organiseChildren does not calculate the X value; instead, each position object stores the current node's level in the tree to calculate it based on the Viewport's current width. Within the Viewport's return statement, each node's X position is calculated by first calculating the width of an even portion of the Viewport by dividing its total length by the number of total levels. The total depth is calculated using the receiveDepth function within FSA.js. Looking at figure 16, this width value would be equal to width/3, and each node's X value would be wX multiplied by its level in the tree to space out each level evenly. It's worth noting that organiseChildren ignores transitions that are self-pointing or those that point to nodes of a higher level to bypass the need for calculations with a circular graph.

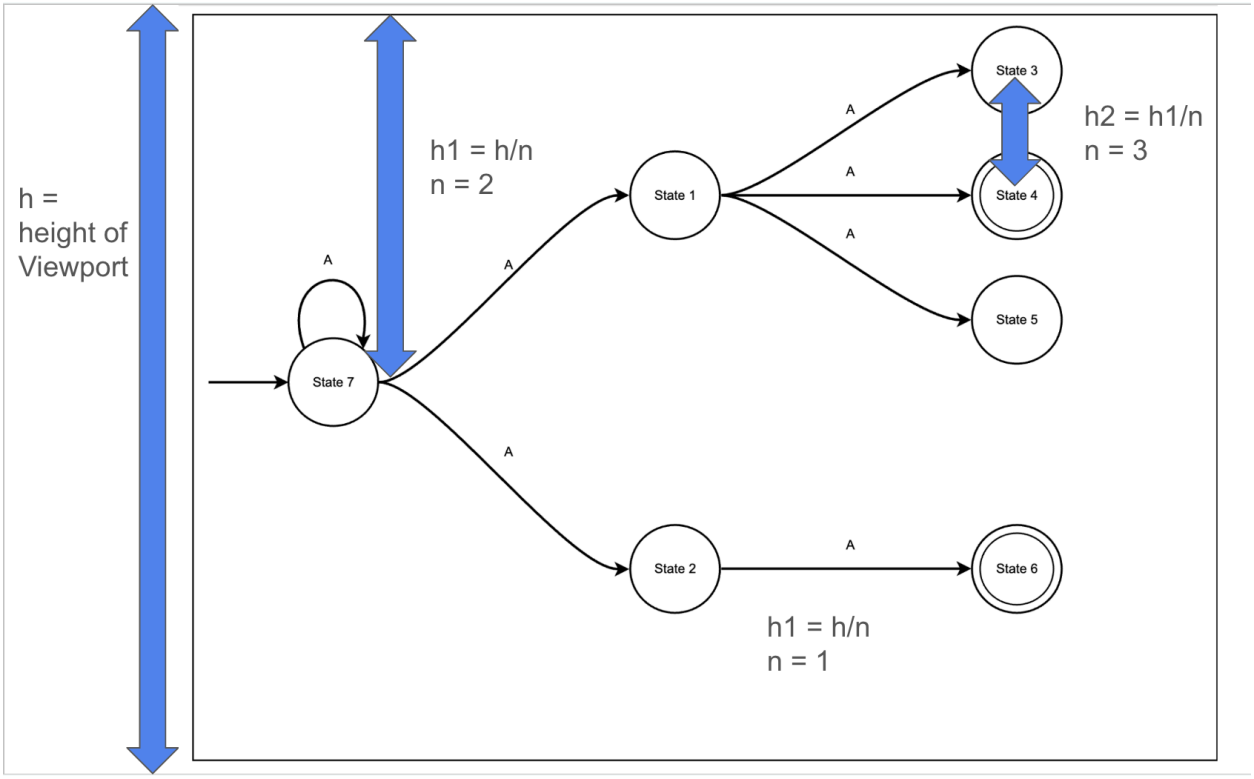


Figure 17: Example FSA after clicking the 'Organise FSA Layout' Button

Y calculations for each node are more complex as they require spacing in relation to their siblings and are not consistent across the entire diagram, like the number of total levels. Figure 17 showcases how the algorithm uses the height of the Viewport to calculate a height slice for each node, dependent on its number of siblings. A height slice is which two coordinates on the Y axis of the Viewport that a node must be in the centre of. For example, as I wanted the start state to be placed in the middle of the Viewport to propagate in each direction, its height slice is the entire height of the Viewport. Height slices are stored as the Y values at the top and bottom of each slice. For the start state, its height slice values would be 0 and 100, starting at 0% of the Viewport's height and then 100%. To calculate a node's height percentage within the Viewport, it is placed in the centre of its height slice by calculating (minHeight + maxHeight) / 2; for the start state, this would equal 50%, placing it in the middle of the Viewport.

Each child node's height slice is a portion of their parent's height slice, calculated by portioning each child evenly within the parent's height slice. This was done as it was the simplest way I could think of fairly making the most of the space while still representing the machine's flow as close as possible. This method is prone to the organisation to be very unbalanced when dealing with deep unbalanced branches, however, most simple finite automata should be well-spaced, allowing for higher readability. For each valid child node, its height slice is calculated by allocating it a portion of its parent's height slice by index; this is passed to the child when recursively calling organiseChildren, so height slices may continue to be portioned. Figure 17

shows how the start state is in the centre of the Viewport's height, then having two children, each of them are in the centre of each half of the start state's slice. In the bottom branch, as State 2 only has one child, this is maintained on the same level with n equaling 1. For State 1's children, we can see this further portioning of height slices. Another limitation of this algorithm is that if each node has many children, nodes can overlap with each other easily; however, given this was the final primary goal that was implemented, I ran out of time to refine the algorithm further.

Alternative Approaches

Upon researching for this feature, I planned to find and implement a known sorting algorithm for graphs, as I felt this would be simpler and more robust than inventing one myself. I found a version of Kahn's Algorithm [51], which topologically sorts directed acyclic graphs to sort the user's machine and display them in sorted topological order. However, I soon realised that the algorithm will not work for graphs with cycles in them, and as an automaton may easily contain a cycle, I needed to come up with a way to adapt this or not use this for automata with cycles. Most algorithms would not perform the Viewport calculations and only identify the topological order of the machine. As I researched and identified that I needed to calculate coordinates for an even format, I decided it was better to create an algorithm that suited my needs rather than trying to find and adapt one. Once I devised the design for the final solution, I originally implemented it by recursively building the nested array from the depth calculations and then calculating the height slices by iterating through the nested array. I chose to refactor my implementation so that the calculations were done within the recursive algorithm, reducing time complexity and repetition.

Summary of Imported React Libraries

Name	**Use**
testing-library [41]	To create mock function components and provide a library for testing a React-based user interface.
react-draggable [15]	To enable draggable React divs and allow users to drag state circles to configure their FSA diagram.
react-xarrows [10]	To draw all arrows between states and display a label acting as the transition's input.
html-to-image [13]	Converts the viewport to an image, which may be downloaded as a PNG or SVG, allowing users to export their FSA as an image.

Testing

How Testing Was Conducted

To test my application, I used an automated JavaScript testing framework called Jest [30] and a library for specifically testing and mocking React components called 'react-testing-library' [41] to allow me to run automated test suites on graphical and non-graphical functionality. Each file within the src/__tests__ directory is a test suite for a specific set of functionality, allowing me to keep testing modular and organised. Non-graphical tests were run by expecting a certain output from functions and were most used for testing FSA.js. Due to the vast library of 'react-testing-library', graphical tests could be performed by generating a mock user interface, simulating user clicks, keyboard presses and moving the pointer to simulate user interactions with the mock interface. All unit and integrated testing was done with Jest and the test suites; however, full system testing was done by using the application in development mode by running npm run dev, which simulated the application.

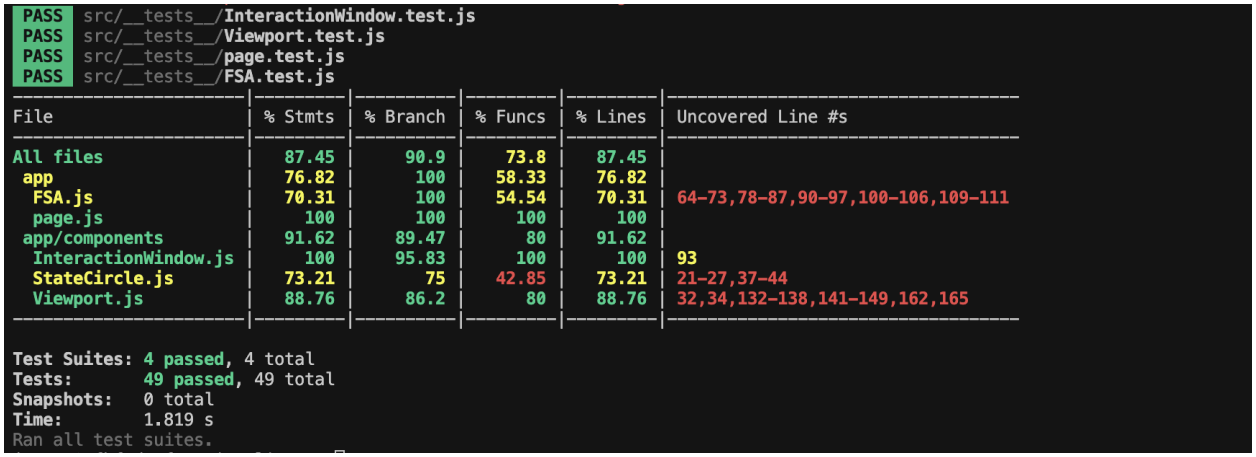


Figure 18: Screenshot of Jest Code Coverage Report During Development
Command: npm run coverage

The specific Jest commands may be found in package.json, but are runnable through npm; running npm run test will run every test across all test suites and provide a report for each test that has failed, npm run <testfilename> will only run a specific suite and npm run coverage will provide a code coverage report, detailing every line of code that is not covered by a test shown in Figure 18.

Test Design

For each new feature developed, I wrote tests that covered all forms of input to ensure they worked for each scenario. This included correct values, testing for erroneous values such as components not rendering, boundary values, edge cases and cases which involved different combinations of features. An example of an edge case would be for running an input on an automaton, testing every type of transition for an empty word, such as at the end of an automaton, self-pointing and testing that the algorithm would correctly make the nondeterministic choice without moving to the next character.

Automated UI Testing

Graphical tests focused on conditional rendering and whether certain elements, text or specific styling of interface components were rendered onto the document after simulated user interaction. The automated testing of the user interface took a great deal of time at the beginning of the project as I was becoming familiar with the style of testing, which involved abstractly representing the user interface and simulating user interactions. Any new piece of functionality which involved libraries or untested structures proved difficult to create tests for, as they would have to be mocked in a new way, and it was unclear whether react-testing-library would be able to recreate them. This included most graphical editing of the state diagram, which involved simulating a drag from one coordinate in the Viewport to another, and testing whether the circle was in the correct position. The test would not work as react-testing-library's drag function would scale the coordinates as the simulated user dragged the circle and not factor in the dimensions of the Viewport. I resolved this by creating the function *getCoords,* which parses the circle's translate style property to extract the coordinates relative to the Viewport. Despite difficulties, I was able to successfully run tests on most parts of the code, mostly achieving my secondary goal of testing the user interface and all parts of the application.

Results of Automated Testing

I chose to use an automated testing framework to allow for continuous integration throughout development and to ensure my code was fully functional, aiming to maintain 100% code coverage of unit tests after the development of each new feature and only merging with main once that metric had been reached. This took a great deal of effort, and usually took longer to implement than the actual graphical features, as the complexity of mocking components and functions takes a while to learn fully, with much of my time spent on resolving tests that did not generate the intended values. Despite this difficulty, this process revealed countless bugs and weaknesses in my implementation and led to many instances of improving my original code, hence why many of the code files are short, as the constant refactoring led to my code being direct and succinct. Additionally, by ensuring 100% code coverage until the end portion of the project, it meant I never had to return to old features for improvement or bug fixing, as each

feature was well-tested and always worked as intended with new features. I can confirm that the tests run can verify the correctness of my application due to routinely outputting test values into the terminal, which showcase the correct values following my implementation and only return true when the correct values are returned and incorrect when false ones are in accordance with my code.

Evaluation and Critical Appraisal

My evaluation begins by critically reflecting on the extent to which my application has achieved each required objective. After this, I evaluate the application against similar works, focusing on the applications mentioned in my context survey, as they hold the most success with my targeted end users. This section concludes with a discussion of the shortcomings of the project and the challenges I faced during development, and a final summary detailing the extent of my success.

Requirements

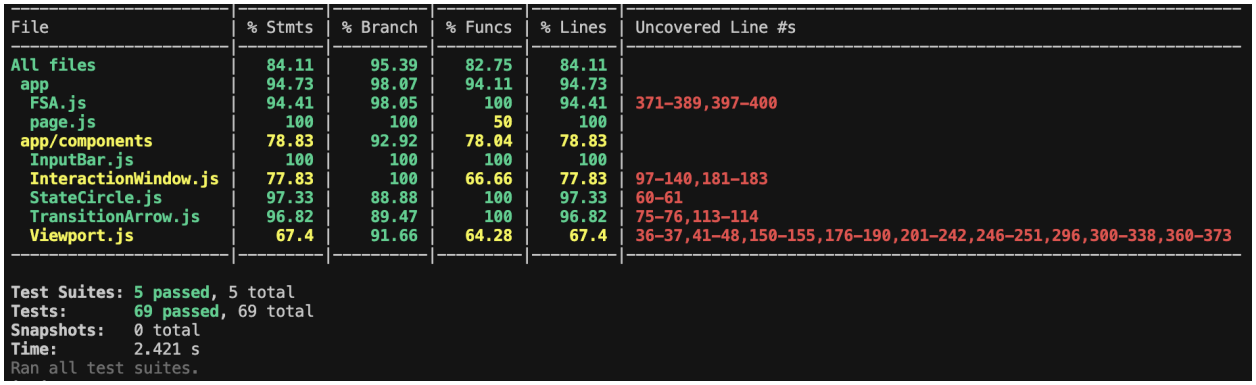
Primary

My application achieves all the required primary goals set out at the start of the project. Users are able to create a deterministic or non-deterministic finite automaton and have it graphically displayed to them. Users are able to drag the state circles to any position in the Viewport to manipulate their exact positions to edit the diagram. The Interaction Window details in real-time the validity and status of the machine and allows users to run any input word on the machine, outputting the execution's result and path on the machine. The 'Organise FSA Layout' button may be pressed to automatically reconfigure the layout of the diagram to an evenly spaced format. My only criticism of the organising feature is that automatons with particularly deep branches and a higher number of children may cause some states to overlap, being unsuitable for lectures; however, if so, the user may be able to drag the circles to adjust the layout. Finally, users may export their finite automaton as either a PNG or SVG image file.

Secondary

Only one of my secondary objectives has been fully met, with the rest having been met to a varying extent. My supervisor and I agreed that we considered all secondary goals to be ambitious and did not expect all of them to be completed. Implementing multiple exporting formats has been met, with the user's automaton able to be exported into two image formats and JSON. Other objectives were not fully achieved due to time constraints and my decision to focus on writing a quality report when entering the end of development. Additionally, external factors limited the amount of time I could focus on the project, which are detailed in the Disruption section.

Partially Completed Objectives



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	84.11	95.39	82.75	84.11	
app	94.73	98.07	94.11	94.73	
FSA.js	94.41	98.05	100	94.41	371-389,397-400
page.js	100	100	50	100	
app/components	78.83	92.92	78.04	78.83	
InputBar.js	100	100	100	100	
InteractionWindow.js	77.83	100	66.66	77.83	97-140,181-183
StateCircle.js	97.33	88.88	100	97.33	60-61
TransitionArrow.js	96.82	89.47	100	96.82	75-76,113-114
Viewport.js	67.4	91.66	64.28	67.4	36-37,41-48,150-155,176-190,201-242,246-251,296,300-338,360-373

```
Test Suites: 5 passed, 5 total
Tests:       69 passed, 69 total
Snapshots:   0 total
Time:        2.421 s
Ran all test suites.
```

Figure 19: Final Code Coverage Report

npm run coverage

My additional objective to perform automated testing on all parts of the program was fully achieved until developing my final design, with the prototype application achieving 100% code coverage before exporting images was added. After image exporting and all features in my final design were added, the final code coverage report, seen in Figure 19, shows that my unit tests cover 84.11% of all lines of code and 95.39% of branches within the code. While not at 100%, I still consider this to be a great achievement, as most applications do not reach 100% code coverage, with many similar works not running any automated testing on their application, showcasing the security of my project. Despite this, as I have not tested image exporting, context menus or organising the layout, I have not fully achieved my goal, but done so to a significant extent.

Usability

I consider my additional objective of having a high level of usability for staff and students of Computer Science to be moderately completed. Given that I received mixed feedback on the usability, with the vast majority of criticism about the control scheme, which has since been improved, and a simpler interaction method is now provided, I feel there is now a high level of usability. Alternatively, neither new control schemes have been academically approved by my end users, and as such, I cannot say this goal has been fully completed. However, I have followed the Interaction Design Process accordingly, performing every stage of the process to a high standard, apart from the final stage of representing the application to gather new feedback.

To further evaluate my application's usability, below are examples that demonstrate how it abides by Jakob's Ten Usability Heuristics. A poster explaining each heuristic may be found in the Appendix.

1. **Visibility of System Status**
   The automaton's status, type and description are shown to the user at all times, keeping them informed.

2. **Match between System and the Real World**
   The diagram notation used is reflective of common finite state automata notation used within and outside of the University of St Andrews for easy interpretation.

3. **User Control and Freedom**
   All error messages can be closed and are shown before incorrect actions are taken, such as the processing of an invalid automaton. When exporting an invalid automaton, users are stopped, and may either choose to continue or cancel the export operation.

4. **Consistency and Standards**
   Right-clicks that open a context menu are a standard for graphical user interfaces, which users should be very familiar with, used by most operating systems and web browsers.

5. **Error Prevention**
   Boundary checks after using 'Organise FSA Layout' move state circles back into the Viewport's boundaries if the organisation button moves them out, if their automaton is too large. Additionally, numerous error messages are displayed to the user, which halt their current action, usually if the automaton or input word is invalid.

6. **Recognition Rather Than Recall**
   With the hotkey controls being able to be shown at all times to the user, this significantly reduces their memory load, as they would not need to have to remember each hotkey binding.

7. **Flexibility and Efficiency of Use**
   The implementation of a simpler and more complex to use but faster control scheme, such as the hotkeys, means that advanced users can use the tool more efficiently and take shortcuts.

8. **Aesthetic and Minimalist Design**
   The design of the application's user interface is minimalist as the Viewport takes up much of the space, with buttons able to hide unnecessary sections, such as the controls, if you

are not using the hotkeys. One way the interface does not follow this is by showing the input bar and machine status at all times; for further minimalism, this section could be able to be minimised.

9. **Recognise, Diagnose, and Recover from Errors**
   All error messages are very clear and plainly explain why the error has occurred, with some directly telling the user how to resolve the issue to proceed.

10. **Help and Documentation**
    The controls being shown to the user are a form of help, to inform them of the hotkey controls, however, no documentation is directly attached to the application and could be added to abide by this heuristic.

As my application abides by all of the usability heuristics in some way and heavily applies a few of them, I would deem my application usable by these standards. Given that I ensured to make suitable changes to the hotkeys, which were the main feature behind the low ratings within the survey, and a new interaction method, which follows common controls has been added, I would conclude my application is moderately usable. As I cannot state how usable my end users would find the final version of the application, I will deem this objective mostly completed.

Uncompleted Objectives

The goals that were not achieved to any extent include: allowing users to control the exact angle of transition edges, utilising animations when simulating an input of the path, allowing users to save a video of those animations and allowing users to fully create their automaton using text entry. As stated in the Disruption section, I was not able to implement these features due to time constraints, however, elements of each feature may be found within the code base. Within *Viewport.js,* a *useEffect* React hook may be found, which contains a basic implementation of how text entry would function by updating the diagram upon any changes to the *machine* prop, which the text entry would do in the Interaction Window. Additionally, the stop button to halt the animation of running an input word's path exists, with the video being a commented-out option within the export drop-down type.

Evaluation Against Similar Works

Starting with comparing the application generally against all available work, my application performs mostly the same in terms of finite automata creation, users may use hotkeys to create an FSA diagram, and an input may be run on it, with its result displayed. However, of the applications I have researched, I have yet to find one that allows the user to neatly organise the diagram's layout at the click of a button, being a piece of functionality that seemingly no other visualiser has achieved. To clarify, some automata visualisers are purely text-based and render

the diagram neatly based on the textual input; however, none offer the ability to organise the layout and then edit this to the user's liking. From my perspective, the usability of the application is high in comparison with other works as it offers two distinct control schemes for users to choose from, whereas most, if not all, other visualisers only offer one control scheme. Furthermore, no other works have performed automated testing for their user interfaces and have achieved the level of code coverage of my application, showcasing the robustness of my application. Another difference is that most applications only offer one way to export an automaton, such as LaTeX code, with none allowing users to export their diagram to an image. In this regard, my application contrasts with other visualisers by allowing users to export the diagram into two image formats and a textual format. Finally, most simulators use animations to highlight the path of an input word on an automaton, with some allowing the user to edit the angle of transitions, features which were not completed for my application.

I will now be comparing the application with each significant work seen in my context survey, comparing and contrasting their features. Compared with finsm.io, my application shares many of the same features, states may be created and dragged to edit the look of the diagram, and the designs share a similar minimalism. However, they contrast by having opposite export types, with finsm.io allowing for exporting to LaTeX code and my application exporting to images. I feel that LaTeX is more useful for staff or students who want an image of the diagram, may simply take a screenshot of the viewport, whereas LaTeX code must be generated and is far more difficult to create without a dedicated export feature for it. Additionally, finsm.io consistently displays the automaton's formal definition, and while my application does display a clear machine description, I feel the use of a formal definition is better. Both JFLAP and jFAST contain numerous more automata and machines to simulate, however, these are not applicable for this project, which is solely focused on finite automata. Some major contrasts with both of these are their dated user interfaces and unfamiliar finite automata notations. Additionally, they both use toolbars for interactivity. One feature that jFAST has that my application does not is the ability to zoom in and out of the diagram, which participants in my survey stated would help create larger automata.

Shortcomings and Challenges

Given this is my first experience developing a project of this size, I faced numerous challenges and shortcomings due to my approach and limited time constraints, both due to the external factors outlined in the Disruption section and the nature of balancing university modules. By starting only having the winter break and the second semester to develop the vast majority of features, it heavily limited the amount I could achieve. Despite this, I feel I have prioritised the core of the application, which focused on creating an artefact of the highest quality instead of the number of features, ensuring that all features were routinely tested and a great deal of thought was put into the design of the application. As such, the application acts as a core foundation of a high-quality application that may be extended in the future.

In contrast to this, my prioritisation of quality over quantity did sacrifice many of my secondary goals, which would have provided useful features for end users. Additionally, usability and automated testing were not originally part of my specification. Given that I spent much of my time creating unit tests with Jest and releasing the anonymous survey to increase usability, these could be seen as redundant, as they do not contribute to any of my original goals. An alternative solution would have been to have focused on implementing as many features as possible, which would have completed more of my requirements and led to a more successful project.

Conclusion

Summary

After being tasked with providing staff and students of the School of Computer Science with a finite automaton simulator and visualiser application, my supervisor and I agreed on primary and secondary goals to include features useful to its end users. The application has been achieved, completing all primary goals fully, one secondary goal fully, with the remaining being partial to varying extents. Key achievements include the creation and dynamic visualisation of both deterministic and nondeterministic finite automata, allowing users to graphically format the diagram and run correct input on the automaton. Additionally, users can export their automaton into various formats to be used for educational material and automatically reconfigure the layout of the diagram. Finally, the application achieves automated UI testing, which simulates user interactions and evaluates the application to be fairly usable. The most significant drawback is too heavy a focus on usability, while valuable for a long-term project or one of larger scale, this only pertained to a single requirement and thus, more focus on other secondary goals would have resulted in a more complete project.

Future Work

If given more time to develop this application, such as another year or continuing this as a PHD project, I would focus on firstly completing the remaining secondary goals by adding new features to support FSA building such as exporting to LaTeX code, and completing my unit testing to achieve 100% code coverage to ensure a thoroughly tested application. At this stage, I would improve the sophistication of the ‘organise layout’ feature to avoid overlapping transitions and format large state diagrams. Secondly, I would continue using Jakob’s Heuristics to improve the usability and continue the Interaction Design Process by sending out another anonymous survey to evaluate the new version of the controls. Once the project receives positive feedback from staff and students, ideally an overall quality rating of 8 or above, I would release the application for use within the school and act as a source of support for any questions or concerns about the application to support students’ learning. During this, I would expand the project by

taking inspiration from other works by implementing the simulation and visualiser of pushdown automata, Turing machines and providing functionality to output the languages and pumping lemmas of the user's automata. If the project is well received and proves to improve students' learning of finite automata, I would aim to write an academic article about the application, to add to the field and share my application so others may improve upon it. Finally, I would aim to release the product commercially, maintaining it and contacting schools who wish to use it, further contributing to the education of finite automata in schools and universities.

Bibliography

[1]
K. C. Dodds, “Implementing a simple state machine library in JavaScript,” *Kentcdodds.com*, 2020. https://kentcdodds.com/blog/implementing-a-simple-state-machine-library-in-javascript (accessed Oct. 08, 2024).
[2]
Linas Spukas, “Finite State Machine in JavaScript,” *DEV Community*, Mar. 29, 2020. https://dev.to/spukas/finite-state-machine-in-javascript-1ki1 (accessed Oct. 08, 2024).
[3]
L. Maldonado, “How to Use Finite State Machines in React,” *Telerik Blogs*, Mar. 17, 2021. https://www.telerik.com/blogs/how-to-use-finite-state-machines-react (accessed Oct. 08, 2024).
[4]
J. Lunde, “@react-hook/mouse-position,” *npm*, Oct. 27, 2021. https://www.npmjs.com/package/@react-hook/mouse-position (accessed Oct. 30, 2024).
[5]
J. Sisley, “jest-css-modules,” *npm*, Jun. 06, 2019. https://www.npmjs.com/package/jest-css-modules (accessed Nov. 25, 2024).
[6]
OpenJSFoundation, “ECMAScript Modules · Jest,” *Jestjs.io*, Nov. 30, 2023. https://jestjs.io/docs/ecmascript-modules (accessed Nov. 26, 2024).
[7]
Haroon Ahamed Kitthu, “How to Create a Functional React Dropdown Menu?,” *Simplilearn.com*, Sep. 27, 2022. https://www.simplilearn.com/tutorials/reactjs-tutorial/how-to-create-functional-react-dropdown-menu (accessed Jan. 24, 2025).
[8]
Ilya Zykin, “How to test a Component’s CSS styles with React-Testing-Library (RTL) and Styled Components.,” *Medium*, Apr. 17, 2019. https://the-teacher.medium.com/how-to-test-a-react-components-css-styles-with-react-testing-library-rtl-styled-components-43f744334528 (accessed Jan. 28, 2025).
[9]
Cathal Mac Donnacha, “How to test a select element with React Testing Library,” *Everyday Frontend* 🚀, Oct. 06, 2021. https://cathalmacdonnacha.com/how-to-test-a-select-element-with-react-testing-library (accessed Jan. 29, 2025).
[10]
Eliav2, “GitHub - Eliav2/react-xarrows: Draw arrows (or lines) between components in React!,” *GitHub*, Jul. 20, 2021. https://github.com/Eliav2/react-xarrows (accessed Feb. 03, 2025).
[11]

S. Bekkhus, “Mock Functions · Jest,” *jestjs.io*, Dec. 29, 2023. https://jestjs.io/docs/mock-functions (accessed Feb. 10, 2025).
[12]
M. Sipser, *Introduction to the theory of computation*. Boston: Course Technology, 2020.
[13]
bubkoo, “html-to-image,” *npm*, Feb. 14, 2025. https://www.npmjs.com/package/html-to-image (accessed Mar. 18, 2025).
[14]
E. Louski, “react-xarrows,” *npm*, Jul. 28, 2021. https://www.npmjs.com/package/react-xarrows (accessed Mar. 18, 2025).
[15]
M. Zabriskie and strml, “react-draggable,” *npm*, 2024. https://www.npmjs.com/package/react-draggable
[16]
Agile Alliance, “What is Agile Software Development?,” *Agile Alliance*, Jan. 16, 2019. https://www.agilealliance.org/agile101/
[17]
Trello, “Trello,” *trello*, 2024. https://trello.com/ (accessed Mar. 18, 2025).
[18]
Qualtrics, “The Leading Research & Experience Software | Qualtrics,” *Qualtrics*, 2005. https://www.qualtrics.com/ (accessed Mar. 18, 2025).
[19]
C. W. Schankula and L. Dutton, “A Web App for Teaching Finite State Automata,” *arXiv.org*, 2024. https://arxiv.org/abs/2410.12115 (accessed Mar. 19, 2025).
[20]
T. M. White and T. Way, “jFAST,” *ACM SIGCSE Bulletin*, vol. 38, no. 1, Mar. 2006, doi: https://doi.org/10.1145/1121341.1121460.
[21]
Microsoft, “Microsoft OneNote Digital Note Taking App | Microsoft 365,” *Microsoft.com*, 2024. http://microsoft.com/en-gb/microsoft-365/onenote/digital-note-taking-app (accessed Mar. 19, 2025).
[22]
S. Rodger and T. Finley, “JFLAP -An Interactive Formal Languages and Automata Package,” 2005. Accessed: Mar. 19, 2025. [Online]. Available: https://www.jflap.org/jflapbook/jflapbook2006.pdf
[23]
Kuldeep Baban Vayadande, P. Sheth, Arvind Shelke, V. A. Patil, Srushti Shevate, and Chinmayee Sawakare, “Simulation and Testing of Deterministic Finite Automata Machine,” *International Journal of Computer Sciences and Engineering*, vol. 10, no. 1, pp. 13–17, Jan. 2022, doi: https://doi.org/10.26438/ijcse/v10i1.1317.

[24]
P. Chakraborty, P. C. Saxena, and C. P. Katti, “Fifty years of automata simulation,” *ACM Inroads*, vol. 2, no. 4, pp. 59–70, Dec. 2011, doi: https://doi.org/10.1145/2038876.2038893.
[25]
E. Gramond and S. H. Rodger, “Using JFLAP to interact with theorems in automata theory,” *ACM SIGCSE Bulletin*, vol. 31, no. 1, Mar. 1999, doi: https://doi.org/10.1145/299649.299800.
[26]
T. Singh, S. Afreen, P. Chakraborty, R. Raj, S. Yadav, and D. Jain, “Automata Simulator: A mobile app to teach theory of computation,” *Computer Applications in Engineering Education*, vol. 27, no. 5, pp. 1064–1072, Jul. 2019, doi: https://doi.org/10.1002/cae.22135.
[27]
D. Abramov and R. Nabors, “Introducing react.dev – React,” *react.dev*, Mar. 16, 2023. https://react.dev/blog/2023/03/16/introducing-react-dev
[28]
Vercel, “Next.js by Vercel - The React Framework,” *nextjs.org*, 2024. https://nextjs.org/ (accessed Mar. 20, 2025).
[29]
M. Tech, “Serverless Functions in Next.js: A Practical Guide - Masader Tech - Medium,” *Medium*, Jun. 06, 2023. https://medium.com/@masader-tech/serverless-functions-in-next-js-a-practical-guide-4499db842d26 (accessed Mar. 20, 2025).
[30]
Jest, “Jest · 🃏 Delightful JavaScript Testing,” *Jestjs.io*, 2017. https://jestjs.io/ (accessed Mar. 20, 2025).
[31]
Babel, “Babel · The compiler for next generation JavaScript,” *babeljs.io*, Mar. 20, 2025. https://babeljs.io/ (accessed Mar. 20, 2025).
[32]
ESLint, “ESLint - Pluggable JavaScript linter,” *ESLint - Pluggable JavaScript linter*, 2013. https://eslint.org/ (accessed Mar. 20, 2025).
[33]
Git, “Git,” *Git-scm.com*, 2024. https://git-scm.com/ (accessed Mar. 20, 2025).
[34]
J. Nielsen, “10 Heuristics for User Interface Design,” *Nielsen Norman Group*, Jan. 30, 2024. https://www.nngroup.com/articles/ten-usability-heuristics/ (accessed Mar. 22, 2025).
[35]
Figma, “Figma: the Collaborative Interface Design tool.,” *Figma*, 2024. https://www.figma.com/ (accessed Mar. 23, 2025).
[36]

A. Miguel, “CS3106 Human Computer Interaction Week 2 Interaction Design,” *St-andrews.ac.uk*, 2023. https://studres.cs.st-andrews.ac.uk/2023_2024/CS3106/Lectures/Week%202/CS3106%202024%20Week%202%20Interaction%20Design.pdf (accessed Mar. 23, 2025).
[37]
Google, “Google Chrome Web Browser,” *Google.com*, 2017. https://www.google.com/intl/en_uk/chrome/ (accessed Mar. 23, 2025).
[38]
E. Saldivar, “Algorithm Approach: Retrieve Depth,” *DEV Community*, Jul. 21, 2021. https://dev.to/esaldivar/algorithm-approach-retrieve-depth-48fk (accessed Mar. 25, 2025).
[39]
Hulya Karakaya, “Creating a React context menu - LogRocket Blog,” *LogRocket Blog*, Dec. 05, 2022. https://blog.logrocket.com/creating-react-context-menu/ (accessed Mar. 28, 2025).
[40]
Stately, “@xstate/react | Stately,” *Stately.ai*, 2024. https://stately.ai/docs/xstate-react (accessed Mar. 30, 2025).
[41]
K. C. Dodds, “Testing Library | Testing Library,” *testing-library.com*, 2018. https://testing-library.com/ (accessed Mar. 30, 2025).
[42]
Nik L, “#8 React.js Alternatives with Their Use-Case Scenarios,” *DEV Community*, Nov. 25, 2023. https://dev.to/nikl/8-reactjs-alternatives-with-their-use-case-scenarios-1h10 (accessed Mar. 31, 2025).
[43]
Angular, “Angular,” *angular.dev*, 2010. https://angular.dev/ (accessed Mar. 31, 2025).
[44]
E. You, “Vue.js,” *Vuejs.org*, 2014. https://vuejs.org/ (accessed Mar. 31, 2025).
[45]
M. Sulikowska, “Naturaily: Web and Mobile Development Company from Poland,” *Naturaily.com*, Jul. 24, 2024. https://naturaily.com/blog/best-nextjs-alternatives (accessed Apr. 01, 2025).
[46]
Gatsby Inc, “The Fastest Frontend for the Headless Web,” *Gatsby*, 2025. https://www.gatsbyjs.com/ (accessed Apr. 01, 2025).
[47]
Remix, “Remix - Build Better Websites,” *remix.run*, 2025. https://remix.run/ (accessed Apr. 01, 2025).
[48]
Eleventy, “Eleventy Home,” *Eleventy*, 2025. https://www.11ty.dev/
[49]

Oracle, “Trail: Creating a GUI With Swing (The Java™ Tutorials),” *docs.oracle.com*, 2024. https://docs.oracle.com/javase/tutorial/uiswing/index.html (accessed Apr. 01, 2025).
[50]
Web Assembly, “WebAssembly,” *webassembly.org*, 2025. https://webassembly.org/ (accessed Apr. 01, 2025).
[51]
Vinay C, “Kahn’s Algorithm for Topological Sorting - Vinay C - Medium,” *Medium*, Jul. 27, 2024. https://medium.com/@vinay.chil/kahns-algorithm-for-topological-sorting-78ef7ba7e04a (accessed Apr. 02, 2025).
[52]
Node.js, “Node.js,” *Node.js*, 2023. https://nodejs.org/en (accessed Apr. 02, 2025).
[53]
Scrum.org, “What Is Scrum?,” *Scrum.org*, 2025. https://www.scrum.org/resources/what-scrum-module (accessed Apr. 04, 2025).

Appendix: User Manual

The application is hosted on the school servers for access on the school network by following this link: https://jeh27.teaching.cs.st-andrews.ac.uk/. If you would like to instead install and execute the application locally, follow the Installing and Executing section.

Installing and Executing

The application requires Node [52] to be installed and was created using version 16.20.2. Once installed, start by running *npm install* to install all React libraries and dependencies for the application. Once installed, this does not have to be run again. To run the application, first build the application by running *npm run build,* and once completed, run *npm run start* to run the application as a server attached to localhost:3000. Next.js will provide a hyperlink to this in the terminal. Locating to localhost:3000 in your browser will display the application.

Using the Application

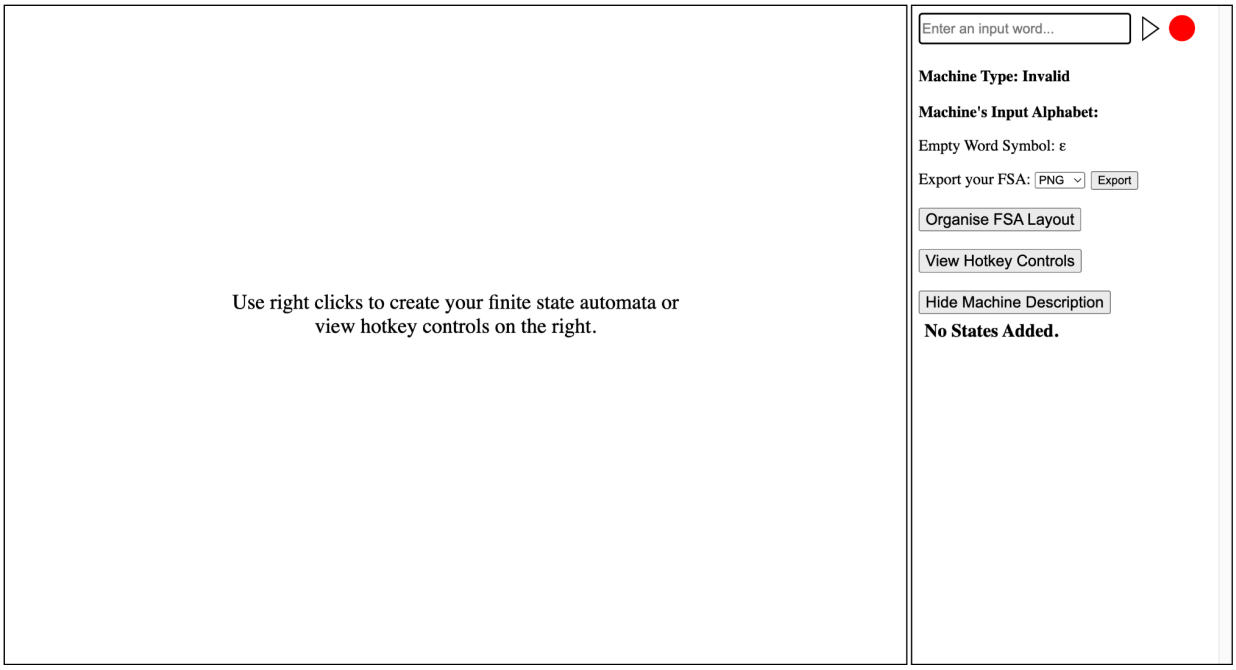


Figure 20: Screenshot of Application

On the left side is the Viewport, which may be used to graphically create a finite state automaton and on the right side is the Interaction Window, which allows you to run an input on your automaton, view information about it, or export it as an image or JSON file.

There are two main control schemes that allow you to create and edit your automaton. The first is through right-clicks and selecting an option from a menu shown. Right-click on the background to either create a new state or clear the Viewport of all states and transitions. Right clicking on a state allows you to either: create a transition by selecting that state as the origin state and then right click on another to finish creating the transition by selecting the destination state, toggle the accept status of a state, make that state the start state of the automaton or delete it and all connected transitions. Right-clicking on a transition allows you to delete it.

The second is hotkey controls that are listed by clicking ‘View Hotkey Controls’, these are combinations of keys and mouse clicks.

The controls are the following:

- Double Click (Blank Space) — Creates a new state at the mouse’s position.
- Click (State) — Allows the user to change the state’s name.
- Drag (State) — Updates the state’s position.
- Shift + Click (Two States) — Creates a transition between the two states.
- Shift + Click (One State x2) — Creates a self-pointing transition.
- Alt + Click (State) — Deletes the state and all connected transitions.
- Alt + Click (Transition) — Deletes the transition.
- Double Click (State) — Toggles whether the state is an accept state or not.
- Control + Click (State) — Turns the selected state into the start state.

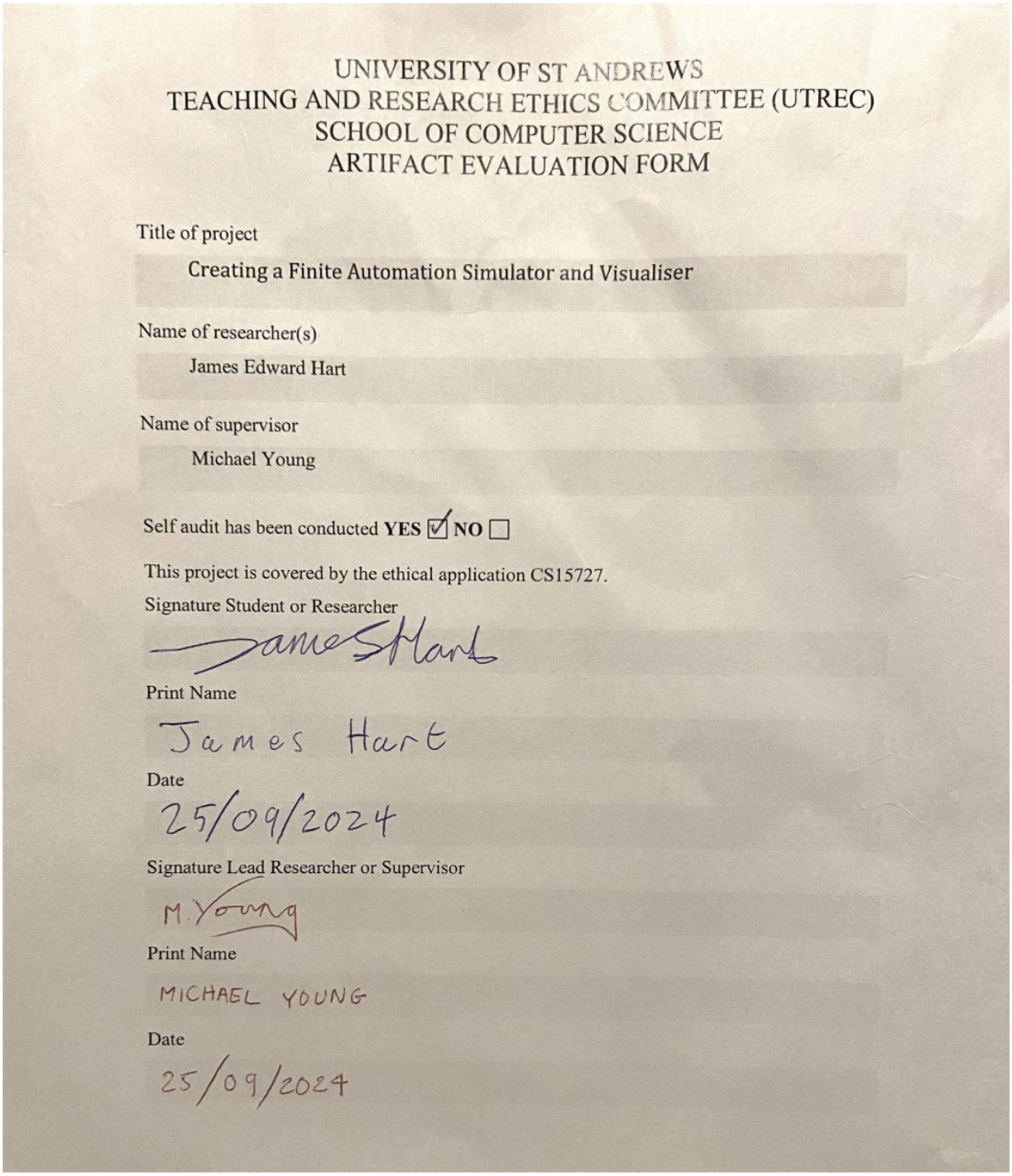
Note: Alt is equivalent to the ‘option’ key on macOS.

To export the automaton, select an option from the dropdown and click export to either download a PNG or SVG file of the diagram or a JSON file detailing the machine built.

Appendix: Other Materials

Artifact Evaluation Form

Ensuring that the project is covered by ethical application CS15727.



UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
ARTIFACT EVALUATION FORM

Title of project

Creating a Finite Automation Simulator and Visualiser

Name of researcher(s)

James Edward Hart

Name of supervisor

Michael Young

Self audit has been conducted **YES** ☑ **NO** ☐

This project is covered by the ethical application CS15727.

Signature Student or Researcher

James Hart

Print Name

James Hart

Date

25/09/2024

Signature Lead Researcher or Supervisor

M. Young

Print Name

MICHAEL YOUNG

Date

25/09/2024

Participant Information Sheet

Participant Information

Creating a Finite Automaton Simulator and Visualiser

James Hart, Michael Young

What is the study about?

We invite you to participate in a research project based on creating an application that allows users to build, appropriately simulate and visualise a finite state machine. The primary set of users will be either lecturers making finite state diagrams for educational material or students making them for practical work within Computer Science modules. This study aims to allow students and lecturers to use a prototype of the application and collect their feedback, intending to make the application more usable and efficient.

A finite state machine is a model of a computational algorithm consisting of a finite number of states the machine may be in and how the machine may transition between them. Based on a given input, the machine will toggle between various states depending on each letter contained in the word. If a state finishes computing each letter of an input word and the final state is an accept state, the machine will accept the word, if not it will reject it. If a word contains a letter which is not computed by any of the states of the machine, the word is invalid.

Why have I been invited to take part?

You are either a lecturer or student in the School of Computer Science.

Do I have to take part?

This information sheet has been written to help you decide if you would like to take part. It is up to you and you alone whether you wish to take part. If you do decide to take part, you will be free to withdraw at any time without providing a reason and with no negative consequences.

What would I be required to do?

You will be asked to complete a questionnaire which asks participants to use the prototype application and then contains 8 questions about their user experience of the prototype. We anticipate this will take 10-15 minutes to complete, depending on the depth of the answers provided.

Are there any risks associated with taking part?

The only possible risk may be that the participant is not familiar with finite state automata or finds the prototype highly unusable, leading to confusion and frustration over not being able to use the application.

Are there any benefits associated with taking part?

In the event the application is deemed usable for teaching, a direct benefit to both lecturers and returning students is the introduction of an application tailored to their user needs during future modules, allowing them to provide feedback before it is introduced and required to use.

Informed consent

It is important that you give your informed consent before taking part in this study, so please do not hesitate to get in touch (using the contact information at the end of this document) with any questions before you provide your consent.

What information about me or recordings of me ('my data') will you be collecting?

We will not collect or process any personal data. All data you provide will be anonymous, which means that no one could use any reasonable means to identify you from the data. As your data will be anonymous, it cannot be withdrawn because we will not know which data is yours.

For more information on the University's data protection and privacy policies visit: https://www.st-andrews.ac.uk/terms/data-protection/

How will this data be managed and used?

The anonymous data collected will be stored securely on a drive on the University network and only the researchers will be able to access it. The data will be analysed as part of the research study and then published in my dissertation.

It is expected that the project to which this research relates will be finalised by April 2025. After the project has been completed, the data will be destroyed.

Ethical Approvals

This research proposal has been scrutinised and subsequently granted ethical approval by the University of St Andrews Teaching and Research Ethics Committee and is covered by the Artifact Evaluation Form for Computer Science project, which abides by the ethics application for "Evaluation of artefacts produced for CS projects" (with the ethics approval code CS15727).

What should I do if I have concerns about this study?

In the first instance, you are encouraged to raise your concerns with the researcher. However, if you do not feel comfortable doing so, then you should contact my Supervisor or School Ethics Contact (contact details below). A full outline of the procedures governed by the University Teaching and Research Ethics Committee is available at https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/complaints/.

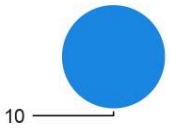
Contact details

Researcher(s)	James Hart	**Supervisor**	Michael Young mct25@st-andrews.ac.uk

Survey - Questions and Results (10 Responses)

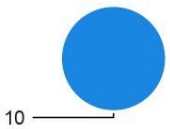
1

Opening Statement - This project is covered by the ethics application for "Evaluation of artefacts produced for CS projects" at the University of St Andrews. The participation in this project is completely voluntary and you can withdraw from the study at any time without giving an explanation and with no disbenefit. You can ask questions about the project and have had them answered satisfactorily. In this project, no personal data will be collected. Data collected in this project will be anonymised and the raw data will be deleted within 3 months after the completion of the project. The data will be only accessible to the named researchers on the project and the data analysis results will be published in a dissertation or an academic publication.



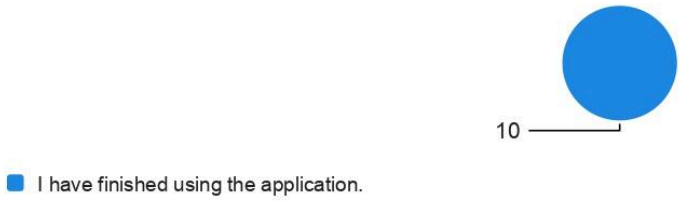
I understand and agree to proceed.

Introductory Brief - You are being invited to participate in a research study titled Creating a Finite Automation Simulator and Visualiser. This study is being done by James Hart from the School of Computer Science at the University of St Andrews. The aim of this project is to develop an application for creating and visualising finite state automata. The intent is for it to be used by lecturers and students for teaching purposes, such as lectures or practical work. As such, participants will be asked to use a prototype of this application and respond to questions about their user experience in a short questionnaire, taking around 10-15 minutes depending on the depth of feedback. Responses will be used to improve the user experience and robustness of the application. If you are interested in taking part, please download a copy of the participant information sheet here and retain this for your records before starting the questionnaire. If you have any questions, please email Michael Young at mct25@st-andrews.ac.uk. Your participation is entirely voluntary, and you can withdraw at any time. You are free to omit any question. In the event you are not aware of what a finite state machine is or need a quick reminder, please read the Participant Information Sheet.



I understand and agree to proceed.

Application Brief - Please run the application by navigating to https://jeh27.teaching.cs.st-andrews.ac.uk on the school network (eduroam). The controls are found on the right panel indicating how you may build your finite state machine. Build any valid finite state machine and test various input words on it. After you are happy with your machine, export it to either a PNG or SVG format. Once you have finished, proceed to the next part of the questionnaire where you will be asked questions based on your experience with the application.



Q1 - Overall how would you rate the artefact as a tool for building finite state automata?

Field	Min	Max	Mean	Responses
Rating	3.00	8.00	5.60	10

Q2 - Please elaborate, indicating what the artefact does well and what may be improved.

Does well: - Seemless creation of transitions - Clean interface - Export is cool What may be improved: - No delete all button - Refreshing clears everything - Control scheme is very poor - Clicking on text when trying to delete a state can result it not being deleted - Too easy to accidentally create stuff - Zooming the page breaks it

The backwards arrow for the start state did not make sense - you should make instructions clearer because I didn't know to eg make a state before making a start state. It's too finnicky. I like that it tells whether it accepts/ rejects and machine type, but you might want an intro on what it actually does

Pros - Smooth interface, - Intuitive controls helpfully detailed - Clean design, - Functionally Valid Improvements - Could do with undo functionality with Ctrl-Z - States which are deleted should decrease global state number value (creating a new state after deleting one that has a higher value than the one just deleted seems a little off). - transition arrow from dragging would be even more user-friendly as I found shift clicks can be a little difficult to grasp. If not dragging, some sort of visual notion that a node has been selected would be helpful. - clicking on a node regularly without shift should also have the ability to create transitions to other nodes by clicking shift and then clicking on the other node. - alt-click to delete an object should denote what an 'object' is. Initially thought it only applied to nodes, later found out transitions could be deleted in the same way. - Ability to edit fsa data in transition table directly would be nice. - node map navigation would be nice (panning zooming, etc) - Clearer indication of 'controls' and 'machine description' would be nice (just a couple headers to indicate the sections)

The display is practical and straightforward, as it contains no unnecessary or distracting UI elements. The use of a colour indicator to show whether the FSA accepts a language is intuitive. Although the controls take a little time to become comfortable with, they eventually become second nature. I also appreciate that the empty string character is provided for easy copying. Controls for MAC or other devices would be appreciated, as there is no alt key on a Mac. Furthermore, an undo keyboard shortcut (Ctrl-Z) would be helpful to prevent accidental deletion of states.

The artefact allows me to construct and test a finite automata quickly. Furthermore it allows me to export the automata to an image making it useful for report referencing. However certain aspects make it quite finicky to use, such as having similar shortcuts making it easy to make mistakes. For example when trying to make a start state I deleted objects or when trying to make a looping transition it turned the state to an accept state. Perhaps a more comprehensive UI with buttons to insert different components would be more intuitive and visually appealing.

It handles all the states and transitions correclty and exporting as an image makes it useful for connecting to external work easier. Can be a bit problematic to use since the controls are so similar and there is no undo however. I would prefer more distinct control selection, perhaps a tool bar with drag and drop functionality

The controls are not particularly intuitive but the work fine. I kept accidentally creating new states because I would click away from the whatever I was editing onto the blank background.

The wavy arrows are cool, however some there are some issues where arrow heads are not pointing where they should, notably on the start state arrow and the loop to self arrow. Currently the system doesn't seem to allow multiple transitions from one state to another, which is fine for DFAs but not really for NFAs. I really like the machine type information, would be interested if there was a conversion functionality for instance a NFA to DFA converter or a RegEx generator. The epsilon transitions are annoying to make but this understandable

It is possible to add all the components of a FSM, the instructions are clear and the saving seems to work but the usability is not good. The major problems I had with it was that it's not possible do undo a mistake (unless I've missed something) so I would often accidentally delete a state that was needed for the machine and would have to add the state back along with all its transitions. The other problem I had was with the way the transitions snap into place, this makes it very hard to see where the transition is going to and where it is coming from if there are many transitions to a node, I tried to make a three state machine and did not manage to make it readable.

The program works decently well, however remembering the hot keys is annoying. Additionally, there does not seem to be a way to add multiple transitions between states, which makes building NFA's challenging

Q3 - How intuitive did you find the artefact to use?

Field	Min	Max	Mean	Responses
Rating	3.00	7.00	5.11	9

Q4 - Was there anything that you found slowed down FSA development?

Accidentally creating new things instead of editing existing ones, remembering controls

The finnicky controls

the shift-click mechanism to create transitions.

Accidental deletion while learning the controls and the inability to undo it.

It was very easy to accidentally add new states, as dragging other states sometimes lead to clicking on the open space. Additionally accidental deletions meant I had to start again.

Having no undo, and controls did not always register, which meant actions like making a start state, and deleting the state could be mistaken for one another. Remaking the states and transitions then took some time

Accidentally creating new states.
Having to check what the controls were to achieve different things (e.g. delete, make start state, make state and accept state,...)

Transitions cover over each other sometimes so trying to select a transition to edit is awkward. Unfamiliar key combinations. Manual testing,i t would be nice to be able save a set of inputs to test all at once to identify if the FSM is compliant with intent. NFA doesn't allow multiple transitions from state 1 to state 2, or at least I couldn't get it to do it, if I want to constrain it to only DFA then maybe have that as a toggle-able behaviour or have a converter functionality rather than a odd hybridised set of constraints when developing. Transitions always being initialised as A, would be more helpful if it prompted what you wanted to make the transition letter before making a transition rather than changing it every time. Still have the edit but allow a set upon create input or something so all of one input can be made at once or don't need to deal with overlap so much

Accidentally deleting crucial states

Hotkeys were not intuitive.

Q5 - How do you find the design of the user interface? Would you prefer your FSA to be displayed in any alternative way?

I think the LHS of the page is good, the rhs is maybea little dense.

UI is generally good, could use maybe more colour/variety

The controls could have been displayed nicer but the FSA design itself was clear

UI is nice. Clean and minimal is the way to go. In terms of mouse graphics when hovering over elements, could do with more 'link select' mouse graphics rather than 'text select' graphics.

Very straightforward. Adding a zoom option for larger FSAs would be helpful.

I would prefer the side bar to possibly be horizontal along the bottom with buttons to press as well as the keyboard shortcuts. Perhaps a different colour background would make the states stand out more.

The FSA follows the format I would expect so the display is good. As stated earlier, a toolbar as part of UI would be helpful

I liked the FSA display, but the controls could be improved.

Is pretty nice, simple but to the point. Would like to be able to export the table as well as the image. The wavy lines are fun additions and it seems to avoid too much overlap generally speaking. I still would like my multiple transitions from same two states however for my NFAs

The arrow head to the start state points the wrong way, other that that it was all good.

UI design is good minus a few visual bugs with arrows.

Q6 - Do you feel that there are any features missing from the application? Or anything useful that could be added to make it easier to use?

Load feature, download in some non-image format

Undo/ redo
Drag and drop interface

Navigation of the node map, ctrl-z functionality, transition table editing

Undo functionality: a button for rearranging the FSA into a grid or an optimal pattern for clearer reading and viewing.

I think more buttons to insert each of the objects would be useful, and perhaps a way to generate example machines.

Undo/Redo Buttons, Toolbar, Perhaps a save functionality (not as a png but as an instance of the fsa, to go back and edit later, would need import functionality too)

I would have liked mouse-only and/or keyboard only controls rather than mouse+keyboard.
Maybe add the idea of a 'selected' object and a panel to edit that object, and the ability to tab around.
A really nice feature would be to visualise the movement from state to state for a given FSA and input, highlighting the current object at each step. This would work nicley for invalid FSA too.

Conversions from NFA to DFA and I guess DFA to NFA, RegEx generation would also be neat but less required
Proper NFA support
Table exporting as well as loading so you could save your progress more easily
Allow sequential test inputs to see if the FSA you created is what you wanted after making edits
A clear button, i don't want to do that manually or reload the site or anything

Ctrl + z or an undo button would make it a lot easier to use and I think using right clicks to change the diagram would be more intuitive.

Converting from an NFA to a DFA could be useful. Additionally, the ability to add multiple transitions between two states is missing. Also, showing the process of recognising a word could be useful, ie with an animation showing which states are active.

Q7 - Did you encounter any errors or unexpected behaviors while using the tool? If so, how did the tool handle them?

Trying to delete an object while clicking on its text

Deletes when you want to make a start state

No errors.

No.

I often found myself accidentally deleting or adding new states. I simply just had to start again, perhaps an undo function would be useful?

The start state arrow faces the wrong direction

No errors

The arrow heads not pointing where they should, for instance directly pointing away from state they were going to. Dragging not working properly when transitions are overlapping state meaning that when you drag nothing happens to the state and instead a new state is added once you move off the FSM. Not being able to select what you want, shift linking two states sometimes fails and then when you try again ends up making a transition in the opposite direction, I assume its not selecting first state consistently which may be due to overlap issues. Also couldn't make my multiple transitions, not sure if this an intended constraint of the program or a bug but it is very annoying. Can sometimes toggle the state accept value when adding a transition looping to itself

No errors

N/A

Q8 - If you used similar FSA-building tools before (either online or university-provided), how does this artefact compare in terms of usability?

I preferred other software as I found it easier to build transitions textually than visually and sometimes they showed the transitions through the machine on a given input

N/A

Has potential to surpass other software due to its simplicity and easy learning curve.

In certain FSA-building tools, transitions occur by holding down the mouse button, thereby displaying the length of the transition in real-time. This feature is quite satisfying (https://madebyevan.com/fsm/).

It is easier to use than a previous tool i used in CS3052, as that required us to manually describe the FSA using a specific text format. Therefore, this was quicker and simpler to use.

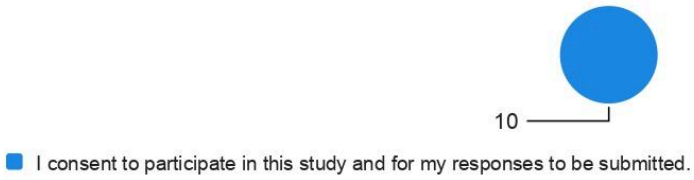
N/A

No similar tools used

There are some additional features to this artefact such as image exporting, with clearer analysis of the type of machine it is, resolutions of the input seems very reliable and responsive. However the FSA's able to generate are limited in sense of NFA not having multi transitions even if an equivalent DFA machine can still be made with this artefact. It doesn't have conversion functionality unlike others, nor export import functionality for saving machine to work on later or adapt.

Having previously worked on a similar tool, this felt roughly comparable in functionality.

Closing Statement - By clicking the 'Submit' button below, you are consenting to participate in this study, as it is described in the participant information sheet, which you can download here. If you did not yet download and keep a copy of this document for your records, we recommend you do that now.



Weekly Meeting Record

Date	What has been done	Plan for next week	Things to Discuss	Notes from meeting
17/09			○ Agree scope of project and technologies - React, ask if it's wise to keep doing the same language. ○ Finite Automaton Simulator and Visualiser ○ Agree which ethics form - test with other lecturers for feedback ○ Agree resources need - none	- Application would be an easy way to markup automata, show the current state, animation to running - Agreed on primary and secondary objectives
25/09				- Signed Artifact form and agreed to submit DOER - Agreed that the project would be started now
18/10	- Next.js built in repo - Mockup finished - Git repo and project are ready - Have a React library for FSAs - Created all user stories	- Create viewport and states - Install Jest	- Cannot update node and npm on lab machines and so current version of Next.js does not work on them - Ask Michael about output, I believe the output is only accept or reject - FSA library will be similar to the text input - Mockup and design choices	- For report, keep planning, top level overview of what you did, how did you decide to do something, this thing VS this thing and what the trade offs were - Can update node and npm, use npm to install latest version of node into project directory
01/11	- Git issues and npm fixed - Circles can now add state - Circles are draggable	- After each feature, will update documentation - No work for next two weeks but will aim to make progress for the 29th	- Feedback on circles - Discuss possible deadlines (Christmas break) - Ask Michael about school funding	- Interim Demo (good to have all primary stuff done) • Secondary and report done after - Write report and user guide (appendix)
22/11	- Circles are finished, very fiddly but now at a stable version	- First batch of jest tests - Make a start on transitions	- How would you like coding to be referenced (citation or link, and assume they are all included in bibliography) - Possible leave of absence and how the diss will work with that	
28/11	- Jest fully set up and working - Fixed issues with Babel - Wrote up reasonings for dependencies, etc	- Plan for holiday is to aim for all primary goals done for the demo, will have a lot of time - Look into previous dissertations - Start report and user guide	- Last week - Ask about format of user guide	- Comment in code with URL with another comment under saying End of section. Then something in report saying 'Some code in the artifact was borrowed from sources online (this is not major) and this is referenced in the code.' - Talk to Graham about dissertation
30/01	- FSA creation in backend - Interaction Window	- Finishing transitions graphically - Dynamic updates - Primary goals	- Interim demo, how shall it go? - Questionnaire, after demo will contact lectures for improvements (how would I do this and how can I make sure the form is ethical?) • When should I do this? Now or later	- Demo = Imagine he's never seen it before **Interim Demo - Week 2:** • Just there to check you are making satisfactory progress • Informal demo to supervisor in regular meeting slot • Upload screenshots of codebase to deliverable in MMS after demo (evidence that the project is going along) - 3 States for machine to tell user in window: • Deterministic, Non-deterministic and invalid • Colours can be done later - ~~Put Machine description at bottom~~ - ~~Export into JSON - machine~~ - Questionarrie should be done near to the end • 5 is a good number • Free text questions • Let them do it when you aren't in the room (could host in, host in NGinXs they just need to go to link) • Qualtrics Form • Release questionarrie after vacation week - Include hosting link to final project in report - Think about doing some every week https://www.geeksforgeeks.org/how-to-export-a-git-project/

06/02 **Interim Demo**	- Transitions can be added or removed with input letters, any form of FSA can be made - Start state can be set - Accept states can be toggled - Multiple input letters on a transition - Empty words can be inputted - Only deterministic FSAs can have their input run	- Test everything that has been implemented - Continue primary goals until vacation week so can do as much as possible before the questionnaire • Vacation Week: ○ Send out questonnaires (with hosted app) ○ Report 1st Draft ○ Secondary goals	- Interim Demo - Should straight for arrows be default? No - Re-evaluate priorities he wants	- Instead of rejected for undef, you would say not a valid word or not a word in input alphabet, and show input alphabet next to machine. Only a reject word if does not go to accept state - For LaTeX, code in that will fix the look of it - Change to breathe-first search for nondeterminism
13/02	- Breathe-first search	- Continue testing and developing		- For report, mention the breathe first over depth first, with a diagram of a nondeterministic machine where depth first is bad - Testing is long because it is difficult, automated test for gui elements, really talk about this and how tis revealed bugs and how complicated and how it leads to a more well developed software . Show the difference between testing FSA and Gui testing and why some test cant be done - Export to JSON and use that for text entry and saving and loading, and later can change it if you wish. Combine them and mention in report or feedback from users.
17/02				
24/02				
Vacation Week				- Send questonarrie out to students and staff for wider pool - GUI testing in report - boast about 100% code coverage - When choosing algoirthms for reorganising layout, write justifications and in report compare various algoirthms - Report structure is only advisory, can group design and implementation in Tasks if that's better - Send michael any report sections as you write them if you want for proof reading
13/03	- Various sections of report - Survey sent out with feedback - Part 6 - Hosting on labs - Questionarie + documents	- Report - Develop more features	- Documents - Show part 6 - Consent form	- Indicate how much time you spent testing - Add primary objective about GUI testing - Because of secondary supervisor, use usability heuristics - During report evaluation say about responses from people, if I were to improve this user interface I would do this based on responses
21/03	- Survey sent out - Report started	- Create UI improvements - Final features	- Report structure • Core features sections • Where to put testing (main body or appendix) - Discuss survey responses - Add extra goals to requirements (usability and save and loading an FSA) - Context survey • 10 heuristics? • How many examples? - Referencing applications and technologies	Context survey - add key takeaways or whats learned Good to have overall design section then technologies, Just makeone design and implementation usable by some group. Use survey results to justify that. Automated testing for everything. Good to have that as an objective. Heading to say what has changeed, why and assert miachale has agreed Software engineering process: continous integration = set of tests you can run after each change, you can then verify it has not brokeen and version control **for issues say what you take away from each bug, instead of 'i had this bug' you can say 'It's clear dependencies are hard to resolve', say something meeaningful and how you reesolved it**

				For babel, if you don't have anything more to say combine them, but should mention any alternatives or issues you had, more reeflective For survey, good for deesign and evaluation, put after Ethics Survery section, bits relevant for each design and evaluation For task 7 treat cycles as a point maybe Testing should be its own section Put trello into reeport itself When results for survey are in take out queestions from appendix Reference technology on the first mention in the text (not heeading)
27/03	- Primary goals met - Anonymous survey section written - Made changes to report from previous meeting	- By Monday • Design • UI Changes • Implementation - Finish rest of report by Thursday next week for Thursday proof read	- Ask if participant rating data should be organised by participant number or order to show split	- Final report draft for Friday for Michael to read - Update final artefact to host servers for final submission (on Friday) - Use bar chart or whisker chart to showcase rating results - UI changes • Avoid the toggle and just have hotkeys (remove blank space click) • Set end of Friday at code freeze
03/04	- Design, Implementation and Testing finished - Code freeze	- Evaluation, Introduction, Conclusion and abstract	- Guidance for upcoming sections	- Make sure that you use one term across the essay • Use automaton and mention machine in code means automaton - Approach for not starting in sem 1 • Look at 2020 dissertation • Section near the start called 'Disruption' ○ Explain the whole thing in an academic way and how it affected you. - Conclusion should have a section called 'future work', say that if you have another year or PHD or new project, possibly make it a commerical project very large big picture • Showing that you understand this stuff - Miriam knows a lot about Hci and how users interact with spftware so say things about user studies and the needs of different group of users. Even if it was just by future work - Remove futre work in evluation - For abstract, use the first sentence from each paragraph and cont • Under 300 words • Should set the scene and summarise the project in 300 words

Usability Heuristics Poster [34]

Nielsen Norman Group

Jakob's Ten Usability Heuristics

1 Visibility *of* System Status

Designs should *keep users informed* about what is going on, through appropriate, timely feedback.

Interactive mall maps have to show people where they currently are, to help them understand where to go next.

2 Match between System and the Real World

The design should speak the users' language. Use words, phrases, and concepts *familiar to the user,* rather than internal jargon.

Users can quickly understand which stovetop control maps to each heating element.

3 User Control *and* Freedom

Users often perform actions by mistake. They *need a clearly marked "emergency exit"* to leave the unwanted action.

Just like physical spaces, digital spaces need quick "emergency" exits too.

4 Consistency *and* Standards

Users should not have to wonder whether different words, situations, or actions mean the same thing. *Follow platform conventions.*

Check-in counters are usually located at the front of hotels, which meets expectations.

5 Error Prevention

Good error messages are important, but the best designs carefully *prevent problems* from occurring in the first place.

Guard rails on curvy mountain roads prevent drivers from falling off cliffs.

6 Recognition Rather Than Recall

***Minimize the user's memory load* by making elements, actions, and options visible. Avoid making users remember information.**

People are likely to correctly answer "Is Lisbon the capital of Portugal?".

7 Flexibility *and* Efficiency of Use

Shortcuts — hidden from novice users — may *speed up the interaction* for the expert user.

Regular routes are listed on maps, but locals with more knowledge of the area can take shortcuts.

8 Aesthetic *and* Minimalist Design

Interfaces should not contain information which is irrelevant. Every extra unit of information in an interface *competes* with the relevant units of information.

A minimalist three-legged stool is still a place to sit.

9 Recognize, Diagnose, *and* Recover from Errors

Error messages should be expressed in plain language (no error codes), precisely indicate the problem, and constructively suggest a solution.

Wrong-way signs on the road remind drivers that they are heading in the wrong direction.

10 Help *and* Documentation

It's best if the design *doesn't need* any additional explanation. However, it may be necessary to provide documentation to help users complete their tasks.

Information kiosks at airports are easily recognizable and solve customers' problems in context and immediately.

NN/g

www.nngroup.com/articles/ten-usability-heuristics/