Senior Design Design Document

Improved Visualization for Formal Languages

Group Members:

Chris Pinto-Font (cpintofont2021@my.fit.edu)

Vincent Borrelli (vborrelli2022@my.fit.edu)

Andrew Bastien (abastien2021@my.fit.edu)

Keegan McNear (kmcnear2022@my.fit.edu)

Faculty Advisor:

Dr. Luginbuhl (dluginbuhl@fit.edu)

Client:

Dr. Luginbuhl

Florida Institute of Technology

September 4, 2025

Table of Contents

1. Introduction

1.1 Purpose

This document presents the system design for the **Improved Visualization for Formal Languages** application. It ensures the design satisfies requirements defined in the Requirements Document. It outlines architecture, modules, data flows, GUI layout, algorithms, and educational animations necessary to provide a modern, user-friendly DFA visualization and learning tool.

1.2 Scope

The application will:

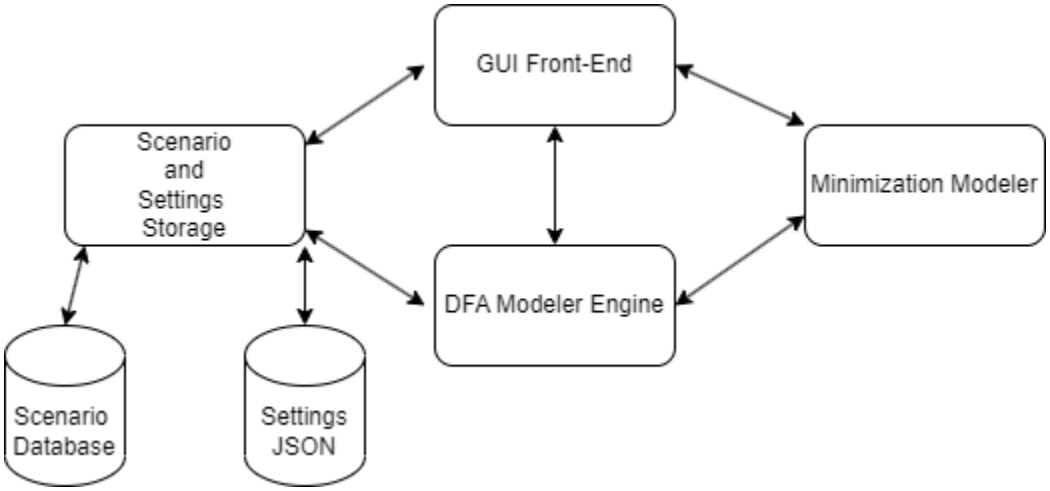
- Allow intuitive creation of DFAs through a graphical editor.
- Plot states, designate initial/final/dead states, and define transitions.
- Animate DFA execution on user-entered strings.
- Automatically minimize and complete DFAs using well-known algorithms.
- Provide an improved GUI toolbox inspired by JFLAP but easier to use.
- Include in-program documentation and tooltips so users learn without leaving the application.

1.3 References

- IEEE Standard for Information Technology – Software Design Descriptions.
- JFLAP open-source project (Java) as an inspiration for GUI and functionality.

2. System Architecture

2.1 High-Level Architecture Diagram



2.2 Component Overview

- **DFA Editor (Front-end): Python-based GUI where users draw states and transitions, input strings, and run animations.**
- **Computation/Algorithm Engine: Executes state transitions, minimization, and completeness algorithms; returns results to GUI.**
- **Documentation/Help System: In-program guides and tooltips explaining DFA concepts and program features.**
- **Data Storage: JSON or SQLite storage for saving/loading DFA projects.**
- **External Libraries: Python GUI library (Qt or Tkinter) and optional Cython modules for performance.**

3. Detailed Design

3.1 DFA Editor Modules

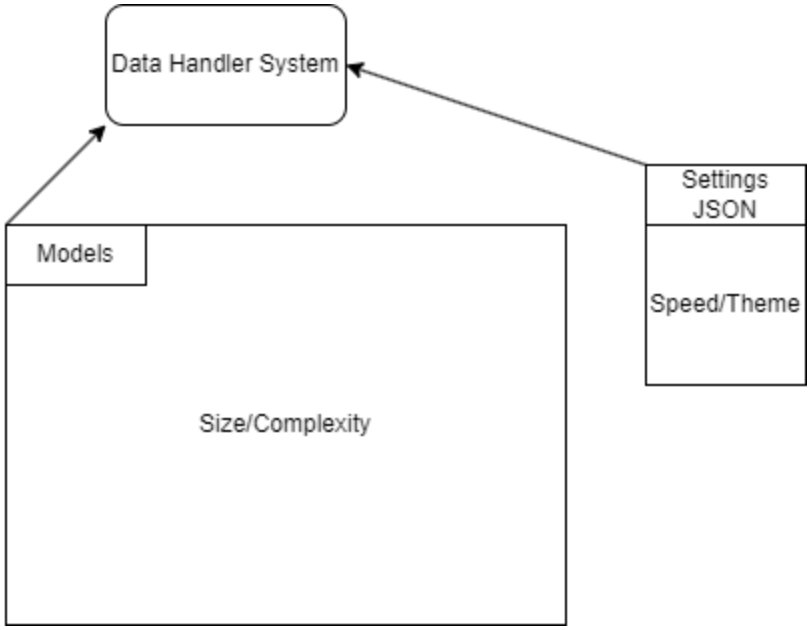
- State/Transition Module: Create, edit, label, and delete states and transitions. Supports multiple symbols per arc.
- Execution Module: Simulate DFA on one or multiple input strings and animate traversal.
- Minimization/Completion Module: Implement algorithms to reduce states and fill missing transitions.
- Documentation Module: Display contextual help, tutorials, and examples in-app.

3.2 Backend/Computation Services

- Simulation Service: Given a DFA and input, step through transitions and produce acceptance/rejection.
- Minimization Service: Apply standard minimization algorithm (partition refinement).
- Completion Service: Detect incomplete DFAs and add dead states/needed transitions.

3.3 Data Storage Model

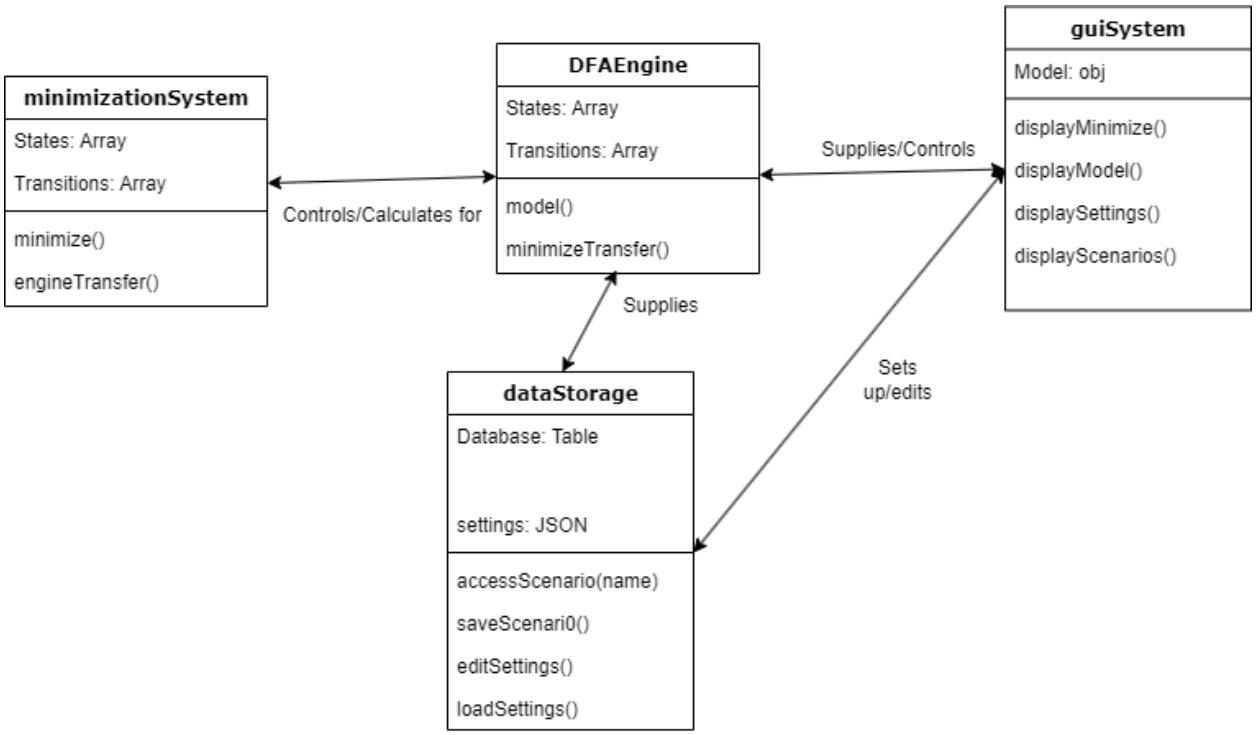
- Model table with past scenarios labeled by complexity/size.
- Settings JSON.
- Both accessed by the data permanence handler class.



3.4 Communication & Security

- Local application; no network traffic for core features.
- All file saves/loads are sandboxed to user's environment.

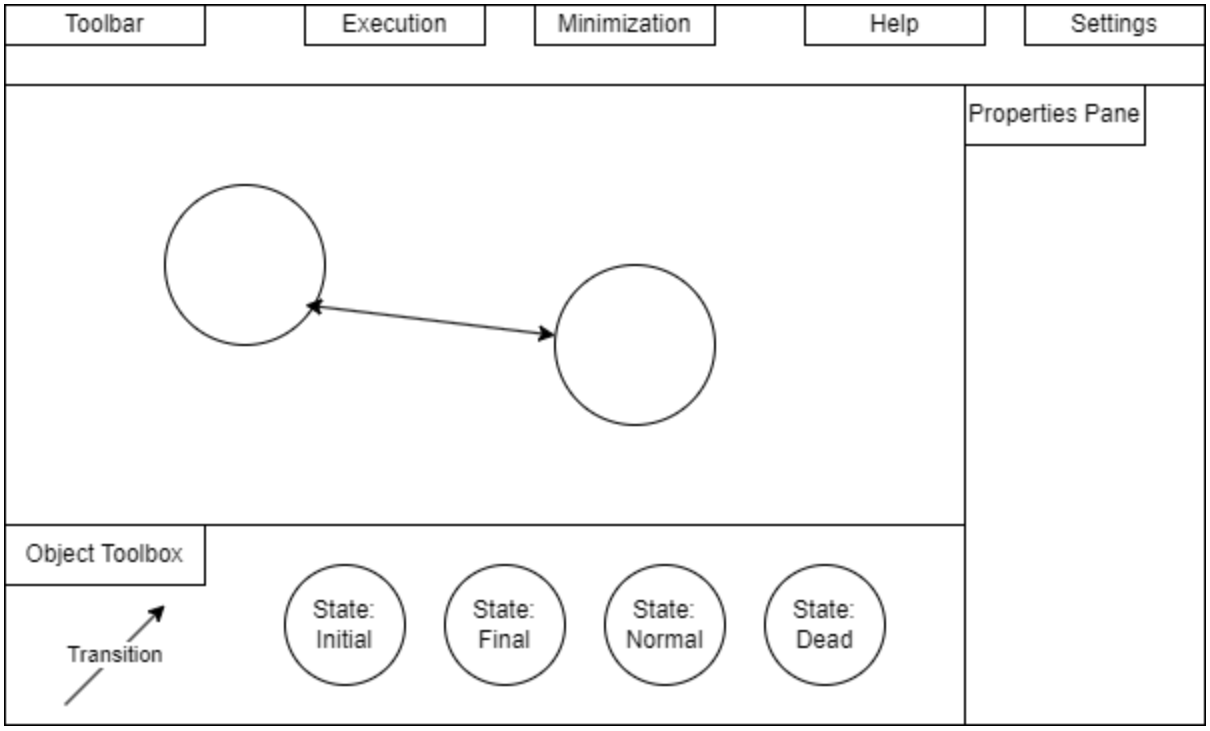
3.5 UML Class Diagram (Mockup)



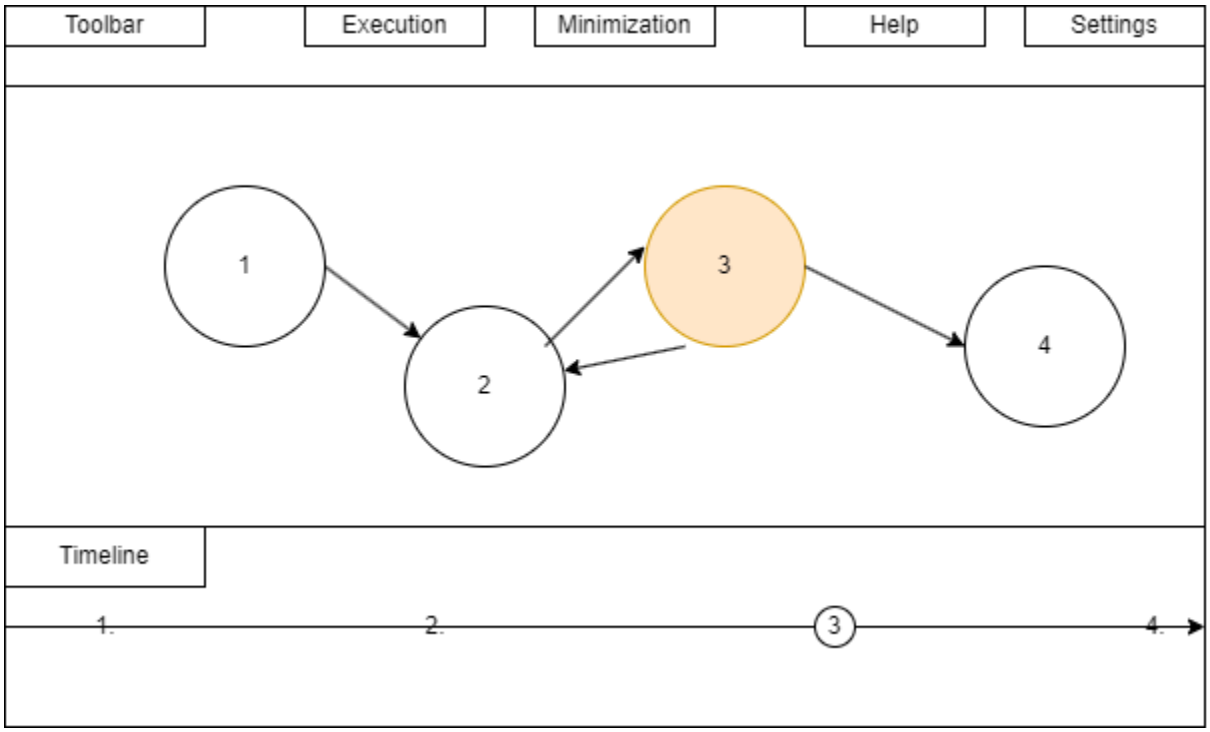
4. Graphical User Interface (GUI)

4.1 Mock-ups of Key Screens

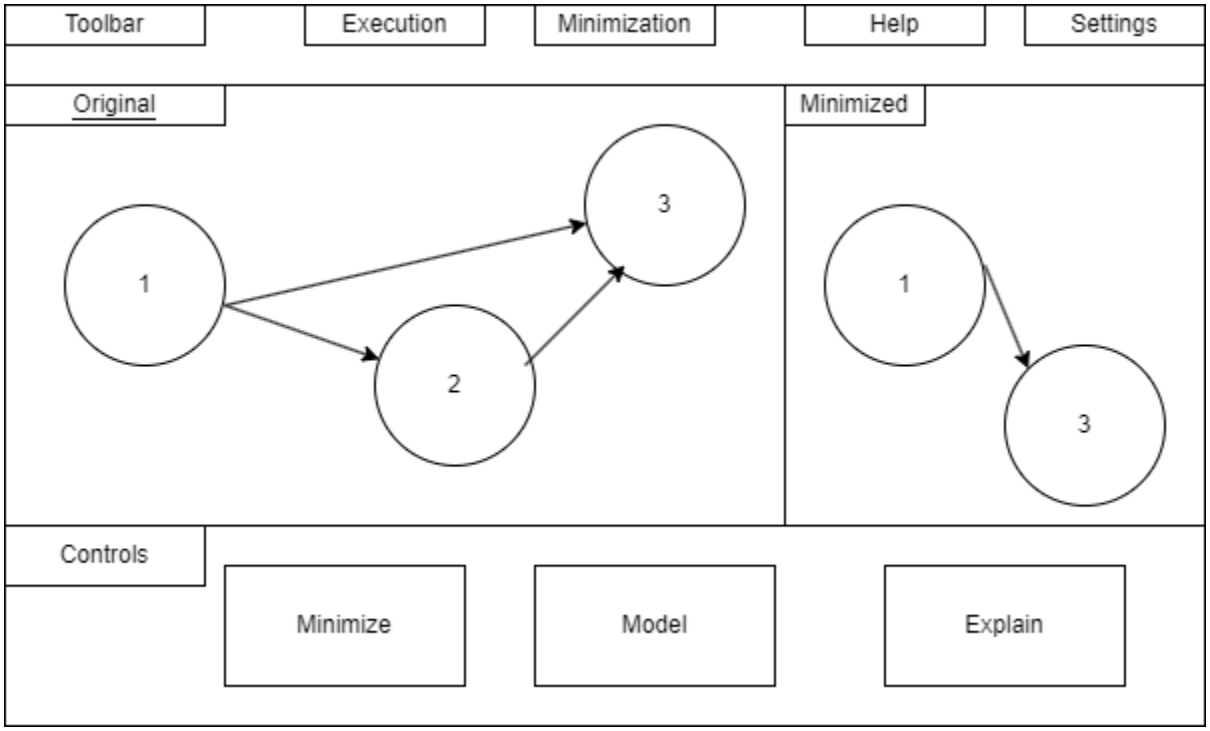
- Main Editor Screen: Canvas with toolbox for states, transitions, and selection; property pane for editing.



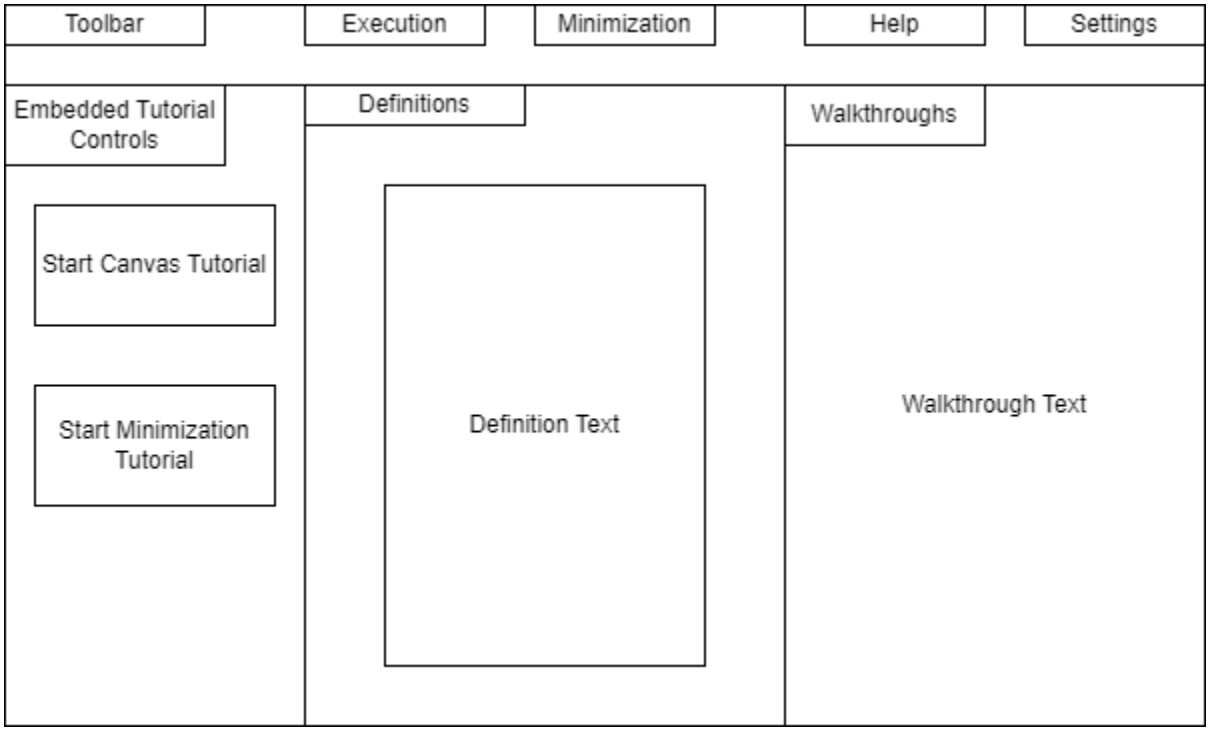
- Execution Screen: Highlight current state as input string is processed step by step.



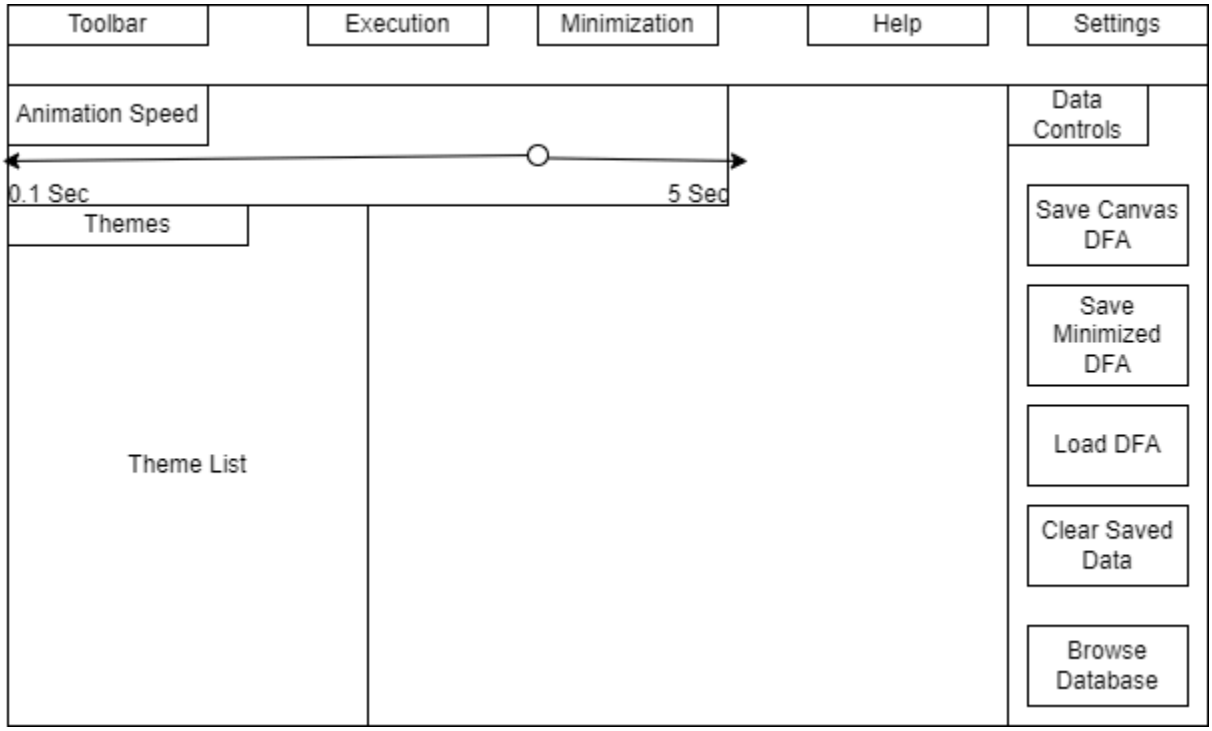
- Minimized DFA View: Side-by-side comparison of original and minimized DFA.



- Help/Documentation Panel: Embedded tutorials, definitions, and walkthroughs.



- Settings Panel: Choose theme, animation speed, and export options.



4.2 Navigation Flow

Left-hand toolbox → Canvas → Simulation controls → Results view.
Tabs or side menu to switch between Editor, Simulation, Minimized View, and Help.

5. Algorithms, Pseudocode & Educational Features

Simulation (simplified pseudocode):

```
function runDFA(dfa, inputString):
    currentState = dfa.initialState
    for symbol in inputString:
        highlight(currentState)
        wait(animationDelay)
        currentState = dfa.transition(currentState, symbol)
    highlight(currentState)
    return currentState in dfa.finalStates
```

Minimization (simplified pseudocode):

```
function minimizeDFA(dfa):
    partition = {finalStates, nonFinalStates}
    repeat:
        newPartition = refine(partition, dfa)
    until newPartition == partition
```

return buildDFA(newPartition)

Educational Component:

- Step-by-step animations show how states are partitioned during minimization.
- Animated tracing highlights transitions on the input string.
- Hovering over a tool shows a pop-up explaining DFA concepts (e.g., “dead state,” “initial state”).
- Optional “learning mode” automatically pauses after each step with a textual explanation.

6. Conclusion

This design integrates modular components to meet functional, performance, and usability requirements for a modern DFA visualizer. The GUI provides a student-friendly toolbox with embedded tutorials and animations. By adhering to IEEE standards and including a built-in educational mode, this design ensures reliability, scalability, maintainability, and effectiveness as a teaching and learning aid for formal languages.