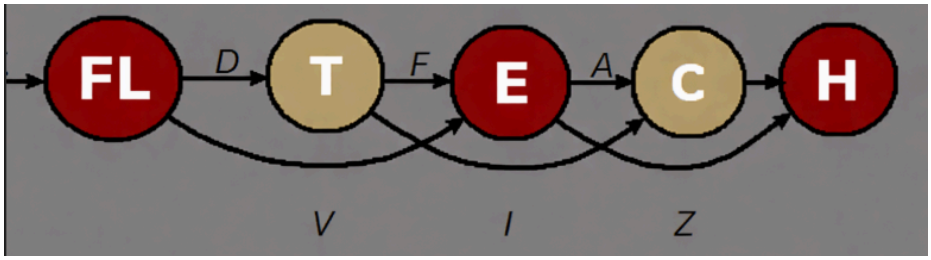


Improved Visualization for Formal Languages

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 & Client:

Dr. David Luginbuhl dluginbuhl@fit.edu

- Florida Institute of Technology, Department of Computer Science -

---

User & Developer Guide

Table of Contents

User Guide

---

1. Introduction

1.1 Purpose of the Manual

This manual provides a comprehensive guide for users of the **DFA Visualizer (newFlap)** program. It explains how to install, navigate, and utilize the application's full set of features, including automata construction, simulation, and educational tools.

The purpose of this program, and the manual at large is to help users:

- Build and analyze DFAs and NFAs
- Understand automata behavior through simulation
- Utilize advanced tools such as regular expression conversion and minimization
- Learn formal language concepts interactively through guided and assessment modes

This manual is intended for:

- Students learning formal languages and automata theory
- Instructors demonstrating DFA/NFA concepts
- Developers or users exploring automata visualization tools

1.2 Overview of the Application

The DFA Visualizer is a standalone desktop application designed to provide a complete environment for working with finite automata.

Key capabilities include:

- **Graph Construction**: Build DFA and NFA graphs interactively
- **Simulation Engine**: Test strings with step-by-step animation
- **Automata Algorithms**:
    - NFA to DFA conversion

    - DFA minimization
- **Automatic Construction**:
    - Build DFA from accepted strings
    - Build DFA from regular expressions
- **Educational Tools**:
    - Tutorial Mode (interactive guidance)
    - Teaching Mode (automated evaluation and feedback)
- **Advanced Features**:
    - Zoomable canvas for large automata
    - Automatic dead state generation
    - Save and load via .newflap files

By combining theoretical correctness with an intuitive graphical interface, the application serves both as a learning tool and a practical automata design system.

Key Features:

- **Graph Editor**: Add states, transitions, start/accept states, undo, redo, and clear
- **Simulation Engine**: Validate strings and visualize execution
- **Conversion Tools**: NFA → DFA conversion
- **Optimization Tools**: DFA minimization
- **Learning Modes**:
    - Tutorial Mode (feature instructions)
    - Teaching Mode (interactive assessment)
- **Auto-Generation**:
    - Build minimal DFAs from accepted strings and Regular Expressions
- **Canvas Repositioning**
    - Zoom in and out of the canvas
- **Intuitive Dead State Additions**
    - Build graph completing dead states with button press

1.3 Document Conventions

- **Bold** text indicates UI items or headers
- Grey Text is code elements
- Numbered lists represent step-by-step procedures

- Bullet points provide feature descriptions or summaries

2. Getting Started

2.1 System Requirements

- Operating System: Windows, macOS, or Linux
- Python 3.8+ (if running from source)
    - OR precompiled `.exe` for Windows users
- Display: Recommended minimum resolution 1366×768

2.2 Installation/Setup Instructions

Option 1: Run from source

1. Download project files from Github Directory
2. Navigate to NewFlap Folder within download
3. Run python -m newFlap

Option 2: Run Executable

1. Download compiled .exe file
2. Double click to run and launch

3. Using System Features

3.1 Feature Overview

The system provides the following major features:

- DFA/NFA graph construction tools
- String simulation with animation
- NFA → DFA conversion
- DFA minimization
- Automatic DFA generation (strings and regex)
- Teaching Mode and Tutorial Mode
- Save/load functionality
- Canvas zoom and navigation
- Automatic dead state generation

3.2 Detailed Feature Walkthrough

3.2.1 Feature 1: DFA/NFA Graph Construction

Users can manually construct automata by:

1. Clicking **Add State** and placing nodes on the canvas
2. Using **Add Transition** to connect states
3. Assigning:
    - A start state
    - Accepting states

The system ensures:

- Clear visual layout
- Reduced transition overlap (improved in final version)

3.2.2 Feature 2: String Simulation & Animation

1. Click **Check String**
2. Enter an input string
3. Use:
    - **Step** to move symbol-by-symbol
    - **Play** for full animation

The system visually:

- Highlights current state
- Shows transition traversal
- Displays acceptance or rejection

3.2.3 Feature 3: Automatic DFA Construction (Strings & Regex)

Build from Strings

- Input accepted strings
- System generates corresponding DFA

Build from Regular Expressions

- Input regex expression
- System constructs DFA using formal algorithms

These features allow rapid automata creation and testing.

3.2.4 Feature 4: Teaching & Tutorial Modes

Tutorial Mode

- Click on interface elements
- Receive contextual explanations
- Explore system interactively

Teaching Mode

1. System generates a random string and alphabet
2. User constructs DFA
3. Submit answer for evaluation

   Feedback includes:

   - Correctness validation
   - Animation of traversal
   - Suggestions for improvement

3.2.5 Feature 5: Save/Load & Dead State Automation

Save/Load

- Save DFA as .newflap file
- Reload and continue editing

Dead State Button

- Automatically creates a dead state
- Ensures DFA completeness
- Connects missing transitions

3.2.6 Feature 6: Zoom & Canvas Navigation

Users can:

- Zoom in/out using GUI buttons
- Use mouse scroll for dynamic zooming
- Reposition automata easily

Features include:

- Cursor-centered zoom
- Scaled rendering of all elements

3.3 Navigation and Interface Tips

- Use zoom to manage large automata
- Space states evenly for clarity
- Use dead state early to avoid errors
- Save work frequently
- Use animation to debug transitions

4. User Specific Examples

4.1 End Users vs. Administrators

Role	Usage
Students	Learning and practicing automata
Instructors	Demonstrating concepts and sharing examples
Developers	Extending system features

4.2 Role-Based Scenarios

4.2.1 Example 1: A Student Learning DFAs

- Opens Tutorial Mode
- Builds DFA step-by-step
- Tests strings
- Gains understanding of transitions

4.2.2 Example 2: A Student Practicing for Exams

- Uses Teaching Mode
- Builds DFA from generated prompt
- Submits for evaluation
- Improves based on feedback

4.2.3 Example 3: A Teacher Assigning Work

- DFA graph in the canvas space
- Downloads it as a .newFlap file

- Sends it to students with a question sheet of strings that may or may not be accepted
- Teaches students to use tool and traverse the DFA

5. Troubleshooting and FAQs

5.1 Common Issues

Issue	Solution
Transitions overlap	Adjust layout or rebuild
DFA rejects valid string	Check missing transitions
Large graph hard to manage	Use zoom feature
Program not launching	Verify Python or use .exe

5.2 FAQ Section

Q: Why does my DFA need a dead state?
A: To ensure completeness for all inputs.

Q: Can I share my DFA?
A: Yes, using .newflap files.

Q: What's the difference between DFA and NFA?
A: DFA is deterministic; NFA allows multiple transitions.

6. Support and Contact Information

6.1 Customer Support

For questions or issues, contact anyone listed on the development team.

6.2 Feedback Mechanisms

Users are encouraged to:

- Report bugs
- Suggest improvements
- Provide usability feedback

Via on-site feedback form.

Developer Guide

---

1. Introduction

1.1 Purpose of the Developer Manual

This manual is intended to guide developers, contributors, and maintainers in understanding, extending, and maintaining the DFA Visualizer (newFlap) application.

It provides detailed documentation on:

- System architecture and design
- Core algorithms and data structures
- GUI and event-handling mechanisms
- Development practices and extension guidelines

The goal of this document is to ensure the system remains maintainable, extensible, and aligned with formal automata theory principles.

1.2 Audience

This document is intended for:

- Software developers contributing to the project
- Computer science students studying automata implementation
- QA testers validating correctness and usability
- Maintainers responsible for feature updates and bug fixes

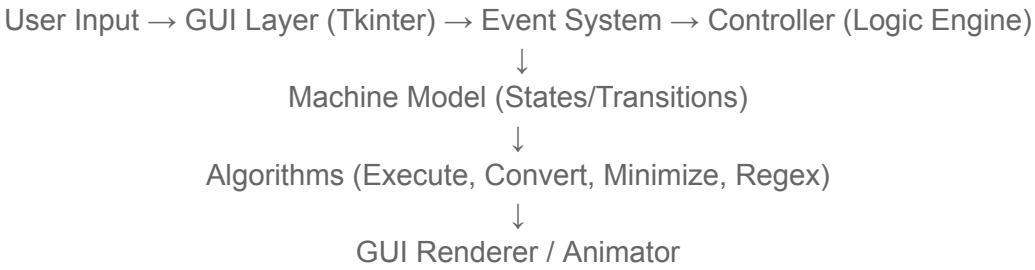
1.3 Document History and Versioning

- Version 1.0 – Initial DFA construction and simulation
- Version 2.0 – Added Teaching Mode and Tutorial Mode
- Version 3.0 – Introduced automatic DFA generation (strings + regex)
- Version 4.0 (Final Release) – Milestone #6:
    - Zoom and canvas scaling
    - Save/load system (.newflap)
    - Dead state automation

- GUI reorganization
- Enhanced teaching mode with animation
- Transition rendering improvements
- Executable packaging (.exe)

2. System Architecture Overview

2.1 Architecture Diagram



2.2 Component Descriptions

2.2.1 Frontend (GUI Layer)

- Built using **Tkinter**
- Responsible for:
  - Canvas rendering of states and transitions
  - User interaction (clicks, drags, input dialogs)
  - Animation controls (step/play)
  - Zooming and canvas scaling

Key modules:

- animator.py → traversal visualization
- tutorial.py → interactive help system
- convert.py / minimize.py → algorithm UI hooks

2.2.2 Backend (Controller + Machine Layer)

The backend is divided into:

Machine Model

- Defines:
  - States (UUID-based)
  - Transitions (multi-symbol, lambda support)

    - Automaton structure

Algorithms

- Execution engine (DFA/NFA simulation)
- NFA → DFA conversion (subset construction)
- DFA minimization
- Regex → DFA compiler (direct construction method)

Teaching Engine

- Generates problems
- Validates user-built DFAs
- Interfaces with animation system

2.2.3 Integration Layer (Event System)

- Centralized event handler (events.py)
- Connects GUI actions to backend logic
- Enables:
    - Loose coupling
    - Easier feature extension
    - Scalable architecture

2.3 Data Flow and Interactions

1. User interacts with GUI (e.g., adds state or runs simulation)
2. Event system captures action
3. Controller processes request
4. Machine model updates state/transition data
5. Algorithm executes if needed
6. GUI re-renders updated automaton

3. Source Code Structure

3.1 Directory Layout

```
newFlap/
 ├── controller/
 │    ├── machine/
 │    │    ├── models.py
 │    │    ├── regex_compiler.py
 │    │    ├── teaching_engine.py
 │    │    └── algorithms/
 │    │         ├── execute.py
 │    │         ├── minimize.py
 │    │         ├── nfa_to_dfa.py
 │    │
 │    └── events.py
 │
 ├── gui/
 │    ├── animator.py
 │    ├── tutorial.py
 │    ├── convert.py
 │    ├── minimize.py
 │    ├── progress.py
 │
 ├── dist/
 │    └── __main__.exe
```

3.2 Module and File Descriptions

3.2.1 Core Files

- models.py → Core data structures for automata
- execute.py → Simulation engine
- minimize.py → DFA minimization
- nfa_to_dfa.py → Conversion logic
- regex_compiler.py → Regex → DFA construction
- teaching_engine.py → Teaching mode logic
- events.py → Event handling system

3.2.2 Configuration Files

- .spec file → PyInstaller configuration
- No external config dependencies required

3.3 Environment Setup

To run from source:

1. Download Newflap folder
2. Navigate to it in downloads
3. Run: **python -m newFlap**

Dependencies:

- Python 3.8+
- Tkinter (built-in)

4. Code Conventions and Guidelines

4.1 Coding Standards

- Follow **PEP 8** guidelines
- Use descriptive variable and function names
- Maintain separation between GUI and logic layers

4.2 Version Control and Branching Strategy

- Use Git for version control
- Recommended workflow:
    - main → stable release
    - feature/* → new features
    - bugfix/* → bug fixes

4.3 Code Commenting and Documentation

- Document all major functions
- Include:
    - Purpose
    - Parameters
    - Return values
- Use inline comments for complex logic (especially algorithms)

5. Detailed Documentation of Methods and Functions

5.1 API Documentation

The DFA Visualizer does not expose a public-facing REST or external API. Instead, it follows a **modular internal API design**, where components communicate through clearly defined Python interfaces.

Internal API Layers

Layer	Responsibility
GUI Layer	Captures user input and renders output
Event System	Routes UI actions to logic
Controller Layer	Executes logic and algorithms
Machine Layer	Maintains automaton state

Example Internal API Flow

```
# GUI event triggers execution
def on_check_string(self):
    input_str = self.input_field.get()
    result = execute_automaton(self.machine, input_str)
    self.animator.animate(result)
```

Here:

- GUI calls controller function
- Controller interacts with machine model
- Result is passed back for visualization

5.2 Class and Function Descriptions

5.2.1 Major Functions

- **Automaton Execution**

  Simulates traversal of a DFA or NFA on a given input string.

```
def execute_automaton(machine, input_string):
    current_states = {machine.start_state}

    for symbol in input_string:
        next_states = set()
        for state in current_states:
            transitions = machine.get_transitions(state, symbol)
            next_states.update(transitions)

        current_states = next_states


    return any(state in machine.accept_states for state in current_states)
```

  Key Characteristics:

  - Supports nondeterminism via state sets
  - Iteratively processes input symbols
  - Returns acceptance result

- **Regex Compilation**

  Implements direct conversion from regex to DFA.

```
def compile_regex(regex):

    postfix = infix_to_postfix(regex)

    syntax_tree = build_syntax_tree(postfix)

    compute_followpos(syntax_tree)
```

return build_dfa_from_tree(syntax_tree)

Key Steps:

1. Insert explicit concatenation
2. Convert to postfix
3. Build AST
4. Compute position sets
5. Generate DFA

- **Minimization**
  - Reduces DFA states using partition refinement.
  - def minimize_dfa(machine):
    - partitions = [machine.accept_states, machine.non_accept_states]
    - while True:
    - new_partitions = refine_partitions(partitions, machine)
    - if new_partitions == partitions:
    - break
    - partitions = new_partitions
    - return build_minimized_dfa(partitions)

- **Teaching Evaluation**
  - Validates correctness of user-created DFAs
  - def evaluate_dfa(machine, test_string):
    - is_deterministic = check_determinism(machine)
    - is_complete = check_completeness(machine)
    - accepts = execute_automaton(machine, test_string)
    - return {
      - "deterministic": is_deterministic, "complete": is_complete, "accepts": accepts
    - }
- **Zoom Scaling**
  - Handles proportional scaling of canvas elements
  - def scale_canvas(canvas, scale_factor, center_x, center_y):
  - for item in canvas.find_all():

canvas.scale(item, center_x, center_y, scale_factor, scale_factor)

Features:

- Scales all objects proportionally
- Maintains relative positioning
- Supports cursor-centered zoom

5.2.2 Methods/Functions

State Creation

```
def create_state(machine, label, x, y):
    state = State(label=label, position=(x, y))
    machine.states.append(state)
    return state
```

State Deletion

```
def delete_state(machine, state):
    machine.states.remove(state)
    machine.transitions = [
        t for t in machine.transitions
        if t.source != state and t.target != state
    ]
```

Transition Creation

```
def add_transition(machine, source, target, symbol):
   transition = Transition(source, target, symbol)
   machine.transitions.append(transition)
```

Transition Validation

```
def is_valid_transition(machine, source, symbol):
    transitions = [t for t in machine.transitions if t.source == source and t.symbol
== symbol]
    return len(transitions) <= 1  # DFA condition
```

Serialization (Save)

```
def serialize_machine(machine):
    return {
        "states": [s.to_dict() for s in machine.states],
        "transitions": [t.to_dict() for t in machine.transitions],
        "start": machine.start_state.id,
        "accept": [s.id for s in machine.accept_states]
    }
```

Deserialization (Load)

```
def load_machine(data):
    machine = Machine()
    # reconstruct states and transitions
    return machine
```

Dead State Generation

```
def add_dead_state(machine):
    dead = create_state(machine, "Dead", 0, 0)

    for state in machine.states:
        for symbol in machine.alphabet:
            if not machine.has_transition(state, symbol):
                add_transition(machine, state, dead, symbol)

    return dead
```

5.3 Utility and Helper Functions

- Transition mapping utilities
    - def get_transition_map(machine):
        - mapping = {}
        - for t in machine.transitions:
            - mapping.setdefault((t.source, t.symbol), []).append(t.target)
        - return mapping
- Graph traversal helpers
    - def reachable_states(machine):
        - visited = set()
        - stack = [machine.start_state]
        - while stack:
            - state = stack.pop()
            - if state not in visited:
                - visited.add(state)
                  stack.extend(machine.get_neighbors(state))
        - return visited
- Validation checks (determinism, completeness)
    - **Determinism**
        - def check_determinism(machine):
        - for state in machine.states:
        - seen = {}
        - for t in machine.transitions:

        - if t.source == state:
        - if t.symbol in seen:
        - return False
        - seen[t.symbol] = True
    - return True
- **Completeness**
    - def check_completeness(machine):
    - for state in machine.states:
    - for symbol in machine.alphabet:
    - if not machine.has_transition(state, symbol):
    - return False
    - return True

6. Development and Debugging Practices

6.1 How to Add New Features

1. Add UI component (button/menu)
2. Register event in event system
3. Implement backend logic
4. Update rendering if necessary
5. Test with sample automata

6.2 Maintenance Guidelines

- Maintain separation:
    - GUI ≠ Logic ≠ Data model
- Avoid modifying core algorithms unnecessarily
- Preserve `.newflap` file compatibility
- Ensure visual clarity (no transition overlap regressions)

6.3 Debugging Procedures

- Use print/log tracing:

print(f"Current state: {state}, symbol: {symbol}")

- Validate:
    - State connectivity
    - Transition correctness
    - Acceptance logic
- Test edge cases:
    - Empty string input
    - Lambda transitions
    - Large automata

6.4 Testing

Testing includes:

- Algorithm correctness:
    - DFA execution
    - Minimization
    - Conversion
- UI testing:

        - Button functionality
        - Canvas rendering
- Teaching Mode:
    - Validation logic
    - Feedback accuracy

Example test:

```
def test_execution():

    machine = build_sample_dfa()

    assert execute_automaton(machine, "abba") == True
```

7. Build, Deployment, and CI/CD Processes

7.1 Build Process

- Built using **PyInstaller**
    - **pyinstaller --onefile --noconsole __main__.py**
- Generates standalone .exe

7.2 Deployment Instructions

- Distribute `.exe` file for end users
- No installation required
- Users run directly

7.3 Continuous Integration/Continuous Deployment (CI/CD)

- Not formally implemented
- Can be extended with GitHub Actions if needed

8. Appendices and Additional Resources

8.1 Glossary of Terms

- **DFA** – Deterministic Finite Automaton
- **NFA** – Nondeterministic Finite Automaton
- **Alphabet** – Set of valid input symbols
- **Dead State** – State that rejects all inputs
- **Regex** – Regular Expression

8.2 External Resources

- Automata Theory textbooks
- Python Tkinter documentation
- Compiler construction references (Aho–Sethi–Ullman)