CLOSURE PROPERTIES SIMULATOR

GROUP 4 MEMBERS:

Kenneth Villabroza

Eduard A. Matubaran

Jim Owen Bognalbal

Hannz Jimdandy Naag

Khynne Mark Elmer Lawan

I. Introduction

Our topic, Closure Properties Simulator, is focused on demonstrating the Kleene star closure property in automata theory. The Kleene star shows that if a language is regular, then its Kleene star is also regular. This means any regular language can be extended to include zero or more repetitions of its strings, and the result will still be regular.

The program takes as input the description of an automaton (states, input alphabet, transition table, start state, and accept states). It then constructively generates a new automaton that represents the Kleene star of the original automaton by adding a new start/accept state and redirecting transitions to enable repetition while accepting the empty string.

This simulator provides both an interactive and constructive (approach, making the Kleene star property more practical and easier to understand.

II. APPROACH

Our program uses both interactive and constructive approaches.

Interactive:

- The program first interacts with the user by asking for the components of the automaton (states, alphabet, start state, accept states, and transitions).
- This makes the program flexible, since any automaton can be provided as input.

Constructive:

- After receiving the input automaton, **A**, the program constructively generates a new automaton **A*** that represents the Kleene star of the language.

Construction steps:

1. Add a new start state S*, which is also an accept state (to allow the empty string).
2. Keep the original accept states as accepting states.
3. Redirect transitions so that:
    - S* behaves like the original start state.
    - Each original accepts state links back to the start behavior, enabling repetition.

III. Theory and Algorithm

The theory behind this topic is the Kleene star closure property in automata theory. The Kleene star of a language L, written as L*, includes all possible strings formed by concatenating zero or more strings from L, including the empty string (ε). This shows that the set of regular languages is closed under the Kleene star operation.

To apply this concept in the program, we designed an algorithm that takes a given automaton **A** and constructs a new automaton **A*** as follows:

1. Add a new start state (S*), which is also an accept state. This ensures that ε (the empty string) is accepted.
2. Preserve the original accept states of the automaton so that all strings accepted by A are also accepted by A*.
3. Redirect transitions:
    - From S* to simulate the behavior of the original start state.
    - From each accept state back to the original start behavior, enabling repetition.

This algorithm was implemented in Python by updating the states, accept states, and transition table of the original automaton. The output automaton represents the Kleene star of the input automaton.

IV. Testing and Evaluation

The program was tested manually by giving different sets of inputs through the console to check if it worked correctly.

First, a basic DFA test was done by entering a simple deterministic finite automaton (DFA) with two states, q0 and q1. The alphabet used was a and b, with q0 as the start state and q1 as the accept state.

The transitions entered were:

q0, a, q1

q1, b, q0

This test was done to check if the program correctly added the new start state S* and created proper looping transitions to follow the rules of the Kleene star operation.

Next, an empty string test was done to make sure that the new start state S* could accept the empty string (ε), which is an important part of the Kleene star process.

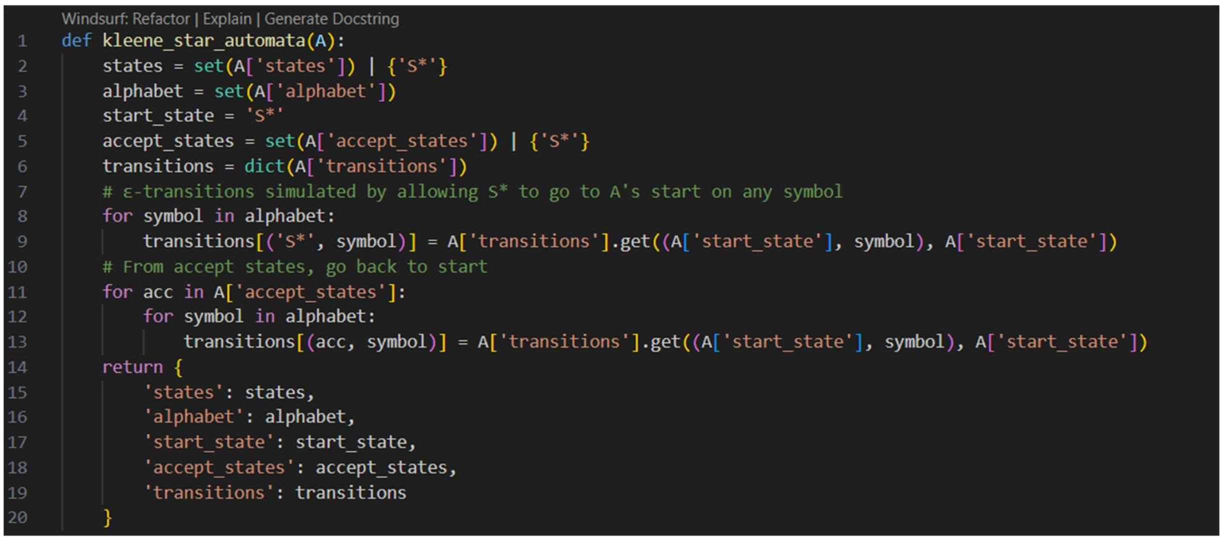
A multiple accept states test was also done by using a DFA with more than one accept state. This test was used to confirm that all accept states correctly looped back to the start state so the automaton could repeat the process as expected.

In addition, invalid input testing was done by entering transitions with errors, such as missing commas, to see how the program handled mistakes or unexpected inputs.

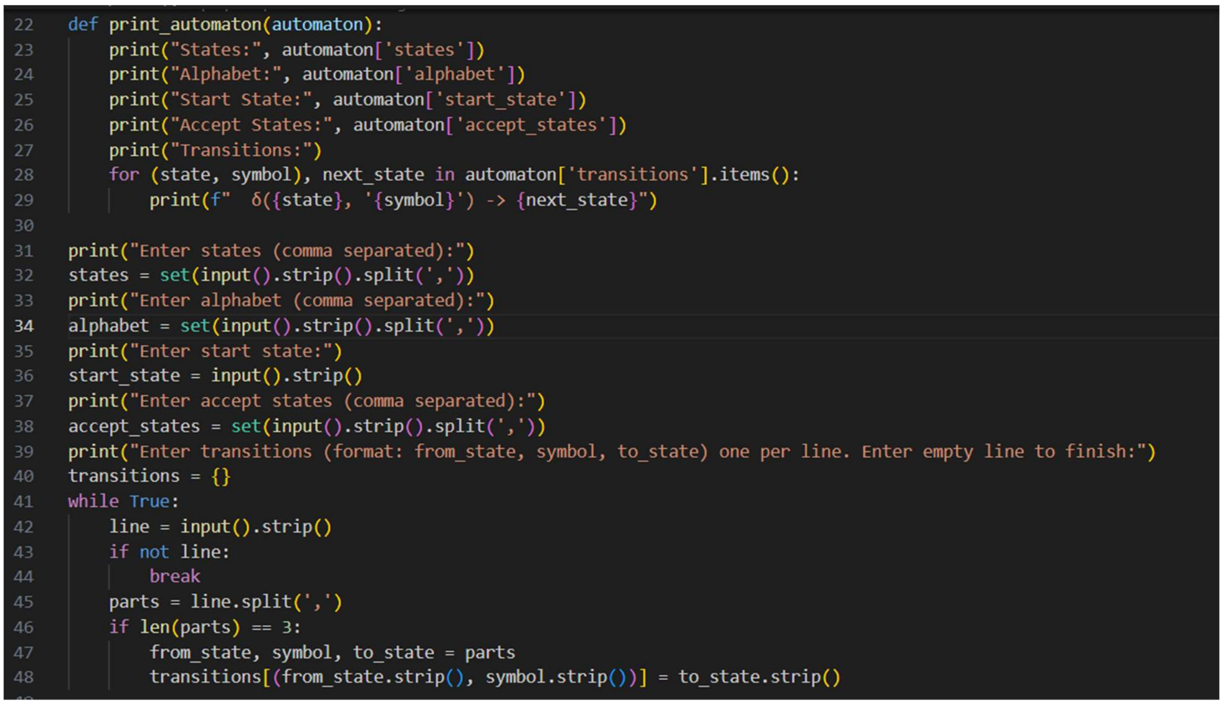
Finally, the output of the program was checked using the print_automaton() function. The results, including the states, alphabet, transitions, and accept states, were compared to the expected behavior of a Kleene star automaton.

From these tests, it was confirmed that the program worked well for simple DFA inputs and was able to build a correct Kleene star automaton.

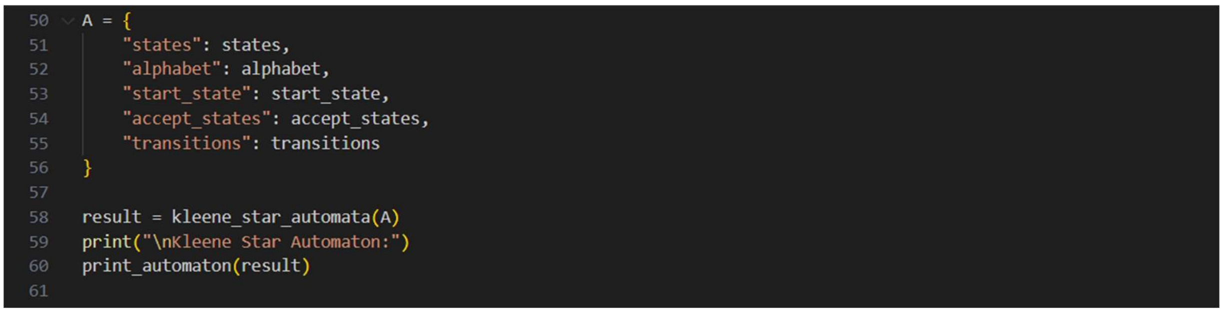
V. Source Code



```python
def kleene_star_automata(A):
    states = set(A['states']) | {'S*'}
    alphabet = set(A['alphabet'])
    start_state = 'S*'
    accept_states = set(A['accept_states']) | {'S*'}
    transitions = dict(A['transitions'])
    # ε-transitions simulated by allowing S* to go to A's start on any symbol
    for symbol in alphabet:
        transitions[('S*', symbol)] = A['transitions'].get((A['start_state'], symbol), A['start_state'])
    # From accept states, go back to start
    for acc in A['accept_states']:
        for symbol in alphabet:
            transitions[(acc, symbol)] = A['transitions'].get((A['start_state'], symbol), A['start_state'])
    return {
        'states': states,
        'alphabet': alphabet,
        'start_state': start_state,
        'accept_states': accept_states,
        'transitions': transitions
    }
```

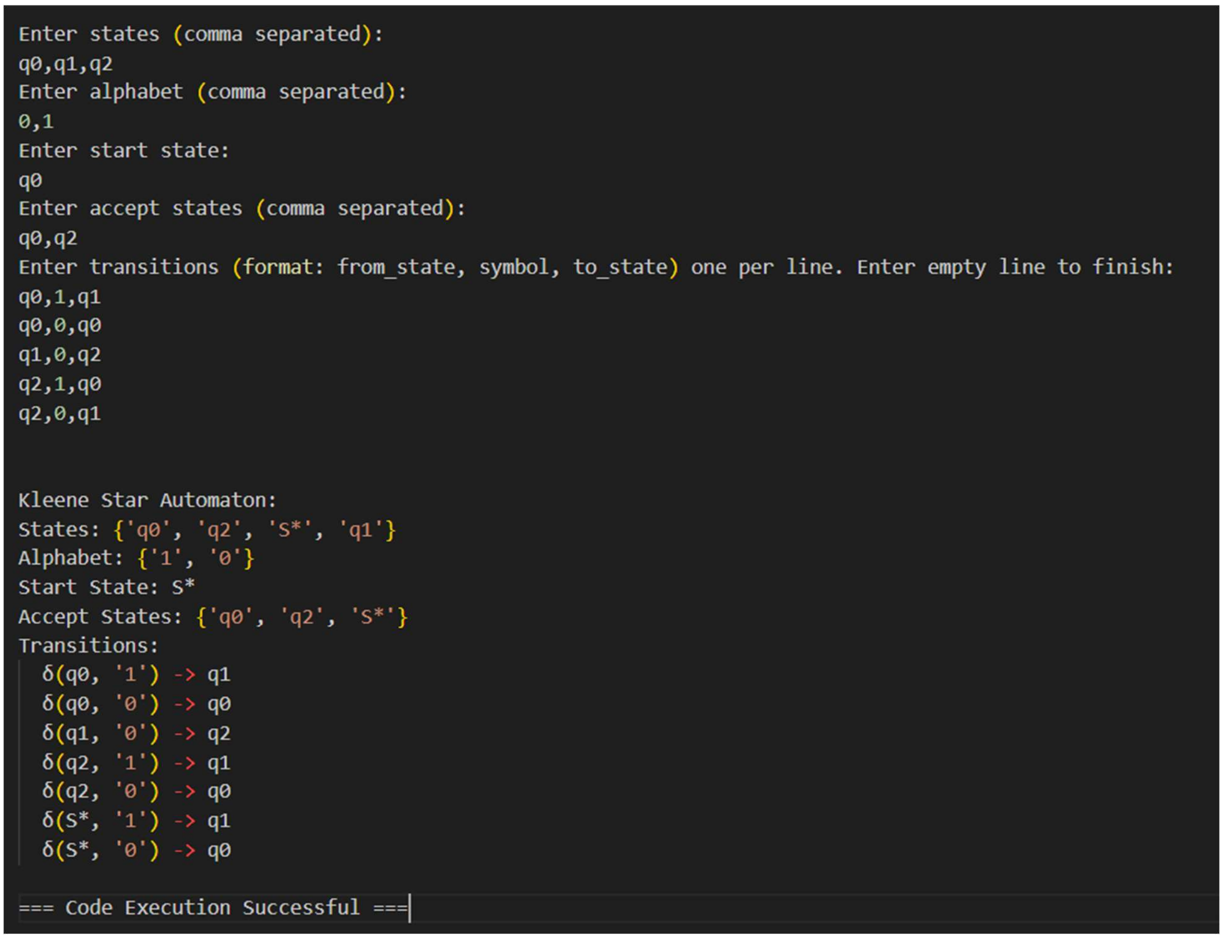


```python
def print_automaton(automaton):
    print("States:", automaton['states'])
    print("Alphabet:", automaton['alphabet'])
    print("Start State:", automaton['start_state'])
    print("Accept States:", automaton['accept_states'])
    print("Transitions:")
    for (state, symbol), next_state in automaton['transitions'].items():
        print(f"  δ({state}, '{symbol}') -> {next_state}")

print("Enter states (comma separated):")
states = set(input().strip().split(','))
print("Enter alphabet (comma separated):")
alphabet = set(input().strip().split(','))
print("Enter start state:")
start_state = input().strip()
print("Enter accept states (comma separated):")
accept_states = set(input().strip().split(','))
print("Enter transitions (format: from_state, symbol, to_state) one per line. Enter empty line to finish:")
transitions = {}
while True:
    line = input().strip()
    if not line:
        break
    parts = line.split(',')
    if len(parts) == 3:
        from_state, symbol, to_state = parts
        transitions[(from_state.strip(), symbol.strip())] = to_state.strip()
```



```python
A = {
    "states": states,
    "alphabet": alphabet,
    "start_state": start_state,
    "accept_states": accept_states,
    "transitions": transitions
}

result = kleene_star_automata(A)
print("\nKleene Star Automaton:")
print_automaton(result)
```

VI. Output



```
Enter states (comma separated):
q0,q1,q2
Enter alphabet (comma separated):
0,1
Enter start state:
q0
Enter accept states (comma separated):
q0,q2
Enter transitions (format: from_state, symbol, to_state) one per line. Enter empty line to finish:
q0,1,q1
q0,0,q0
q1,0,q2
q2,1,q0
q2,0,q1


Kleene Star Automaton:
States: {'q0', 'q2', 'S*', 'q1'}
Alphabet: {'1', '0'}
Start State: S*
Accept States: {'q0', 'q2', 'S*'}
Transitions:
  δ(q0, '1') -> q1
  δ(q0, '0') -> q0
  δ(q1, '0') -> q2
  δ(q2, '1') -> q1
  δ(q2, '0') -> q0
  δ(S*, '1') -> q1
  δ(S*, '0') -> q0

=== Code Execution Successful ===
```

VII. AI Assisted Part

In this topic, we used AI tools to help us with part of the coding. The AI specifically assisted in writing the logic for the transition redirection in the Kleene star automaton. The suggested code was:

```
    # ε-transitions simulated by allowing S* to go to A's start on any symbol
    for symbol in alphabet:
        transitions[('S*', symbol)] = A['transitions'].get((A['start_state'], symbol),
A['start_state'])
    # From accept states, go back to start
    for acc in A['accept_states']:
        for symbol in alphabet:
            transitions[(acc, symbol)] = A['transitions'].get((A['start_state'],
symbol), A['start_state'])
```

This part was important because it made sure that:

1. The new start state S* connects back to the original start state.
2. The accept states loop back to the start so repetition is possible, which follows the Kleene star property.

To improve and verify this AI-assisted code, we tested the program with different input DFAs, including a simple DFA, an empty string case, and multiple accept states. We checked the printed automaton results against the expected behavior of a Kleene star automaton. Through this process, we confirmed that the AI suggestion was correct and fully understood how it worked.

References:

https://www.tutorialspoint.com/automata_theory/kleene_closure_in_automata_theory.htm

https://www.geeksforgeeks.org/theory-of-computation/kleenes-theorem-in-toc-part-1/

https://web.stanford.edu/class/archive/cs/cs103/cs103.1204/lectures/16-DFA-NFA/Part%20III.pdf