Documentation

https://github.com/BVlad917/flcd_labs

Write a program that:

1. Reads the elements of a FA (from file).

2. Displays its elements, using a menu: the set of states, the alphabet, all the transitions, the initial state and the set of final states.

3. For a DFA, verifies if a sequence is accepted by the FA.

Deliverables:

1. FA.in - input file (*on Github*)
2. Source code (*on Github*)
3. Documentation. It should also include in BNF or EBNF format the form in which the FA.in file should be written (*on Moodle and Github*)

Max grade = 9

Max grade = 10: Use FA to detect tokens <identifier> and <integer constant> in the scanner program

Analysis: The finite automata will be read from an input file. It will support two main operations: we can verify if the finite automata is a DFA or not and we can verify if a given sequence is accepted by the finite automata. Internally, the finite automata will store all relevant information, such as: the states, the alphabet, the transitions, and the initial and final states. Finally, the finite automata will be used in the Scanner to check if a given sequence is a valid integer or a valid identifier given our language specification.

Implementation:

letter ::= “a” | “b” | “c” | … | “z” | “A” | “B” | “C” | … | “Z”

digit ::= “0” | “1” | “2” | … | “9”

state ::= letter{letter|digit}

accepted ::= letter|digit|”_”|”+”|”-”|”#”

transition ::= “(”state”,”accepted”,”state”)”

states ::= “Q=”state{“ ”state}

alphabet ::= “S=accepted{“ ”accepted}”

transitions ::= “T=”transition{“ ”transition}

initial_state ::= “IS=”state

final_states ::= “FS=”state{“ ”state}

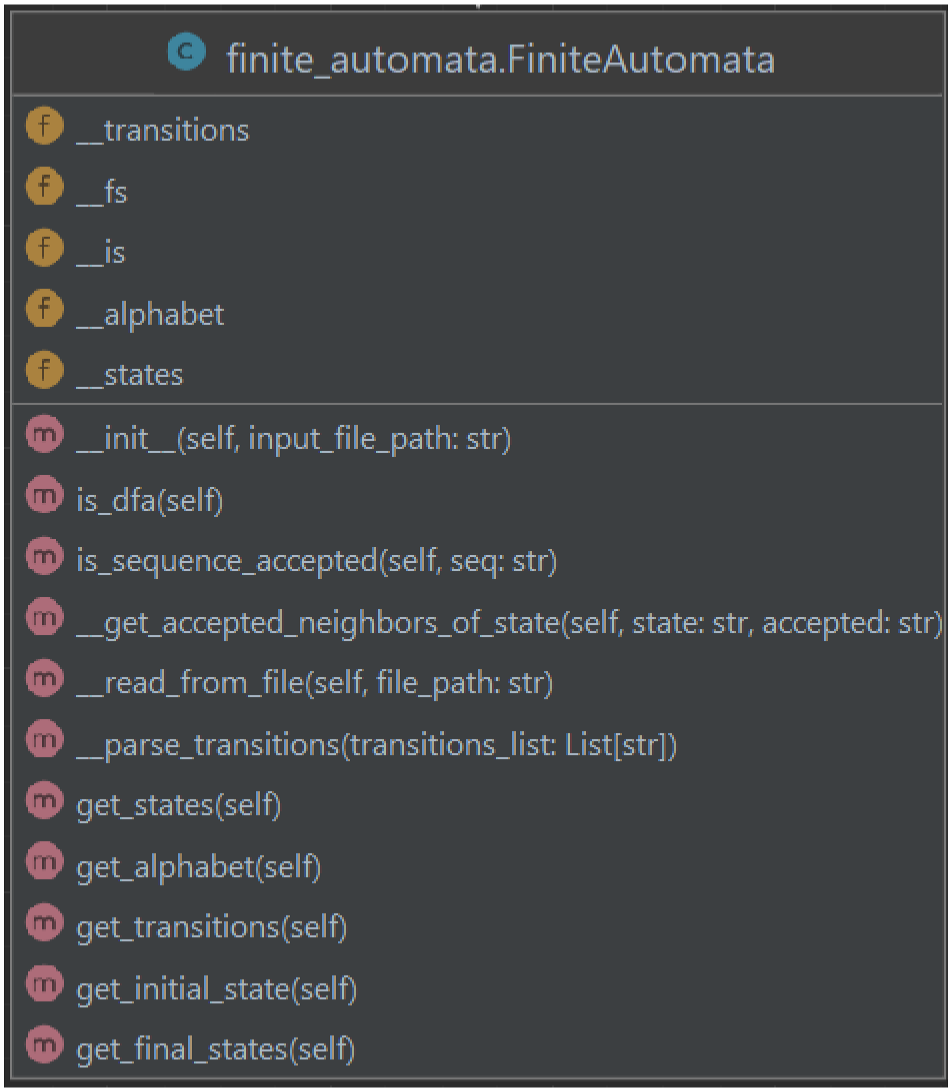
comment_line ::= “#”{digit|letter|” ”|”^” |”(”|”)”|”-”|”?”|”[”|”]”|”+”|”-”|”*”|”$”|”’”|”|”}

input_file ::= {comment_line} ”\n” states “\n” alphabet “\n” transitions “\n” initial_state “\n” final_states

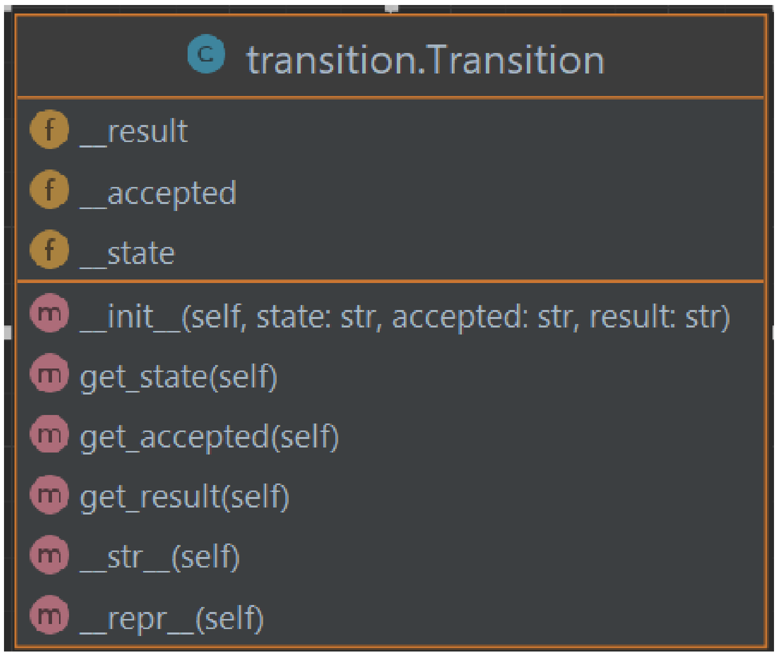
The input file is read and parsed when the finite automata is initialized (in the constructor, so it needs a file path as input). The two main methods the FA implements are: verify if the FA is a DFA or not, and verify if a given sequence is accepted by the FA.

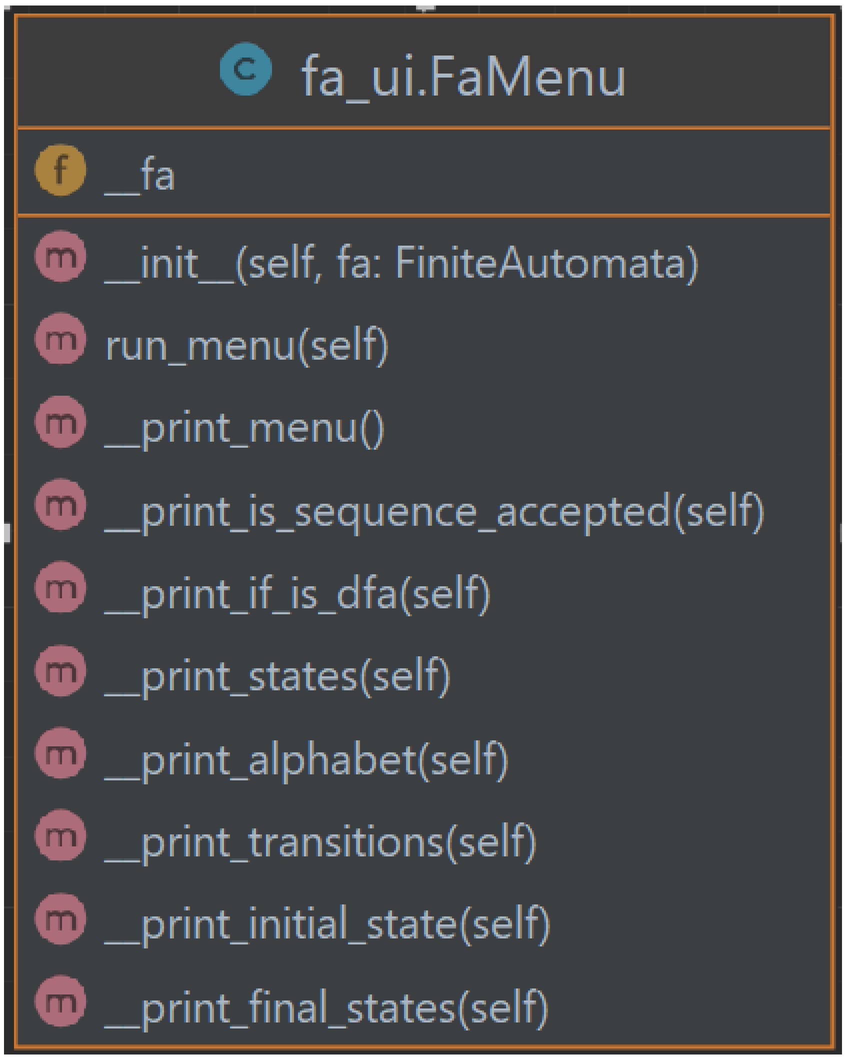
To check if the FA is a DFA or not, we create a dictionary where the keys are “(state, accepted)” pairs, and the values are “result” (i.e., the key is “(from, accepted)” and the value is “to”). We group this dictionary by key (i.e., put in a list all “results” which have the same key), and we check that all the values in this grouped dictionary (where the values are now lists) have length of at most 1.

To verify if a given sequence is accepted by the FA, we use a modified DFS algorithm in which we push on a stack pairs where the first element is the current state and the second element is the currently remaining sequence. Each time we pop an element from the stack we take the state’s neighbors (which can be accessed using the first element in the currently remaining sequence) and we push that neighbor state along with the currently remaining sequence WITHOUT the first character. If we get to a point where the currently remaining sequence is empty AND the current state is a final state, then the sequence is accepted by the FA. If we get to the end of the DFS algorithm and the condition was never satisfied, then the sequence is not accepted by the FA.



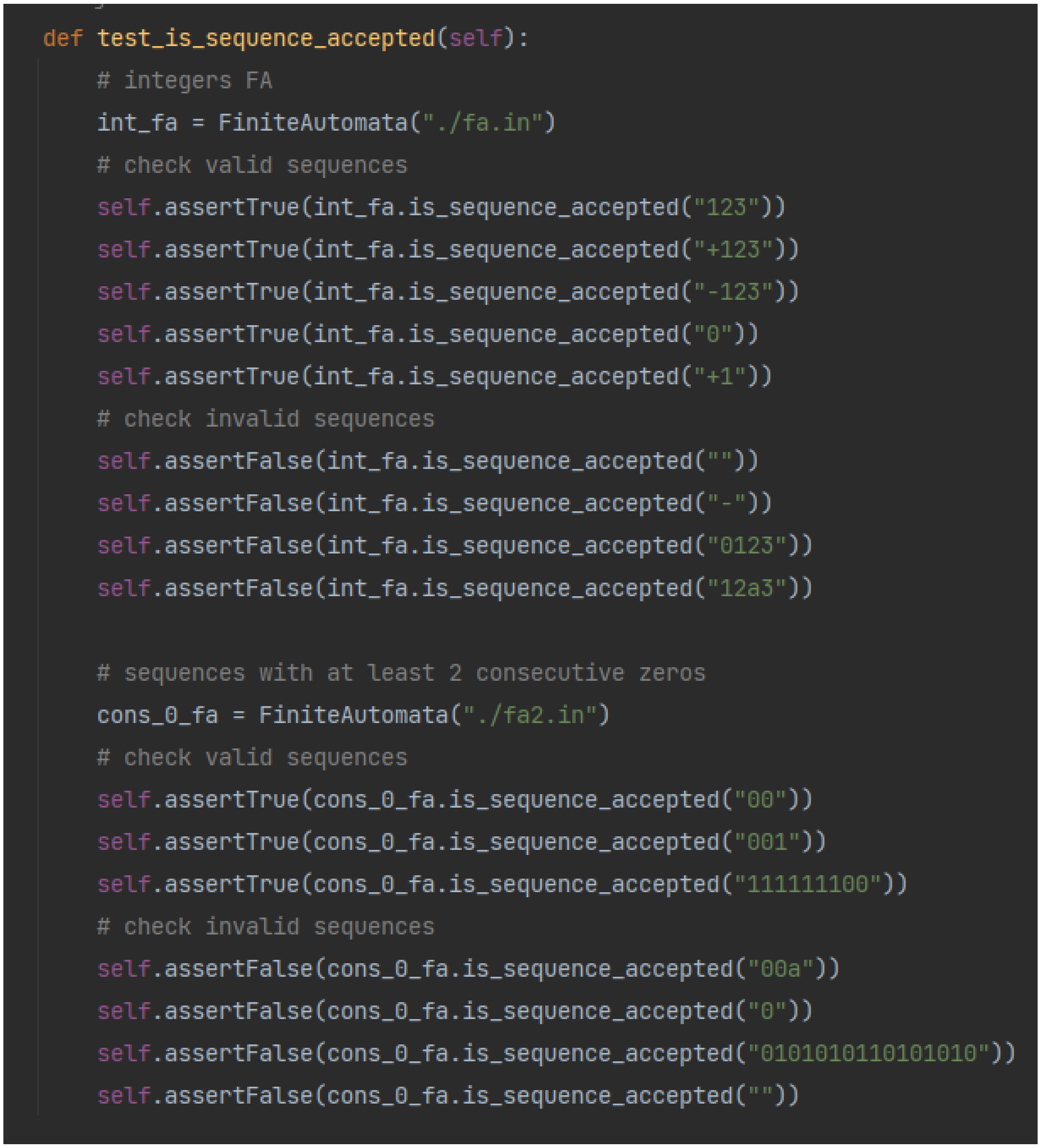
```
finite_automata.FiniteAutomata
__transitions
__fs
__is
__alphabet
__states
__init__(self, input_file_path: str)
is_dfa(self)
is_sequence_accepted(self, seq: str)
__get_accepted_neighbors_of_state(self, state: str, accepted: str)
__read_from_file(self, file_path: str)
__parse_transitions(transitions_list: List[str])
get_states(self)
get_alphabet(self)
get_transitions(self)
get_initial_state(self)
get_final_states(self)
```


```
transition.Transition
__result
__accepted
__state
__init__(self, state: str, accepted: str, result: str)
get_state(self)
get_accepted(self)
get_result(self)
__str__(self)
__repr__(self)
```


```
fa_ui.FaMenu
__fa
__init__(self, fa: FiniteAutomata)
run_menu(self)
__print_menu()
__print_is_sequence_accepted(self)
__print_if_is_dfa(self)
__print_states(self)
__print_alphabet(self)
__print_transitions(self)
__print_initial_state(self)
__print_final_states(self)
```

Testing: We created tests for the two main methods of the Finite Automata program (checking if the FA is a DFA or not, and checking if a given sequence is accepted by the FA)

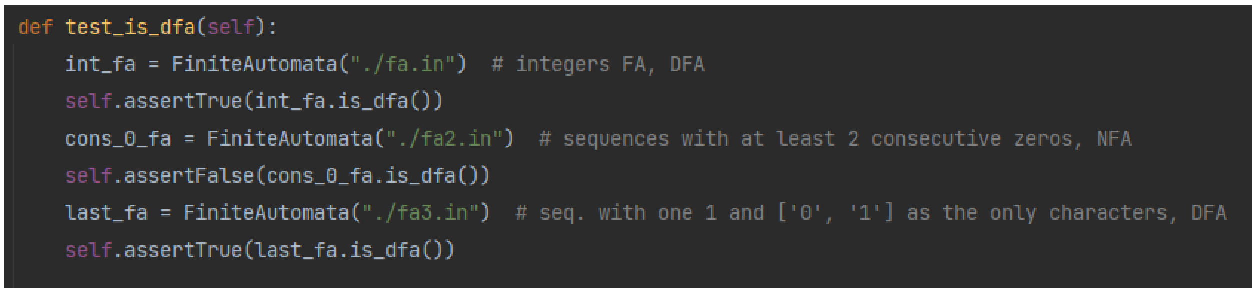
Tests for checking if a given sequence is accepted:



```python
def test_is_sequence_accepted(self):
    # integers FA
    int_fa = FiniteAutomata("./fa.in")
    # check valid sequences
    self.assertTrue(int_fa.is_sequence_accepted("123"))
    self.assertTrue(int_fa.is_sequence_accepted("+123"))
    self.assertTrue(int_fa.is_sequence_accepted("-123"))
    self.assertTrue(int_fa.is_sequence_accepted("0"))
    self.assertTrue(int_fa.is_sequence_accepted("+1"))
    # check invalid sequences
    self.assertFalse(int_fa.is_sequence_accepted(""))
    self.assertFalse(int_fa.is_sequence_accepted("-"))
    self.assertFalse(int_fa.is_sequence_accepted("0123"))
    self.assertFalse(int_fa.is_sequence_accepted("12a3"))

    # sequences with at least 2 consecutive zeros
    cons_0_fa = FiniteAutomata("./fa2.in")
    # check valid sequences
    self.assertTrue(cons_0_fa.is_sequence_accepted("00"))
    self.assertTrue(cons_0_fa.is_sequence_accepted("001"))
    self.assertTrue(cons_0_fa.is_sequence_accepted("111111100"))
    # check invalid sequences
    self.assertFalse(cons_0_fa.is_sequence_accepted("00a"))
    self.assertFalse(cons_0_fa.is_sequence_accepted("0"))
    self.assertFalse(cons_0_fa.is_sequence_accepted("0101010110101010"))
    self.assertFalse(cons_0_fa.is_sequence_accepted(""))
```

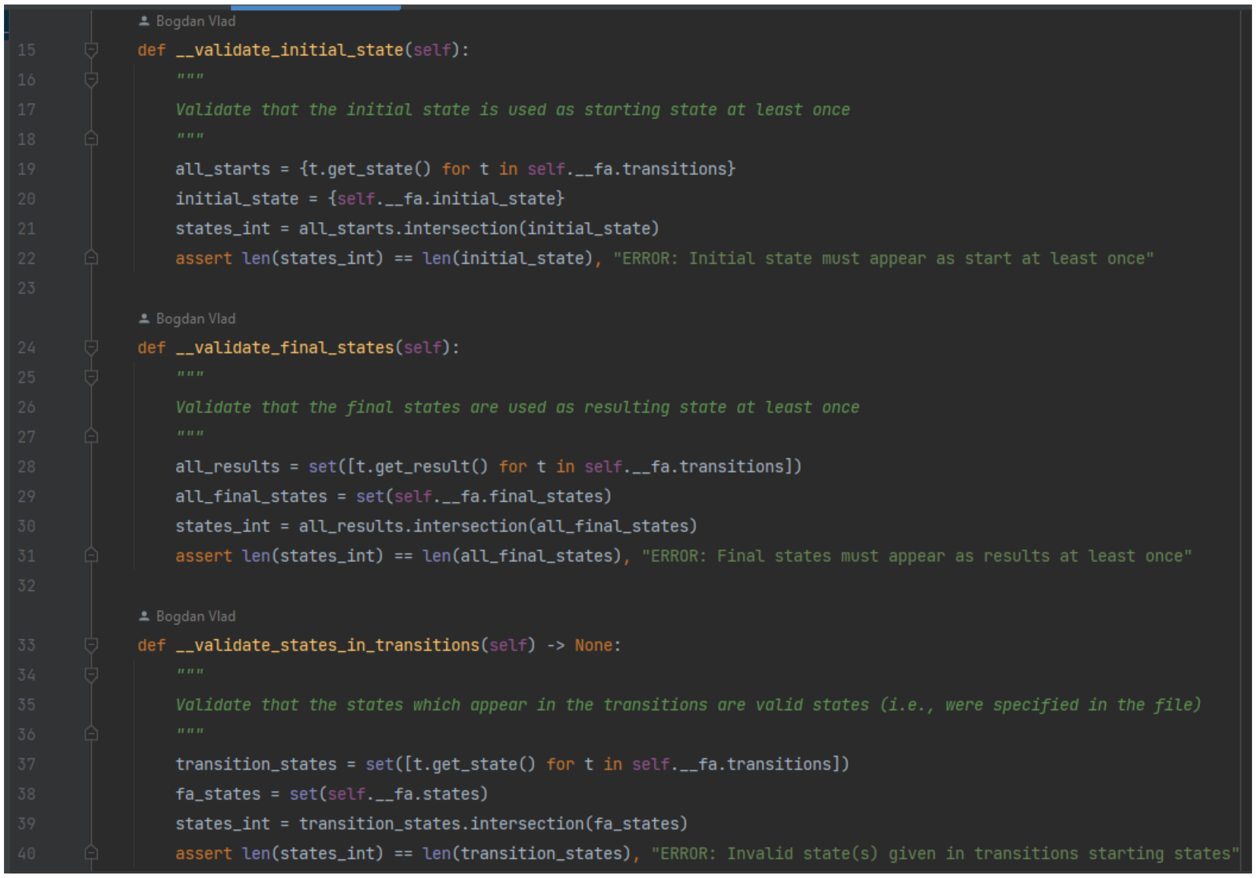
Tests for checking if the FA is a DFA or not:



```
def test_is_dfa(self):
    int_fa = FiniteAutomata("./fa.in")  # integers FA, DFA
    self.assertTrue(int_fa.is_dfa())
    cons_0_fa = FiniteAutomata("./fa2.in")  # sequences with at least 2 consecutive zeros, NFA
    self.assertFalse(cons_0_fa.is_dfa())
    last_fa = FiniteAutomata("./fa3.in")  # seq. with one 1 and ['0', '1'] as the only characters, DFA
    self.assertTrue(last_fa.is_dfa())
```

Validation: We added the following validations, which are run at the beginning (just after the FA is read from a file):

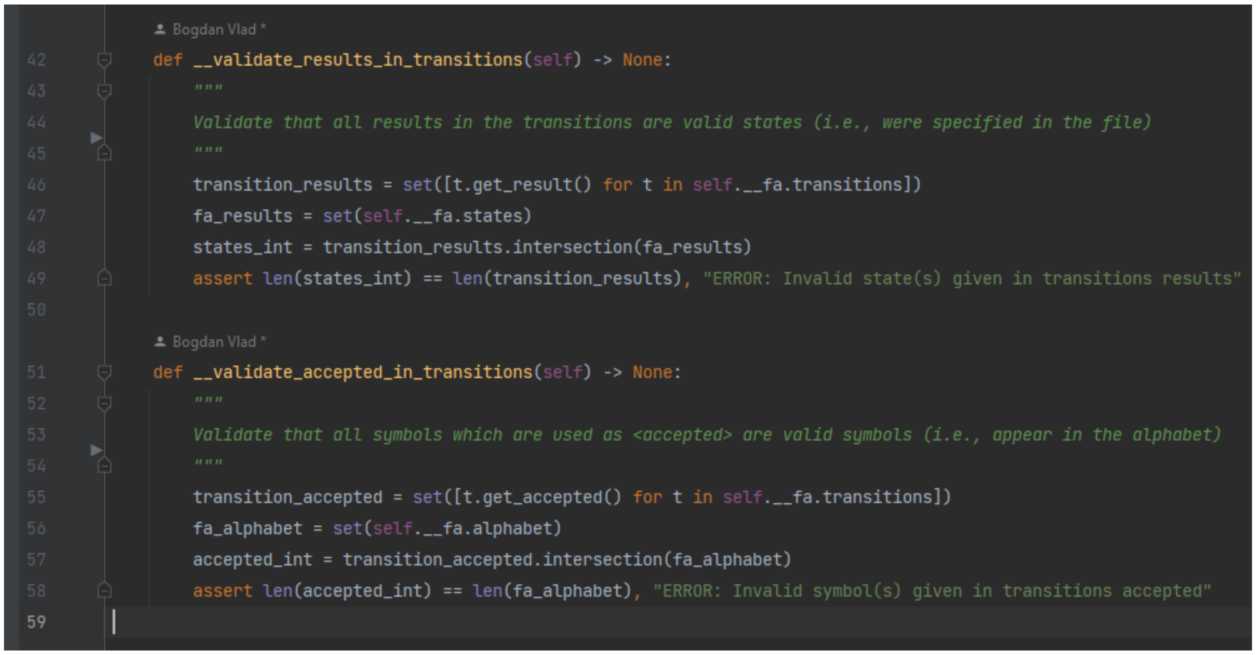
1. Validate that the initial state is used as starting state at least once
2. Validate that the final states are used as resulting state at least once
3. Validate that the states which appear in the transitions are valid states (i.e., were specified in the file)
4. Validate that all results in the transitions are valid states (i.e., were specified in the file)
5. Validate that all symbols which are used as <accepted> are valid symbols (i.e., appear in the alphabet)



```
    Bogdan Vlad
    def __validate_initial_state(self):
        """
        Validate that the initial state is used as starting state at least once
        """
        all_starts = {t.get_state() for t in self.__fa.transitions}
        initial_state = {self.__fa.initial_state}
        states_int = all_starts.intersection(initial_state)
        assert len(states_int) == len(initial_state), "ERROR: Initial state must appear as start at least once"

    Bogdan Vlad
    def __validate_final_states(self):
        """
        Validate that the final states are used as resulting state at least once
        """
        all_results = set([t.get_result() for t in self.__fa.transitions])
        all_final_states = set(self.__fa.final_states)
        states_int = all_results.intersection(all_final_states)
        assert len(states_int) == len(all_final_states), "ERROR: Final states must appear as results at least once"

    Bogdan Vlad
    def __validate_states_in_transitions(self) -> None:
        """
        Validate that the states which appear in the transitions are valid states (i.e., were specified in the file)
        """
        transition_states = set([t.get_state() for t in self.__fa.transitions])
        fa_states = set(self.__fa.states)
        states_int = transition_states.intersection(fa_states)
        assert len(states_int) == len(transition_states), "ERROR: Invalid state(s) given in transitions starting states"
```



```
    Bogdan Vlad *
    def __validate_results_in_transitions(self) -> None:
        """
        Validate that all results in the transitions are valid states (i.e., were specified in the file)
        """
        transition_results = set([t.get_result() for t in self.__fa.transitions])
        fa_results = set(self.__fa.states)
        states_int = transition_results.intersection(fa_results)
        assert len(states_int) == len(transition_results), "ERROR: Invalid state(s) given in transitions results"

    Bogdan Vlad *
    def __validate_accepted_in_transitions(self) -> None:
        """
        Validate that all symbols which are used as <accepted> are valid symbols (i.e., appear in the alphabet)
        """
        transition_accepted = set([t.get_accepted() for t in self.__fa.transitions])
        fa_alphabet = set(self.__fa.alphabet)
        accepted_int = transition_accepted.intersection(fa_alphabet)
        assert len(accepted_int) == len(fa_alphabet), "ERROR: Invalid symbol(s) given in transitions accepted"
```