Fontys University of Applied Science

ALE2 REPORT

Use Your Machine Automata Application

Long Nguyen
3621340

Table of Contents

Chapter 1. Test Plan 2
Plan 2
Result 2
Chapter 2. Application Overview 2
Chapter 3. Design 2
Design 2
Automaton 2
Parser 3
Converting NFA to DFA 4
Regular Expression 4
Detailed Algorithms 4
FSA graph traversal 4
Process Blocked by Adding Pushdown Automaton 5
Plan 5
Why did this happened? 5

Chapter 1. Test Plan

Plan

Only integration tests are done, where major functions are tested:

- Regular expression
  - Test with regexes associated with finite and infinite language.
- Automaton
  - Tests will run using input files

In above automated tests, all public features (functions, except graph generation, file generation) are asserted.

Result

Unit tests are not made, due to lack of time to go into details. Also, there was a big change in design, which was inconvenient for unit test.

Test coverage report could not be generated due to my lack of skill in the test library/tool.

Chapter 2. Application Overview

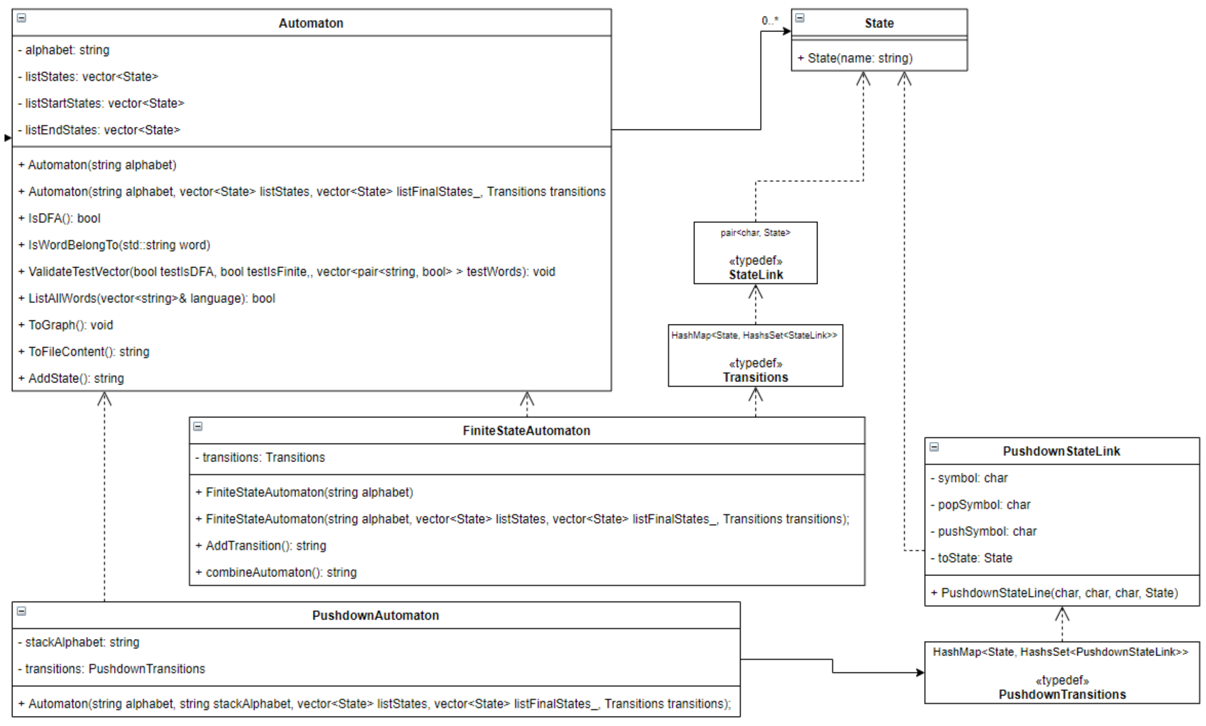
Majors features:

- Read an input regular expression, then show the NFA constructed from the input, and the DFA converted from the previous NFA. **(Big feature 1)**
- Read the automaton (FSA or PDA) from an input file, show the automaton, check the tests in the file and show errors if the test does not match the automaton **(Big feature 2)**. What are checked:
  - If the automaton is a DFA.
  - If the automaton accept a finite language (if the automaton is not a PDA).
  - If the word belongs to the automaton.

Chapter 3. Design

Design

Automaton


Automaton
- alphabet: string
- listStates: vector<State>
- listStartStates: vector<State>
- listEndStates: vector<State>
+ Automaton(string alphabet)
+ Automaton(string alphabet, vector<State> listStates, vector<State> listFinalStates_, Transitions transitions
+ IsDFA(): bool
+ IsWordBelongTo(std::string word)
+ ValidateTestVector(bool testIsDFA, bool testIsFinite,, vector<pair<string, bool> > testWords): void
+ ListAllWords(vector<string>& language): bool
+ ToGraph(): void
+ ToFileContent(): string
+ AddState(): string
0..*
State
+ State(name: string)
pair<char, State>
«typedef»
StateLink
HashMap<State, HashsSet<StateLink>>
«typedef»
Transitions
FiniteStateAutomaton
- transitions: Transitions
+ FiniteStateAutomaton(string alphabet)
+ FiniteStateAutomaton(string alphabet, vector<State> listStates, vector<State> listFinalStates_, Transitions transitions);
+ AddTransition(): string
+ combineAutomaton(): string
PushdownStateLink
- symbol: char
- popSymbol: char
- pushSymbol: char
- toState: State
+ PushdownStateLine(char, char, char, State)
PushdownAutomaton
- stackAlphabet: string
- transitions: PushdownTransitions
+ Automaton(string alphabet, string stackAlphabet, vector<State> listStates, vector<State> listFinalStates_, Transitions transitions);
HashMap<State, HashsSet<PushdownStateLink>>
«typedef»
PushdownTransitions

Why class State does not stores isStart, isFinal? Because start/final states of different automatons are required many times and are extracted from the hash table frequently, hence start/final states should not be stored in class State. Why?

- O(n) to extract
- Lengthy code (for loop, check if,…)
- Have to sync between listFinalStates with State.isFinal.

Why not let PushdownAutomaton be the parent of FiniteStateAutomaton, the algorithms can be generalized to work with PushdownAutomaton and can work with FSA?

- FSA is more specific, and can have its better, simpler algorithms.
- The code would be too big to maintain if it is generalized.
- I think this can be done, but I have not had much thoughts on it.

Why not leave a general Transition field in the Automaton?

- I use Adjacent List, extending the Transition (for example, the stack in PDA) makes it more laborous and increase the complexity in using the Transition. (That's why all information is stored in PushdownStateLink)

Why listStates is a list instead of a HashSet?

- The input size is small, O(logN) lookup HashSet does not give much improvement over O(N) list where the number of states is small (below 1000).
- They are subclasses of C++ Cointainer, and I used the list in a general way, hence are easily interchanged, and would be done if time allows.

Automaton presentation

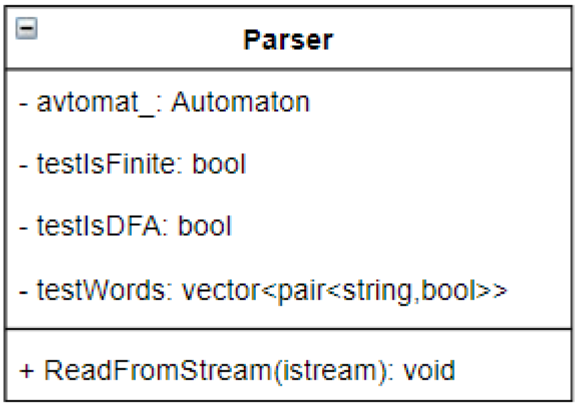
The automaton is presented as a graph with |listStates| vertices, with start and end states.

Edges are presented as an adjacent list: HashMap<State, HashSet<StateLink>>. It is a map where each states store a set of outgoing edges (StateLink). In the StateLink, edge information is store (transition symbol, push/pop stack symbols, …). Adjacent list provides O(|V| + |E|) graph traversal, V and E are the sets of vertices and edges, respectively.

Parser

The parser receives a stream (file or string input), and returns the automaton.

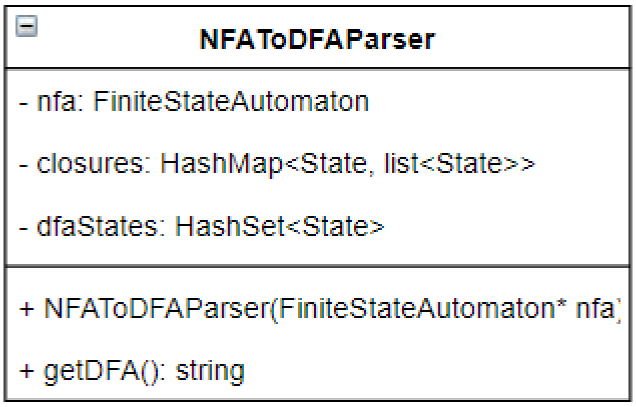
Fields with test prefix are test vector parsed from file


Parser
- avtomat_: Automaton
- testIsFinite: bool
- testIsDFA: bool
- testWords: vector<pair<string,bool>>
+ ReadFromStream(istream): void

Why automaton parser does not belong to class automaton?:

- Also parse test vectors (isDfa, isFinite, check if words belong to automaton) -> those test vectors do not belong to the automaton.

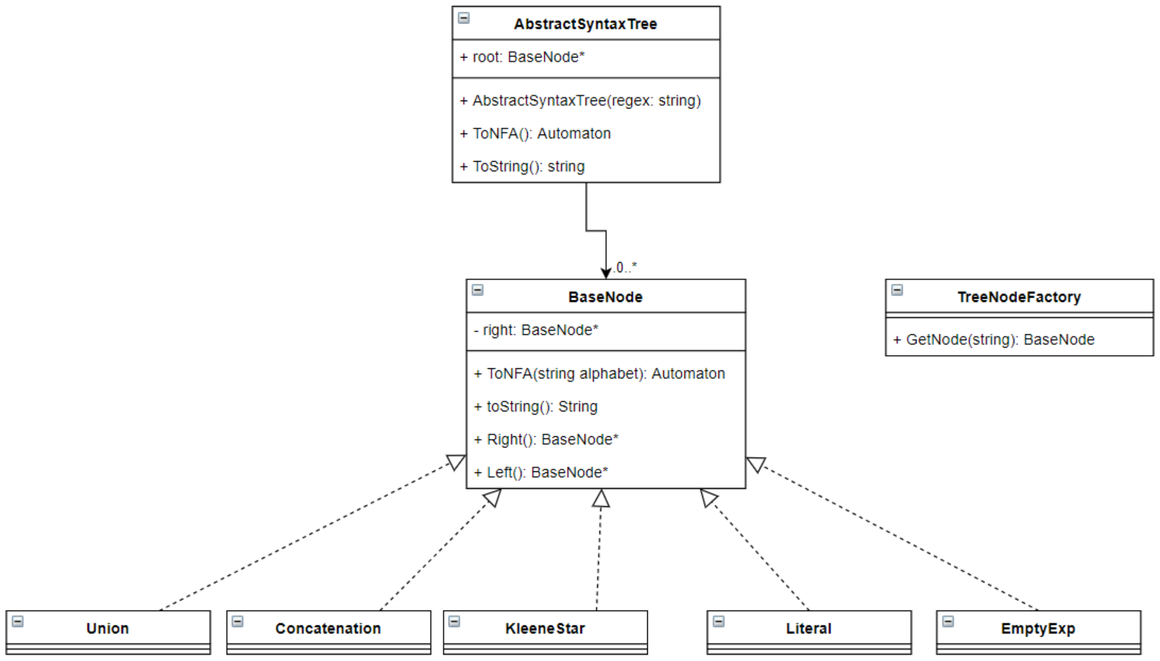
Converting NFA to DFA


NFAToDFAParser
- nfa: FiniteStateAutomaton
- closures: HashMap<State, list<State>>
- dfaStates: HashSet<State>
+ NFAToDFAParser(FiniteStateAutomaton* nfa)
+ getDFA(): string

Why not convert to DFA inside NFA class?

- The conversion requires algorithms and data structures, which are not of interest of NFA class. They should not be stored in the NFA whole lifetime.

Regular Expression



The regex string is parsed inside the AST constructor.

Why this was not the case with automaton, but is the case here?

- Parsing the regex is much simpler.
- After a simple tokenization, the tree must be built, which requires the inside knowledge of the AST.

Detailed Algorithms

FSA graph traversal

This includes checking if a word belongs to the FSA, if the FSA is finite, and populates the languages.

The traversal can be done with an DFS. Until satisfying the condition (reaching the end states in checking if the word belongs to the FSA, or visited all states in checking finite FSA or populating the languages). A Visited set is assigned, storing all history pairs (State, wordIndex), indicating that this state was reached, matching the "wordIndex"-th character of the word, to make sure this state is not visited again.

A tricky case is while checking if the FSA is finite, there could be a loop in the graph. To handle this 2 values are assigned to every states: canReachEndState and belongsToCycle. If there is a back edge in the DFS tree (and edge creating a cycle), then the belongsToCycle of all nodes in the cycles are set to true. Besides, every node that can reach the end states is set to canReachEndState. After DFS, the FSA is finite if: there exist **NO** node, that belongsToCycle **AND** canReachEndState. That means, if there are cycles in the FSA, but there are no path going from start states to end states, passing such cycles, then the FSA is still finite.

Process Blocked by Adding Pushdown Automaton

When adding PDA was required, I realized there need to be an FSA subclass to the Automaton, and the Automaton must be an Abstract Class (at the time, Automaton was still the FSA). Hence the FSA (was still Automaton at the time) must be moved to a new class FSA, in a way that not break the previous features.

Plan

- Create a PushdownAutomaton class
- In parser, return a PushdownAutomaton if the input file is a pushdown automaton
- Implement Pushdown Automaton methods that override Automaton methods
- Implement FiniteStateAutomaton class, with methods copied from Automaton and overriding Automaton methods.
- Return the FiniteStateAutomaton in the parser (if the input is FSA)
- Delete related Automaton method content, make them virtual.
- Make Automaton an ABC.

Why did this happened?

Because I did not study the requirements carefully. It was thought that only additional features/algorithms on Automaton were needed. I didn't know Pushdown Automaton would have different way of working. The lesson is to understand all requirements before designing.

Another lesson is that refactoring consumes a lot of time.