

CS2001/CS2101 Week 3 Practical

Finite State Machines

4th October 2020

Matriculation Number: 200002548

Tutor: Alex Konovalov

Contents

1	Introduction	2
2	Design	2
2.1	FSMInterpreter Class	3
2.2	FiniteStateMachine Class	3
2.2.1	The load() method	4
2.2.2	The isLayoutValid() method	4
2.2.3	The isLogicValid() method	4
2.2.4	The run() method	5
2.3	StateInputKey Class	5
3	Testing	5
3.1	Basic Stackscheck Tests	5
3.2	Comprehensive Testing using Stackscheck	5
4	Conclusion	14

1 Introduction

The aim of this practical was to create a finite state machine (FSM) interpreter in Java. The interpreter had to construct a FSM by parsing a transition table read in from a text file and then compute the output generated by the FSM from its current state and the input provided. The interpreter had to be able to handle multiple different inputs, from the text file describing the FSM and the input string from standard input. The interpreter should output 'Bad description' if the machine's description is not well-formed and 'Bad input' if the input characters entered are not part of the set of inputs described in the FSM's description. The interpreter should run with the following command:

```
java fsminterpreter description.fsm < input.txt
```

An example of a valid transition table used to describe the FSM is shown below:

```
1 a z 2
1 b y 1
2 a x 3
2 b w 1
3 a v 2
3 b u 3
...
```

Where the current states, input symbols, output symbols and next states are shown in each column from left to right. States had to be numeric with inputs and outputs composed of single characters. The initial state of the FSM would be the state of the first row in the file.

Following this specification, I successfully built a FSM interpreter in Java. The program handles a variety of different transition tables and checks for errors with the layout of the FSM description, the logic within the FSM and also the string used as input symbols for the FSM. The program is designed to be as intuitive and simple as possible, whilst also having complex functionality to handle different possibilities with regards to the FSM description and input.

2 Design

The solution to this practical is split into three classes of which are featured in the UML class diagram in figure 1. Overall the practical was designed to be as efficient and maintainable as possible. The data structures used and methods designed below show how this was achieved.

2.1 FSMInterpreter Class

As specified, the `fsminterpreter` class features the main method. This class serves as an interface between the user and the finite state machine behind it. The class receives a path to the FSM description (.fsm file) and an input string from standard input. It then creates a `FiniteStateMachine`, passing in the input string and a `Scanner` object to read from the FSM description. Therefore this class creates a FSM in an abstract way, with all the details being abstracted from the `FiniteStateMachine` class. The benefit of this approach is that the interpreter class only has to deal with managing input and not with the inner workings of the FSM.

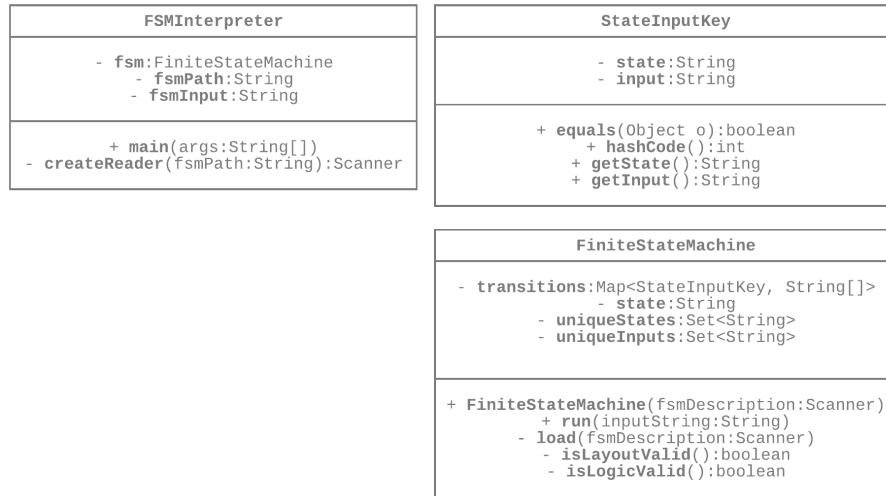


Figure 1: UML class diagram of solution

2.2 FiniteStateMachine Class

The `FiniteStateMachine` class is where most of the logic behind the finite state machines behaviour exists. This class features the `transitions` hash-table which takes a `StateInputKey`, `String[]` key-value pair. This hash-table stores all of the information from the FSM description about the current states, inputs, outputs and next states. The current states and inputs of the FSM are stored as keys within `StateInputKey` objects and the outputs and next states are the corresponding values. The hash-table was used to store these values rather than arrays or lists because it allows for constant time - $O(1)$ lookups. Therefore because a FSM performs so many lookup operations to fetch the next state and output a result, this will have a huge performance benefit. Furthermore it made implementation easier as an extra unconditional loop was not needed to iterate through an array or list. Hash-tables also automatically

remove duplicates and so it made removing lines in the FSM description which were exactly the same, automatic. The `state` of the FSM is stored as a string. There are also two sets called `uniqueStates` and `uniqueInputs` which store all of the unique states and inputs from the FSM description and are key when checking if the next state is a current state and if there is any state-input pairs missing from the FSM description. The `FiniteStateMachine` class has four methods (not counting the constructor), `load()`, `run()`, `isLayoutValid()` and `isLogicValid()`.

2.2.1 The `load()` method

The purpose of the `load` function is to load data from the specified FSM description and store it within the `transitions` hash-table. This is done by adding a new key-value pair to the hash-table. This key is created by instantiating a new `StateInputKey` object comprised of the state and input from the current row in the file. However a check is carried out to ensure that the number of symbols per line in the file is equal to four. This is because if the row had any other number of symbols then it is not a valid transition table. This method then sets the value of the FSM's `state` to the first state present in the FSM description.

2.2.2 The `isLayoutValid()` method

The design of this finite state machine keeps the loading and validating of the FSM description separate. This helps make implementation easier and the code more maintainable. This method iterates through the hash-map, and if either of the current or next states are not represented as numerical values then the method returns `false`. This method checks to see if the length of the `transitions` hash-map is greater than zero, and if returns `false`. Furthermore if either of the inputs or outputs are made up of strings that are not a character long then the method returns `false`. This is because the specification detailed that inputs and outputs must be single characters. Also if characters were used then the program would crash with `IllegalArgumentException` and not return '`Bad description`' when a input or output longer than one character was present in the FSM description.

2.2.3 The `isLogicValid()` method

This method iterates through hash-map again. It first performs a check to see if each next state in the hash-map is equal to a current state. If not another the method returns `false`. The method then checks to see if there are any missing rows within the transitions table. This is achieved as follows. We take the sets of the current states and inputs and obtain the product of the length of the sets. If the resultant number is greater than the number of transitions then there is a state-input pair missing and so the FSM description is invalid.

2.2.4 The `run()` method

Finally, the last and most important method within the `FiniteStateMachine` class. The `run()` method is where the FSM actually runs through the hash-table using the specified input and outputs the results, moving from state to state. The input string is iterated over. If `transitions` hash-table does not contain a input-state key pair, created with the current state of the machine and input symbol, then the method outputs 'Bad input'. If not then the FSM outputs the relevant output symbol and moves on to the next state. The `run()` method is designed to output the result from one row of the FSM immediately, rather than waiting for the whole FSM to finish. This is so we can have the correct FSM behaviour. If this was not the case, and a large transition table and large input sequence was entered into the program, then it would take forever for any output to be received.

2.3 StateInputKey Class

The `StateInputKey` class is needed to be able to use a pair of variables as a key to a `HashMap` as the default Java `HashMap`'s only allow for one variable to be used as a key. The class overrides the `equals()` and `hashCode()` methods so that when used within the `HashMap` the `contains()` function calculates the right hash values, taking into account the two variables, in order to lookup the values associated with them.

3 Testing

The interpreter was tested various ways. Using `stacscheck` the program was tested with the basic example tests found at `/cs/studres/CS2001/Practicals/W03-FSM/Tests/` and a more comprehensive set of tests of my own.

3.1 Basic Stacscheck Tests

The interpreter was built and ran successfully, completing all eight of the tests, and outputting the expected result. This can be shown in figure 2. This shows that the interpreter has some amount of robustness, however more comprehensive testing is needed.

3.2 Comprehensive Testing using Stacscheck

There were numerous extreme edge cases the interpreter had to be tested against. Below I have explained what each test case is testing for, what the expected result is, what the actual result was, as well as detailing the inputs provided to the interpreter.

All the tests below were ran with `stacscheck` and the test files are provided under the `test/` directory in this assignment. The output of `stacscheck` is shown

A terminal window titled 'frr1@trench:~/Documents/CS2101/Exercises/W03' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the command 'staccoscheck /cs/studres/CS2001/Practicals/W03-FSM/Tests/' and its output. The output indicates that the submission directory 'src' was found in the current directory. It then lists eight tests: 'BUILD TEST - basic/build : pass' and seven 'COMPARISON TEST' entries, all of which passed. The final summary is '8 out of 8 tests passed'.

```
frr1@trench:~/Documents/CS2101/Exercises/W03
File Edit View Search Terminal Help
frr1@trench:~/Documents/CS2101/Exercises/W03 $ staccoscheck /cs/studres/CS2001/Practicals/W03-FSM/Tests/
Testing CS2001 Week 5 Practical (FSM)
- Looking for submission in a directory called 'src': found in current directory
* BUILD TEST - basic/build : pass
* COMPARISON TEST - basic/Test01_simple/progRun-expected.out : pass
* COMPARISON TEST - basic/Test02_bigger/progRun-expected.out : pass
* COMPARISON TEST - basic/Test03_description/progRun-expected.out : pass
* COMPARISON TEST - basic/Test04_missinginput/progRun-expected.out : pass
* COMPARISON TEST - basic/Test05_illegal/progRun-expected.out : pass
* COMPARISON TEST - basic/Test06_minimal/progRun-expected.out : pass
* COMPARISON TEST - basic/Test07_also-description/progRun-expected.out : pass
8 out of 8 tests passed
frr1@trench:~/Documents/CS2101/Exercises/W03 $
```

Figure 2: Basic staccoscheck tests running successfully

below in figure 3. All of the tests ran successfully after a reasonable amount of errors and debugging. In its final state, the interpreter produced no errors and is a robust program that follows the specification and is programmed defensively where the specification becomes vague. For instance in Test 11 when no input is supplied the interpreter should not crash and instead produce no output, this was done successfully.

Furthermore, the program handles dealing with duplicate state/input pairs excellently. If two rows have the same state and input but different outputs and or next states then the interpreter flags that the description is invalid. If however the state and input pairs have the same outputs and next states then the description is still valid and the interpreter correctly produces the intended output.

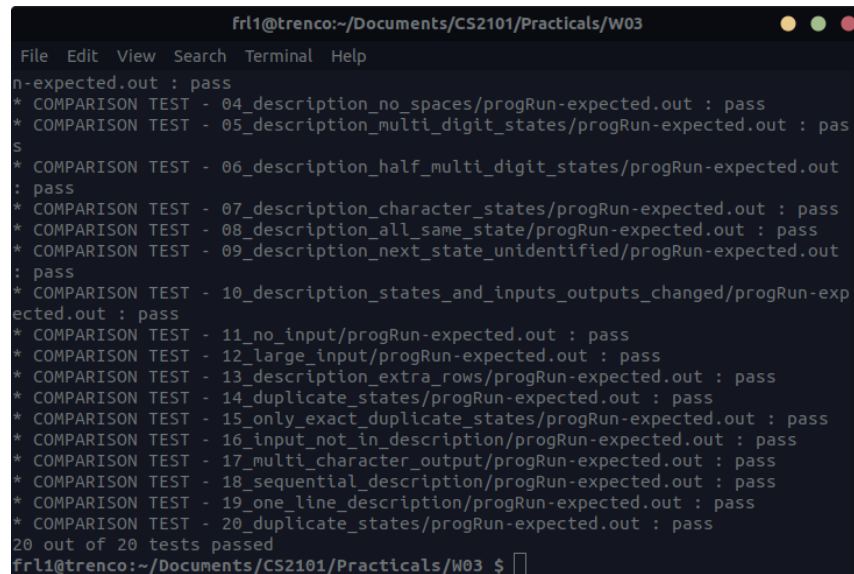
A screenshot of a terminal window with a dark background. The title bar shows the user 'frl1@trencor' and the path '~/Documents/CS2101/Practicals/W03'. The terminal displays the output of a test suite. It starts with 'n-expected.out : pass', followed by 20 comparison tests, each marked with an asterisk and 'pass'. The tests cover various FSM descriptions like '04_description_no_spaces', '05_description_multi_digit_states', etc. The final line shows '20 out of 20 tests passed' and the prompt 'frl1@trencor:~/Documents/CS2101/Practicals/W03 \$'.

Figure 3: Comprehensive tests running successfully in stacscheck

Test 01 - Empty FSM Description

Input: a

Expected Result: Bad description

Result: Bad description

no file...

Figure 4: description.fsm

This is correct behaviour, having no FSM description should flag up to the user that there is something wrong with their FSM description.

Test 02 - FSM Description Only Whitespace

Input: a

Expected Result: Bad description

Result: Bad description

Figure 5: description.fsm

Obviously, having a FSM description full of whitespace should be invalid and be flagged to the user.

Test 03 - FSM Description with Whitespace and Characters Interchanged

Input: a

Expected Result: Bad description

Result: Bad description

```
1 a h 2
2 a i 1
```

Figure 6: description.fsm

This is also an invalid form of an FSM description because the program is looking for characters separated by whitespace, not the other way around.

Test 04 - FSM Description No Spaces

Input: a

Expected Result: Bad description

Result: Bad description

```
1a2a2a2
2a1a1a1
```

Figure 7: description.fsm

Without any spaces it is impossible to determine where in the FSM description states, input and output symbols are, therefore this is an invalid description.

Test 05 - FSM Description with Multi-digit States

Input: ab

Expected Result: hi

Result: hi

```
10 a h 20
10 b r 10
20 b i 10
20 a e 10
```

Figure 8: description.fsm

Multi-digit states work with the FSM because it has been implemented in with the design to store states as strings rather than characters. Therefore they work just like single-digit states and don't cause any errors anywhere else and so we get a correct result from this test.

Test 06 - FSM Description with Multi and Single Digit States

Input: aba
Expected Result: zyx
Result: zyx

```
10 a z 20
10 b f 10
20 b y 1
20 a e 10
1 a x 10
1 b e 20
```

Figure 9: description.fsm

If both single and multi-digit states work on their own then they should both work together. We see this happening in the test above.

Test 07 - FSM Description with States Represented as Characters

Input: abc
Expected Result: Bad description
Result: Bad description

```
a a z b
b b y c
c c x a
```

Figure 10: description.fsm

In the specification it was said that states can only be represented numerically and so if states are instead represented as characters the description is invalid.

Test 08 - FSM Description All in the Same State

Input: abccd
Expected Result: hello
Result: hello

```

1 a h 1
1 b e 1
1 c l 1
1 d o 1

```

Figure 11: description.fsm

As we can see above the program works if there is only one state within the transition table, this is correct behaviour as having one state is still valid within a FSM.

Test 09 - FSM Description with Unidentified Next State

Input: a

Expected Result: Bad description

Result: Bad description

```

1 a h 2
2 a i 3

```

Figure 12: description.fsm

A FSM description with a next state that is not already a state in the transition table is invalid since that next state does not have a entry in the transition table.

Test 10 - FSM Description with Positions of States and Inputs/Outputs Changed

Input: a

Expected Result: Bad description

Result: Bad description

```

a 1 2 h
b 2 1 e

```

Figure 13: description.fsm

Obviously changing the columns of states, inputs and outputs causes the FSM description to be invalid since characters may be used to represent numbers for states and the input/output symbols may not match up with next states.

Test 11 - No Input

Input:

Expected Result:

Result:

```
1 a h 2
2 a i 1
```

Figure 14: description.fsm

If there is no input into the FSM then the FSM should just do nothing. Therefore the interpreter displays the correct output.

Test 12 - Large Input

Input: aaaaaaaaaaaaaaaaaa

Expected Result: hihihihihihihihih

Result: hihihihihihihihih

```
1 a h 2
2 a i 1
```

Figure 15: description.fsm

The FSM should be able to handle a large input just as it would a small one. All that is required is more computing time for searching in the transition table.

Test 13 - FSM Description with Extra Rows

Input: aa

Expected Result: Bad description

Result: Bad description

```
1 a h 2 a b c d
2 a i 1 a b c d
```

Figure 16: description.fsm

The specification said that the FSM description must exist at 4 symbols separated by spaces, therefore if more rows are in the description, it should become invalid.

Test 14 - FSM Description with Duplicate State/Input Pair

Input: a

Expected Result: Bad description

Result: Bad description

```

1 a i 1
1 a x 1
2 b a 2

```

Figure 17: description.fsm

A FSM description with a duplicate state/input pair should be invalid since the FSM is non-deterministic since one state and input corresponds to two different outputs and or next states, which is not possible.

Test 15 - FSM Description with Only Exact Duplicate States

Input: a

Expected Result: i

Result: i

```

1 a i 1
1 a i 1

```

Figure 18: description.fsm

An FSM description with only exact duplicate states means that the FSM only has one unique row in the transition table and so behaves like a single row. Since a transition table with a single row is valid the FSM runs through this row and outputs the result.

Test 16 - Input not in FSM Description

Input: f

Expected Result: Bad input

Result: Bad input

```

1 a i 1
1 b x 1

```

Figure 19: description.fsm

If an input is not featured in the transition table as an input then the input is an invalid input into the FSM as there will be no method to deal with it.

Test 17 - Multi Character Output

Input: ab

Expected Result: Bad description

Result: Bad description

```
1 a yes 1
1 b no 1
```

Figure 20: description.fsm

The specification insisted that inputs and outputs be represented as single characters and so the FSM description is invalid because multi character outputs are given.

Test 18 - Sequential FSM Description

Input: aaaaaaaaaa

Expected Result: Hello_World!

Result: Hello_World!

```
1 a H 2
2 a e 3
3 a l 4
4 a l 5
5 a o 6
6 a _ 7
7 a W 8
8 a o 9
9 a r 10
10 a l 11
11 a d 12
12 a ! 1
```

Figure 21: description.fsm

This was just an interesting test to check on the correct operation of running the FSM with the transition table. The test just requires one input but produces a different character as it move up a state. The interpreter produces the correct result.

Test 19 - One Line FSM Description

Input: a

Expected Result: i

Result: i

```
1 a i 1
```

Figure 22: description.fsm

Even though it is a small FSM description it is still valid and is ran by the interpreter.

Test 20 - Exact Duplicate Rows

Input: ba

Expected Result: hi

Result: hi

```
1 a i 1
1 a i 1
1 b h 1
```

Figure 23: description.fsm

If two rows in the FSM description are exactly the same then the description is still valid as both rows still produce the same result and move to the same state.

4 Conclusion

In this practical I have created a FSM interpreter in Java that has demonstrated to be robust and fit for purpose. As a direct entry student I had never written a Java program before that was more than the `main()` and one other method. Therefore in this task not only did I learn about the operation of finite state machines, but I developed my programming ability in Java and learnt about the nuances of the language. Many of the features of Java are shared with C++ and so thanks to my knowledge of C++ I could get to grips with the language faster.

I found the difficulty of the practical to be not as harsh as I originally thought. The class structure and methods became clear the more I progressed into the practical. At the start of programming I was storing the transition table as four lists within that `FiniteStateMachine`. However when I learned how to implement hash-tables in Java using `HashMap`'s and the efficiency of constant time lookups I redesigned the whole class around the use of the hash-tables. This made the whole code much easier to read as less unconditional loops were needed to iterate over lists. Furthermore using the `StateInputKey` class helped me understand more about the function of `HashMap`'s within Java.

Given more time I would like to create another Java program to automatically generate test transition tables to load into the FSM and then link the two

together. It would be an interesting thought to have one machine creating the transition tables (using some rules) which then tell another machine how to process inputs in order to generate some output.

References

- [1] Java Program to Check if a String is Numeric, n.d., accessed 28-09-2020,
<https://www.programiz.com/java-programming/examples/check-string-numeric>
- [2] Oracle, String(Java Platform SE 7), n.d., accessed 29-09-2020,
<https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>
- [3] Oracle, HashSet(Java Platform SE 7), n.d., accessed 29-09-2020,
<https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html>
- [4] SudoRahul, Convert Character to ASCII Numeric Value in Java, 09-05-2013, accessed 04-10-2020,
<https://stackoverflow.com/questions/16458564/convert-character-to-ascii-numeric-value-in>
- [5] karim79, Iterate Through a HashMap, 30-07-2009, accessed 04-10-2020,
<https://stackoverflow.com/questions/1066589/iterate-through-a-hashmap>
- [6] Tomasz Nurkiewicz, How to Create a HashMap with Two Keys (Key-Pair, Value), 03-02-2013, accessed 04-10-2020,
<https://stackoverflow.com/questions/14677993/how-to-create-a-hashmap-with-two-keys-key-p>