**CSC615M MP4: Combination of Turing Machines**

Ryan Austin Fernandez

CS-ST Student

2401 Taft Avenue,

Manila, Philippines 1004

(+63)917-621-6469

ryan\_fernandez@dlsu.edu.ph

**ABSTRACT**

Turing Machines are involved in the Church-Turing thesis, which states that any computation that can be done can be modelled using a Turing Machine. To show this, complex computations are performed such that each computation is built up from simpler Turing Machines. This project aims to show that this is possible, in the hopes that appreciation will be gained for this model of computation. Using predefined Turing Machines, the system was designed with each Turing machine as a command in the Command Design Pattern and the model was implemented in Python. The testing was successful, with the model being able to do simple computations such as gcf and square root.

**I. INTRODUCTION**

Turing Machines provide a generalized model for computation. In the Church-Turing thesis, it is stated that any computation that can be regarded as possible to carry out can be performed by a Turing machine with suitable instructions. Therefore, any possible computation can be modelled as an algorithm to be carried out by a Turing Machine (Denning, P.J., Dennis, J.B., & Qualitz, J.E., 1978).

This raises the question of whether a low level simulation of Turing Machines is possible, such that with elementary Turing Machines, simple computations such as increment or decrement can be built upon.

This project’s goal is to simulate more complex Turing Machines with simple Turing Machines to be described in the next section. The project’s scope is also further described in the next section.

This project’s significance is to see the importance of the Church-Turing thesis, and how it can be used to model basic computation, such as greatest common divisor or square root.

**II. DESIGN**

The elementary Turing Machines described in the project operate on the general notion that Turing Machines operate on a tape with #-delimited natural numbers in unary notation. This means an input of x1,x2 would look like #1x1#1x2#.

The elementary Turing machines implemented are as follows:

shL k / shR k – shift left/right moves the tape head k numbers to the left or right

copy k – copies the kth number to the left of the head and places it to the right of the initial head position. The head ends up on the right of the new copy.

const k – prints the constant k in unary notation to the right of the tape head. Tape head remains to the left of the constant.

move m,n – deletes m numbers to the left of the initial tape head and moves n numbers from the right of the initial tape head location to the left. Tape head ends at the right of all numbers.

pushL – equivalent to move 1, 1

inc – Current number is copied and incremented. Tape head is to the left of the incremeneted copy.

dec - Current number is copied and decremented. Tape head is to the left of the decremeneted copy.

add – adds the two numbers to the right of the tape head and leaves the result there.

mult - multiplies the two numbers to the right of the tape head and leaves the result there.

monus - subtracts the two numbers to the right of the tape head and leaves the result there. If the first number is less than the second number, a 0 is set as the result.

swap – swaps the two numbers to the right of the initial tape head.

gotoEQ – checks if the first number to the right of the tape head is equal to the second number

gotoNE – checks if the first number to the right of the tape head is not equal to the second number

gotoGE – checks if the first number to the right of the tape head is greater than or equal to the second number

gotoGT – checks if the first number to the right of the tape head is greater than the second number

gotoLE – checks if the first number to the right of the tape head is less than or equal to the second number

gotoLT – checks if the first number to the right of the tape head is less than the second number

There is also a goto option which does an unconditional jump to an instruction.

Combined Turing Machines are a set of instructions of the form

Instruction no.,label,[param1,…],[next instruction]

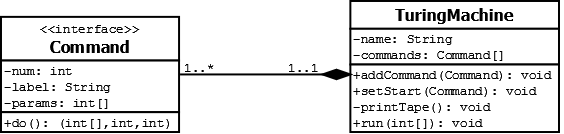
For goto type instructions, it is implied that if the condition is evaluated to false, the program executes the next statement in sequence i.e. no jump.

All these Turing Machines were abstracted in the Command Design Pattern, where their execution is abstracted as a do() method in the Command interface.

The do() method takes in the list of integers on the tape, the current tape head location, and then outputs the new tape, the new tape head location, and the new instruction counter. HALT returns NIL instead of the usual 3-tuple.

A Turing Machine, then, is just composed of a list of commands to be executed as specified.

Figure 2.1 shows the UML diagram of the system.



**Figure 2.1. – UML Diagram of the System**

The algorithm now for running the Turing Machine is in Table 2.1.

|  |
| --- |
| **Table 2.1. – Algorithm for running the Turing Machine** |
| def run(input)  input – integers on the tape  curr = start #start instruction  index = 0  list = input  print("START")  printTape()  while curr is not NIL:  ret = curr.do(list,index)  print("After " + curr)  if ret is None:  curr = None  else:  list = ret[0]  index = ret[1]  curr = commands[ret[2]]  printTape() |

**III. IMPLEMENTATION**

The system was implemented in Python. The project had four modules: TuringMachine.py for modelling the actual machine, Command.py to model the states, FileReader.py to read the file, and Main4.py to act as the driver.

The TuringMachine class used a dictionary to map instruction ids to commands. The commands simple extended the Command class that acts as an interface.

**IV. TESTING**

**V. CONCLUSION**

In conclusion, the project was successfully implemented. The program can successfully simulate any given deterministic two-way accepter. It can be used as a basis for implementing combinations of Turing Machines.

**APPENDIX A – SELF-ASSESSMENT RUBRIC**

|  |  |
| --- | --- |
| Criterion | Grade |
| I/O Modules | 29 |
| Core Process | 30 |
| Quality of Testing | 20 |
| Documentation | 20 |
| TOTAL | 99 |

**REFERENCES**

1. Denning, P.J., Dennis, J.B., & Qualitz, J.E. (1978). *Machines, languages, and computation*. New Jersey: Prentice Hall.