CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

# Survey of Imperative Style Turing Complete proof techniques and an application to prove Proteus Turing Complete

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in
Computer Science

By

Isaiah Martinez

December 2024

The thesis of Isaiah Martinez is approved:

| | |
|---|---|
| Kyle Dewey, PhD., Chair | Date |
| John Noga, PhD. | Date |
| Maryam Jalali, PhD. | Date |

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus lacinia odio vitae vestibulum vestibulum. Cras venenatis euismod malesuada. Maecenas vehicula felis quis eros auctor, sed efficitur erat suscipit. Curabitur vel lacus velit. Proin a lacus at arcu porttitor vehicula. Mauris non velit vel lectus tincidunt ullamcorper at id risus. Sed convallis sollicitudin purus a scelerisque. Phasellus faucibus purus at magna tempus, sit amet aliquet nulla cursus.

Table of Contents

List of Figures

This list must reference the figure, page it appears, and subject matter.

List of Tables

This list must reference the table, page it appears, and subject matter.

# List of Illustrations

This list must reference the illustration, page it appears, and subject matter.

Abstract

TITLE GOES HERE

By

Isaiah Martinez

Master of Science in Computer Science

Abstract which will cover the contents of the entire paper in less than 350 words or so. 1.5 pgs double spaced

state research probem briefly describe methods and procedures used in gathering data or studying the problem give a condensed summary of the findings of the study

# Chapter 1

# Introduction

## 1.1 Outline

This project aims to understand how Turing Completeness is demonstrated across different disciplines and apply them to a novel programming language, Proteus. The goal of this thesis is to show Proteus is Turing Complete. As such, I will first describe some major concepts such as Turing Machines, Turing Completeness, and major theorems that will be utilized. I will supplement this with some programming language design details which prove important for showing that Proteus is Turing Complete. Afterwards, I will describe Proteus in detail.

The following chapter will describe the different approaches from each domain showing Turing Completeness. These include Computer Engineering, Computer Science, and Mathematics where the different proofs will be discussed in detail.

With this understanding of showing a system is Turing Complete, I will outline a proof to show that Proteus is Turing Complete. With the proof outlined, I will discuss the design and approach. After this, I will follow the outline to flesh out the proof.

After demonstrating that Proteus is Turing Complete, I will reflect on the knowledge gained and applied towards this project. I will remark on some points of improvement, and then conclude the Thesis.

## 1.2 Turing Machines

Alan Turing is generally considered the father of computer science for his numerous contributions including: formalization of computation theory, algorithm design, complexity theory, as well as creating the idea of the Turing Machine. A Turing machine can be described as a machine/automata

that is capable of performing operations towards some desired goal given an input. In a sense, it was designed to be capable of performing any single computable task, such as addition, division, concatenating strings, rendering graphics, etc. [10]. TMs are at the highest level of computational power, i.e. capable of handling any computation [11].

There are two different kinds of TMs: Deterministic and Non-deterministic. Regardless of such a construction, both are equivalent in power to each other, and as such remain interchangeable until practice [1]. Assume all TMs discussed are Deterministic unless otherwise stated.

Figure 1.1 is an example of a Turing Machine designed using JFLAP, see https://www.jflap.org/ for more information on this software. In fact JFLAP has been tested to grade students homework on computation [12], showcasing just how useful this tool is. The described machine takes an input string of 1's followed by a 0. The machine then outputs whether there is an even number of 1's or not. We will revisit this exact machine later in section 2.3.

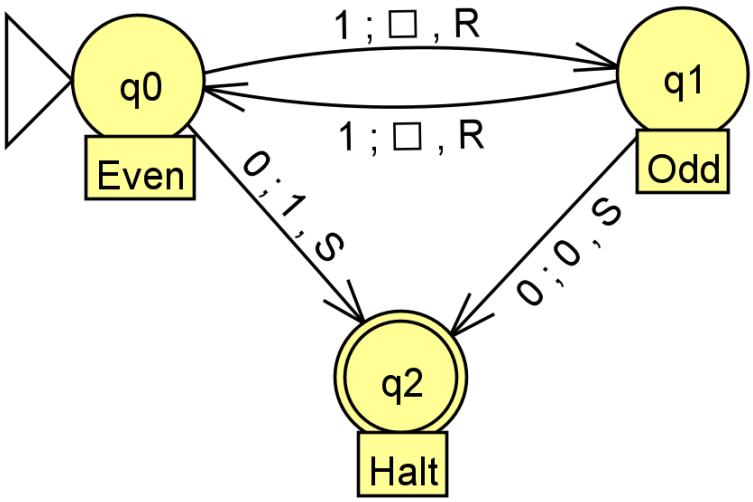


Figure 1.1: A TM that determines if there is an even number of 1's on the tape.

### 1.2.1 Oracles

Of course there also exists the Oracle, sometimes called Turing Machines with an oracle, which is capable of solving problems that TMs cannot. It does so by having the ability to respond to any given problem from the TM it is connected to. For example, the oracle would be able to solve the

Halting Problem for the associated Turing Machine, but not the Halting Problem in general for all Turing Machines. The reason the oracle is not considered more powerful is because in practice (i.e. reality), is because there is no such all-knowing source to retrieve information from. As a result, I will disregard the Oracles for the rest of the thesis.

### 1.2.2 Universal Turing Machines

A simple abstraction of the standard TM is a Universal Turing Machine. A UTM is capable of solving any computable problem, given the exact process/rules of the TM that will solve it. In essence it is a machine that is not hard-coded with what to perform when given input. The UTM will read the input, and respond based on the rules given. As a result, the UTM is equivalently as powerful as a TM. The only functional difference is the usability of the UTM towards a larger number of problems as opposed to the TM being created for a singular problem.

### 1.2.3 The Church-Turing Thesis

According to the Church-Turing Thesis, every effectively calculable function can be computed by a Turing Machine. As explained by Robin Gandy, the main idea was to show that there is an upper bound on the computation of TMs. This upper bound does not exist for humans and is therefore the basis of separation between computation power of TMs and humans.

**Theorem 1** *(Church-Turing Thesis) Every effectively calculable function can be computed by a Turing Machine.*

He proposed a series of four Principles that we still use today as a basis for determining what one of the definitions of a TM is capable of computing. Any automata that violates any of these Principles is said to have ”free will”, which in context means being able to compute any non-computable function. As an example, the Oracle machine would be capable of computing a non-computable function, namely the halting problem, and thus would have ”free will”. We disregard such automata as there aren’t any systems that exist in reality as of yet to display ”free will”. As a

result, TMs are the most powerful automata that can compute any calculable function. This is why the Church-Turing thesis is generally assumed to be true [2].

### 1.2.4 Rice's Theorem

Another famous theorem to consider is Rice's Theorem. It is widely known in the domain of computability theory as it defines the bounds of research in a formal manner. Below is the Formal statement of Rice's Theorem, which will be supplemented with a concise statement summarizing the theorem. Wolfram Rice Thm

**Theorem 2** *(Rice's Theorem) Let $\varphi$ be an admissable numbering of partial computable functions. Let P be a subset of $\mathbb{Z}$. Suppose that:*

1. *P is non-trivial: P is neither empty nor $\mathbb{N}$ itself.*
2. *P is extensional: $\forall m,n \in \mathbb{N}$, if $\varphi_m = \varphi_n$, then $m \in P \Leftrightarrow n \in P$.*

*Then P is undecidable.*

*I.e. The only decidable index sets are $\emptyset$ and $\mathbb{N}$.*

Rice Thm wiki

This directly translates to programming languages as stating: All non-trivial smenatic properties of programs are undecidable. Thus, Rice's theorem is a generalization of the Halting Problem. Because of this theorem, it is impossible to design a program that determines if a given program is able to execute without error. Another view of thie theorem is that it states only trivial properties of programs are algorithmically decidable. Eg. If a program has an if statement inside the code.

## 1.3 Turing Completeness

Turing Completeness is a closely related term when discussing Turing Machines. For a system to be Turing Complete, it must be capable of performing any computation that a standard TM can perform. An equivalent description would be that for a system to be TC, it must simulate a UTM.

By transitivity, if any system is proven to be TC, then it must be equivalent in power to all other systems that are TC. Therefore, all TC systems are considered the most powerful computation machine.

### 1.3.1 Considering the Practicality of Turing Complete Programming Languages

Despite all TC systems being equivalent in computation power, this does not mean that all are practically useful. This is because despite the TC being able to simulate any TM, it may have a more complex method for simulation or calculation of the same problem. This is considered a non-issue as the length of time needed for computation is not considered when discussing TMs and TC systems. This is only a factor for practical purposes, such as programming languages, space and time complexity are of major importance.

#### 1.3.1.1 Esoteric Progamming Languages

Esoteric programming languages are designed to demonstrate a key concept with language design, but are often done so in a joking manner. An example of a highly simplistic well-known esoteric TC language is brainfuck. I will describe the way brainfuck operates and then provide several example programs with explanations.

The language has only 8 instructions, a data pointer, and an instruction pointer. It uses a single dimensional array containing 30,000 byte cells, with each cell initialized to zero. The data pointer points to the current cell within the array, initialized to index 0. The instruction pointer points to the next instruction to be processed, starting from the first character given in the code. Any characters besides those used in the instructions are considered comments and will be ignored. Instructions are executed sequentially unless branching logic is taken via the '[' or ']' instructions. The program terminates when the instruction pointer moves beyond the final command. Additionally, it has two streams of bytes for input and output which are used for entering keyboard input and displaying output on a monitor using the ASCII encoding scheme. [Wikipedia citation brainfuck](https://en.wikipedia.org/wiki/Brainfuck) [Github basics of Brain-

fuck](https://gist.github.com/roachhd/dce54bec8ba55fb17d3a)

The 8 instructions are as follows:

| | |
|---|---|
| $>$ | Increments the data pointer by one. (This points to the next cell on the right). |
| $<$ | Decrement the data pointer by one. (This points to the next cell on the left). |
| $+$ | Increments the byte at the data pointer by one. |
| $-$ | Decrements the byte at the data pointer by one. |
| . | Output the byte at the data pointer. |
| , | Accept one byte of input, storing its value in the byte at the data pointer. |
| **[** | If the byte at the data pointer is zero, then instead of moving the instruction forward to the next command, go to the matching ']' command. (Jump forwards). |
| **]** | If the byte at the data pointer is non-zero, then instead of moving the instruction forward to the next command, go to the matching '[' command. (Jump backwards). |

Table 1.1: Brainfuck Instruction Set

Each '[' or ']' must correspond to match with it's complement symbol, namely ']' and '[' respectively. Also, when input is read with the ',' command the given character from a keyboard input will have its value read as a decimal ASCII code (eg. '!' corresponds to 33. 'a' corresponds to 97, etc.). The decimal value is what is then converted to binary and stored within the current byte.

BRAINFUCK CITATION FROM STACK OVERFLOW

Here is a simple program that modifies the value of the first cell in the 30,000 byte array.

```
++      Add 2 to the byte value in cell 0
[-]     Decrement the value of the current cell until it reaches 0
```

In fact, we can remove the comments and put the code onto a single line to achieve the same result. Recall that comments include any character that is not listed as one of the 8 aforementioned instructions.

```
++[-]
```

An equivalent program in python is seen below:

```
# Let 'Array' be our 30,000 byte array
Array[0] += 2
while (Array[0] != 0):
    Array[0] -= 1
```

Below is an example program that outputs Hello World. At the end of each line is the end result of the operations done in the line as a comment. Each line prints a new character.

```
>++++++++[<+++++++++>-]<.       H
>++++[<+++++++>-]<+.            e
+++++++..                       l
+++.                            l
>>++++++[<+++++++>-]<++.        o
------------.                   [space]
>++++++[<+++++++++>-]<+.        W
<.                              o
+++.                            r
------.                         l
--------.                       d
>>>++++[<++++++++>-]<+.         !
```

For an in-depth breakdown of brainfuck with examples and guiding logic, see: [Github basics of Brainfuck](https://gist.github.com/roachhd/dce54bec8ba55fb17d3a).

One common technique utilized when creating programming languages is to bootstrap them. This means that the developers will write a compiler for the language, using the language itself. This is done for many reasons, but the reason for introducing it here is to show how brainfuck is capable of complex logic that is more practically useful than simple programs as seen above. Below is the current smallest bootstrapped compiler for brainfuck.

PLACE SAYING ITS THE SMALLEST BRAINFUCK COMPILER [SMALLEST BRAINFUCK COMPILER][https://brainfuck.org/dbfi.b]

```
>>>+[[-]>>[-]++>+>+++++++[<++++>>++<-]++>>+>+>+++++[>++>++++++<<-]+>>>,
<++[[>[->>]<[>>]<<-]<[<]<+>>[>]>[<+>-[[<+>-]>]<[[[-]<]++<-[<+++++++++>[
<->-]>>]>>]]<<]<]<[[<]>[[>]>>[>>]+[<<]<[<]<+>>-]>[>]+[->>]<<<<[[<<]<[<]
+<<[+>+<<-[>-->+<<-[>+<[>>+<<-]]]>[<+>-]<]++>>-->[>]>>[>>]]<<[>>+<[[<]
<]>[[<<]<[<]+[-<+>>-[<<+>++>-[<->[<<+>>-]]]<[>+<-]>]>[>]>]>[>>]>>]<<[>>
+>>+>>]<<[->>>>>>>>>]<<[>.>>>>>>>]<<[>->>>>>]<<[>,>>>]<<[>+>]<<[+<<]<]
```

As we quickly found out, using brainfuck in any practical sense is simply too much work due to its extreme inefficiency. It also is extremely difficult to understand without comments indicating the goal of each step. Due to the simplicity of the language, it is very useful for studying Turing Completeness. There exist many other esoteric TC programming languages, but the reason for choosing brainfuck in particular is its simple instruction set. As a result, we will now look at more useful and practical programming language paradigms. These languages within these paradigms will be much more efficient and legible, at the cost of increased complexity in instruction set.

#### 1.3.1.2 Procedural Languages

Procedural Programming Languages are designed to be read linearly in execution order, top to bottom. The main idea behind this design of languages is to create procedures and subprocedures (equivalently routines and subroutines), to achieve a larger goal. For example to calculate the sum of squares in code you may design it in the C code as follows:

```
float squareNumber(float a) {
    return a * a;
}
float findSumOfSquares (float a, float b) {
    return squareNumber(a) + squareNumber(b);
```

```
}
```

When working in Procedural Programming Languages, variables are used to store and modify data. These variables may be locally or globally defined, which is where we define the concept of scope. Scope refers to the current lens in which we view code and the system memory. It identifies which variables exist, what values they have, and what operations are being performed. The below example in C provides insight into the importance of scope.

```
float globalVariable;


float foo (float bar) {
    float localVariable;
    ...
}


float baz (float qux) {
    float localVariable;
    ...
}
```

Notice that we are able to utilize a variable named localVariable in the two functions foo and baz. This is allowed because when the scope is inside of either function, the other function does not exist. The variable globalVariable is available to both because it is outside of the scope of both functions. This means that any other function in the code in the same scope as the globalVariable is capable of accessing its value.

I will now describe the importance of functions in Procedural Programming Languages. Functions are designed to complete a single goal and return a single output. They are capable of accepting $0^+$ inputs and outputting 0 or 1 outputs. These functions are capable of calling other functions,

including themselves. When a function calls itself, this is called recursion. Here is an example constructing the fibonacci sequence in C in 2 ways: without recursion, and with recursion.

```
//Given an integer n, calculate the first n numbers
//of the fibonacci sequence without recursion

void sequentialFibonacci (int n) {
    if (n < 1) {
        printf("Input must be an integer greater than 0");
        return;
    }

    int sub1 = 0;
    int sub2 = 1;

    for (int i = 1; i <= n; i+=1) {
        if (i > 2) {
            int curr = sub1 + sub2;
            sub1 = sub2;
            sub2 = curr;
            printf("%d ", curr);
        }
        else if (i == 1) {
            printf("%d ", sub1);
        }
        else if (i == 2) {
            printf("%d ", sub2);
        }
```

```
    }
}

/*****************************************************************/

//Recursive case

//keep track of current Index, given amount of fibonacci numbers
//to print, and propogate the two subnumbers to the next step
void recursiveFibonacci (int currIndex, int n, int sub1, int sub2) {
    if (currIndex < n) {
        printf("%d ", sub1 + sub2);
        recursiveFibonacci(currIndex + 1, n, sub2, sub1 + sub2);
    }
    return;
}

//handle base cases (exit conditions)
//otherwise start the recursive process
void startRecursiveFibonacci (int n) {
    if (n < 1) {
        printf("Input must be an integer greater than 0");
    } else if (n == 1) {
        printf("%d ", 0);
    } else if (n == 2) {
        printf("%d %d ", 0, 1);
    } else {
```

```
            printf("%d %d ", 0, 1);
            recursiveFibonacci(0, n - 2, 0, 1);
        }
        return;
    }
```

As shown above, recursion is a powerful tool to simplify the amount of lines needed to code the main functionality of the procedure. It simplifies the amount of lines written because the overall logical design is more complex.

Through the use of scope and compartmentalizing procedures, Procedural programming is a very straightforward and capable design paradigm for software development. Some well known languages that are Turing Complete from this paradigm are:

- C
- Pascal
- COBOL
- Fortran
- ALGOL
- Basic

Although these languages are very old, with some coming from the 1960s, some still find modern use. Linus Torvalds, the creator of the Linux kernel and git, chose C to be the main language for developing both of these well known pieces of software. Both are still actively developed and improved to this day and remain majorly written in C. [SOURCE TO SEE THAT FILES FOR GIT ARE IN C][https://git.kernel.org/pub/scm/git/git.git/tree/] [SOURCE TO SEE THAT LINUX KERNEL FILES ARE IN C][https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/] Additionally, Richard Stallman led the development for the GNU operating system using C. SOURCE

TO SHOW C IS PREFERRED Although most users are on the Windows or Apple platform for PCs, the GNU operating system with the Linux kernel is still a popular choice amongst users looking for a different experience. [SOURCE SHOWING MARKET SHARE OF OS][https://gs.statcounter.com/os-market-share/desktop/worldwide/] Besides C, COBOL remains a language that is used professionally for banking. Many banks still use COBOL their business application and management. [COBOL USED BY BANKS][https://increment.com/programming-languages/cobol-all-the-way-down/]

#### 1.3.1.3 Object Oriented Languages

A different scheme altogether for a programming language is an Object Oriented Language. Developing in this language paradigm is known as a Object Oriented Programming. OOP is structured entirely different than Procedural Programming. Instead of defining procedures to solve the problem, we utilize a new idea of coding. We outline classes, which are representations of some system that we wish to design. Classes contain 3 parts: Data Members, Constructors, and Methods. Data Members are used to describe what the Class is. For example, if I want to model a school, an important data member would be the amount of students enrolled. Constructors are ways to create an instance of the class. This is where the object is created. In the school example, perhaps there would be two ways to create a school: with a total amount of students enrolled already, and another without. Both are valid as adding an existing school to the digital system would use the first constructor, while creating a new school would utilize the second constructor. Methods are ways we modify the attributes of the objects. Perhaps a certain amount of students enroll into the newly created school. We must have a way to update the amount of students for any school. These 3 parts form the basis of what OOP looks like. Below is a snippet of Java code demonstrating these principles.

```
class School {

    private int numEnrolledStudents;
```

```
    public School () {
        this.numEnrolledStudents = 0;
    }

    public School (int numAlreadyEnrolled) {
        this.numEnrolledStudents = numAlreadyEnrolled;
    }

    public int getNumEnrolled() {
        return this.numEnrolledStudents;
    }

    public void setNumEnrolled(int numStudents) {
        this.numEnrolledStudents = numStudents;
        return;
    }
}

class RunCode {
    public static void main(String[] args) {
        School NewSchool = new School();

        System.out.println("The number of students enrolled
            in the new school is: " + NewSchool.getNumEnrolled()); // 0

        NewSchool.setNumEnrolled(100);
```

```
        System.out.println("The number of students enrolled
            in the new school is: " + NewSchool.getNumEnrolled()); // 100

        School CSUN = new School(32172);

        System.out.println("The number of students enrolled
            at CSUN is: " + CSUN.getNumEnrolled();                // 32172
    }
}
```

[SRC FOR CURRENT NUM CSUN STUDENTS][https://www.usnews.com/best-colleges/california-state-university-northridge-1153]

There are more advanced features such as Inheritance that allow for more complex design models. Furthermore, Java contains Modifiers which are used to change the permission of which piece of code is capable of being accessed by another piece of code. In the above example, only the school object is capable of managing the data of numEnrolledStudents. Through the use of the methods getNumEnrolled and setNumEnrolled, any other class can modify the value of the class, but only through the reference of the school object,

Some well known languages that are Turing Complete from this paradigm are:

- Java
- C++
- Scala
- PHP
- Perl
- Swift

#### 1.3.1.4 Multi Paradigm Languages

Some languages allow for the combination of OOP and Procedural Programming. In such paradigms, the code allows for both to be run at the same time and enjoys the benefits of both approaches, at the cost of increased design overhead of the project. Here is an example snippet of Python code that demonstrates both at the same time:

```
def findSumOfSquares(num1, num2):
return (num1 ** 2) + (num2 ** 2)


class Homework:
    def __init__(self, problem):
        self.problem = problem


    def problem(self, problem):
        self.problem = problem


HW = Homework("What is the sum of squares of 2 and 3?")
print(HW.problem)


print(findSumOfSquares(2, 3))


#####       Printed to Terminal     #####


What is the sum of squares of 2 and 3?
13
```

In the code example, we utilize Procedural Programming to create the findSumOfSquares func-

tion. Through the usage of OOP, we create a Homework object that has a single data member, a single problem. By accessing the problem within the Homework object, we are able to print it out, then use the function to solve it.

Some well known languages that are Turing Complete from this paradigm are shown below. Notice that some languages mentioned in the previous sections may show up in the list:

- JavaScript
- C++
- Python
- R
- Perl
- Fortran

Multi Paradigm languages have a lot of flexibility for the applications that they can be used to create. The top frontend frameworks for web development use Javascript as their main language including React, Vue, Svelte, and more. [JS IN FRONTED][https://www.simform.com/blog/javascript-frontend-frameworks/] Python is very popular for its legible and flexible code. With its libraries such as Tensorflow and Keras, Machine Learning and other AI subgenres are easier to implement than in other languages. The popular LLM ChatGPT is primarily written in python. [chatgpt uses python][https://enjoymachinelearning.com/blog/is-chatgpt-written-in-python/] R is another language that is popular for its data science capabilities. It is heavily used within the Sciences (alongside python) because of its simplistic syntax, as well as its numerous libraries for data analysis. [LINK TO MY PROJECT ON ML AND DL IN PYTHON AND R][INSERT LINKS] [R for data science][https://www.simplilearn.com/what-is-r-article]

#### 1.3.1.5 Functional Programming Languages

Consider these next two sections as an outline and series of notes on what to do. They aren't fleshed out like the rest of the document that is written.

NOT SURE IF I WANT TO KEEP THE FOLLOWING 2 SECTIONS OR NOT: FUNCTIONAL PROGRAMMING LANGUAGES, LOGIC PROGRAMMING LANGUAGES. I WOULD PROBABLY KEEP IT SIMPLE WITH 1-3 THINGS TO DESCRIBE ABOUT THE WAY THE LANGUAGE IS STRUCTURED W/ 1-2 CODE EXAMPLES AND EXPLANATIONS. (I ESTIMATE 3-5 PAGES PER SECTION DEPENDING ON SIZE OF EXAMPLES). I THINK I'VE SHOWN MY POINT, WHICH IS THAT THE DIFFERENT PARADIGMS ARE CAPABLE OF BEING TC DESPITE BEING SO DIFFERENT FROM EACH OTHER. IT MIGHT BE COOL/INTERESTING TO SHOW THE DIFFERENT THINKING INVOLVED FOR LANGUAGES LIKE PROLOG. I COULD ALSO COME BACK TO ACKNOWLEDGE CERTAIN ASPECTS OF LANGUAGE DESIGN PRINCIPLES HERE, WHICH WOULD REQUIRE FURTHER READING. I DO ADMIT THAT THIS IS SOMEWHAT OUTSIDE THE SCOPE OF THIS THESIS BECAUSE IM NOT EXACTLY LOOKING AT THE STYLE OF LANGAUGE OF PROTEUS, NOR HOW IT COMPARES TO THESE PARADIGMS TO CATEGORIZE IT.

describe design of these languages Some well known languages that are Turing Complete from this paradigm are:

- Lisp
- Haskell
- Elixir
- OCaml
- Go
- Rust

DESCRIBE SOME ACTUAL USAGE OF THESE PROGRAMS IN MODERN SOFTWARE DEVELOPMENT

RUST IS BEING USED TO TRY TO CREATE A NEW AND DIFF LINUX KERNEL. GO IS NEW POPULAR PROGRAMMING LANGUAGE FOR WEB DEVELOPMENT, DEVOPS, ETC.

#### 1.3.1.6 Logic Programming Languages

The most well known Logic Programming Language is Prolog. describe design of prolog Describe how it basically relies on recursion to do stuff. Other languages exist, but are not popular when compared to the aforementioned languages in the previous sections.

IS PROLOG EVEN USED TODAY FOR STUFF IN INDUSTRY???

## 1.4 Proteus

The main goal of this thesis is to outline a proof demonstrating that a novel prototype language, Proteus, is TC. In this section, I will describe in detail what Proteus is.

### 1.4.1 Proteus Description

( EN - IS IT ALRIGHT TO COMPLETELY USE THE WHOLE LINE OF SOMEONE ELSES WORK???? WHOLE PARAGRAPHS??? I CAN REWORD IF NEED BE, BUT THE FIRST SENTENCE IS PRETTY GOOD IMO )

Proteus is a programming language and compiler being developed as a project for CSUN's Autonomy Research Center for STEAHM in collaboration with the NASA Jet Propulsion Laboratory (JPL). JPL system engineers needed a safer language to develop autonomous systems reliably, which is why Proteus was created. Proteus allows for the creation of different models: actors and hierarchical state machines. It is compiled to C++ with the C++17 standard [3].

Proteus is a programming language that follows the Actor model paradigm, which is somewhat related to the OOP paradigm. The difference lies in that Actor model allows for concurrent

computation, while OOP generally runs sequentially. This means that parallelism is inherently existent in the language. [NEED SOURCE HERE] Furthermore, because of the design of the events and event queue for Actors, any code involving them is run sequentially. This means that Proteus also supports the Procedural Programming langauge paradigm. Thus, Proteus is a Multi Paradigm language that enjoys the ability to utilize features such as scope, recursion, and so forth.

#### 1.4.1.1 Actors

Actors are independent entities within concurrent systems. By allowing several actors to operate independently, there is: no sharing of resources, concurrent runtime, and only interact amongst each other via a message system. Communication is asynchronous because the messages get buffered by the system until the recipient can handle them. Actors can send messages, modify local state, or create more actors based on the message handling.

#### 1.4.1.2 Hierarchical State Machines

Hierarchical State Machines allow developers to model the system that they are developing for. These HSMs are an extension to the standard definition of a state machine as HSMs allow states to be HSMs themselves. This allows for simplification of the states and transitions amongst states allowing simpler models for usage in the real-world.

Actors have $0^+$ HSMs while each HSM belongs to exactly one actor. Event Handlers are the way that messages are sent amongst machines as well as how the machine perform state transitions. Actors and states are statically defined, which means that they cannot be created nor destroyed at runtime. When compiled, Actors and states are created as C++ structs.

### 1.4.2 Proteus Grammar

Below is the Grammar for Proteus. It outlines the command followed by the definition for writing the command. Anything outlined in single quotations indicates text to be written explicitly. It is to be read as: 'OPERATION' can be written as 'HOW TO WRITE THE OPERATION'.

```
Program: DefEvent* DefGlobalConst* DefFunc* DefActor+
DefActor: 'actor' ActorName '{' ActorItem* '}'
ActorItem: DefHSM | DefActorOn | DefMember | DefMethod
DefActorOn: 'on' EventMatch OnBlock
DefHSM:   'statemachine' '{' StateItem* '}'
DefState: 'state' StateName '{' StateItem* '}'
StateItem: DefOn | DefEntry | DefExit | DefMember |
            DefMethod | DefState | InitialState
DefOn: 'on' EventMatch OnBody
EventMatch: EventName '{' [VarName (',' VarName)*] '}'
OnBody: GoStmt | OnBlock
OnBlock: Block
DefEntry: 'entry' '{' Block '}'
DefExit: 'exit' '{' Block '}'
DefMember: Type VarName '=' ConstExpr ';'
DefMethod: 'func' FuncName FormalFuncArgs ['->' Type] Block
InitialState: 'initial' StateName ';'
Block: '{' Stmt* '}'
Stmt: IfStmt | WhileStmt | DecStmt | AssignStmt | ExitStmt |
            ApplyStmt | SendStmt | PrintStmt | PrintlnStmt
DefEvent: 'event' EventName '{' [Type (',' Type )*] }' ';'
DefFunc: 'func' FuncName FormalFuncArgs ['->' Type] Block
DefGlobalConst: 'const' Type VarName '=' ConstExpr ';'
ExitStmt: 'exit' '(' NUMBER ')' ';'
ReturnStmt: 'return' Expr ';'
DecStmt: Type VarName '=' Expr ';'
AssignStmt: VarName '=' Expr ';'
```

```
ApplyStmt: ApplyExpr ';'

SendStmt : HSMName '!' EventName ExprListCurly ';'

PrintStmt : 'print' ExprListParen ';'

PrintlnStmt : 'println' ExprListParen ';'

FormalFuncArgs : '(' [Type VarName (',' Type VarName)*] ')'

ExprListParen :'(' [Expr (',' Expr)*] ')'

ExprListCurly :'{' [Expr (',' Expr)*] '}'

Type: 'int' | 'string' | 'bool' | 'actorname' | 'statename' |
            'eventname'

GoStmt: JustGoStmt | GoIfStmt

JustGoStmt: 'go' StateName Block

GoIfStmt: 'goif' ParenExpr StateName Block
            ['else' (GoIfStmt | ElseGoStmt)]

ElseGoStmt: 'go' StateName Block

IfStmt: 'if' ParenExpr Block ['else' (IfStmt | Block)]

WhileStmt: 'while' ParenExpr Block

ParenExpr: '(' Expr ')'

ConstExpr: IntExpr | BoolExpr | StrExpr

Expr: ValExpr | BinOpExpr | ApplyExpr

BinOpExpr: ValExpr BinOp Expr

BinOp: '*' | '/' | '%' | '+' | '-' | '<<' | '>>' | '<' | '>' |
            '<=' | '>=' | '==' | '!=' | '^' | '&&' | '||' | '*=' |
            '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '^='

ApplyExpr: FuncName ExprListParen

ValExpr: VarExpr | IntExpr | StrExpr | BoolExpr | ActorExpr |
            StateExpr | EventExpr | ParenExpr

VarExpr: VarName
```

```
IntExpr: NUMBER

StrExpr: STRING

BoolExpr: BOOL

ActorExpr: 'actor' ActorName

StateExpr: 'state' StateName

EventExpr: 'event' EventName

StateName: NAME

ActorName: NAME

FuncName: NAME

VarName: NAME

EventName: NAME
```

Looking at the grammar is similar to looking at the pieces of a puzzle without actually arranging the pieces together. Below is an example written in Proteus code that showcases Actors and HSMs in a system.

There are a total of 3 events: POWER_ON which accepts a boolean as input, POWER_OFF, and NEXT with the latter two not accepting an inputs. There are 2 actors: Main and Driver. Main has a single state machine with 2 states: On and Off. Main defines an internal boolean for whether Mode2 is enabled. By default it is initialized to false. The HSM within Main is initalized to Off, and switches to On when the POWER_ON event is registered. Furthermore, it updates the value for Mode2 being enabled with the input for POWER_ON. When turned On, there is a defined Mode1 that is the initial mode of the machine. It then defines what the machine does when it is turned on and off. In both cases, it outputs a message indicating the status of the power state of the machine (on prints on, and off prints off). Mode1 prints to the output the current Mode, and then has logic determining what to do when the NEXT message is received. If the machine has mode2_enabled set to true, then it should should go to Mode2. Mode2 simply prints the current mode when it is entered. The second actor is the Driver. It determines the actions to be taken by Main in a series of messages (events) that are broadcasted from its internal statemachine. Upon turning on

the machine, it will send the event for Main to turn on with an input of true. Then it sends Main the NEXT event twice. It then tells Main to power off with the POWER_OFF event.

```
event POWER_ON {bool};
event POWER_OFF {};
event NEXT {};
actor Main {
    bool mode2_enabled = false;
    statemachine {
        initial Off;
        state Off {
            on POWER_ON {x} {go On {mode2_enabled = x;}}
        }
        state On {
            initial Mode1;
            entry {println(\turning on");}
            exit {println(\turning off");}
            on POWER_OFF {} {go Off {}}
            state Mode1 {
                entry {println(\mode 1");}
                on NEXT {} {goif(mode2_enabled) {Mode2 {}}}
            }
            state Mode2 {
                entry {println(\mode 2");}
            }
        }
    }
}
```

```
actor Driver {
    statemachine {
        entry {
            Main ! POWER_ON {true};
            Main ! NEXT {};
            Main ! NEXT {};
            Main ! POWER_OFF {};
        }
    }
}
```

From the above example, we can see the OOP and Procedural Programming properties that Proteus is capable of. In fact, we can see the property of scope in action. The local variable mode2_enabled is only accessible within Main and the HSM within Main. Furthermore, we see that the order of events is run in sequence. Main accepts these events in the order received in the internal event queue, and is capable of responding based on the internal state conditions as well as the event received. When this code is run, the output is seen below:

```
turning on
mode1
mode2
mode1
turning off
```

The goal of this Thesis is to analyze this language to prove that it is TC. This is done by looking at the grammar and type of programming paradigm. With this understanding, we will also look at different approaches for proving Turing Completeness. Afterwards, we will demonstrate that Proteus is in fact turing complete using some of the methods seen in the next chapter.

# Chapter 2

## Different Approaches for Proofs to Demonstrate Turing Completeness

### 2.1 Overview

We will be exploring the different approaches to demonstrate TC for different systems. I have outlined the approaches based on their respective discipline, increasing in abstraction. With each discipline comes a more theoretical view and understanding of TMs and TC systems. My intention is to add clarity on the logic for these proofs/techniques. For example, in the Computer Engineering perspective, TM is created from its mechanical properties through the usage of logic gates. This is vastly different compared to how Mathematicians show TC, which is through the use of Lambda calculus – a model for representing mathematical logic. All proofs are equivalent in goal, however. These are not the only perspectives and types of proofs for showing Turing Completeness as well as TMs. This is simply a survey into what TMs and Turing Completeness looks like across the disciplines.

### 2.2 Computer Engineering

In this section, we will analyze what a TM looks like from a physical perspective. This may seem contradictory because the TM is described as a theoeretical machine. But in fact, the very computers that we use today are are capable of processing TC systems through the usage of programming languages. This means that they are limited TMs, because they are bounded only in memory. In this approach, we will look at the core components of Computer Engineering to create a TM.

### 2.2.1 Logical Design of a TM

To define what a TM does, we must explore what it is capable of. Recall Theorem 1 in 1.2.3, "Every effectively calculable function can be computed by a Turing Machine." Every effectively calculable function, as Turing and Church understood, was any mathematical calculation. This means that a TM must have some ability to perform any operation on numbers, such as the basic operations of addition, subtraction, multiplication, and division. Furthermore, they must be capable of combining these together to form more complex operations such as exponential arithmetic. Beyond the mathematical aspect, they must allow for some sort of logic handling. [BIOLOGY PAPER ON TM]

#### 2.2.1.1 Architecture

Looking at modern day computer architecture, there are several components that work independently but operate concurrently. It is based off of the Modified Harvard Structure which is a variation of the Harvard computer architecture and Von Neumann architecture. It combines both approaches towards computer architecture to handle many tasks that were difficult to handle using one of either architecture.

The von Neumann Architecture which has a centralized CPU to handle tasks for the computer. All processes are handled by the CPU directly. It contains several parts inside for processing data. Inside the CPU is an ALU with registers, as well as a Control Unit. The ALU processes arithmetic and logical computation, with the assistance of registers to store data at each step. The Control unit determines the commands to be given to the ALU and other parts of the computer. There is an associated Memory Unit which is where the bulk of memory storage lies. Outside of the CPU are the Input and Output devices.

citation for von neumann architecture image

The von Neumann architecture has several limitations, with one of the biggest criticisms being that it is bottlenecked by the throughput between the CPU and memory. Essentially, the CPU will eventually have more processing power than the bus can handle to write/read from memory. This

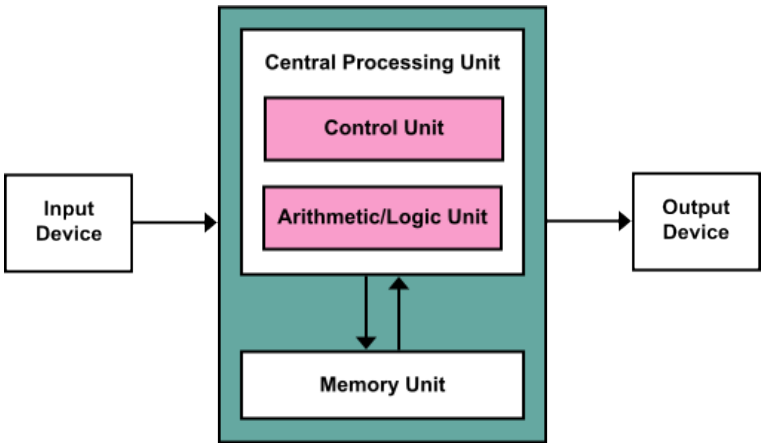


Figure 2.1: Von Neumann architecture.

causes the CPU to wait until the bus is freed to continue processing. As an alternative, we will now look at the Harvard Architecture

The Harvard architecture looks at the problem as says that if the CPU is too large and complex, then each individual component should be separated. This allows for the tasks to be distributed evenly amongst the several smaller components like the ALU and Instruction memory as opposed to having them live inside the CPU. The CPU is capable of simultaneous reads and writes. However, a similar bottleneck occurs where the bus connecting each of the components is the limiting factor. Below is a diagram demonstrating the Harvard Architecture.

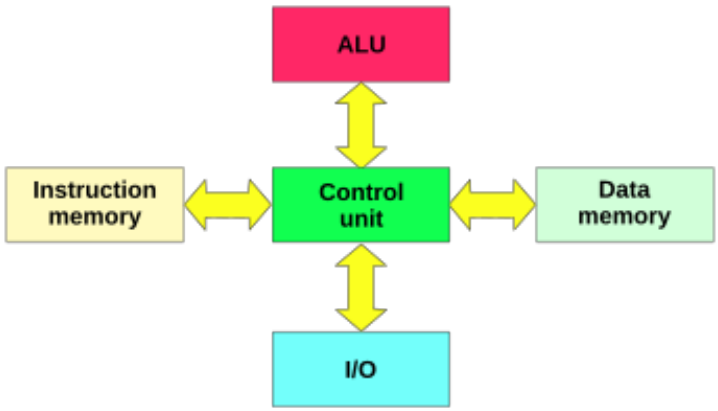


Figure 2.2: Harvard architecture.

citation for harvard architecture image

However, modern day computers utilize a mixed computer architecture called the Modified Harvard architecture. It combines both architectures into a single model. This usually is of the form of separating the components of the computer, but allowing several smaller memory caches for the CPU. This is why modern computers utilize several components such as the CPU, GPU, Main Memory (SDD or HDD) and so forth. Furthermore, this advancement allows for paralellism or multi-core systems to arise. Some notable examples include the NVIDIA RTX 4080 which has over 8000 cores SOURCE or for the intel i9-13900ks CPU to have 16 cores SOURCE. By allowing each one to independently operate, but still have the CPU as the ”brains” of the operation. We will now dive slightly deeper into the discussion to see what lies beneath these components within modern computer systems.

#### 2.2.1.2 Logic Gates

The basic building blocks for devices such as the ALU, CPU, and such are logic gates. These are simplistic logical components that allow for processing of data and performing operations on them.

The core of the CPU relies on the ALU. The ALU is contains registers which simply hold data inside. Registers are associated with an address for referencing purposes.

Within the ALU there are smallers components that perform specific operations such as addition and subtraction. These smaller components utilize logical gates such as the OR gate to compute the result with the given input. See the example below for a 2-Bit ALU that can process OR, AND, XOR, and addition calculations.

img src

### 2.2.2 Constructing the TM

With the ability to construct logic gates, we can create more complex components such as the ALU, CPU, and more. Accompanied with the ability to store memory, as well as have a way to interact with the system through Input and Output, we are able to create a functional computer. In

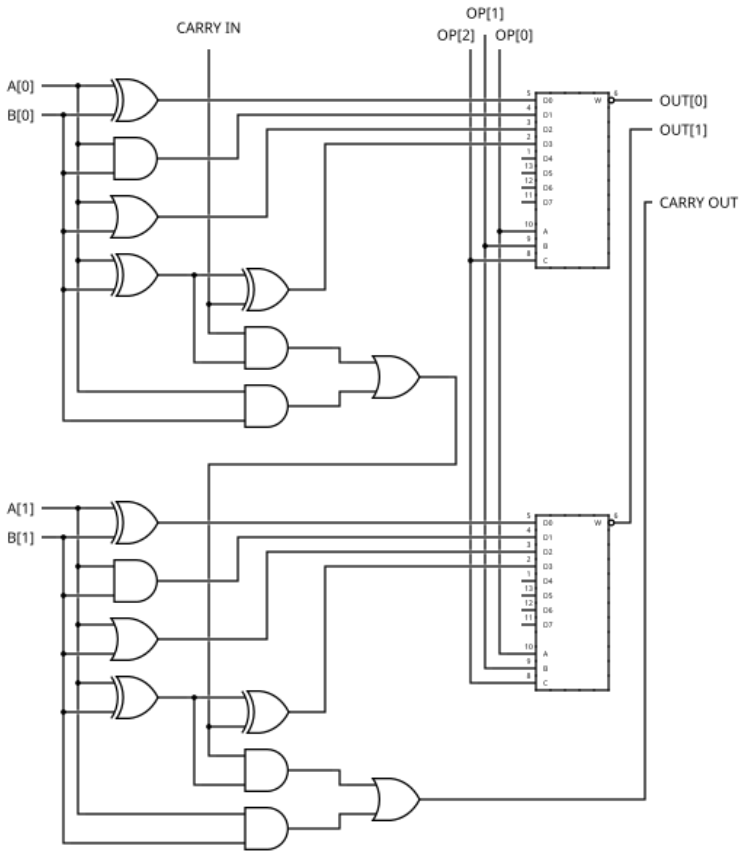


Figure 2.3: A 2-Bit ALU.

the following examples, developers have successfully created computers in the well-known video games of Minecraft and Terraria respectively, Computer in Minecraft Computer in Terraria. In fact, we will see later on that the computer built inside Terraria is verifiably TC via a method discussed in 2.3.2.1.

This means that to construct a TM on a physical level (of course alleviating the restriction of unbounded memory), these would be the minimum requirements.

SRC - https://www.nand2tetris.org/ SRC -

## 2.3 Computer Science

In this section, we will conceptualize what a TM looks like under the lens of Computer Science. There are 2 main perspectives: that of the Automata Theory and the Software Engineering ap-

proach. The Automata Theory approach utilizes theoretical designs more reminiscent of those listed by Turing and Gavin. The Software Engineering approach instead applies it to a problem to showcase Turing Completeness via programs and code.

### 2.3.1 Automata Theory

We will now abstract from the physical understanding of how to create a TM, to creating a theoretical one using Automata theory. In automata theory, Turing Machines are described using logical notation. The definition of a TM is stated within [4] and [13].

**Definition 1** *A Turing Machine M is defined by:*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$$

*where:*

$Q$ *is the set of internal states,*

$\Sigma$ *is the input alphabet,*

$\Gamma$ *is the finite set of symbols called the tape alphabet,*

$\delta$ *is the transition function,*

$\square \in \Gamma$ *is a special symbol called the blank,*

$q_0 \in Q$ *is the initial state,*

$F \subseteq Q$ *is the set of final states.*

The transition function $\delta$ is defined as

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}.$$

This means that for a given $\delta$ transition with inputs $q \in Q$ and $a \in \Gamma$, the tape will move to another state $x \in Q$, write nothing to the tape (indicated by $\square$) or some symbol $y \in \Gamma$, and choose to move the tape head Left one cell, Right one cell, or to Stay at the current cell. An example transition can

be written:

$$\delta(q_0, a) = (q_1, d, R)$$

where the internal state is $q_0$, and we read input token a. After the transition, we have internal state $q_1$, wrote symbol d onto the tape, and moved to the right one cell. See the below diagram demonstrating this change:

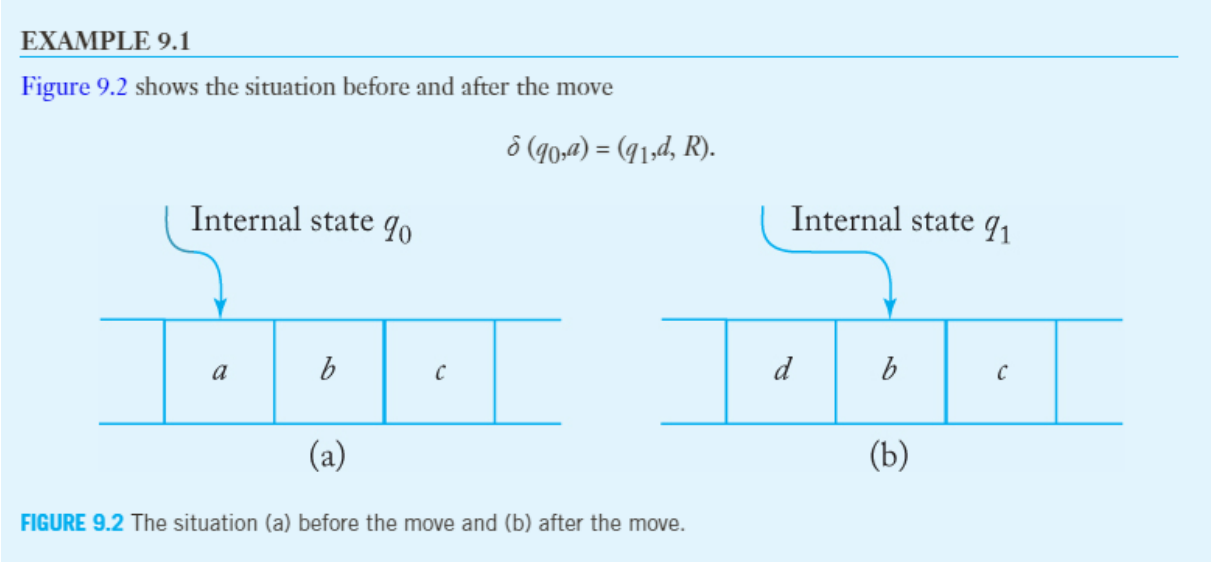


Figure 2.4: Delta transition example from [4].

Recall Figure 1.1 which represents a simplistic TM. In formal nomenclature, it can be written as follows:

$Q = \{q_0, q_1, q_2\}$ with associated labels {Even, Odd, Halt}

$\Sigma = 0, 1$

$\Gamma = 0, 1$

$F = \{q_2\}$

$q_0 \in$ Q as the initial state

and

$$\delta(q_0,0) = (q_2,1,S),$$

$$\delta(q_0,1) = (q_1,\square,R),$$

$$\delta(q_1,0) = (q_2,0,S),$$

$$\delta(q_1,1) = (q_0,\square,R).$$

#### 2.3.1.1 Notable examples using Formal language

The proofs constructed using formal language usually modify the given system to meet these requirements.

In "Magic: the Gathering is Turing Complete", the authors modified the way the game is understood between 2 players. They make the system force moves through clever leverage of the cards and their functions within the game [6].

In a different paper, "Turing Completeness and Sid Meier's Civilization", the system was also creatively modified to demonstrate Turing Completeness. In each game, they constructed UTMs by utilizing the layout of the maps, as well as mechanics for changing states of the roads within the game [7].

To show that Java Generics are TC, the authors showed that by creating a subtyping machine, it corresponds to only a small portion of the Java Generics while simulating TMs. Following this discovery, they simulate a TM and then show that the given inputs are undecidable [5]. I.e. leveraging an extension of Rice's Theorem, see **??**.

The idea of describing the system as a TM and showing it has an undecidable input is a common practice. This technique is also seen in a paper titled "The Game Description Language is Turing Complete" [8].

### 2.3.2 Software Implementation

As opposed to the various theoretical approaches seen previously in section 2.3.1, this section outlines a different perspective. Instead of constructing a TM within the compounds of the system,

an equivalent proof is to implement a program that demonstrates TC. By implementing any known TC program successfully implies that the overall system is TC.

One such implementation would be to create a functional example of a known TC cellular automata. Cellular automata are models of computation which use grids of cells. Each cell contains a finite number of states, belonging to only one at any given time. There are rules that determine what state a cell should become. These rules are applied to all cells simultaneously, and thus form the next step in the sequence. These steps are made sequentially to show the changes over time. This makes all cellular automata 0-player games, meaning after an initial configuration there is no further input from the user. With the work from Stephen Wolfram and other researchers such as Matthew Cook, some of these rules of cellular automata have been shown to be TC. Famous examples of cellular automata include Conway's Game of Life and Rule 110.

https://mathworld.wolfram.com/CellularAutomaton.html wiki cell automata

Cellular automata are sorted into 4 classes:

- Class 1: Nearly all initial patterns evolve quickly into a stable, homogenous state. Any randomness in the initial pattern disappears.

- Class 2: Nearly all initial patterns evolve quickly into stable or oscillating structures. Some of the randomness in the initial pattern may filter out, but some remains. Local changes to the initial pattern tend to remain local.

- Class 3: Nearly all initial patterns evolve in a pseudo-random or chaotic manner. Any stable structures that appear are quickly destroyed by the surrounding noise. Local changes to the initial pattern tend to spread infinitely.

- Class 4: Nearly all initial patterns evolve into structures that interact in complex and interesting ways, with the formation of local structures that are able to survive for long periods of time. Class 2 type stable or oscillating structures may be the eventual outcome, but the number of required to reach this state may be very large, even when the initial pattern is realtively simple. Local changes to the initial pattern may spread indefinitely.

Wolfram conjectured that many class 4 cellular automata are capable of universal computation. Both Conway's Game of Life and Rule 110 exhibit "Class 4 behavior" and have been proven to be Turing Complete [9].

Wiki src SRC - Cellular Automata: A Discrete Universe by Andrew Ilachinski

#### 2.3.2.1 Conway's Game of Life

Conway's Game of Life is a 2D grid of cells extending infinitely in the x and y directions. Each cell contains only 2 states: Alive (On) or Dead (Off). The rules of CGoL are simple:

1. Any live cell with fewer than two live neighbors dies. (Underpopulation)
2. Any live cell with two or three live neighbors lives on to the next generation. (Survival)
3. Any live cell with more than three live neighbors dies. (Overpopulation)
4. Any dead cell with exactly three live neighbors becomes a live cell. (Reproduction)

They are demonstrated in the following graphic:

src for graphic

CGoL is considered undecidable. This is because given any initial pattern and a desired pattern at some later generation, there is no algorithm to determine whether the desired pattern will exist. As such, it is analogous to the Halting Problem.

#### 2.3.2.2 Rule 110

Whereas CGoL is created on a 2D plane, Rule 110 lives in the 1D space. There is an infinite tape of cells that each may exist in one of two states: 0 or 1. By looking at three cells in series, one can find what the next state of the middle cell will be. Below are the rules for Rule 110:

1. 111 makes 0
2. 110 makes 1

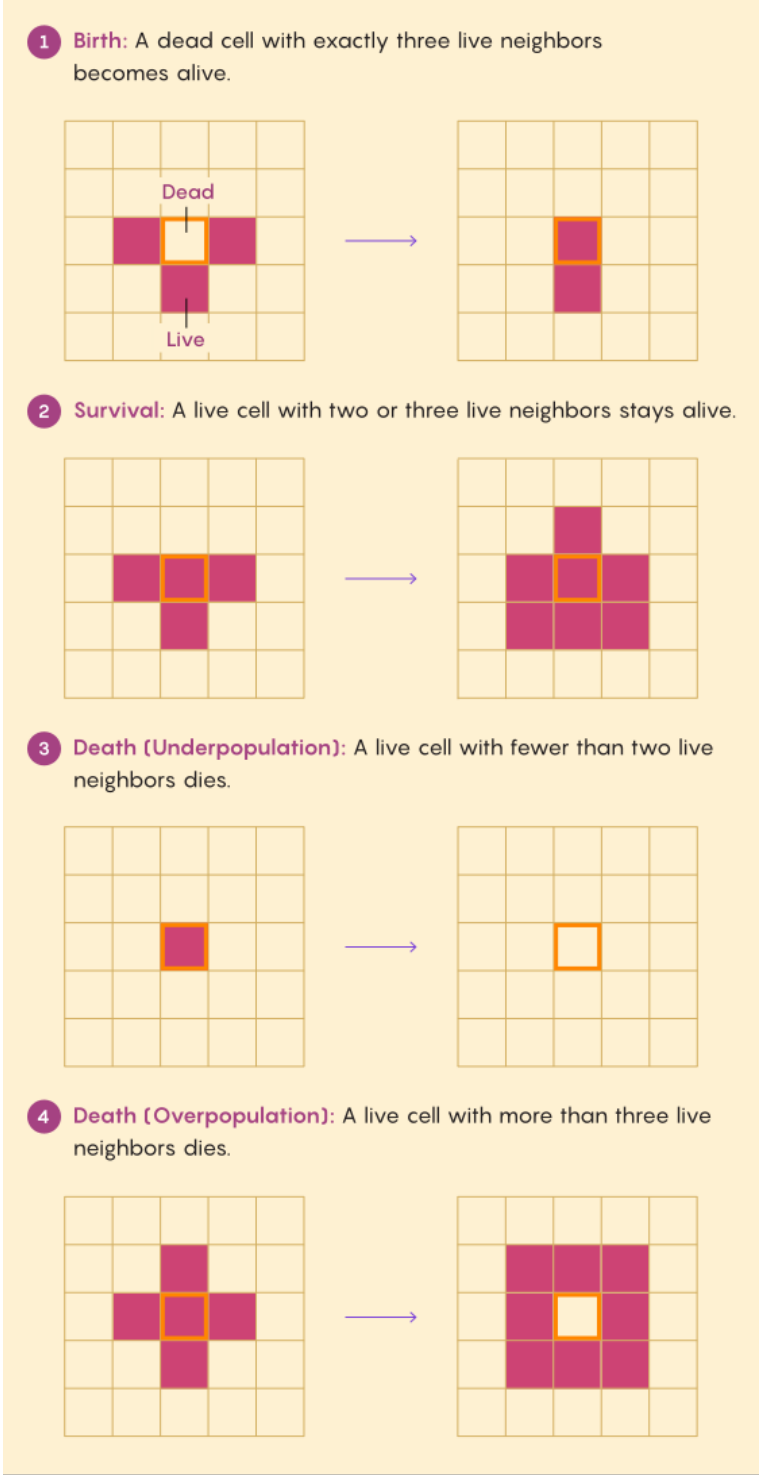


Figure 2.5: rules of conway's game of life visualized

3. 101 makes 1

4. 100 makes 0

5. 011 makes 1

6. 010 makes 1

7. 001 makes 1

8. 000 makes 0

Here is an associated graphic:


Figure 2.6: Rules for Rule 110

Rule 110 is one of the simplest TC system that is known. This makes it a relatively easy system to create to demonstrate Turing Completeness as opposed to CGoL.

SRC - WIKI page for graphic

#### 2.3.2.3 Programmable Calculator

An entirely different approach to create a program that demonstrates Turing Completeness is to model the behavior of TMs directly. This means that you create a system that does everything that a TM can do. Recalling the Church-Turing Thesis (Theorem1), it must be able to calculate any function. In a basic sense, this means that the system is capable of:

- Reading/Writing memory
- Elementary Arithmetic/Logical operations
- Conditional Logic
- Looping Logic

Programmable Calculators meet all of these requirements. By being able to store values into variables which can be referenced later, it can read/write memory. Because it is a calculator, it is capable of performing arithmetic operations. If statements and while loops are sufficient for handling the conditional and looping logic. A programmable calculator thefore is TC.

A simplistic set of steps is outlined below:

1. Start by making a basic arithmetic calculator.
2. Then add the ability to store values into variables.
3. Afterwards, create functionality for if statements, allowing boolean logic.
4. Finally, create looping logic with while statements.

TC language in Ruby using Bable-Bridge

This much simpler approach is easier to digest and reason out with some software engineering design principles.

#### 2.3.2.4 Interpreter for a known Turing Complete language

Alternatively, to show a programming language is TC, one can create an interpreter for a known TC language. Many programming languages feature complex grammars and rulesets, which is why TC esoteric programming languages are preferred. In fact brainfuck, as seen in section 1.3.1.1, is used to demonstrate TC for its concise ruleset.

c bf interpreter: https://thesharperdev.com/how-to-write-a-brainfuck-interpreter-in-c/

novel lang meep that compiles to bf and thus is TC: https://github.com/srijan-paul/meep https://injuly.in/blog/bf

python bf interpreter: https://martin-ueding.de/posts/creating-a-brainfuck-interpreter/

### 2.4 Mathematics

In this section, I will take a look at the mathematical system that is most well known for being TC, Lambda Calculus. This is an abstract form of understanding functions and their capabilities. It

was actually designed by Church, and proven to be TC later on based off the work of Turing and Church by a famous mathematician: Stephen Cole Kleene [14].

### 2.4.1 Lambda Calculus

Lambda calculus upon initial inspection seems like a very abstract form of functions and relations within mathematics. It can be understood to those in Computer Science as a very abstract programming language, and actually forms the basis of Functional Programming Languages [16]. SRC - functional programming.

Lambda Calculus is a form of expressing functions in a simple manner that allows for creating any complex system [15]. At its core, it consists of three inductive rules defining what lambda terms are. Each lambda term is a valid statement in lambda calculus:

1. $x$: A **variable** to represent a character or string. This is to be understood as a parameter for functions.

2. $\lambda x.M$: A lambda **abstraction** that is a function definition. This function takes the bound variable $x$ as input, and returns the body $M$.

3. $(M\ N)$: An **application** where it applies the function $M$ to argument $N$.

There also exist reduction operations to improve legibility but retain equivalent logical meaning:

1. $(\lambda x.M[x]) \rightarrow (\lambda y.M[y])$: $\alpha$-conversion, which renames the bound variables in the expression. This is be used to avoid name collisions.

2. $((\lambda x.M)N) \rightarrow (M[x := N])$: $\beta$-reduction, which replaces bound variables with the argument expression in the body of the abstraction. This is used to simplify chained functions being written out.

Parentheses may be used to to disambiguate terms from each other. This is especially useful when constructing complex applications using lambda calculus.

SRC - Lambda Calculus wiki

I will define an equivalent TM to the previously mentioned TM seen in Figure 1.1 and in section 2.3.1. Recall that the goal of the TM was to determine if there are an even or odd number of '1's in a sequence.

We construct the list of Natural Numbers, $\mathbb{N}$, as follows:

$$0 \equiv \lambda sz.s(z)$$

$$1 \equiv \lambda sz.s(s(z))$$

$$2 \equiv \lambda sz.s(s(s(z)))$$

and so on...

Now we construct the ideas of Arithmetic Boolean Logic, and other necessary logical operators. Treat 'f' as a function and variables as only locally defined to their respective operator. These examples can be seen here: SRC. Some notation can be interpreted as SKI combinator calculus SRC.

EN - This should be cut so that it doesnt extend past the confines of the page

$$K := \lambda xy.x \equiv X(X(XX)) \equiv X'X'X'$$

$$S := \lambda xyz.(xz)(yz) \equiv X(X(X(XX))) \equiv XK \equiv X'(X'X')$$

$$I := \lambda x.x \equiv SKS \equiv SKK \equiv XX$$

$$Y := \lambda g.(\lambda x.g(xx))(\lambda x.g(xx))$$

$$
\begin{aligned}
SUCC &:= \lambda nfx.f(nfx) \\
PRED &:= \lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u) \\
&\equiv \lambda n.n(\lambda gk.\ ISZERO\ (g\ 1)k\ (PLUS(g\ k)1))(\lambda v.0)0 \\
PLUS &:= \lambda mnfx.nf(mfx) \\
&\equiv \lambda mn.n\ SUCC\ m \\
SUB &:= \lambda mn.n\ PRED\ m \\
MULT &:= \lambda mnf.m(n\ f) \\
&\equiv \lambda mn.m(PLUS\ n)0 \\
DIV &:= \lambda Y(\lambda gqab.\ LT\ a\ b\ (PAIR\ q\ a)(g\ (SUCC\ q)(SUB\ a\ b)\ b))0 \\
MOD &:= \lambda ab.\ CDR\ (DIV\ a\ b) \\
TRUE &:= \lambda xy.x \equiv K \\
FALSE &:= \lambda xy.y \equiv 0 \equiv \lambda x.I \equiv KI \equiv SK \equiv X(XX) \\
NOT &:= \lambda pab.pba \equiv \lambda p.p\ FALSE\ TRUE \\
ISZERO &:= \lambda n.n(\lambda x.\ FALSE)\ TRUE \\
LT &:= \lambda ab.\ NOT\ (LEQ\ b\ a) \\
LEQ &:= \lambda mn.\ ISZERO\ (SUB\ n\ m) \\
PAIR &:= \lambda xyf.fxy \\
CAR &:= \lambda p.p\ TRUE \\
CDR &:= \lambda p.p\ FALSE \\
NIL &:= \lambda x.\ TRUE \\
NULL &:= \lambda p.p(\lambda xy.\ FALSE) \\
LENGTH &:= Y\lambda(gcx.\ NULL\ xc(g\ (SUCC\ c)\ (CDR\ x)))0
\end{aligned}
$$

Now we can combine these lambda functions from a higher abstraction level to perform the operation.

```
Obtain the Length of the List.
With the list length, subtract 1 from it.
Take the mod of the result.
If the new result is 0, then that means it was even.
If instead it was 1, then it was odd.
```

Resulting in the following simplified lambda calculus operation:

$$MOD\ (SUB\ (LENGTH\ (\textbf{input})\ 1))\ 2$$

with **input** being the input string. One can expand this result to the above lambda calculus notation, resulting in an extraneously long sequence. See the below image for an example of an expanded lambda calculus function that determines if a number is even or odd, i.e. it's cardinality SRC - REDDIT PROGRAMMER HUMOR.

$$(\lambda m.(\lambda g.(\lambda x.g(\lambda a.x(x)(a)))(\lambda x.g(\lambda a.x(x)(a))))(\lambda f.(\lambda n.(\lambda a.(\lambda b.(\lambda p.(\lambda a.(\lambda b.p(b)(a))))$$
$$((\lambda m.(\lambda n.(\lambda n.n(\lambda x.(\lambda a.(\lambda b.b)))(\lambda a.(\lambda b.a)))((\lambda m.(\lambda n.n((\lambda n.(\lambda f.(\lambda x.n(\lambda g.(\lambda h.h(g(f))))$$
$$(\lambda u.x)(\lambda u.u)))))(m)))(m)(n))))(b)(a))))(n)(\lambda f.(\lambda x.f(f(x))))(\lambda j.(\lambda n.n(\lambda x.(\lambda a.(\lambda b.b)))$$
$$(\lambda a.(\lambda b.a)))(n))(\lambda j.f(j))((\lambda m.(\lambda n.n((\lambda n.(\lambda f.(\lambda x.n(\lambda g.(\lambda h.h(g(f))))(\lambda u.x)(\lambda u.u)))))(m)))$$

Figure 2.7: Expanded lambda calculus function to determine the cardinality of a number.

With the ability to define any calculable function, Lambda Calculus is TC, as stated in the Church-Turing Thesis, Theorem 1.

# Chapter 3

# Proteus is Turing Complete

This section will describe how we will construct the proof showing that Proteus is TC.

## 3.1 Useful information to be used in the proof

First, I will discuss features about Proteus programs on a theoretical level. Then, I will discuss features about the Proteus language that allow for the creation of a TM. This is the background information that will guide the construction of the proof outline and ultimately the proof itself.

### 3.1.1 Undecidable input

Because Proteus is a higher-level programming language, we can leverage the usage of Rice's Theorem, Theorem 2. Thus, given any input it is impossible to determine an answer to the Halting Problem. Furthermore, one cannot determine if there is an actor that will be told to switch to a particular state. With this knowledge, it is understood that any given Proteus program is undecidable. Thus, I will look at how to create a TM in Proteus.

### 3.1.2 Requirements of a TM

In this section, I will point out critical pieces of Proteus that prove useful to create a TM. We can see that the core features to create a TM, seen previously in sections 2.2.1 and 2.3.2.3, include:

1. Arithmetic and Logical Processing
2. Memory storage and manipulation
3. Conditional Logic

4. Looping Logic

5. Input/Output

Recall the proteus grammar seen in section 1.4.2. I will now describe from the Proteus grammar how to construct/use Proteus creating each part of the TM.

#### 3.1.2.1 Arithmetic and Logical Processing

The grammar provides the following definitions for arithmetic and logical processing:

- BinOp
- Type
- ConstExpr

'BinOp' handles all binary operations for both arithmetic and logical calculation. Some features include addition, subtraction, multiplication, division, modular arithmetic, equivalence relations, and, and or. Looking at brainfuck in section 1.3.1.1, one can notice that the only necessary mathematical operations are addition and subtraction. Furthermore, the only logical processing is seen in the looping mechanism. If the value at the pointer is 0 and the input token is a '[', then the loop is skipped. This means that there is an equality check which returns a boolean result.

Looking deeper at the types of Proteus, type consists of all the possible types that are built into the language:

- int
- string
- boolean
- actorname
- statename

- eventname

Despite allowing for division, the set of integers is closed under truncation, which is how Proteus handles cases where normally it wouldn't be. eg. 5 / 2 = 2.5, but under truncation 5 / 2 = 2. These truncation rules are similar to those seen in other languages such as Java and C, in JAVA in C.

'ConstExpr' describes the 3 simple data types: Int, String, and Boolean. These 3 types are capable of mimicking the behavior of brainfuck as well.

#### 3.1.2.2 Memory Storage and Manipulation

The grammar provides the following definitions for memory storage and manipulation:

- DefHSM
- DefState
- DefGlobalConst
- DecStmt
- AssignStmt
- SendStmt

'HSM' are Hierarchical State Machines which are actors in the language. These state machines utilize states to determine logical processing. These logical processes may utilize local or global variables that are stored, via the 'DecStmt' and 'DefGlobalConst' definitions respectively.

To modify data, the 'AssignStmt' was defined which allows for modifying the value of a given variable. State Machines can modify state via the 'SendStmt' command. Utilizing 'SendStmt', state machines can modify the state of themselves and other state machines as well.

#### 3.1.2.3 Conditional Logic

The grammar provides the following definitions for conditional logic:

- GoStmt
- JustGoStmt
- GoIfStmt
- ElseGoStmt
- IfStmt

Conditional Logic or Branching is necessary for a TM to compute any calculable function (see: Theorem 1). 'GoStmt' is considered either a 'JustGoStmt' or a 'GoIfStmt', which are used to switch between states of a given HSM. Similarly, the 'ElseGoStmt' switches to a particular state of a given HSM if the condition from the 'GoIfStmt' fails.

The 'IfStmt' is utilized for conditional logic within the processing of the state machines, and is akin to the standard if statements in other programming languages. It is defined recursively to allow for nested "If ... else if.... else ..." statements. These definitions allow for conditional statements to occur for a given HSM and within the code itself.

#### 3.1.2.4 Looping Logic

The only looping logic that can be seen in the garmmar that is built in, is the:

- WhileStmt

This is the only necessary form of looping, as it can be broken by conditional statements and is capable of performing like other loops such as the do-while, for, and so forth. This allows for more complex logical processing, such as recursion, which is a necessary requirement for TMs to perform any calculation. A simplistic example of a problem that requires recursion would be the

Ackermann Function. See the definition of the Ackermann function here:

$$A(0,n) = n+1$$

$$A(m+1,0) = A(m,1)$$

$$A(m+1,n+1) = A(m,\ A(m+1,n))$$

Although being able to compute the Ackermann function requires recursion, it doesn't conclude that any system that can compute it is TC. It was created to show that not all total computable functions are primitively recursive [17]. The Ackermann function exists to show that not all functions can be represented with for loops, which is what primitive recursive functions are [18]. Nonetheless, all computable functions (regardless of their expression) are capable of being calculated by a TM, as stated by the Church-Turing Thesis (Theorem 1).

#### 3.1.2.5 Input/Output

Looking at the grammar definitions for:

- Stmt
- PrintlnStmt
- PrintStmt
- SendStmt

From 'Stmt' I would like to highlight the 'SendStmt' command. 'SendStmt' is utilized to send events to a particular State Machine (i.e. an output). By default, all actors are able to receive events. 'PrintlnStmt' and 'PrintStmt' are the standard print and println commands that are well known from other languages which serve as output to the console. Although there is no explicit way to allow for input from the systems grammar dynamically, this is unnecessary as it can be preconfigured before runtime. Thus, there exists a way to send inputs before the program is run via static input of values.

## 3.2 Proteus Turing Machine Description

By showing that any input to Proteus programs are undecidable and it is possible to create a TM in Proteus, Proteus can be shown to be TC. This proof leverages the usage of both the Church-Turing Thesis, Theorem 1, and Rice's Theorem, Theorem 2.

I will explicitly create a TM using the built-in features seen previously in section 3.1.2. After showing how to create a TM within Proteus, I will use Proteus to implement Conway's Game of Life and Rule 110 . This is to demonstrate that the system is TC. Recall demonstrating an implementation of CGoL or Rule110 indicates the system is TC from sections 2.3.2.1 and 2.3.2.2.

1. Define the set of internal states
2. Define the initial state
3. Define the final state
4. Define the input alphabet
5. Define the tape alphabet
6. Define the state transitions
7. Define the blank symbol

A Proteus program will simulate the TM by having several parts. The tape is a series of state machines, HSMs, that will be ordered as $c_0, c_1, \ldots, c_n$ arbitrarily with $c_n$ being the last non-empty cell. This order will be consistent and not allow state machines to swap places with each other in the sequence. There will be an additional state machine which functions as the read/write head. This state machine will be the one describing what the state of the TM and overall program is. It contains a queue of events to be broadcast, with each entry in the queue containing a single event and target state machine. Below is an image describing the system as a whole, followed by another image which shows the states and logical flow of the proposed TM.

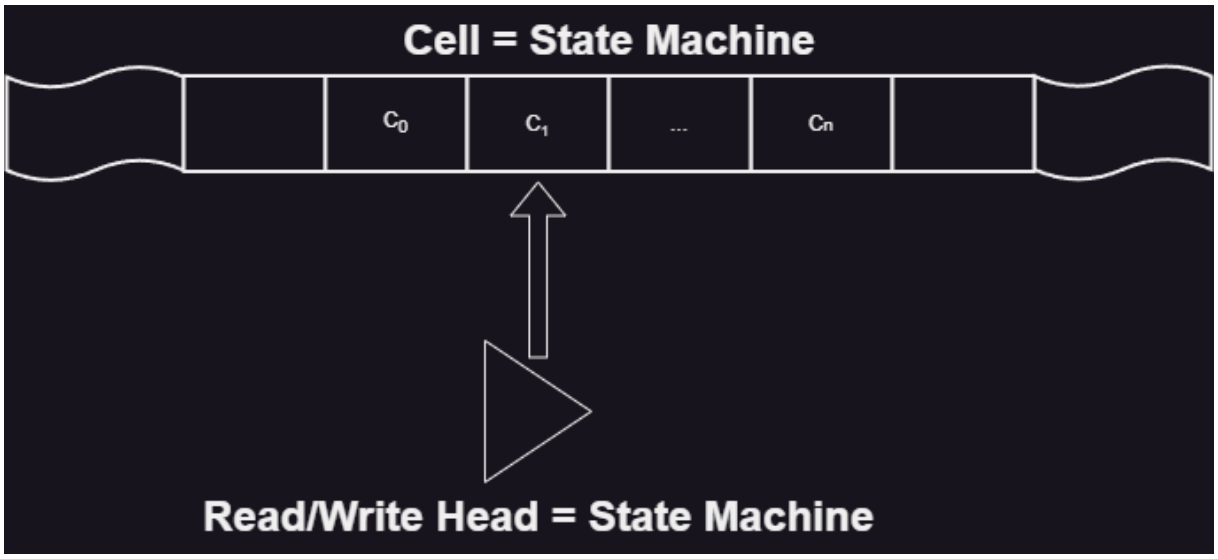


Figure 3.1: Design of TM in Proteus

When the Proteus program is run, the read/write head will enter the 'ProgramOn' State. If the tape is empty, as in there are no state machines that are created by the programmer, then the read/write head enters the 'ProgramOff' state and halts. If instead the cell is non-empty, then the read/write head enters the 'Read' state which begins the process of reading information from the tape. If a write is to be issued, then the read/write head enter the 'Write' state and writes the new data in the current cell. After the write, the read/write enters the 'Read' state once again.

The logic for movement to an adjacent cell is mirrored on the left and right sides. I will describe the movement to the left-adjacent cell. From the current cell, the read/write head enters the 'BoundLeft' state and determines whether it encounters another symbol or the blank. If there is a symbol, then it still lies within the non-empty tape information and returns to the 'Read' state. If instead there is a blank, this means that there is no state machine defined, and has extended past the bounds of the tape with given information. The read/write head moves one cell to the right, then returns to the 'Read' state to continue processing. Because Proteus does not allow for dynamic state machine creation, the read/write head leaves the blank unmodified along the tape.

To exit the program and enter the halting state, all state machines within the tape must enter the 'Off' state, indicated by the 'O' within Figure 3.2. The read/write head enters the 'FindStart' state from the 'Read' state to prepare for halting. In the 'FindStart' state, the read/write head will move

to the left, one cell at a time. When it encounters a blank cell, it moves the read/write head to the adjacent right cell and enters the 'Check' state. In the check state, the read/write head moves one cell at a time to the right and checks if they are in the 'Off' state. Upon encountering a cell that is not in the 'Off' state the read/write head enters the 'Read' state for further processing. If every cell is in the 'Off' state, then the read/write head will encounter a blank on the next cell after $c_n$. In this case, all state machines are in the 'Off' state and the read/write head enters the 'ProgramOff' state to halt.

Because the read/write head is itself a state machine, it contains an event queue for events to be broadcasted. Each event is associated to a single state machine. In order to find the proper state machine to send the event to, the read/write head must search for it within the bounds of the non-empty tape. This searching cannot utilize 'FindStart', because if all state machines are off and there are some nonzero number of events still in the queue, then the machine will still have events to process, but end up in the 'ProgramOff' state. As such, it must search for them and find them using an unoptimized algorithm such as brute forcing all possible movements across the non-empty tape. Note that I will explicitly describe what values $x, y,$ O must be in the following section 3.3.

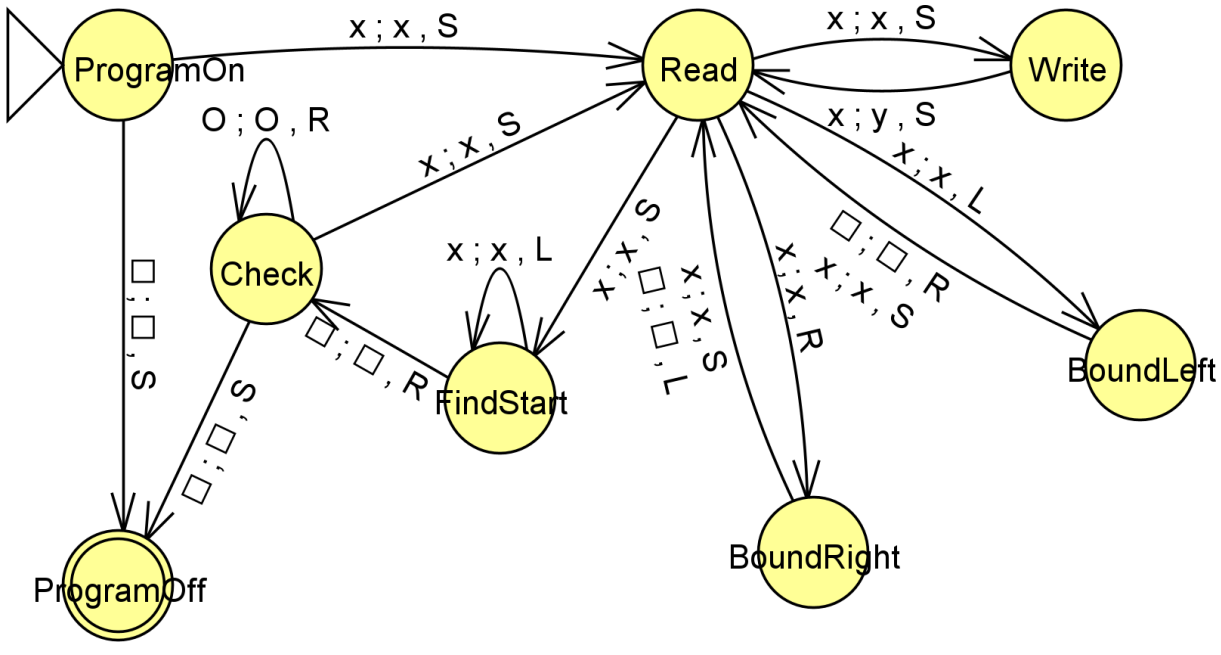


Figure 3.2: TM State Diagram in Proteus

### 3.3 Proof

To begin, I will describe the set of internal states of the TM, specifically of the state machine that is the read/write head. There are two internal states that define the current status of the read/write head: 'ProgramOn' and 'ProgramOff'. 'ProgramOn' means that the program is currently running, while 'ProgramOff' means that the program has halted. The read/write head also contains the following additional states:

- Read state: Reads the current state of the cell at the read/write head.
- Send state: Broadcasts the event at the front of the event queue to the current cell.
- Move state: Moves the read/write head to the next/previous cell in the tape.

Let the set of internal states, be defined as follows:

$$Q = \{\text{'ProgramOn'}, \text{'ProgramOff'}, \text{'Read'}, \text{'Send'}, \text{'Move'}\}$$

$q_0$ is a state demonstrating that the program is currently running, i.e. the initial state. To differentiate 'On' for the program (read/write head) versus the cell, the read/write head's initial state is defined as 'ProgramOn'. Similarly, I named the final state of the Program as 'ProgramOff' for consistency. Fis the set of halting states, and in this case contains only the element 'ProgramOff'. Therefore, the initial state of the program and the halting state of the program can be described as:

$$q_0 = \text{'ProgramOn'}$$

$$F = \{\text{'ProgramOff'}\}$$

The input alphabet consists of the symbols that appear as already existing on the tape. Recall that each cell is a state machine in Proteus, seen in section **??**. The starting states for each cell (state machine) will be one of the following states: 'On' or 'Off'. With this information we have the input alphabet:

$$\Sigma = \{\text{'On'}, \text{'Off'}\}$$

The symbols that can be written to and from the tape consist of the states within each state machine. These are user defined, but also include the previously defined states: 'On' and 'Off'. I will assume there is some number of states $n \in \mathbb{Z}_{\geq 0}$ indicating that there are $0^+$ additional states designed for each state machine by the programmer. Each state, $s_i$, are programmer created states which are not the 'On' or 'Off' states. All states must output what the current state is of the state machine as a string. This is to ensure that the read/write head is capable of knowing what state at the current location. The last symbol to be included is the blank. The blank represents that the cell has not changed state. Thus, the list of symbols that can be written to and from the tape is as follows:

$$\Gamma = \{\text{'On'}, \text{'Off'}, s_0, \ldots, s_n, \square\} \text{ for } n \in \mathbb{Z}_{\geq 0}$$

The transition function is what allows the read/write head to change states. In order to change the state of the overall program from 'ProgramOn' to 'ProgramOff' all internal state machines must be turned off. This means that all non-blank cells on the tape must be in the 'ProgramOff' state. Therefore, we can define transition states as:

DEFINE ALL THE TRANSITIONS FOR EACH OF THE INTERNAL STATES: READ, SEND, MOVE, PROGON, PROGOFF

$$\delta = \text{hahahahaha}$$

The blank symbol refers to a lambda transition, similar to those used in JFLAP (see section 1.2 for information on JFLAP). These blank symbols allow for transitioning between states without changing the internal state, or writing to the tape.

In summary, the following definitions create a TM for an arbitrary Proteus program:

$$Q = \{\text{'ProgramOn'}, \text{'ProgramOff'}, \text{'Read'}, \text{'Send'}, \text{'Move'}\}$$

$$F = \{\text{'ProgramOff'}\}$$

$$q_0 = \text{'ProgramOn'}$$

$$\Sigma = \{\text{'On'}, \text{'Off'}\}$$

$$\Gamma = \{\text{'On'}, \text{'Off'}, s_0, \ldots, s_n, \square\} \text{ for } n \in \mathbb{Z}_{\geq 0}$$

with the transition functions:

$$\delta(q_0, 0) = (q_2, 1, S),$$

$$\delta(q_0, 1) = (q_1, \square, R),$$

$$\delta(q_1, 0) = (q_2, 0, S),$$

$$\delta(q_1, 1) = (q_0, \square, R).$$

### 3.4 Implementing Conway's Game of Life

Implementation of CGoL that is a demonstration of TC.

### 3.5 Implementing Rule 110

Implementation of Rule 110 that is a demonstration of TC.

# Chapter 4

## Conclusion

Succinctly describe what techniques were used. Compare and constrast them? Perhaps a discussion section?

## Bibliography


[1] Q. Gao and X. Xu, "The Analysis and Research on Computational Complexity," pp. 3467–3472.

[2] R. Gandy, "Church's Thesis and Principles for Mechanisms," The Kleene Symposium, pp. 123–148, Jun. 1980.

[3] B. McClelland, "Adding Runtime Verification to the Proteus Language," CSUN, May 2021.

[4] P. Linz, An Introduction to Formal Languages and Automata. Jones & Bartlett Learning, 2016.

[5] R. Grigore, "Java generics are turing complete," ACM SIGPLAN Notices, vol. 52, no. 1, pp. 73–85, Jan. 2017, doi: https://doi.org/10.1145/3093333.3009871.

[6] A. Churchill, S. Biderman, and A. Herrick, "Magic: The Gathering is Turing Complete."

[7] A. de Wynter, "Turing Completeness and Sid Meier's Civilization," IEEE Transactions on Games, vol. 15, no. 2, pp.292-299, June 2023.

[8] A. Saffidine, "The Game Description Language Is Turing Complete," IEEE Transactions on Computational Intelligence and AI in Games, vol. 6, no. 4, pp. 320–324, Dec. 2014, doi: https://doi.org/10.1109/tciaig.2014.2354417.

[9] P. Rendell, "A Universal Turing Machine in Conway's Game of Life," 2011 International Conference on High Performance Computing & Simulation, Istanbul, Turkey, 2011, pp. 764-772, doi: 10.1109/HPCSim.2011.5999906.

[10] S. S. T. Gontumukkala, Y. S. V. Godavarthi, B. R. R. T. Gonugunta and S. M., "Implementation of Tic Tac Toe Game using Multi-Tape Turing Machine," 2022 International Conference on Computational Intelligence and Sustainable Engineering Solutions (CISES), Greater Noida, India, 2022, pp. 381-386, doi: 10.1109/CISES54857.2022.9844404.

[11] Jeffrey Outlaw Shallit, A second course in formal languages and automata theory. Cambridge ; New York: Cambridge University Press, 2009.

[12] M. Biçer, F. Albayrak and U. Orhan, "Automatic Automata Grading System Using JFLAP," 2023 Innovations in Intelligent Systems and Applications Conference (ASYU), Sivas, Turkiye, 2023, pp. 1-4, doi: 10.1109/ASYU58738.2023.10296744.

[13] E. Luce and S. H. Rodger, "A visual programming environment for Turing machines," Proceedings 1993 IEEE Symposium on Visual Languages, Bergen, Norway, 1993, pp. 231-236, doi: 10.1109/VL.1993.269602.

[14] Dezani-Ciancaglini Mariangiola and J. R. Hindley, "Lambda-Calculus," Wiley Encyclopedia of Computer Science and Engineering, pp. 1–8, Sep. 2008, doi: https://doi.org/10.1002/9780470050118.ecse212.

[15] M. Dezani-Ciancaglini and J. R. Hindley, “Lambda-Calculus,” Nov. 2007, Available: Here

[16] R. Rojas, “A Tutorial Introduction to the Lambda Calculus,” 2015, Available: Here

[17] CWoo, “Ackermann function is not primitive recursive,” Mar. 2013, Available: https://www.cs.tau.ac.il// nachumd/term/42019.pdf

[18] W. Dean and A. Naibo, Recursive Functions, Fall 2024 Edition. Stanford University: Metaphysics Research Lab, Stanford University, 2024. Available: https://plato.stanford.edu/archives/fall2024/entries/recursive-functions/