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:

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

California State University, Northridge

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

Abstract

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

By

Isaiah Martinez

Master of Science in Computer Science

General purpose programming languages are provably Turing Complete. Proofs showing a given system is Turing Complete come in many varieties depending on the discipline and perspective of the system. Proteus is a programming language developed for the Autonomy Research Center for STEAHM at California State University, Northridge in collaboration with the NASA Jet Propulsion Laboratory. This thesis looks into different methods of demonstrating a given system is Turing Complete and applying them to Proteus. After completing the proof, there is a reflection of the methods discussed and utilized.

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, followed by code implementations to demonstrate that fact.

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 Turing Machine. A Turing machine (TM) can be described as a machine/automaton 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 (DTM) and Non-Deterministic (NDTM). Regardless of such a construction, both are equivalent in power, and as such remain interchangeable until implementation [1]. Assume all TMs discussed henceforth are Deterministic unless otherwise stated, i.e. DTMs.

Figure 1.1 is an example of a TM designed using JFLAP, see the official websiteJFLAP for more information on this software (https://www.jflap.org/). 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, i.e. its parity.

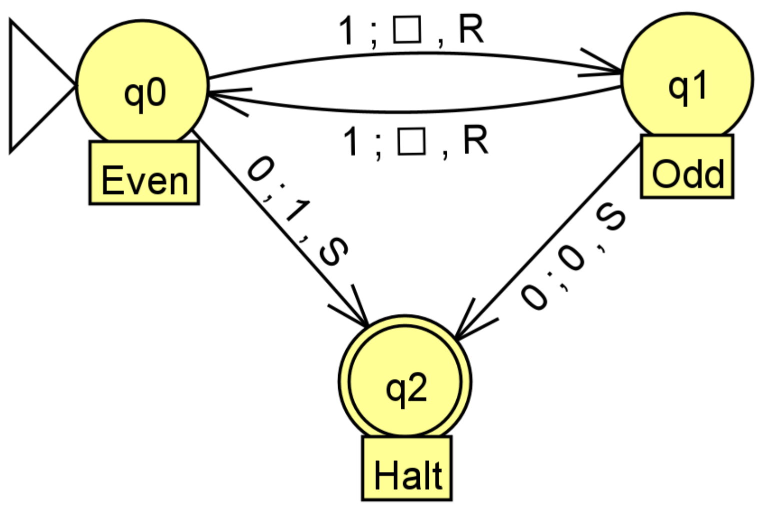

Figure 1.1: A TM that determines the parity of the length of a sequence of 1's on the tape.

1.2.1 Oracles

There also exists the Oracle, sometimes called Turing Machines with an oracle, which is capable of solving problems that TMs cannot. The TM is given a problem and then consults the oracle which is said to contain the answer to any problem associated with the TM. The oracle has the ability to solve the problem and returns that information to the TM. This does have its own limitations, however. The oracle would be able to solve the Halting Problem for the associated TM, but not the Halting Problem in general for all TMs. Although oracles are considered more powerful in theory, they are disregarded in reality 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 (UTM). The UTM functions similar to the TM, except it requires the instructions needed to simulate any other TM in order to function. In essence it is a machine that is a blank slate that when given an operation, the input, and a means to perform the operation, performs the operation on the given inputs. 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

Robin Gandy formalized the works of Turing and his mentor, Alonzo Church, into a concise theorem [2]. This theorem is well-known in computability theory, and discusses the capabilities of functions, automata, and more.

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

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 [2]. This upper bound does not exist for humans and is therefore the basis of separation between computation power of TMs and humans.

He proposed a series of four Principles that are still used today as a basis for determining what a TM is capable of computing. Any automaton 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 in the domain of computability theory is Rice’s Theorem. It defines the bounds of research for programming languages, and more generally computing systems, in a formal manner. Below is the formal statement of Rice’s Theorem, which will be supplemented with a summarization of the theorem [26, 27].

Theorem 2 *(Rice’s Theorem) Let φ 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 $\mathbb{N}$.

This directly translates to programming languages as stating: All non-trivial semantic 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 seperate given program is able to execute without error. Another view of this theorem is: 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 (TC), 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 TM. 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, where 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 [28, 29].

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 then converted to binary and stored within the current byte [30].

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 like the one above, read: [29]. 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 [31, 32].

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

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. However, 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. The 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, in the C code below, I have 2 functions that are used to find the sum of the squares of 2 given numbers.

```
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 a non-negative number of 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.

Recursion relies on defining the exit conditions, sometimes called base cases, which stop the recursion from propogating further. The other essential part of defining a recursive function is the set of conditions in which to recurse. Typically, using recursion uses less overall lines, but comes at the increased complexity of the implementation. Another consideration would be the space and time complexity being compounded at each step of recursion. However, recursion remains an elegant approach towards progamming nonetheless.

See section B.1 in Appendix B for an example constructing the fibonacci sequence in C in 2 ways: without recursion, and with recursion.

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 old, with some coming from the 1960s, they may 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 [33, 34]. Additionally, Richard Stallman led the development for the GNU operating system using C [35]. 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 [36]. Besides C, COBOL remains a language that is used professionally for banking. Many banks still use COBOL their business application and management [37].

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). OOP is structured entirely different than Procedural Programming. Instead of defining procedures to solve the problem, OOP utilizes a new idea of coding. It outlines classes, which are descriptions of the system, and defines objects, which are the implementations of the parts of the system. Classes contain 3 parts: Data Members, Constructors, and Methods.

1. Data Members: Used to describe attributes about the class.
2. Constructors: Ways to create an instance (object) of this class.
3. Methods: Ways of manipulating the Data Members associated with the class of objects or a particular object.

For example, if I want to model a school, an important data member would be the amount of students enrolled. Perhaps there would be two ways to create a school: with a total amount of students already enrolled, and another with no students. 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. Each academic year, a certain amount of students enroll into either school. There must be a way to update the amount of students for any school. See section B.2 in the Appendix B for a snippet of Java code demonstrating these principles and implementing the above school example.

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 [38]. 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

Java is utilized today for a variety of applications including: embedded systems, android mobile apps, and web-apps [73, 74]. PHP is a language that is mostly used for web development. It is primarily used as a scripting language for servers [79]. It's most commonly seen in the Laravel framework as well as for WordPress [80, 81]. C++ also sees a wide variety of applications that it's used for. Some things include OSs like Microsoft Windows and Linux [75, 76]. Firefox, an open source browser, is also made using C++ [77, 78]. Java, PHP, and C++ remain some of the most desired languages for jobs in the market today [82–84].

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. The most desired languages today are Python and Javascript due to their simplicity and numerous list of libraries to code just about any project of any scale [82–89].

Here is an example snippet of Python code that demonstrates both object oriented and procedural programming at the same time:

```
# Sequential Programming
def findSumOfSquares(num1, num2):
```

```
    return (num1 ** 2) + (num2 ** 2)


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

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



# Creating the 'Homework' object and printing a method from it
HW = Homework("What is the sum of squares of 2 and 3?")
print(HW.problem)

# Utilizing a function created from sequential programming
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 function. Through the usage of OOP, we create a homework object that has a single data member, a string containing the 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 [39]. 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 [40]. 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 [41, 42].

1.3.1.5 Functional Programming Languages

Functional Programming Languages take an entirely different approach to coding. These languages are based on lambda calculus and are not often taught to new programmers due to their unique style. They usually follow a simple design of pure functions:

1. Referential Transparency: The same output is produced for all arguments given.
2. Immutability: There is no modification to the arguments given, I/O streams, or local/global variables.

FP has the unique property of being immutable for all functions and arguments [90–93]. FP functions also only use the arguments given for a particular task. Below are some simplistic code examples in Haskell:

```
-- Function to add two numbers
addTwoNumbers :: Num a => a -> a -> a
addTwoNumbers x y = x + y

-- Function to find the square of a number
findSquare :: Num a => a -> a
findSquare x = x * x



-- Function to find the sum of squares
findSumOfSquares :: Num a => a -> a -> a
findSumOfSquares x y = addTwoNumbers (findSquare x) (findSquare y)

-- Main function
main :: IO ()
main = do
    let num1 = 2
    let num2 = 3
    let result = findSumOfSquares num1 num2
    putStrLn ("The sum of squares is: " ++ show result)

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

    The sum of squares is: 13
```

This looks like a more advanced version of lambda calculus seen in 2.4.1. It operates by defining the function and its arguments. Once the definition is completed, you write out the logic for the function. One can chain these outputs like the functions in section 1.3.1.2. To implement loops, recursion is required.

```
-- Function to compute the nth Fibonacci number
fibonacci :: Integer -> Integer
fibonacci n
    | n < 0     = error "Negative input is not allowed"
    | n == 0    = 0
    | n == 1    = 1
    | otherwise = fibonacci (n - 1) + fibonacci (n - 2)

-- Function to generate a list of Fibonacci numbers up to the nth number
fibonacciList :: Integer -> [Integer]
fibonacciList n = [fibonacci x | x <- [1..n]]

-- Main function to print Fibonacci numbers up to the nth Fibonacci number
main :: IO ()
main = do
    let n = 4  -- User-defined bound for Fibonacci sequence
    let fibNumbers = fibonacciList n
    putStrLn ("The first " ++ show n ++
        " Fibonacci numbers are: " ++ show fibNumbers)

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

The first 4 Fibonacci numbers are: [1,1,2,3]
```

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

- Lisp
- Haskell
- Elixir
- OCaml
- Scala

Haskell and Elixir are used by many companies as a way for writing their software [94–97]. They are used for a variety of purposes such as security, browsers, and more. Coq is a proof assistant that allows users to codify their proofs and assert their findings. Coq was developed using OCaml [98]. For my experience using Coq alongside other proof assistants including Dafny and TLA+, see Appendix A.

Some benefits of using functional programming are:

- Unit Testing
- Debugging
- Concurrency

Unit Testing is made easier because of the immutable property. By not allowing for modification of items, it is easier to determine which tests fail. Furthermore, it forces more thought to be considered when designing the program.

Debugging is also made easier for similar reasons. Because functions are explicitly defined with inputs and ouputs with the immutability on all elements, this means that when there is an error it is simple to find. When there is a logical error, the solution is apparent more rapidly because of the logical flow of the program. When there is a compiler error, then instead the stack trace returns where the problem is.

Concurrency is also inherently allowed in the language because each function operates from a single thread. This is why Erlang or Elixir is used for telecommunications from companies like CISCO or banking companies such as Goldman Sachs [99].

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

Proteus is a programming language and compiler being developed as a project for CSUN's Autonomy Research Center for STEAHM (ARCS) 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 [70, 71]. 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 [25].

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 or modify local state 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. This is because in Proteus 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.

A real-world example would be that a manual car is within the ”Drive” state. Within this drive state, it has a number of sub-states to determine which gear is being used at the moment, eg. 1-6. The car would still be able to transition into the ”Neutral” gear (state), despite also being within the ”Drive” state.

Actors have a non-negative amount of associated HSMs while each HSM belongs to exactly one actor. Event Handlers are used to manage messages 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 [3].

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. See Appendix C for the grammar.

Looking at the grammar is similar to looking at the pieces of a puzzle without actually arranging the pieces together. It gives insight into what the final image might be, but won’t help unless you actually start using the pieces. 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 any 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 state machine. 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 Protes and to prove that it is TC. This is done by looking at the grammar and type of programming paradigm. Furthermore, there is an extensive analysis into proofs for proving Turing Completeness. By combining this knowledge together, I will then demonstrate that Proteus is in fact TC using some of the methods seen in the next chapter.

Chapter 2

Different Approaches for Proofs to Demonstrate Turing Completeness

2.1 Overview

I 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 level, starting from the Hardware and increasing in theory at each step. 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. By looking at these different approaches, I can apply them to Proteus.

2.2 Computer Engineering

In this section, I will analyze what a TM looks like from a physical perspective. This may seem contradictory because the TM is described as a theoeretical machine from Turing. 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, I 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, the Church-Turing Thesis, ”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 and subtraction. Furthermore, they must be capable of combining these simplistic operations together to form more complex operations such as exponential arithmetic. Beyond the mathematical aspect, they must allow for logical processing including logical AND, logical OR, and Logical XOR [19].

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 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 such as the keyboard and monitor [103].

Figure 2.1 visually describes the architecture [43]. 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 causes the CPU to wait until the bus is

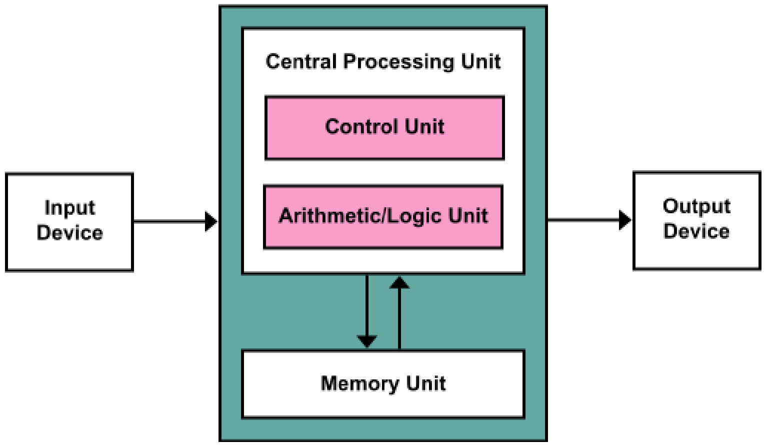

Figure 2.1: Von Neumann architecture visualized [43].

freed to continue processing. As an alternative, the Harvard Architecture was created.

The Harvard architecture separates the large and complex CPU from the von Neumann architecture into its several smaller components. 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. This allows the CPU to be capable of simultaneous reads and writes. However, a similar bottleneck occurs where the bus connecting each of the components is the limiting factor. See figure 2.2 for an illustration of this concept [44].

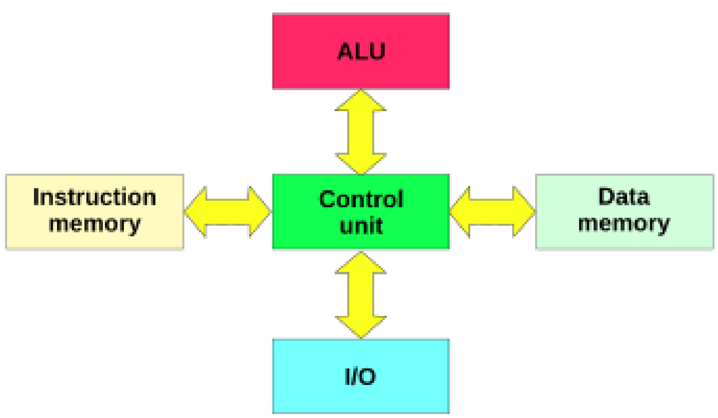

Figure 2.2: Harvard architecture visualized [44].

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 or for the intel i9-13900ks CPU to have 16 cores [45, 46]. By allowing each one to independently operate, but still have the CPU as the "brains" of the operation. With this knowledge of the architecture, I will now dive 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. This is because the CPU determines what operations to perform on what data. The ALU then receives the command to perform some arithmetic logic, and returns the result. The ALU contains registers which hold the temporary data for calculations. Eg. the multi-step operation $(2*4)+1$ requires several cycles and the ability to store the intermediary step $(2*4)$ without sending it as the result. These smaller registers are associated with an address for referencing purposes, and are not seen by the CPU. Instead the CPU manages it's own memory with caches and main memory.

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 inputs. Figure 2.3 describes a highly simplistic view of what a 2-Bit ALU may look like [100].

It accepts two inputs, i_a and i_b which are the inputs to the given operation. These inputs are fed into the respective operation handler, SUBtract, AND, ADDition, OR, XOR, or MOVe. The operation handles these inputs and uses the logical gates to perform the desired operation. This

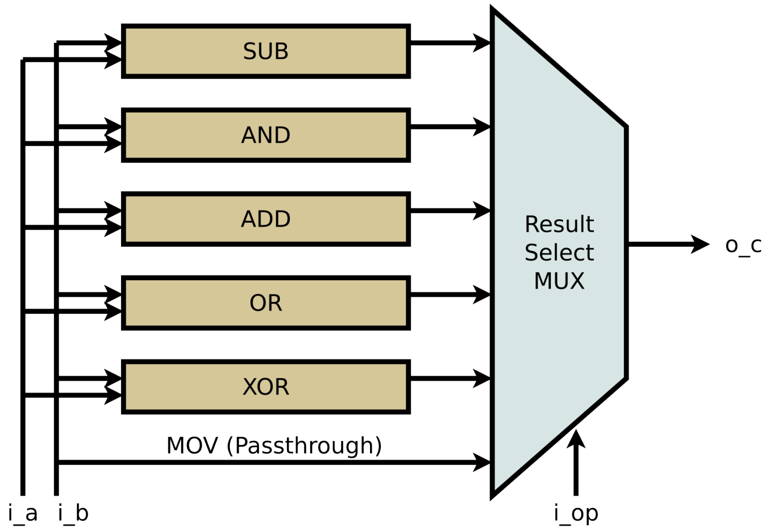

Figure 2.3: A simplistic 2-Bit ALU design [100].

result then gets sent to the MUX which will also take in the operation i_{op} and only accept the output from the desired operation. This result is then output as o_c.

This simplistic design lacks the ability to handle multi-step operations. It also lacks the actual layout of the ADD or OR operations. It instead abstracts them to display the circuit in a more digestible manner. As such, I have provided a more in-depth version of an ALU which lacks many of the operations seen in Figure 2.3. It explicitly shows the physical design of a different ALU [47].

Figure 2.4 only has 3 operations: OR, AND, XOR. The inputs are labeled A and B with the operation defined as $OP[1], OP[2], OP[3]$. $A[0]$ refers to the least-significant bit of A while $A[1]$ refers to the most-significant bit. These inputs are fed to all operation sequences (OR, AND, XOR), and sent to both the top and bottom MUX. These MUX then output the desired result from the $OP[NUMBER]$ desired as $OUT[0]$ and $OUT[1]$.

Overall, by combining these circuits together, one can create more powerful mechanisms such as logarithms, division, and more. To construct a physical TM using such circuitry, assuming unbounded memory amongst other conditions, is possible.

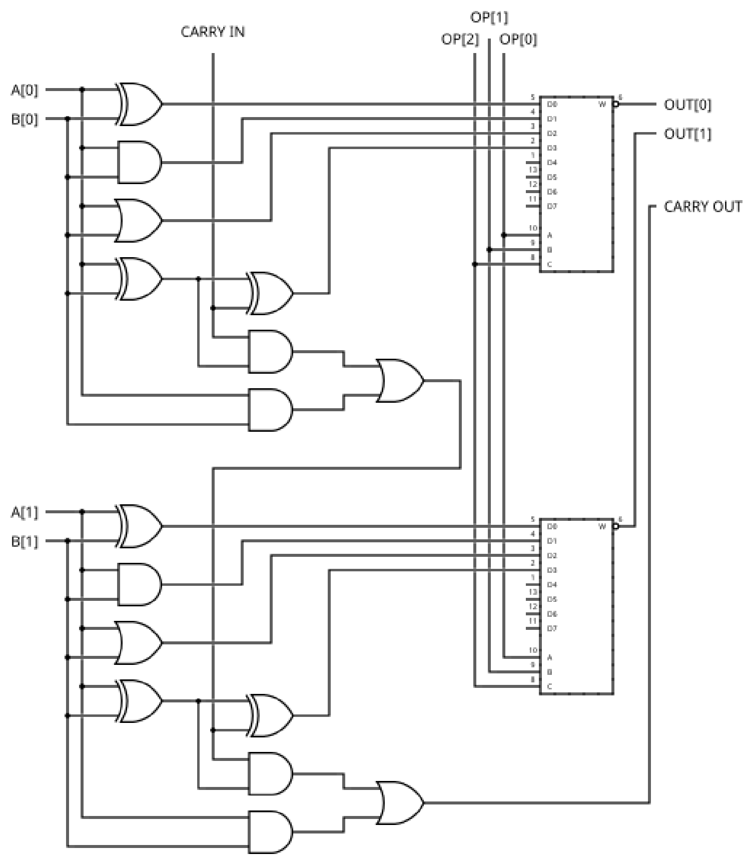

Figure 2.4: A lower level look at a different 2-Bit ALU design [47].

2.2.2 Constructing the TM

With the ability to construct logic gates, one can create more complex components such as the ALU, which would be used within the CPU, and overall create the computer from the bottom-up. Accompanied with the ability to store memory, as well as have a way to interact with the system through Input and Output, it is possible to create a functional computer. Developers have successfully created computers in the well-known video games of Minecraft and Terraria respectively in [20–22]. In fact, I will discuss a method to demonstrate Turing Completeness that verifies that the computer built inside Terraria is verifiably TC via a method discussed in 2.3.2.1.

In summary, to construct a TM on a physical level (of course alleviating the restriction of unbounded memory), these would be the minimum requirements [23, 24]:

- ADD

- SUB
- MUX
- OR
- AND
- XOR

Which themselves rely on logic gates as building blocks. Therefore, if it is possible to create/simulate logic gates, or to create the above shown processes, then it is capable of satisfying any arithmetic or logical calculation. As such, the Church-Turing Thesis, Theorem 1, is met and a TM can be created.

2.3 Computer Science

In this section, I will conceptualize what a TM looks like under the lens of Computer Science. I will now abstract from the physical understanding of how to create a TM, to creating a theoretical one using Automata theory or implementing a system that is TC using code. There are 2 main perspectives: that of Automata Theory and the Software Engineering approach. The Automata Theory approach utilizes theoretical designs based off of those utilized by Turing and Gavin. The Software Engineering approach instead applies it to a problem to showcase Turing Completeness via programming with code.

2.3.1 Automata Theory

In automata theory, Turing Machines are described using logical notation. The definition of a TM has several interpretations, but I will outline a slightly more advanced description [4,13]. Definition 1 allows for the machine to stay at the current cell. This will be needed later on for the proof, but as stated in section 1.2.3, all TMs are equivalently the most powerful computational machine.

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,

Σ is the input alphabet,

Γ is the finite set of symbols called the tape alphabet,

δ 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 δ is defined as

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

This means that for a given δ transition with inputs $q \in Q$ and $a \in \Gamma$, the tape will move to another state $x \in Q$, clear the current cell (indicated by $\square$) or some symbol $y \in \Gamma$, and choose to move the tape head Left one cell (L), Right one cell (R), or to Stay at the current cell (S). An example transition can be written:

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

where the internal state is q_0, and reads the input token a. After the transition, the internal state becomes q_1, the symbol d is written onto the tape at the current cell, and the cell pointer is moved to the right one cell. Figure 2.5 demonstrates the change.

Recall Figure 1.1 which represents a simplistic TM. This TM reads an input containing a string of 1's followed by a single 0. It then outputs whether the parity of the amount of 1's. This output overwrites the value at the cell containing 0, with 0 meaning odd and 1 meaning Even. In formal

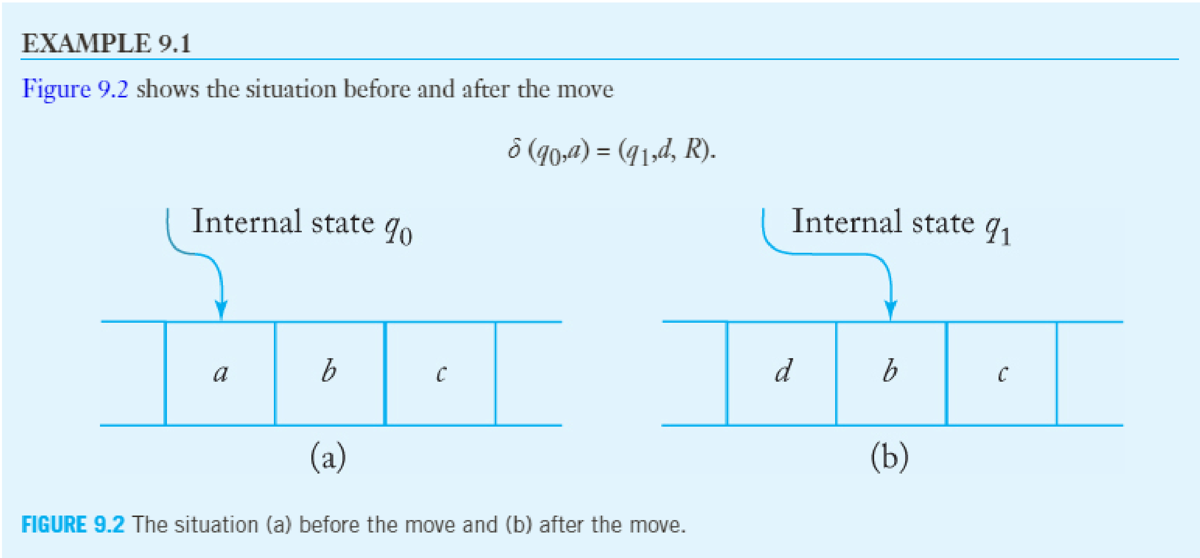

Figure 2.5: Delta transition example from [4].

nomenclature, the described TM can be written as follows:

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

$\Sigma = \{0, 1\}$

$\Gamma = \{0, 1, \square\}$

$F = \{q_2\}$

$q_0 \in$ Q as the initial state

and the following delta transitions

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

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

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

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

2.3.1.1 Notable examples using formal language

The proofs constructed using formal language usually modify the given system to meet these requirements. They must first demonstrate that the input to the system is Undecidable or able to be reduced to the Halting problem. Furthermore, they must create a TM using the system to demonstrate that it is capable of calculating anything.

(EN - I can explain deeper how they work in this section, but it is pretty complex and so i avoid that by briefly describing them)

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]; leveraging an extension of Rice’s Theorem, Theorem 2.

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. This approach utilizes the capabilities of the system as opposed to adjusting how they are understood/used. Instead of constructing a TM within the confines of the system, an equivalent proof is to implement a program that demonstrates Turing Completeness.

By implementing any known TC program implies that the overall system is TC. For example, the popular Sony PlayStation game ”LittleBigPlanet” released for the PlayStation 3, is TC as shown by someone implementing Conway's Game of Life inside of it [104].

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 TC cellular automata include Conway's Game of Life, Rule 110, and Langton's Ant [9,48,49,101,102].

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 relatively 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, 50]. Although not discussed further, Langton's Ant is another TC cellular automaton [101, 102].

2.3.2.1 Conway's Game of Life

Conway's Game of Life (CGoL) is a 2D grid of cells extending infinitely in the cartesian plane. Each cell may switch between only 2 states: Alive (On) or Dead (Off). The rules of CGoL are simple and are illustrated in Figure 2.6 [51]. 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.

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)

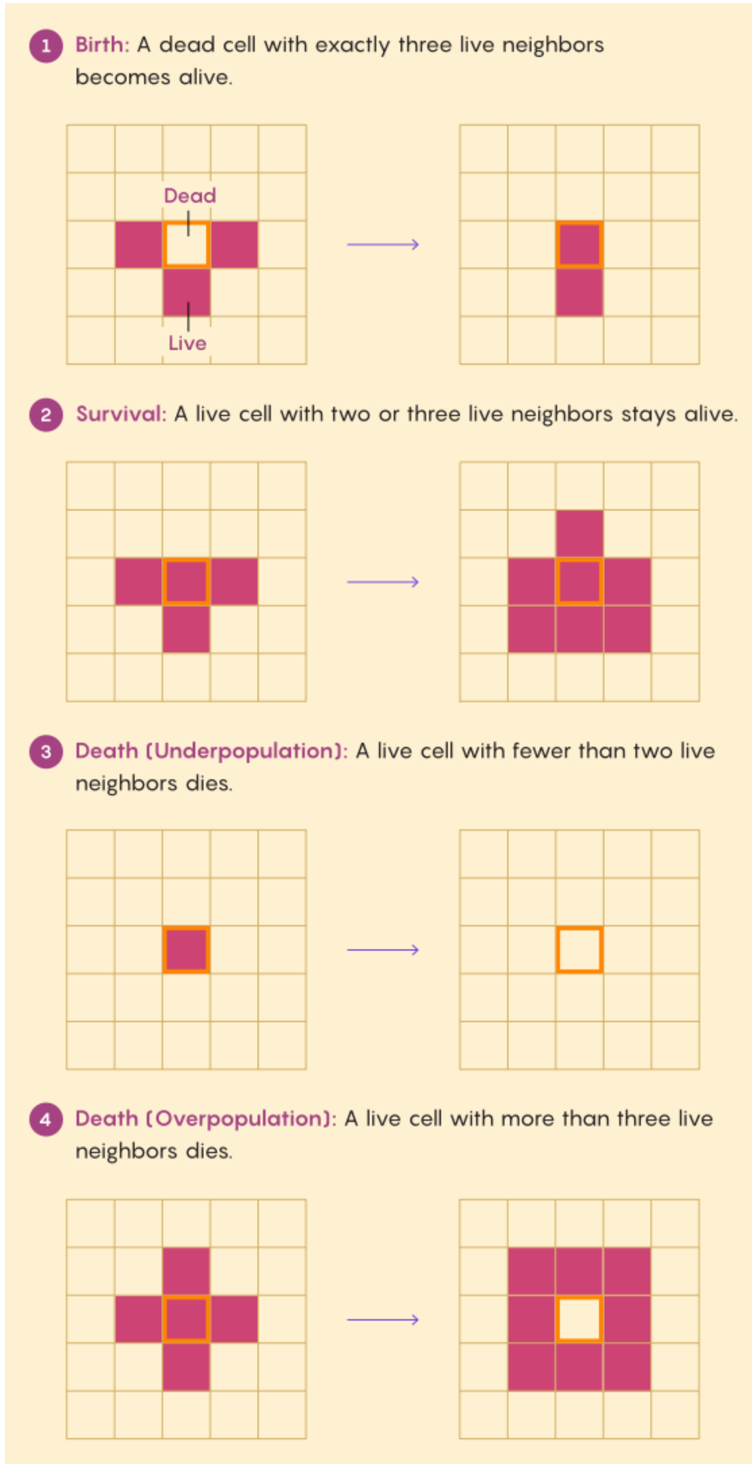

Figure 2.6: Rules of Conway's Game of Life visualized [51].

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 with an associated graphic in Figure 2.7 [52]:

1. 111 makes 0
2. 110 makes 1
3. 101 makes 1
4. 100 makes 0
5. 011 makes 1
6. 010 makes 1
7. 001 makes 1
8. 000 makes 0

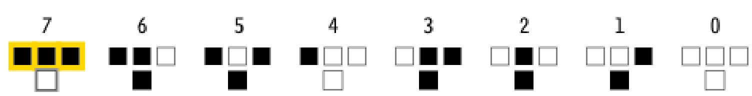

Figure 2.7: Rules for Rule 110 visualized [52].

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

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. Allowing boolean operators is also trivial to add in addition to the arithmetic operations. If statements and while loops are sufficient for handling the conditional and looping logic. A programmable calculator thefore is TC [53]. This much simpler approach is clear and follows software engineering design principles. A simplistic set of steps is outlined below:

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

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, seen in section 1.3.1.1, is often used to demonstrate a programming language is TC for its concise ruleset [54–57].

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 Alonzo 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, 58].

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. $(\lambda x.M[x]) \rightarrow (\lambda y.M[y])$: α-conversion, which renames the bound variables in the expression. This is be used to avoid name collisions.
2. $((\lambda x.M)N) \rightarrow (M[x := N])$: β-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 [59].

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 the parity of the total number of 1's in the given string.

I construct the list of Natural Numbers, $\mathbb{N}$, as follows using the idea of succession:

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

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

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

and so on...

Now I 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 [60]. Some notation can be interpreted as SKI combinator calculus [61].

$$\begin{aligned}
K &:= \lambda xy.x \equiv X(X(XX)) \equiv X'X'X' \\
S &:= \lambda xyz.(xz)(yz) \equiv X(X(X(XX))) \equiv XK \equiv X'(X'X') \\
I &:= \lambda x.x \equiv SKS \equiv SKK \equiv XX \\
Y &:= \lambda g.(\lambda x.g(xx))(\lambda x.g(xx)) \\
SUCC &:= \lambda nfx.f(nfx) \\
PRED &:= \lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u) \\
&\equiv \lambda n.n(\lambda gk.\ ISZERO\ (g\ 1)k\ (PLUS(g\ k)1))(\lambda v.0)0 \\
PLUS &:= \lambda mnfx.nf(mfx) \\
&\equiv \lambda mn.n\ SUCC\ m \\
SUB &:= \lambda mn.n\ PRED\ m
\end{aligned}$$

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

Now I will combine these lambda functions to perform the operation on the given input string. I list out the order of operations to produce the desired result.

1. Obtain the length of the List.
2. With the list length, subtract 1 from it.
3. Take the mod of the result.
4. If the new result is 0, then that means it was even.
5. 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 Figure 2.8 for an example of an expanded lambda calculus function that determines the parity of a given number [62, 63].

$$\begin{gathered}(\lambda m.(\lambda g.(\lambda x.g(\lambda a.x(x)(a)))(\lambda x.g(\lambda a.x(x)(a))))(\lambda f.(\lambda n.(\lambda a.(\lambda b.(\lambda p.(\lambda a.(\lambda b.p(b)(a)))) \\ ((\lambda m.(\lambda n.(\lambda n.n(\lambda x.(\lambda a.(\lambda b.b)))(\lambda a.(\lambda b.a)))((\lambda m.(\lambda n.n((\lambda n.(\lambda f.(\lambda x.n(\lambda g.(\lambda h.h(g(f)))) \\ (\lambda u.x)(\lambda u.u)))))(m)))(m)(n))))(b)(a))))(n)(\lambda f.(\lambda x.f(f(x))))(\lambda j.(\lambda n.n(\lambda x.(\lambda a.(\lambda b.b))) \\ (\lambda a.(\lambda b.a)))(n))(\lambda j.f(j))((\lambda m.(\lambda n.n((\lambda n.(\lambda f.(\lambda x.n(\lambda g.(\lambda h.h(g(f))))(\lambda u.x)(\lambda u.u)))))(m)))\end{gathered}$$

Figure 2.8: Expanded lambda calculus function to determine the parity of a number [62, 63].

With the ability to define any calculable function, Lambda Calculus is TC, satisfying the Church-Turing Thesis, Theorem 1. This is reminiscent of brainfuck, seen in section 1.3.1.1, which is so abstract that it becomes unusable for practical purposes.

Chapter 3

Proteus is Turing Complete

This section will describe how I 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. The outline of this chapter is as follows:

1. Show all inputs (Proteus programs) are Undecidable.

2. Create a TM in Proteus.

3. Show CGoL in Proteus code.

4. Show Rule 110 in Proteus code.

On a technical level, by the second step, Proteus is proven TC. By adding CGoL and Rule 110, I demonstrate how to utilize Proteus code, as well as showcase its property of being TC.

3.1.1 Undecidable input

Because Proteus is a higher-level programming language, I 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. Now 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. Recall 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.

Here is the list of types for Proteus:

- 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 capable of responding (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 [64, 65].

'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 their own state as well as other state machines.

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 grammar 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. The Ackermann function is defined:

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

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

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

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 CGoL 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.

I will construct the TM in the following steps:

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

I will now describe how to create a TM within Proteus from a higher abstraction layer with section 3.3 then going into the explicit formal definition. The tape is a series of state machines, HSMs, that will be ordered as $c_0, c_1, \ldots, c_{\mathrm{n}}$ 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. The read/write head 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. Whenever a non-empty cell is encountered by the read/write head, it will broadcast what the current state is. Figure 3.1 shows the design of the TM in a simplified manner.

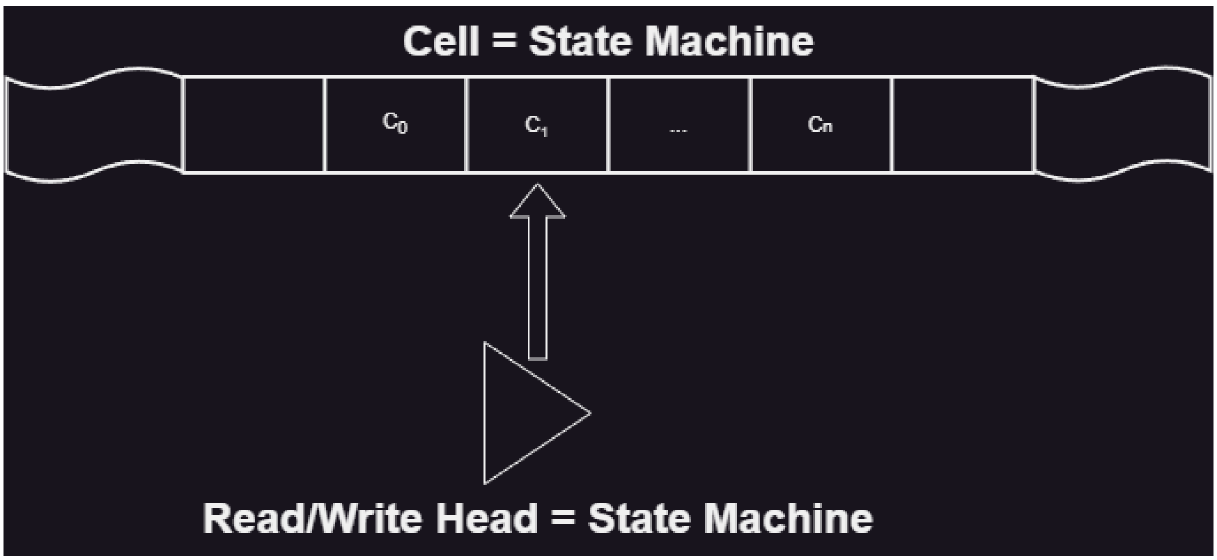

Figure 3.1: Design of a TM in Proteus

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

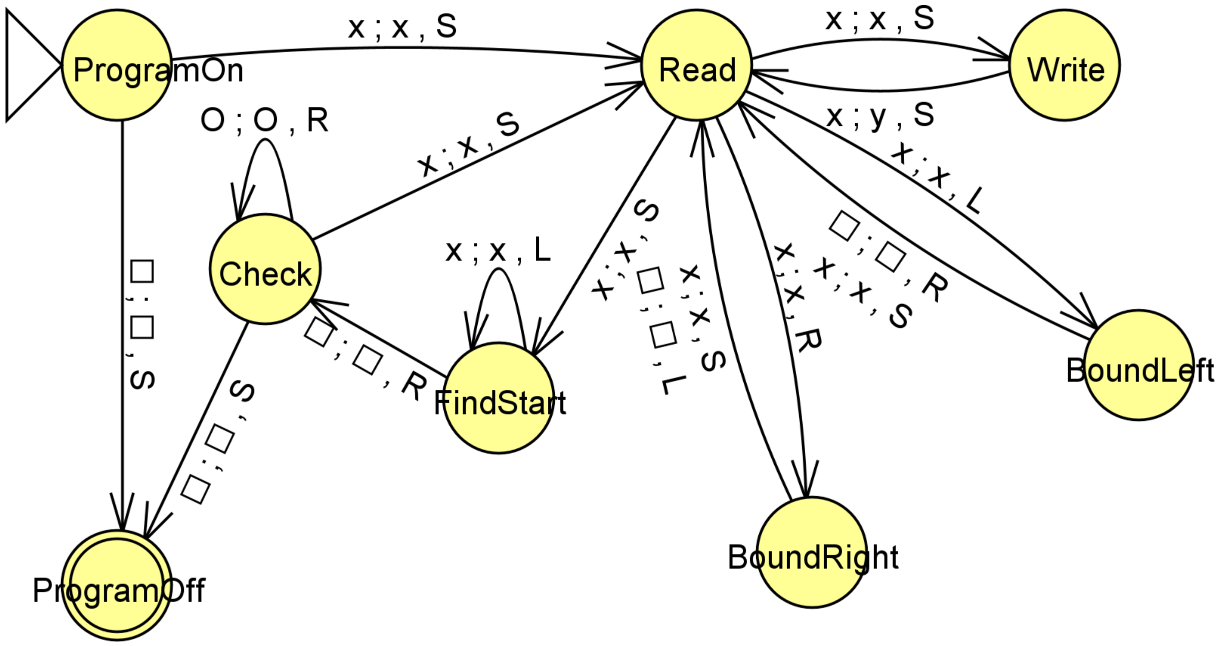

Figure 3.2: TM State Diagram in Proteus

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.

3.3 Definition of a Turing Machine in Proteus

The list of internal states describes the states that the read/write head has. The list of states is seen in Figure 3.2 and will be described formally below.

$$Q = \{\text{'ProgramOn'}, \text{'ProgramOff'}, \text{'Read'}, \text{'Write'}, \text{'BoundLeft'}, \text{'BoundRight'}, \text{'FindStart'}, \text{'Check'}\}$$

The initial state of the program is 'ProgramOn', thus

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

There is only a single halting state, 'ProgramOff'. Therefore, the set of Final states is written,

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

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 3.2. 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 exists a non-negative number of states defined by the programmer. Each programmer created state, s_i, is not to be either of the ’On’ or ’Off’ states. Because Γ must contain the ’$\square$’ symbol, it indicates that it is capable of writing blanks to the cells. Because Proteus disallows dynamic creation and deletion of state machines, I will say that writing a blank means the state machine at the specific cell is regarded as being in the ’Off’ state. Notice that in the described TM, there is no overwriting of data within the non-empty cells with a blank. Because JFLAP does not allow for multiple characters to be a single element within the set of Γ, I use the symbol O to represent the ’Off’ state in Figure 3.2.

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

The transition function determines the conditions for the read/write head to change states. These transitions can be seen clearly in Figure 3.2.

$$\text{let } x, y \in \Gamma$$

$$\delta(\text{'ProgramOn'}, x) = (\text{'Read'}, x, S)$$

$$\delta(\text{'ProgramOn'}, \square) = (\text{'ProgramOff'}, \square, S)$$

$$\delta(\text{'Read'}, x) = (\text{'Write'}, x, S)$$

$$\delta(\text{'Read'}, x) = (\text{'BoundLeft'}, x, L)$$

$$\delta(\text{'Read'}, x) = (\text{'BoundRight'}, x, R)$$

$$\delta(\text{'FindStart'}, x) = (\text{'BoundLeft'}, x, S)$$

$$\delta(\text{'Write'}, x) = (\text{'BoundLeft'}, y, S)$$

$$\delta(\text{'BoundLeft'}, x) = (\text{'Read'}, x, S)$$

$$\delta(\text{'BoundLeft'}, \square) = (\text{'Read'}, \square, R)$$

$$\delta(\text{'BoundRight'}, x) = (\text{'Read'}, x, S)$$

$$\delta(\text{'BoundRight'}, \square) = (\text{'Read'}, \square, L)$$

$$\delta(\text{'FindStart'}, x) = (\text{'FindStart'}, x, L)$$

$$\delta(\text{'FindStart'}, \square) = (\text{'Check'}, \square, R)$$

$$\delta(\text{'Check'}, x) = (\text{'Read'}, x, S)$$

$$\delta(\text{'Check'}, \text{'Off'}) = (\text{'Check'}, \text{'Off'}, R)$$

$$\delta(\text{'Check'}, \square) = (\text{'ProgramOff'}, \square, S)$$

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

$$Q = \{\text{'ProgramOn'}, \text{'ProgramOff'}, \text{'Read'}, \text{'Write'}, \text{'BoundLeft'}, \text{'BoundRight'}, \text{'FindStart'}, \text{'Check'}\}$$

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

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

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

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

with the transition functions:

$$\begin{aligned}
&\text{let } x, y \in \Gamma \\
\delta(\text{'ProgramOn'}, x) &= (\text{'Read'}, x, S) \\
\delta(\text{'ProgramOn'}, \square) &= (\text{'ProgramOff'}, \square, S) \\
\delta(\text{'Read'}, x) &= (\text{'Write'}, x, S) \\
\delta(\text{'Read'}, x) &= (\text{'BoundLeft'}, x, L) \\
\delta(\text{'Read'}, x) &= (\text{'BoundRight'}, x, R) \\
\delta(\text{'FindStart'}, x) &= (\text{'BoundLeft'}, x, S) \\
\delta(\text{'Write'}, x) &= (\text{'BoundLeft'}, y, S) \\
\delta(\text{'BoundLeft'}, x) &= (\text{'Read'}, x, S) \\
\delta(\text{'BoundLeft'}, \square) &= (\text{'Read'}, \square, R) \\
\delta(\text{'BoundRight'}, x) &= (\text{'Read'}, x, S) \\
\delta(\text{'BoundRight'}, \square) &= (\text{'Read'}, \square, L) \\
\delta(\text{'FindStart'}, x) &= (\text{'FindStart'}, x, L) \\
\delta(\text{'FindStart'}, \square) &= (\text{'Check'}, \square, R) \\
\delta(\text{'Check'}, x) &= (\text{'Read'}, x, S) \\
\delta(\text{'Check'}, \text{'Off'}) &= (\text{'Check'}, \text{'Off'}, R) \\
\delta(\text{'Check'}, \square) &= (\text{'ProgramOff'}, \square, S)
\end{aligned}$$

3.4 Creating Conway's Game of Life

To show some practical usage of Proteus as a language while also showing it TC, I will show how to simulate CGoL. First, I will describe the design for the overall program, then supplement it with the actual code implementation in Proteus.

3.4.1 Design of the Program

Recall CGoL in section 2.3.2.1. To represent the standard two-dimensional (2D) grid, I will use state machines for each cell. To remain consistent, I will be simulating the cartesian plane (2D) and assuming an arbitrary cell is the origin, (0,0). This is to simplify the referencing of cells within the cartesian plane (2D).

To begin, let there be a non-zero amount of cells that are to be initialized to some starting state. I will define the bounds of the grid of initialized cells with $n, m \in \mathbb{N}$. Each cell will be labeled with it's respective position according to the origin, (0,0). Figure 3.3 illustrates the design of the grid with each of the cells. Note that I have only depicted the cells within the first quadrant because these are the cells where the initialization process is described [72].

Also in Figure 3.3, there is a 'Controller'. This is because all the cells within the plane will be state machines, alongside this 'Controller'. The 'Controller' does not exist within the plane described, and is to be viewed as a separate entity entirely.

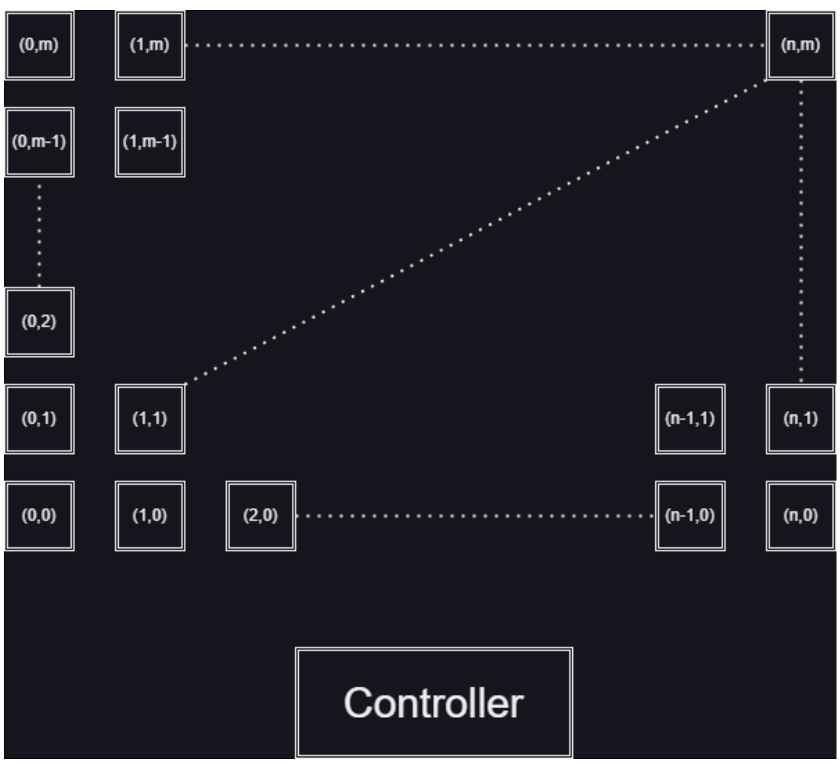

Figure 3.3: System design for Conway's Game of Life in Proteus

There will be two types of state machines: The cell, and the controller. I will now describe the data associated with each state machine.

The cell has 5 local variables:

1. myName: the name of the cell as a string. (Final)
2. Xcoord: the x coordinate as an integer. (Final)
3. Ycoord: the y coordinate as an integer. (Final)
4. currCellIsOn: holds the value of the current state of the cell (true == On).
5. nextStateCurrCellIsOn: holds the value of the next state of the cell (true == On).

Each cell also has 3 states:

1. Display: broadcasts currState to whatever machine requested the current state of the cell. Is the initial state the cell is in.
2. CalculateNext: Asks the neighbors in the 4 cardinal directions for their Display. Then determines based on the rules of CGoL what the next state of this cell should be.
3. Update: updates currState to the value of nextStateCurrCellIsOn

The controller is much simpler. It has no local variables and only 2 states:

1. Setup: Determines cells to turn on/off when starting the program
2. nextStep: broadcasts a message to all Cells to (1) calculate their next state and (2) update after finding their next state.

3.4.2 Implementation of Conway's Game of Life

In total there are 4 events that will be used in the Proteus code:

1. getDisplay(myName): the current cell requests another cell for their currState value. It passes the current cells name so that the receiver may transmit the data back to this cell.
2. calculateNextState(): this event is sent from the controller to all cells. It instructs the cells to calculate their next state (true == On).
3. updateAllCells(): this event is sent from the controller to all cells. It instructs the cells to update their currCellIsOn to the value of nextStateCurrCellIsOn.
4. initializeCell(Value): this event is sent from the controller to all programmer-defined cells. It instructs the cell to update it's currCellIsOn to the given Value, with Value $\in \{$true,false$\}$.

The Proteus code for a given cell at coordinates (X,Y) are as follows: (EN - if Proteus does not accept 'HSMName' to be defined dynamically, then instead add 4 vars to store the name of each of the neighbors statically. Another option would be to leave it outside of a variable and hardcode it)

```
actor cellXY{
    string myName = "cellXY";
    int Xcoord = [X];
    int Ycoord = [Y];
    bool currCellIsOn = false;
    bool nextStateCurrCellIsOn = false;
    statemachine {
        initial Display;
        state Display {
            on getDisplay {otherCellName} {otherCellName ! currCellIsOn}
            on calculateNextState {} {go calculateNext {}}
```

```
    on updateAllCells {} {go Update {}}
    on initializeCell {Value} {currCellIsOn = Value}
}
state calculateNext {
    int neighborTop = [Y]coord + 1;
    int neighborBot = [Y]coord - 1;
    int neighborLeft = [X]coord - 1;
    int neighborRight = [X]coord + 1;
    string neighborTopName = "cell" + Xcoord + neighborTop;
    string neighborBotName = "cell" + Xcoord + neighborBot;
    string neighborLeftName = "cell" + neighborLeft + Ycoord;
    string neighborRightName = "cell" + neighborRight + Ycoord;
    int count = 0;
    if (neighborTopName ! getDisplay {myName}) {
        count += 1;
    }
    if (neighborBotName ! getDisplay {myName}) {
        count += 1;
    }
    if (neighborLeftName ! getDisplay {myName}) {
        count += 1;
    }
    if (neighborRightName ! getDisplay {myName}) {
        count += 1;
    }
    if ((!(currCellIsOn)) && (count == 3)) {
        nextStateCurrCellIsOn = true;
```

```
            } else if ((currCellIsOn) && ((count == 2) || (count == 3))) {
                nextStateCurrCellIsOn = true;
            } else if ((currCellIsOn) && (count < 2)) {
                nextStateCurrCellIsOn = false;
            } else if ((currCellIsOn) && (count > 3)) {
                nextStateCurrCellIsOn = false;
            }
            go Display {}
        }
        state Update {
            currCellIsOn = nextStateCurrCellIsOn;
            go Display {}
        }
    }
}
```

Assuming that there are a nonzero amount of cells that are to be initialized to a desired state, the Proteus code for the controller is as follows. Let $cellXY, \ldots, cellAB$ be the cells to be defined with initial states, and associated states $s0, \ldots, sAB$. In order to send an event to all the cells, I will write $cellSS, \ldots, cellFF$ to represent the start and finishing cells in this list of possible cells.

```
actor controller{
    statemachine {
        initial Setup;
        state Setup {
            cellXY ! initializeCell { s0 };
            ...
            cellAB ! initializeCell { sAB };
            go nextStage {}
```

```
        }
        state nextStage {
            cellSS ! calculateNextState {};
            ...
            cellFF ! calculateNextState {};
            cellSS ! updateAllCells {};
            ...
            cellFF ! updateAllCells {};
        }
    }
}
```

The only thing missing is to tell the controller how many times to repeat the next stage. This is not a requirement for satisfying the capability of CGoL, but an extra nicety that would allow users to view the results at each stage. As such, the above Proteus code demonstrates CGoL, and is verifiably TC as shown previously in section 2.3.2.1.

3.5 Creating Rule 110

Another way to demonstrate that Proteus is TC is by implementing Rule 110. I will similarly start by outlining the design of the program followed by the implementation in Proteus code.

3.5.1 Design of the Program

Recall Rule 110 in section 2.3.2.2. To represent the standard one-dimensional (1D) grid, I will use state machines for each cell. To remain consistent, I will be simulating the 1D plane and assuming an arbitrary cell is the origin, 0. This is to simplify referencing particular cells within the infinite 1D plane.

To begin, let there be a non-zero amount of cells that are to be initialized to some starting state.

I will define the bounds of the grid of initialized cells with $n \in \mathbb{N}$. Each cell will be labeled with it's respective position according to the origin, 0.

Figure 3.4 illustrates the design of the line with each of the cells. Note that I have only depicted the cells within the first n cells because these are the cells where the initialization process is described.

Similar to demonstarting CGoL seen in the previous section, there is also a 'Controller' which is not a part of this 1D plane but is capable of interacting with all cells in Figure 3.4. All the cells within the plane will be state machines, alongside this 'Controller'.

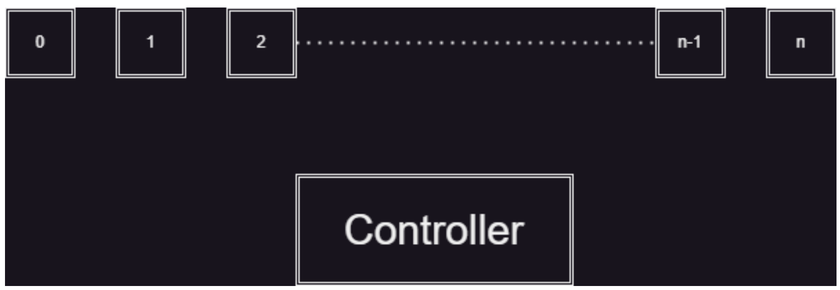

Figure 3.4: System design for Rule 110 in Proteus

There will be two types of state machines: The cell, and the controller. I will now describe the data associated with each state machine.

The cell has 4 local variables:

1. myName: the name of the cell as a string. (Final)
2. coord: the coordinate of the cell as an integer. (Final)
3. currCellIsOn: holds the value of the current state of the cell (true == On).
4. nextStateCurrCellIsOn: holds the value of the next state of the cell (true == On).

Each cell also has 3 states:

1. Display: broadcasts currState to whatever machine requested the current state of the cell. Is the initial state the cell is in.

2. CalculateNext: Asks the adjacent neighbors for their Display. Then determines based on the rules of Rule 110 what the next state of this cell should be.

3. Update: updates currState to the value of nextStateCurrCellIsOn

The controller is simpler. It has no local variables and only 2 states:

1. Setup: Determines cells to turn on/off when starting the program

2. nextStep: broadcasts a message to all Cells to (1) calculate their next state and (2) update after finding their next state.

3.5.2 Implementation of the Rule 110

In total there are 4 events that will be used in the Proteus code:

1. getDisplay(myName): the current cell requests another cell for their currState value. It passes the current cells name so that the receiver may transmit the data back to this cell.

2. calculateNextState(): this event is sent from the controller to all cells. It instructs the cells to calculate their next state (true == On).

3. updateAllCells(): this event is sent from the controller to all cells. It instructs the cells to update their currCellIsOn to the value of nextStateCurrCellIsOn.

4. initializeCell(Value): this event is sent from the controller to all programmer-defined cells. It instructs the cell to update it's currCellIsOn to the given Value, with Value $\in \{$true,false$\}$.

The Proteus code for a given cell at coordinate X are as follows: (EN - if Proteus does not accept 'HSMName' to be defined dynamically, then instead add 2 vars to store the name of each of the neighbors statically. Another option would be to leave it outside of a variable and hardcode it)

```
actor cellXY{
    string myName = "cellX";
    int coord = [X];
    bool currCellIsOn = false;
    bool nextStateCurrCellIsOn = false;
    statemachine {
        initial Display;
        state Display {
            on getDisplay {otherCellName} {otherCellName ! currCellIsOn}
            on calculateNextState {} {go calculateNext {}}
            on updateAllCells {} {go Update {}}
            on initializeCell {Value} {currCellIsOn = Value}
        }
        state calculateNext {
            int neighborLeft = coord - 1;
            int neighborRight = coord + 1;
            string neighborLeftName = "cell" + neighborLeft;
            string neighborRightName = "cell" + neighborRight;
            bool valNeighborLeft = neighborLeftName ! getDisplay {myName};
            bool valNeighborRight = neighborRightName ! getDisplay {myName};
            if ((valNeighborLeft) && (currCellIsOn)
                    && (valNeighborRight)) {
                nextStateCurrCellIsOn = false;
            } else if ((valNeighborLeft) && (currCellIsOn)
                    && (!(valNeighborRight))) {
                nextStateCurrCellIsOn = true;
            } else if ((valNeighborLeft) && (!(currCellIsOn))
```

```
                && (valNeighborRight)) {
            nextStateCurrCellIsOn = true;
        } else if ((valNeighborLeft) && (!(currCellIsOn))
                && (!(valNeighborRight))) {
            nextStateCurrCellIsOn = false;
        } else if ((!(valNeighborLeft)) && (currCellIsOn)
                && (valNeighborRight)) {
            nextStateCurrCellIsOn = true;
        } else if ((!(valNeighborLeft)) && (currCellIsOn)
                && (!(valNeighborRight))) {
            nextStateCurrCellIsOn = true;
        } else if ((!(valNeighborLeft)) && (!(currCellIsOn))
                && (valNeighborRight)) {
            nextStateCurrCellIsOn = true;
        } else {
            nextStateCurrCellIsOn = false;
        }
        go Display {}
    }
    state Update {
        currCellIsOn = nextStateCurrCellIsOn;
        go Display {}
    }
  }
}
```

Assuming that there are a nonzero amount of cells that are to be initialized to a desired state, the Proteus code for the controller is as follows. Let $cellX, \ldots, cellY$ be the cells to be defined with initial states, and associated states $s1,\ldots,sN$. In order to send an event to all the cells, I will write $cellS,\ldots,cellF$ to represent the start and finishing cells in this list of possible cells.

```
actor controller{
    statemachine {
        initial Setup;
        state Setup {
            cellX ! initializeCell { s1 };
            ...
            cellY ! initializeCell { sN };
            go nextStage {}
        }
        state nextStage {
            cellS ! calculateNextState {};
            ...
            cellF ! calculateNextState {};
            cellS ! updateAllCells {};
            ...
            cellF ! updateAllCells {};
        }
    }
}
```

The only thing missing is to tell the controller how many times to repeat the next stage. This is not a requirement for satisfying the capability of Rule 110, but an extra nicety that would allow users to view the results at each stage. As such, the above Proteus code demonstrates Rule 110, and is verifiably TC as shown previously in section 2.3.2.2.

With this, I have concluded showing that Proteus is TC with 3 verified methods:

1. Constructing a TM with an undecidable input

2. Implementing Conway's Game of Life in a Proteus program

3. Implementing Rule 110 in a Proteus program

Chapter 4

Conclusion

This chapter will summarize, reflect, and look beyond the scope of this thesis project.

4.1 Summary

Previously, several methods of demonstrating that a system can proven TC were discussed in chapter 2. Of these, the automata theory section 2.3.1, and software engineering approaches section 2.3.2, were utilized in this proof for their direct application to Proteus. Utilizing Rice's Theorem, Theorem 2, I demonstrated that with an undecidable input and the construction of a TM in the system that Proteus is TC. Because Proteus is a programming language, I also implemented some software implementations of TC programs. The choice of CGoL and Rule110 is because of the application of each state machine representing a cell on a grid; similar to the infinite tape described in the proof. After showing how to create a TM in Proteus, the proof is technically complete. I went further to showcase Proteus code in action and doubling as a secondary proof/implementation of Turing Completeness. This concludes the goal of the thesis.

4.2 Personal Reflections and Remarks

Turing Completeness was a topic discussed only in my higher level courses (300+) for my undergraduate degree. Even when it was discussed, it was seldomly discussed on a very deep level outside of the Automata classes, which admittedly makes sense. However, it seems that such a theoretical machine of such power and capability would be discussed across the other classes. It seemed to me that perhaps there was a fundamental reason which spearheaded my interest into the topic. Turing Machines seemed simplistic to understand based on the Church-Turing Thesis, Theorem 1. Despite this, there exist many equivalent definitions of a TM from automata theory, such as a DTM and a NDTM. Then I was interested in some systems that were unintentionally TC such as Excel or Java Generics [5, 66]. There exist many unassuming systems that are TC by allowing for the creation of a TM or through other various methods [67, 68].

Working with NASA JPL and through CSUN STEAHM ARCS, I was introduced to the Proteus project where I was able to assist by analyzing the theoretical implications of the system that would eventually be used to power rovers for interstellar research. My intentions were to show the computation power of this language and highlight some theoretical implications that may have otherwise been unthought of.

4.2.1 Importance of Turing Completeness

For a system to be shown TC, Theorem 1, means that it can compute just about any practical result. This means that it would be a major security vulnerability if a bad actor was able to utilize the TM for nefarious deeds. Thus, it's a matter of security as well as theory. By having a system that is TC, one must be extremely careful about how to handle security access and permissions. Mismanagement of memory may allow for adversaries to obtain access to memory they shouldn't and furthermore exploit the system. Such exploits have already been seen in systems that were shown TC [69].

4.3 Further Research and Improvements

This section outlines alternative proofs to show Proteus is TC. It also describes some improvements to the proof provided in chapter 3.

4.3.1 Utilizing the other approaches discussed

Omitting the brainfuck interpreter was intentional. This is because to properly implement an interpreter for brainfuck, I would have needed to implement a system to represent an input to the system, which is not available explicitly from the Proteus Grammar. This can be circumvented by some creative usage of state machines and understanding how to statically implement the input from the user, which would have otherwise been impossible. By instead writing out the input values statically encoded as an event, it would be possible to create a proper brainfuck interpreter. This seemed unnecessary complicated, but remains possible and an area for improvement.

Because of the truncation to maintain closure of the integers with arithmetic, there is no easy way to create a calculator using the Proteus Grammar directly. The other parts of the calculator to demonstrate a TC system are applicable, seen in section 2.3.2.3. If one was able to represent decimal values by utilizing state machines for holding the whole value portion and the decimal portion, then it would be possible to create a programmable calculator. The crux of the proof remains how to represent and achieve these numbers. This is also a possible improvement and method to demonstrate that Proteus is TC.

The mathematical approach for lambda calculus (section 2.4) is not applicable because Proteus is not a Functional Programming language and would require more in-depth theoretical definitions.

Using the Modified Harvard Architecture to create a TM, seen within section 2.2.1.1, would quickly prove to be very verbose. This is because defining the separate components and building a set of MUX along with other logical gates, would require many state machines with associated events. Even more, it would still require a way to interact with the I/O system which may be modeled similar to the one previously mentioned for the brainfuck interpreter. As a result, it isn't quite capable of receiving input naturally. Additionally, the restriction of truncation along the integers proves challenging in representing arithmetic such as division within the ALU. Nevertheless it is possible due to Proteus being TC, but remains a notion for future work.

4.4 Concluding Comments

By analyzing how other disciplines define Turing Completeness and prove systems TC is remarkable. These seemingly complex proofs are all equivalent in determining that a system is TC, the most powerful theoretical machine that exists. By performing this survey into theory across the disciplines, the gap between them reduces.

Proving that a novel programming language, Proteus, is TC through several methods discussed shows that these methods are useful and verifiable. Perhaps not all are applicable, but all are equivalent in purpose and truth.

References

[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: https://www.researchgate.net/publication/228107078_Lambda-Calculus

[16] R. Rojas, “A Tutorial Introduction to the Lambda Calculus,” 2015, Available: https://arxiv.org/pdf/1503.09060

[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/

[19] A. Hjelmfelt, E. D. Weinberger, and J. Ross, “Chemical implementation of neural networks and Turing machines.,” Proceedings of the National Academy of Sciences, vol. 88, no. 24, pp. 10983–10987, Dec. 1991, doi: https://doi.org/10.1073/pnas.88.24.10983.

[20] “I Made a Working Computer with just Redstone!,” www.youtube.com. https://www.youtube.com/watch?v=CW9N6kGbu2I

[21] From Scratch, “I Made a 32-bit Computer Inside Terraria,” YouTube, Jun. 24, 2023. https://www.youtube.com/watch?v=zXPiqk0-zDY

[22] misprit7, ”computerraria”, Feb. 16, 2023, GitHub repository, https://github.com/misprit7/computerraria

[23] “Home — nand2tetris,” nand2tetris, 2017. https://www.nand2tetris.org/

[24] Noam Nisan, ELEMENTS OF COMPUTING SYSTEMS : building a modern computer from first principles. 2020.

[25] B. McClelland et al., “Towards a Systems Programming Language Designed for Hierarchical State Machines,” pp. 23–30, Jul. 2021, doi: https://doi.org/10.1109/smc-it51442.2021.00010.

[26] Sakharov, Alex. “Rice’s Theorem.” From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein, Wolfram.com, 2024. https://mathworld.wolfram.com/RicesTheorem.html

[27] Wikipedia Contributors, “Rice’s theorem,” Wikipedia, Sep. 24, 2019. https://en.wikipedia.org/wiki/Rice%27s_theorem

[28] “Brainfuck,” Wikipedia, Oct. 05, 2021. https://en.wikipedia.org/wiki/Brainfuck

[29] 262588213843476, “Basics of BrainFuck,” Gist, Oct. 29, 2024. https://gist.github.com/roachhd/dce54bec8ba55fb17d3a

[30] speeder, “How does the Brainfuck Hello World actually work?,” Stack Overflow, May 30, 2013. https://stackoverflow.com/questions/16836860/how-does-the-brainfuck-hello-world-actually-work/19869651#19869651

[31] "brainfuck - Esolang," Esolangs.org, 2023. https://esolangs.org/wiki/Brainfuck#Self-interpreters

[32] Brainfuck.org, 2024. https://brainfuck.org/dbfi.b

[33] "git.git - The core git plumbing," Kernel.org, 2024. https://git.kernel.org/pub/scm/git/git.git/tree/

[34] "kernel/git/torvalds/linux.git - Linux kernel source tree," Kernel.org, 2024. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/

[35] "Source Language (GNU Coding Standards)," Gnu.org, 2024. https://www.gnu.org/prep/standards/html_node/Source-Language.html

[36] StatCounter, "Desktop Operating System Market Share Worldwide — StatCounter Global Stats," StatCounter Global Stats, 2019. https://gs.statcounter.com/os-market-share/desktop/worldwide/

[37] "On the past, present, and future of COBOL – Increment: Programming Languages," increment.com. https://increment.com/programming-languages/cobol-all-the-way-down/

[38] "How Does California State University–Northridge Rank Among America's Best Colleges?," @USNews, 2015. https://www.usnews.com/best-colleges/california-state-university-northridge-1153

[39] H. Shah, "7 Frontend JavaScript Frameworks Loved by Developers in 2022," Simform - Product Engineering Company, Feb. 17, 2022. https://www.simform.com/blog/javascript-frontend-frameworks/

[40] D. K. K. has years of experience as a S. D. S. H. enjoys coding, teaching, and has created this website to make M. L. accessible to everyone, "Is ChatGPT Written In Python??? [We FINALLY Found The Proof]≫ EML," enjoymachinelearning.com, Feb. 09, 2023. https://enjoymachinelearning.com/blog/is-chatgpt-written-in-python/

[41] Simplilearn, "What is R: Overview, its Applications and what is R used for — Simplilearn," Simplilearn.com, Oct. 25, 2021. https://www.simplilearn.com/what-is-r-article

[42] J. C. Luna, "Learn R, Python & Data Science Online," www.datacamp.com, Mar. 2023. https://www.datacamp.com/blog/top-programming-languages-for-data-scientists-in-2022

[43] Kapooht, "English: von Neumann Architecture," Wikimedia Commons, Apr. 28, 2013. https://commons.wikimedia.org/wiki/File:Von_Neumann_Architecture.svg

[44] "File:Harvard architecture.svg - Wikimedia Commons," Wikimedia.org, May 11, 2010. https://commons.wikimedia.org/wiki/File:Harvard_architecture.svg

[45] "NVIDIA GeForce RTX 4080 Specs," TechPowerUp, Oct. 13, 2023. https://www.techpowerup.com/gpu-specs/geforce-rtx-4080.c3888

[46] "Intel® CoreTM i9-13900KS Processor (36M Cache, up to 6.00 GHz) - Product Specifications," Intel. https://www.intel.com/content/www/us/en/products/sku/232167/intel-core-i913900ks-processor-36m-cache-up-to-6-00-ghz/specifications.html

[47] "File:2-bit ALU.svg - Wikibooks, open books for an open world," Wikibooks.org, Oct. 18, 2011. https://en.wikibooks.org/wiki/File:2-bit_ALU.svg

[48] Wikipedia Contributors, "Cellular automaton," Wikipedia, Dec. 05, 2019. https://en.wikipedia.org/wiki/Cellular_automaton

[49] E. W. Weisstein, "Cellular Automaton," From Mathworld–A Wolfram Web Resource, mathworld.wolfram.com. https://mathworld.wolfram.com/CellularAutomaton.html

[50] A. Ilachinski, Cellular Automata. World Scientific, 2001.

[51] "Math's 'Game of Life' Reveals Long-Sought Repeating Patterns — Quanta Magazine," Quanta Magazine, Jan. 18, 2024. https://www.quantamagazine.org/maths-game-of-life-reveals-long-sought-repeating-patterns-20240118/

[52] "File:One-d-cellular-automaton-rule-110.gif - Wikimedia Commons," Wikimedia.org, Nov. 20, 2018. https://commons.wikimedia.org/wiki/File:One-d-cellular-automaton-rule-110.gif

[53] Code & Optimism, "How to write a Turing-Complete Programming Language in 40 minutes in Ruby using Bable-Bridge," YouTube, Sep. 19, 2012. https://www.youtube.com/watch?v=_Uoyufkb5lk

[54] M. Kenyon, "How to Write a Brainfuck Interpreter in C#," Thesharperdev.com, Oct. 12, 2019. https://thesharperdev.com/how-to-write-a-brainfuck-interpreter-in-c/

[55] "Compiling to Brainf#ck - Meep.," InJuly, 2024. https://injuly.in/blog/bfinbf/index.html

[56] srijan-paul, "GitHub - srijan-paul/meep: A programming language that compiles to brainfuck.," GitHub, 2020. https://github.com/srijan-paul/meep

[57] M. Ueding, "Creating a Brainfuck interpreter," Martin Ueding, Apr. 19, 2023. https://martin-ueding.de/posts/creating-a-brainfuck-interpreter/

[58] "Functional Programming," learn.saylor.org. https://learn.saylor.org/mod/book/tool/print/-index.php?id=33044&chapterid=13087

[59] Wikipedia Contributors, "Lambda calculus," Wikipedia, Jan. 02, 2020. https://en.wikipedia.org/wiki/Lambda_calculus

[60] "Collected Lambdas," jwodder.freeshell.org. https://jwodder.freeshell.org/lambda.html

[61] Wikipedia Contributors, "SKI combinator calculus," Wikipedia, Oct. 28, 2024. https://en.wikipedia.org/wiki/SKI_combinator_calculus

[62] "Reddit - Dive into anything," Reddit.com, 2017. https://www.reddit.com/r/ProgrammerHumor/comments/78z90f/when_you_need_to_know_if_a_number_is_even_or_odd/

[63] "Reddit - Dive into anything," Reddit.com, 2017. https://www.reddit.com/r/ProgrammerHumor/comments/78z90f/comment/doylzry/

[64] "What is truncation in Java - Javatpoint," www.javatpoint.com, 2021. https://www.javatpoint.com/what-is-truncation-in-java

[65] GeeksforGeeks, "trunc() , truncf() , truncl() in C language," GeeksforGeeks, Jan. 25, 2018. https://www.geeksforgeeks.org/trunc-truncf-truncl-c-language/

[66] "The Excel Formula Language Is Now Turing-Complete," InfoQ. https://www.infoq.com/articles/excel-lambda-turing-complete/

[67] "Accidentally Turing-Complete," beza1e1.tuxen.de. https://beza1e1.tuxen.de/articles/accidentally_turing_complete.html

[68] G. Branwen, "Surprisingly Turing-Complete," gwern.net, Dec. 2012, Available: https://gwern.net/turing-complete

[69] "A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution," Blogspot.com, 2021. https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html?m=1

[70] Wikipedia Contributors, "Actor model," Wikipedia, Nov. 14, 2019. https://en.wikipedia.org/wiki/Actor_model

[71] L. Nigro, "Parallel Theatre: An actor framework in Java for high performance computing," Simulation Modelling Practice and Theory, vol. 106, p. 102189, Jan. 2021, doi: https://doi.org/10.1016/j.simpat.2020.102189.

[72] "Cartesian Plane - Definition, Meaning, Quadrants, Examples," Cuemath. https://www.cuemath.com/geometry/cartesian-plane/

[73] "What Is Java Used For: 12 Real World Java Applications," www.softwaretestinghelp.com. https://www.softwaretestinghelp.com/real-world-applications-of-java/

[74] "What is Java Used For?," Codecademy News, Jun. 24, 2021. https://www.codecademy.com/resources/blog/what-is-java-used-for/

[75] Simplilearn, "Top 7 Practical Applications of C++ and the Way to Build a Career in the Field," Simplilearn.com, Jan. 16, 2020. https://www.simplilearn.com/c-plus-plus-programming-for-beginners-article

[76] "Uses of C++ — 10 Reasons Why You Should Use C++," EDUCBA, Jul. 19, 2019. https://www.educba.com/uses-of-c-plus-plus/

[77] "mozilla/gecko-dev," GitHub, Feb. 01, 2024. https://github.com/mozilla/gecko-dev

[78] "The new, fast browser for Mac, PC and Linux — Firefox," Mozilla, 2000. https://www.mozilla.org/en-US/firefox/

[79] Astari S., “What Is PHP? Learning All About the Scripting Language,” Hostinger Tutorials, Apr. 30, 2019. https://www.hostinger.com/tutorials/what-is-php/

[80] http://facebook.com/syedbalkhi, “What is PHP? How PHP is Used in WordPress?,” WPBeginner, 2019. https://www.wpbeginner.com/glossary/php/

[81] T. Otwell, “Laravel - The PHP Framework For Web Artisans,” Laravel.com, 2015. https://laravel.com/

[82] L. dev, “Top 8 Most Demanded Programming Languages in 2023,” Devjobsscanner, Jun. 22, 2023. https://www.devjobsscanner.com/blog/top-8-most-demanded-programming-languages/

[83] L. Whitney, “Top 10 programming languages employers want in 2023,” TechRepublic, Feb. 03, 2023. https://www.techrepublic.com/article/top-programming-languages-employers-want/

[84] “11 of the Most In-Demand Coding Languages,” Indeed Career Guide. https://www.indeed.com/career-advice/career-development/most-in-demand-coding-languages

[85] “Search results,” PyPI. https://pypi.org/search/

[86] npm, “npm — build amazing things,” Npmjs.com, 2019. https://www.npmjs.com/

[87] Yarnpkg.com, 2019. https://yarnpkg.com/

[88] “10 Hardest and Easiest Programming Languages in 2024 - GUVI Blogs,” Mar. 01, 2023. https://www.guvi.in/blog/easiest-programming-languages-to-hardest-ranked/

[89] “The 10 Most Popular Coding Languages to Learn in 2023 — BestColleges,” www.bestcolleges.com. https://www.bestcolleges.com/bootcamps/guides/most-important-coding-languages/

[90] J. M. Fernandes, “Functional Programming: With Examples and Lots of Cats,” Medium, Sep. 29, 2021. https://medium.com/arctouch/thinking-functional-now-with-example-and-cats-8b9c2478b9af

[91] GeeksForGeeks, “Functional Programming Paradigm - GeeksforGeeks,” GeeksforGeeks, Jan. 02, 2019. https://www.geeksforgeeks.org/functional-programming-paradigm/

[92] “Referential transparency,” Wikipedia, Jan. 05, 2021. https://en.wikipedia.org/wiki/Referential_transparency

[93] Claudiu, “What is referential transparency?,” Stack Overflow, Oct. 17, 2008. https://stackoverflow.com/a/9859966

[94] beam-community, “elixir-companies/priv/companies at main · beam-community/elixir-companies,” GitHub, 2015. https://github.com/beam-community/elixir-companies/tree/main/priv/companies

[95] erkmos, “GitHub - erkmos/haskell-companies: A gently curated list of companies using Haskell in industry,” GitHub, 2017. https://github.com/erkmos/haskell-companies?tab=readme-ov-file

[96] beam-community, “GitHub - beam-community/elixir-companies: A list of companies currently using Elixir in production.,” GitHub, 2015. https://github.com/beam-community/elixir-companies

[97] “Haskell in industry - HaskellWiki,” Haskell.org, 2018. https://wiki.haskell.org/Haskell_in_industry

[98] coq, “GitHub - coq/coq: Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs.,” GitHub, Sep. 04, 2024. https://github.com/coq/coq?tab=readme-ov-file

[99] F. Cesarini, “Companies Who Use Erlang,” Erlang Solutions, Sep. 11, 2019. https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus/

[100] “A Simple ALU, drawn from the ZipCPU,” zipcpu.com. https://zipcpu.com/zipcpu/2017/08/11/simple-alu.html

[101] I. Stewart, “The Ultimate in Anty-Particles,” Scientific American, vol. 271, no. 1, pp. 104–107, Jul. 1994, doi: https://doi.org/10.1038/scientificamerican0794-104.

[102] Wikipedia Contributors, “Langton,” Wikipedia, Feb. 16, 2024.

[103] Y. N. Patt and S. J. Patel, Introduction to computing systems : from bits and gates to C and beyond. Boston: Mcgraw-Hill\Higher Education, Cop, 2005.

[104] ashandore, “LittleBigLife - The Game of Life in LittleBigPlanet,” YouTube, Dec. 15, 2008. https://www.youtube.com/watch?v=13GOFa1C4e4

Appendix A

Proof Assistants

Initially, my goal was to learn how to use proof assistants to assert properties about Proteus. I began to learn how to write basic and simple proofs using Coq. Coq was powerful enough to show properties of systems that were publishable. Whilst learning how to use Coq, I was learning to use TLA+ which proved equally as powerful, but not suited for my use case. TLA+ is used for simultaneous computing as well as state machines which seemed very appealing upon initial inspection, since Proteus uses state machines as well. However, this didn't extend well into demonstrating parts of Proteus that were not related to state machines. I then considered using a different proof assistant like Dafny, Lean, or Twelf to see which would be better suited for proving something about Proteus. I continued learning how to use the proof assistants and aimed to apply them to whatever proof I would eventually create about Proteus.

The big idea hit me one day; To show that Proteus was Turing Complete. It was one of the biggest things to show in theory and was able to utilize all my knowledge and experience from this discipline. There was only 1 problem, I couldn't use any proof assistant. This is because fundamentally, it is impossible to show Turing Completeness using a proof assistant.

Recall Turing Completeness means that the shown system is equivalently as powerful as a TM, see section 1.3. Also, recall the Church-Turing Thesis, Theorem 1, which means that the definition of a TM is to be able calculate any calculable function. Additionally, recall the proof of showing a system is TC in section 2.3.1.1. The purpose of a proof assistant is to assert that the given proof is valid, step by step until the goal is reached. Therefore for all inputs for any proof assistant, they are decidable (assuming they are well-written). Thus, they are not capable of satisfying the Church-Turing Thesis, Theorem 1, and also cannot be as powerful as a TM. As such, it is impossible to use a proof assistant to show Turing Completeness. This is when I abandoned using a proof assistant for proving Proteus TC.

Appendix B

Code Segments

B.1 Fibonacci in C

This code segment shows an example for 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 i = 1;
    int sub1 = 0;
    int sub2 = 1;
    for (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;
}
```

B.2 Java School Example

Java code to demonstrate OOP using the school example mentioned in 1.3.1.3

```
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
    }
}
```

Appendix C

Proteus Grammar

```
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
```