Documented Design

## System Overview

The system will be composed of a series of windows. These windows will include a few options, but the main window will be used for the majority of the time. Here, you can define the parameters of the Turing Machine to run. This will include:

* Machine variant
* Tape count
* Transition functions of machine
* Speed of operation
* Initial state
* Tape contents
* Read/Write head positions

Users can also use this window to run the machine, as well as save and import it to and from text files. There will also be a series of preinstalled examples in the main window that the user can open and test. The system also contains the option to create a diagrammatic layout of the defined machine. This will be handled by another window, which can be opened from the toolbar of the main window. The project allows users to set some parameters of the machine to load every time the program is open. The configuration file contains:

* Minimum speed of machine
* Maximum speed of machine
* Default case character
* Full speed interval value

There will also be another window that can be opened to give the user instructions on how to use each machine and how their transition function syntaxes work. There will also be details on certain parameters that must be met before a machine can run, and also details on how to use built-in escape characters in the Turing machines.

## Module Diagram

Run Machine Variant Code

(De)Compile Machine

Machine Variant Code

Machine

Execute Instruction

Information Window

Read information on Turing machine

Configure Turing machine

Options Window

Main Window (Simulator)

Display as FSM

Finite State Machine

Rename Window

Rename Turing machine

Import Turing machine

Export Turing machine

Edit Turing machine

## Module Overview

Simulator

**Includes Simulator.cs and:**

* **TabObj.cs and TapeObj.cs**
* **MachineData.cs**

The Simulator module handles all of the GUI operations of the main window. This section of the program contains the sections:

* Input Tabs
* Set UI Tags during Creation
* Update UI Post-Creation
* Tape Instancing
* Runtime
* Utility
* Open Forms

Input Tabs

This section of the module handles machines. Machines in my program are contained in the tabs of the InputWindow control. When a new tab is added, a new machine is created in the program. The way this section handles machines is:

* Creating new machines;
* Opening existing machines from text files;
* Deleting machines;
* Saving machines to text files;
* Duplicating machines;
* Renaming machines;
* Controlling what can/can’t be pasted into the machine’s text editor.

Set UI Tags during Creation

A relatively small section, this contains subroutines that work to set the initial values of the UI directly after creating a new or opening an existing machine. These subroutines perform the following functions:

* Instantiate the correct number of tapes into the program;
* Set the initial value of the machine variant;
* Set the values of the tapes and update their read/write heads to match their contents.

Update UI Post-Creation

This section is used to update the UI’s values after the machine has been created/opened. These functions include:

* Updating what settings are available depending on machine variant;
* Updating the speed of operation of the machine;
* Updating the list of available initial states for the machine to start in;
* Ensure the tape’s contents is the correct size to fit in the program window;
* Update the stored value of the initial state of the tape(s);
* Update the stored value of the position of the read/write head;
* Update the UI’s representation of the read/write head;
* Correctly mapping new values to the UI when changing what machine is selected;
* Setting the new height of the program if more than one tape is used.

Tape Instancing

Tape objects need to be produced by the program in order to allow the user to use more than one tape. This section allows the user to create new tapes to represent existing data, delete unnecessary objects, or create completely new tapes:

* Add a new tape without updating the machine’s values. This is useful when changing machines;
* Add a new tape to the program with new values which are added to the machine;
* Delete a tape without removing information from the machine. This is useful when changing machines;
* Delete a tape and remove the information that tape held from the machine.

Runtime

This section of the program controls the operation of the machine. It disables and enables controls that are needed to run the program correctly and also calls the subroutines used to run the program:

* Set the machine up to run. This is done by disabling all controls that cannot be used by the program while the machine is running, and loading all of the information into the Machine module;
  + The section must also be able to set the machine up to ‘step’ the machine, performing only one instruction and then pausing, not running the program one instruction after the other.
* Ensure that a machine can be run and is not written incorrectly;
* Pause the machine;
* Reset the machine completely, resetting the UI;
* Halt the machine, displaying information about the result of its operation;
* Perform one instruction in the operation of the machine once every *x* milliseconds, where *x* is specified by the user;
* Perform *y* instructions in the operation of the machine once every 10 milliseconds, where *y* is the number of instructions to be performed specified by the user;

Utility

This section includes methods used by the other sections to function. These are split from their sections because they either need to be used in multiple places in the program, or because moving them to another location makes the maintenance of the program easier. The functions contained by this section can:

* Disable controls in the program;
* Enable controls in the program;
* Clear the contents of the debug box;
* Calculate the string to be used to represent the position of the read/write head along the tape;
* Find the line number of the current selection in the text editor;
* Interact with the UniversalMachine module to produce a Universal Turing machine;
* Import and export user-defined preferences from the config.ini file.

Open Forms

This section of the program has subroutines that open other windows. These windows are parts of their own modules that serve other purposes. The windows that can be opened from this section are:

* Finite State Machine representation;
* Settings window;
* Information window.

Machine

**Includes Machine.cs and:**

* **Other Machine.cs files (Excluding UniversalMachine.cs)**
* **Tape.cs, MultiTape.cs**
* **Transition Function classes**

The Machine module performs the computations to run the Turing machines. The module is called by the Simulator module when the machine is run by the user. The functions of this module are:

* Take information from the Simulator module, compiling this as local copies to be able to quickly reference the information;
* Return Boolean values to indicate if the machine is compiled, and if the machine is halted;
* Perform one instruction of the machine when called by the Simulator module;
* Update the UI values of the Simulator module after performing a step;
* Decompile the machine by making the machine variant value equal to -1. When this is true the machine is marked as not compiled;
* Find the list of characters that can override the default case character in single-tape machines;
* Find the list of characters for each tape in a multi-track machine that can override the default case character;
* Find the list of sequences of read keys that can override the default case character in a multi-tape and universal machine.

Finite State Machine

**Includes FiniteStateMachine.cs and all Graphics objects.**

This module contains functions that work to produce a graphical representation of a Turing machine as an FSM. The module uses a depth-first algorithm to instantiate each PictureBox object that represents a state. Using the locations of these objects, the module draws lines using a Canvas object that makes the ‘edges’ of the FSM – the arrows that represent the transitions between states. Each arrow in the diagram contains a label that displays what input causes the transition, and what the output of the transition is (what is written to the tape and where the read/write head moves).

UniversalMachine

**Includes UniveralMachine.cs**

This module is a static class that contains subroutines to allow the user to convert a Deterministic or Read-Only machine into a Universal Turing machine (UTM). The subroutines used perform the following functions:

* Generate a set of transition functions for the UTM to use to run the input machine.
* Convert an input machine into a set of tapes to be placed on the UTM.

Settings

The Settings module is a small module that gets preferences from the user. These preferences are:

* Minimum and maximum speeds of the Turing machine;
* Default case character;
* Number of steps performed by the machine per step in Full Speed mode.

When the options are confirmed by the user, the machine validates the options, and if they pass, they are returned to the Simulator module and are set.

Information

This module does not contain any user-created subroutines, but is still an important section of my program. This module contains documentation on how to use my program to produce Turing machines. The information is split into:

* General Information, explaining the background information behind Turing machines;
* Transition Functions. This section describes, with examples, the syntax used in my program for the transition functions of all the different Turing machines.
* Universal Machines. These machines can be difficult to understand, so a section is used to explain what their purpose is, and how to format them in my program.
* Default Cases. The concept of default case functions are explored in this section of documentations. Included is an explanation of how each machine interacts with default case functions.

## Pseudocode Machine

The following is a pseudocode example of the Deterministic Turing machine; the default setting for the program. Here, take commands as the list of transition functions given by the user, tape as an object containing a string of characters and an integer pointer, state as the current state of the machine and isHalted and isStepCompleted two Boolean values to show whether or not the machine can continue.

PROCEDURE Main(tape, commands, state, IsHalted)

IsStepCompleted <- False

FOREACH command in commands:

IF !IsStepCompleted THEN

IF command.InitialState == state THEN

Step(command, tape, state, IsStepCompleted)

ENDIF

ELSE

break

ENDIF

ENDFOR

IF !IsStepCompleted THEN

IsHalted <- True

ENDIF

ENDPROCEDURE

PROCEDURE Step(command, tape, state, IsStepCompleted, OverrideString)

IsMatch <- False

IF tape.Pointer >= tape.Length THEN

IF command.ReadKey == ‘\_’ THEN

IsMatch <- True

ENDIF

ELSE IF tape.Pointer < 0 THEN

IF command.ReadKey == ‘\_’ THEN

IsMatch <- True

ENDIF

ELSE

IF command.ReadKey == tape.Tape[tape.Pointer] THEN

IsMatch <- True

ELSE IF command.ReadKey == DefaultCaseKey THEN

IF tape.Tape[tape.Pointer] NOT IN OverrideString THEN

IsMatch <- True

ENDIF

ENDIF

IF IsMatch THEN

tape.Tape[tape.Pointer] <- command.WriteKey

IF command.MoveKey == ‘r’ THEN

tape.Pointer += 1

ELSE IF command.MoveKey == ‘l’ THEN

Tape.Pointer -= 1

ENDIF

state <- command.FinalState

IsStepCompleted <- True

ENDIF

ENDPROCEDURE

The six machines in the program use this basic algorithm. Each one has its own small variation on the algorithm, such as not being able to write on the tape, or allowing for non-deterministic instructions.

Read-Only Variation

The only variation in the operation for this Turing machine is that the tape’s contents is not edited once the instruction is matched.

Non-Deterministic Variation

In Non-Deterministic machines, the program will choose a random instruction that matches the current state and read key. This is done using the below algorithm:

PROCEDURE Step(Commands, command, tape, state, IsStepCompleted)

ReadKey <- tape.Tape[tape.Pointer]

MatchingCommands <- []

FOREACH command in Commands

IF command.InitialState == state THEN

IF command.ReadKey == ReadKey THEN

MatchingCommands.Append(command)

ELSE IF command.ReadKey == DefaultCaseCharacter THEN

IF ReadKey NOT IN OverrideString THEN

MatchingCommands.Append(command)

ENDIF

ENDIF

ENDFOR

IF MatchingCommands.Count > 0 THEN

randInt <- Random(1 to MatchingCommands.Count)

command = MatchingCommands[randInt - 1]

ELSE

return

tape.Tape[tape.Pointer] <- command.WriteKey

IF command.MoveKey == ‘r’ THEN

tape.Pointer += 1

ELSE IF command.MoveKey == ‘l’ THEN

tape.Pointer -= 1

ENDIF

state <- command.FinalState

IsStepCompleted <- True

ENDPROCEDURE

This procedure collects all of the matching transition functions and chooses a random one to perform.

Multi-Tape Variation

The main change in this Turing machine involves the handling of default-case functions. This machine uses a more complex priority system than a binary yes/no that the other machines use. The priority is determined by how many defined characters are used – the more defined a function is, the higher its priority.

PROCEDURE Step(command, tape, state, IsStepCompleted, OverrideSequences)

IsMatch <- true

Read <- tape.ReadTapes()

Priority <- COUNT OF DefaultCaseCharacter IN command.ReadKeys

OverridePriority <- 0

OverrideSequence <- “”

FOREACH override in OverrideSequences

IF override != command.ReadKeys THEN

TempOverridePriority <- COUNT OF DefaultCaseCharacter in override

IF TempOverridePriority < OverridePriority THEN

FOR i in tape.Tapes

IF override[i] == Read[i] THEN

doesOverrideMatch <- true

ELSE IF override[i] == DefaultCaseCharater THEN

doesOverrideMatch <- true

ELSE

doesOverrideMatch <- false

break

ENDIF

ENDFOR

IF doesOverrideMatch THEN

OverridePriority <- TempOverridePriority

OverrideSequence <- override

ENDIF

ENDIF

ENDFOR

IF OverridePriority < Priority THEN

return

ENDIF

IF isMatch

FOR i in tape.Tapes

WriteTapes()

MoveTapes()

state <- command.FinalState

isStepCompleted <- true

ENDIF

ENDPROCEDURE

Multi-Track Machine

The Multi-Track machine’s step method functions identically to that of single tape machines. The only difference in the operation of this machine is the main() method. There is a loop that encloses the ‘stepping’ of a tape, so that the step() method is called for each :

PROCEDURE Main(tapes, commands, states, IsHalted, IsTapeHalted)

IsStepCompleted <- [tapes.Count]

FOREACH i in range tapes.Count

IF !IsTapeHalted[i] THEN

FOREACH command in commands:

IF !IsStepCompleted THEN

IF command.InitialState == states[i] THEN

Step(command, tape[i], states[i], IsStepCompleted[i])

ENDIF

ELSE

break

ENDIF

ENDFOR

IF !IsStepCompleted THEN

IsTapeHalted <- true

ENDIF

IsStepCompleted[i] <- false

ENDIF

ENDFOR

IsHalted <- IsAllHalted(IsTapeHalted)

ENDPROCEDURE

FUNCTION IsAllHalted(IsTapeHalted)

FOREACH tape in IsTapeHalted

IF tape == false THEN

return false

ENDIF

ENDFOR

return true

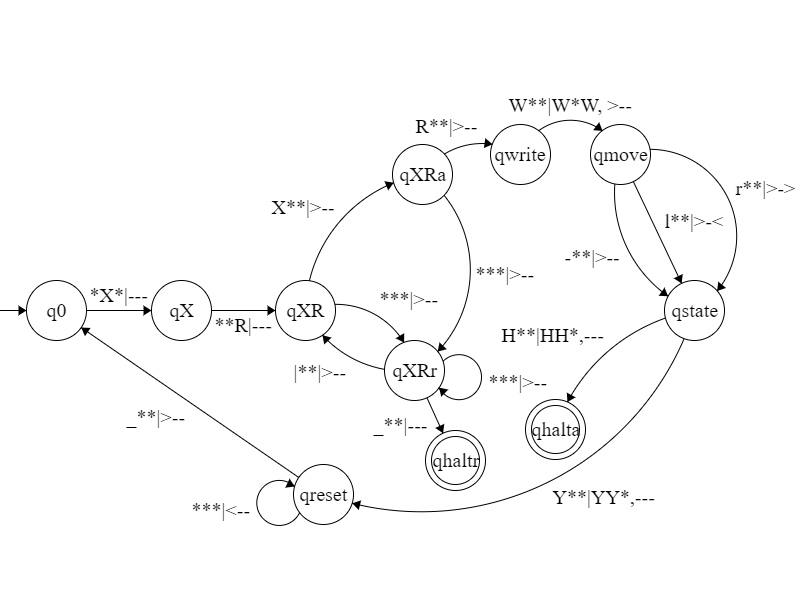
ENDFUNCTION

## Finite State Machine for Universal Turing Machine

The Universal Turing machine (UTM) uses an automatic process to generate the code to simulate another Deterministic Turing machine (DTM). This process does not produce a UTM that just ‘puppeteers’ the DTM by predetermining how it will behave when looking at certain functions. The process creates a set of transition functions that can simulate any DTM that contains the same alphabet, running state set and halting state set as the one that was selected to be generated.

For example, if I had a Deterministic Turing machine with alphabet [0, 1, -], running states [a, b] and halting states [h], running the conversion process would produce a UTM that could successfully run any DTM with those same properties.

The below Finite State machine shows the algorithm used for the UTM simulation. Because mapping a complete UTM would require dozens, if not hundreds of states, this shows a simplified form of the algorithm.

The characters used for this machine represent the following:

* X – The current state of the simulated machine.
* R – The currently selected character on the read-write head.
* W – Any character in the alphabet to replace the current character with.
* Y – A final (non-halting) state of the machine.
* H – A final (halting) state of the machine.

Using an example of a parity bit checker, the DTM’s attributes would be defined as:

* Alphabet = {0, 1, -}
* Running States = {a, b}
* Halting States = {o, e}

The syntax used for the transitions in this FSM is:

input|output

The output is split into:

write, move

When the tape is not written to at all by the machine, the write field is left blank in the FSM.

Each of these fields has three values. The first values in the {input}, {write} and {move} fields correspond to the Instruction Tape in the UTM. The second is the State Tape, and the third is the Data Tape.

The move keys in the transitions can have three values, ‘>’, ‘<’, and ‘-’. The ‘>’ character means that the read/write head on the tape will move one space to the right. The ‘<’ character is one space to the left, and ‘-’ means that the tape’s head should not move at all.

So the state qX would be split into qa and qb, with each of those states allowing to read and write characters 0, 1, and -, move either right, left, or not at all, and then transition into any one of the running states or halting states.

## Validating Turing Machines

To ensure the data given by the user is valid and will work, the values given are checked every time they need to be used. This is to make sure that incorrect entries will not create runtime errors if they are accepted as inputs for the machine. The data to be checked is the tape, tape count, transition code and initial state settings:

* The tape cannot contain the default case character, which is ‘\*’ by default.
* The tapes on a UTM must follow these additional rules:
  + The Instruction Tape in a UTM must be in the correct format. Functions are separated by ‘|’, and each function contains five fields. The two states at each end of the function must both be a single alphabetical character. The read and write keys must be anything other than blank characters (spaces), and the move keys can only be ‘r’, ‘l’ or ‘-‘. The regular expression for this rule is:

^(\w\S\S(r|l|-)(\w))(\|\w\S\S(r|l|-)\w)\*$

* + The State Tape of the UTM can only contain one character. This is because this tape must only store the current state and nothing else.
* The tape count must not exceed one for single-tape machines. There is no limit for multi-tape and multi-track machines.
* There must be a selected initial state for all required occurrences of it. A multi-tape machine must have a valid initial state for each tape, where all of the other machines only need a valid state for the first tape’s initial state box (all of the other boxes will not be visible for these machines to avoid confusion).
* Each transition function must adhere to these rules:
  + Exactly five attributes for the function, excluding a read-only function which must have four.
  + The read, write and move attributes must be a single character, excluding a multi-tape function, where these attributes must be as long as the tape count.
    - The move key for every machine variant can only contain ‘r’, ‘l’ or ‘\_’.
  + The combination of the initial state and read key must be unique for every transition function, except in a Non-Deterministic machine where these can be repeated (with different following values).
  + No machine can have a two identical transition functions.
  + The initial and final states cannot consist of spaces.

The regular expressions for each machine variant are shown below:

* For Deterministic, Non-Deterministic and Multi-Track machines:
  + The (\S)+ sections are used for the states. The states in these machines can be of any length, and contain any character. However, they cannot have spaces in them as spaces are used to mark the boundary between one field and the next.
  + Each field is separated by a space, represented by \s.
  + The read and write keys can be any character, except for a space (blank spaces are represented using ‘\_’ in these fields). This is represented in the expression by \S.
  + The move key of the transition function can have three values: right, left, or stay. The characters for these directions are given as ‘r’, ‘l’ and ‘\_’. In the regular expression, the move key is given as (r|l|\_) where | acts as ‘OR’ operations between the three characters.

^(\S)+\s\S\s\S\s(r|l|\_)\s(\S)+$

* + For example, the line:

set 0 0 r set

* + Would be accepted. However, the line:

set 0 0 f set

* + Would be invalid because its move key is not allowed to be the character ‘f’.
* For Read Only Turing machines:

^(\S)+\s\S\s(r|l|\_)\s(\S)+$

* + For example, the line:

set 0 r set

* + Would be accepted, but the line:

set 0 0 r set

* + Would not be accepted as a write-key is not allowed in this machine.
* For Multi-Tape and Universal Turing machines:
  + The syntax for these machines are the same as Deterministic Machines. However, the input and output fields must be defined for all of the tapes in the machine. This means that they must be repeated *k* times, where *k* is the number of tapes in the machine.
  + The attribute “Tapes” is used to store the number of tapes in the machine. The phrase {Tapes} specifies that the previous entry in the expression needs to be repeated by that many times.

^(\S)+\s\S{Tapes}\s\S{Tapes}\s(r|l|\_){Tapes}\s(\S)+$

* + For example, in a machine of tape count 3 this line would be accepted:

set 000 000 rrr set

* + But this line would not be accepted as the read key field is not of the correct length:

set 0 000 rrr set

## Integrity Measures

In addition to these rules when attempting to run a machine, when loading a machine from an external text file, a user may open the wrong file or open a corrupted file. Using exception handling, if an error is encountered when attempting to open a text file, this file will be rejected and an error message will be shown to the user.

There can also be errors when representing a Turing machine as an FSM. These can be caused by a lack of RAM in the computer running the program or any unforeseen bugs. In order to prevent these issues from crashing the program, exception handling is used to ensure that when an exception is thrown, it is caught and the FSM will fail to represent, with the window closing.

## Format of External Files

Text documents are used to store data for this program. They are used for two functions:

* Storing Turing machines
* Storing user preferences

Turing machine files can be opened by the user, producing a new machine tab containing all of the information stored in the file. The user can also save a created Turing machine to a text file, which when reopened will load all of the information from the machine when saved.

There is one options file stored by the program. This is kept in the debug folder of the program, and is used on start-up to load previously configured auxiliary settings for the program, such as the lower and upper limits of the speed of the machine. This file is updated every time the preferences are edited by the user.

Turing Machine files

Line 0: Machine Variant

Repeated by the number of

tapes in machine.

Line 1: Number of Tapes (Only present if using a machine where tapes can be more than 1).

Line 1/2: Tape

Line 2/3: Pointer

Line 3/4: Initial State (If multi-tape machine is used, only one occurrence at the end. If multi-track machine is used, repeat for each tape).

Line 4/5+: Transition Function Set

The Machine Variant is an ID number assigned to each machine. This is determined by its order in the selection box in the program. The order is:

* 0: Deterministic
* 1: Read-Only
* 2: Non-Deterministic
* 3: Multi-Tape
* 4: Multi-Track
* 5: Universal

These are numbered 1-6 in the selection box.

Example Turing Machine File

1

10101011110

0

a

a 1 r b

b 1 r a

a 0 r a

b 0 r b

a \_ \_ even

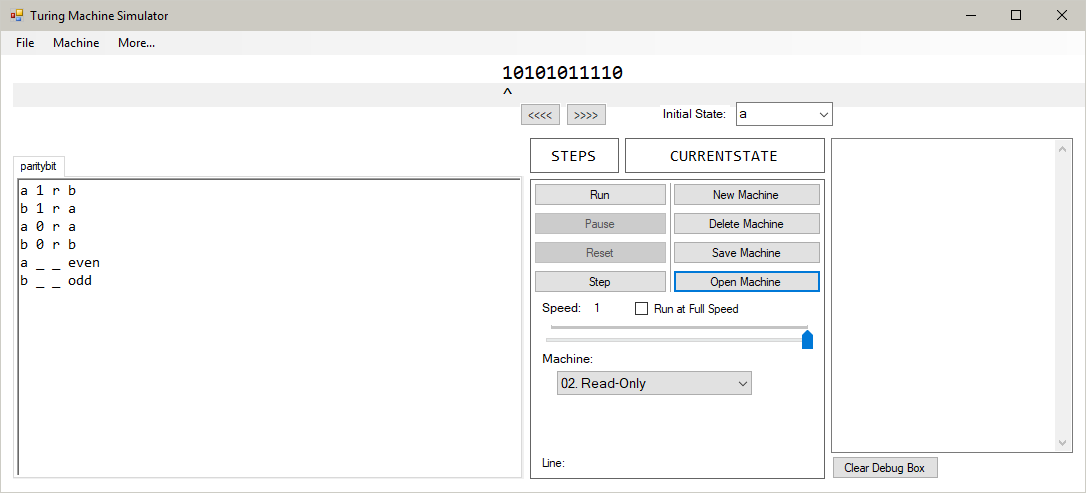
b \_ \_ odd

This is the text data for a Turing Machine. Its function is to find out the parity of a binary sequence – if there are an odd or even number of 1s in the sequence. The first line of the text file contains the number ‘1’. This shows that the machine variant for this Turing Machine is a Read-Only machine.

Because the machine is Read-Only, we know that there will only be one tape. Therefore, the next line will not contain a number specifying how many tapes there are, but will contain the tape that is to be placed onto the tape when the file is imported into the program.

Line 3 contains the number ‘0’. This number is used to show where the read/write head is in relation to the beginning of the tape. As this value is 0, we know that the read/write head is at the first character in the tape. The next line contains the initial state of the program. This tells us that the machine should start in state ‘a’. Because this machine is Read-Only, there is only one tape. Therefore, all of the lines after contain transition functions.

After loading in the text file, the fields in the program are filled in as follows:



Example Turing Machine File 2

4

3

1011+1100

0

set

1+1

0

set

1000111+10111101

0

set

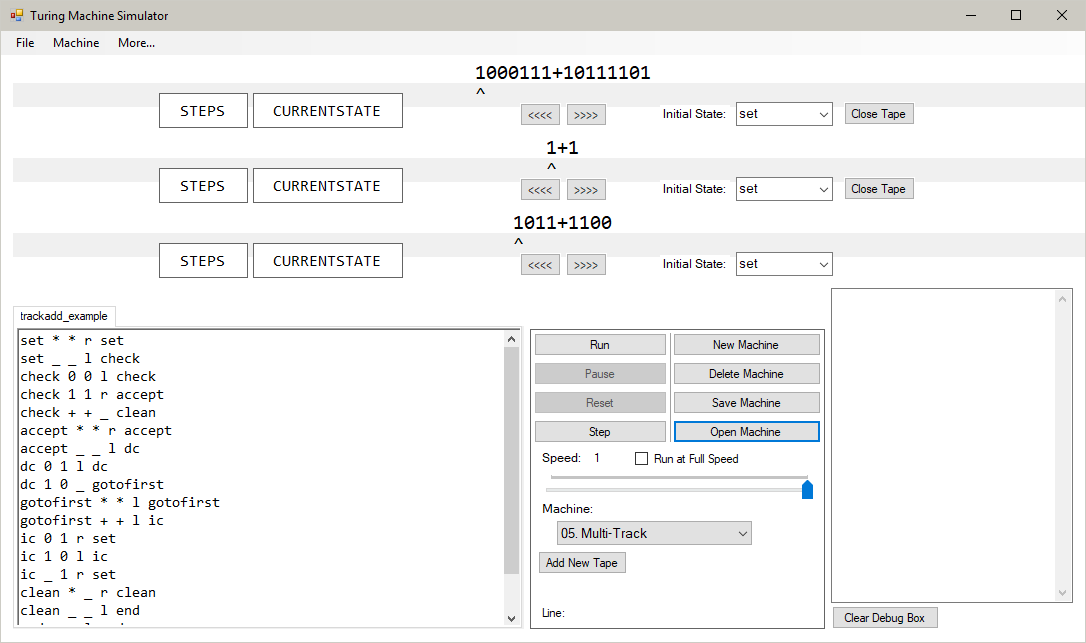
set \* \* r set

This example does not contain the full set of transition functions, for the sake of compactness. The Turing Machine’s function is to add together multiple numbers at once.

The first line of the machine contains the number ‘4’. This number, when checking with the list shown above, means that the machine is a Multi-Track machine. Therefore, there can be multiple tapes in the machine. Because of this, the next line to be read will be interpreted as the number of tapes in the machine. This line reads ‘3’, and so the program gathers three sets of tapes, pointers and states from the file.

Because the machine is a Multi-Track and not a Multi-Tape, each tape has its own initial state as each tape works independently. Therefore, the program will read the next three lines for each tape. The first tape’s parameters are given as ‘1011+1100’, ‘0’ and ‘set’. This means the tape is going to contain the string ‘1011+1100’, its read/write head will be set to the first character on the tape, and its initial state will be ‘set’. This process is repeated for the next two tapes, after which the program reads in the transition functions.

After processing this information, the program loads in the machine:



Configuration Save File

Line 0: Maximum Speed

Line 1: Minimum Speed

Line 2: Default Case Character

Line 3: Full Speed Interval Value

Example Configuration File

1000

1

\*

50

This is an example of the ‘config.ini’ file in the program’s Debug folder. It contains the four options described above. The first two lines contain the numbers ‘1000’ and ‘1’. These are the maximum and minimum values for the speed slider in the options pane of the program.

The next line contains the character ‘\*’. This is the character the program will identify as the default case character. The last line of the program contains the number to be used for the interval at full speed. With the number ‘50’, the machine will perform 50 steps before it updates the UI values of the program to speed up its operation.

## Object Orientated Programming (OOP)

Classes/ Objects

Each Object Definition below is an object used in my program. My rules for producing these objects are shown below:

* An attribute that is edited outside of the constructor is a private variable with Get/Set methods for use outside of the object.
* Attributes that are not referenced outside of the object do not need Get/Set methods.
* Procedures and functions that are not referenced outside the object do not need to be public.
* Attributes that are inherited from a base class are defined in that base class as protected, not private, attributes.
* Methods that are only called inside the class are set to private.

ReadOnlyTransition = Class

Protected:

InitialState: STRING

ReadKey: CHAR

MoveKey: CHAR

FinalState: STRING

Public:

PROCEDURE Constructor

GET methods

ENDCLASS

StandardTransition = Class(ReadOnlyTransition)

Protected:

WriteKey: CHAR

Public:

PROCEDURE Constructor

GET methods

ENDCLASS

MultiTapeTransition = Class(StandardTransition)

Private:

OVERRIDE ReadKey: LIST<CHAR>

OVERRIDE WriteKey: LIST<CHAR>

OVERRIDE MoveKey: LIST<CHAR>

Public:

PROCEDURE Constructor

GET methods

ENDCLASS

Tape = Class

Private:

Contents: LIST<CHAR>

Pointer: INTEGER

Public:

PROCEDURE Constructor

PROCEDURE AddFront, AddBack, AddMiddle

PROCEDURE TrimEdges

FUNCTION GetLength

GET/SET methods

ENDCLASS

MultiTape = Class

Private:

Contents: LIST<LIST<CHAR>>

Pointer: LIST<INT>

Public:

PROCEDURE Constructor

PROCEDURE AddFront, AddBack, AddMiddle

PROCEDURE TrimEdges

FUNCTION GetLength

FUNCTION GetTapes

GET/SET methods

ENDCLASS

TapeObj = Class(System.Windows.Forms.Panel)

Private:

Controls: CONTROLCOLLECTION

Public:

PROCEDURE Constructor

GET methods for specific controls:

Tape

Pointer

InitialState

InitialStateLabel

CurrentState

Steps

Close

ENDCLASS

TabObj = Class(System.Windows.Forms.TabPage)

Private:

Controls: CONTROLCOLLECTION

Public:

PROCEDURE Constructor

GET methods for specific controls:

CodeBox

ENDCLASS

MachineData = Class

Private:

Machine: INT

Tapes: INT

InitialState: LIST<STRING>

InitialPointer: LIST<INT>

Pointer: LIST<INT>

Public:

PROCEDURE Constructor

PROCEDURE ResetPointers, SetPointers

PROCEDURE AddTape, PROCEDURE CloseTape

GET/SET methods

ENDCLASS

Graph = Class

Private:

isReadOnly: BOOL

FUNCTION ReassembleLine

FUNCTION AssembleLine

FUNCTION GetReadKeys

FUNCTION GetWriteKeys

FUNCTION GetMoveKeys

Public:

PROCEDURE Constructor

Get Only:

Contents: DICTIONARY<STRING, LIST<STRING>>

Neighbours: DICTIONARY<STRING, LIST<STRING>>

States: LIST<STRING>

StatesHalting: LIST<STRING>

ENDCLASS

Node = Class(System.Windows.Forms.PictureBox)

Public:

PROCEDURE Constructor

PROCEDURE SetHalting

ENDCLASS

TransitionLabel = Class(System.Windows.Forms.Label)

Public:

PROCEDURE Constructor

ENDCLASS

EdgeCanvas = Class(System.Windows.Forms.Panel)

Private:

Graph: GRAPH

Public:

PROCEDURE Constructor

PROCEDURE PaintEvent

PROCEDURE DrawEdges

ENDCLASS

Inheritance

TabObj

TabPage

Node

PictureBox

TapeObj

EdgeCanvas

TransitionTag

Panel

Label

System.Windows.Forms

ReadOnlyTransition

MultiTapeTransition

StandardTransition

Node

PictureBox

Panel

TabObj

TapeObj

EdgeCanvas

TransitionTag

Label

TabPage

Composition/Aggregation

The following chart indicates whether a link between two objects is defined as composition, aggregation or association. The definitions of these terms are used as follows:

* **Composition (Filled Diamond):** An instance of Class A contains an instance of Class B. This instance of Class B is integral to the operation of some part of Class A (only a section of Class A needs to be dependent on Class B to meet this requirement). When the instance of Class A is destroyed, the instance of Class B contained by it is also destroyed.
* **Aggregation (Hollow Diamond):** An instance of Class A contains an instance of Class B. The instance of Class B is not used by Class A for some computation and acts solely as a container for the instance. When the instance of Class A is destroyed, the instance(s) of Class B contained by it are destroyed as well.
* **Association (Arrow):** An instance of Class A and an instance of Class B interact here. Neither contain the other, and the lifetimes of the two instances are independent. They are usually known by each other through an object reference and exchange information which can be used for computation.

**TapeObj**

**Simulator**

**FiniteStateMachine**

**TabObj**

**Graph**

**EdgeCanvas**

**Node**

**TransitionTag**

**MachineData**

**InfoWindow**

**OptionsMenu**

**RenameWindow**

The Simulator Form object is linked to multiple objects by composition. These objects are ‘bound’ to the form. This means that when the form is closed and destroyed, these objects are also destroyed. While this is true, all of these objects can be destroyed separately (tabs can be closed, tapes can be closed, etc.) without destroying the Simulator object. Also, the objects that are linked by composition are integral to the operation of a certain function in the instance of the Simulator class. For example, the Simulator class cannot alter certain parameters of the machine without using an instance of the OptionsMenu class to get the user to input information.

The Simulator Form subclass also contains a number of objects that are linked by aggregation. These include the FiniteStateMachine and InfoWindow objects. These objects’ lifetimes depend on the Simulator still existing – but they are not integral to the operation of the Simulator object. The objects linked by composition provide a necessary service to the Simulator form to allow it to run. An example of this is the TapeObj class, which stores the contents of a tape in the machine – an integral part of the program.

The MachineData class is an object that is not linked by composition or aggregation, rather by association. This object is not integral to the operation of the Simulator object, as the information given by this object is already stored in the list of TabObj objects. This is used in the Simulator object as an object reference, allowing easier access to information. The object reference can change to any instance of the MachineData class, without the previously referenced object being destroyed. Thus, the two are not linked by composition, as the object is not integral to the Simulator’s operation, and they are not linked by aggregation, as the lifetime of the object does not depend on the Simulator.

While the MachineData class is only linked to the Simulator class by association, it is linked to the TabObj class by composition. Each instance of the TabObj class contains an instance of the MachineData class that is used to store information about that machine. Without this MachineData instance inside the TabObj, the object cannot function. In addition to this, when a TabObj instance is deleted from the Simulator, the MachineData object that is contained within the TabObj is also destroyed. Therefore, the MachineData class is linked to the TabObj class by composition.

The FiniteStateMachine class also has links to other classes. An instance of this class contains a number of Node and TransitionTag objects, a Graph and an EdgeCanvas. The TransitionTag objects are not used in the FiniteStateMachine to perform any computations. Therefore, their link is only one of aggregation, as their lifetimes are still bound to their container object. The link between the other objects and the FiniteStateMachine class is compositional. This is because the FiniteStateMachine class needs to use these objects in its computations in order to function correctly. In addition, their lifetimes are bound to the lifetime of the FiniteStateMachine instance that they are contained by – when the FiniteStateMachine instance is destroyed, so are these objects.

The EdgeCanvas class contains an object reference to a Graph object. This instance of the graph object will be the same instance as the one contained by the instance of the FiniteStateMachine that contains the EdgeCanvas object. The operation of the EdgeCanvas is facilitated by the Graph in the DrawEdges() subroutine, which produces the edges and TransitionTag objects in the FiniteStateMachine form. Despite the fact that the presence of a non-null Graph reference is imperative to the operation of the EdgeCanvas, the Graph is not contained by the object – it is contained by the FiniteStateMachine object. Therefore, these objects are linked by association. This can also be shown by the fact that the lifetime of the Graph object is not determined by the EdgeCanvas object. They are destroyed at the same time (when their container, the FiniteStateMachine object, is destroyed) but the destruction of the EdgeCanvas is not the cause of the Graph object to be destroyed.

Graph Abstraction

I decided to represent my graph of states using a dictionary as an adjacency list. The dictionary’s key is the initial state of the transition function. In the graph, this is the node the edge originates from. The value of each key is another dictionary. This dictionary’s key is the final state of the transition function, which, in the graph is the node that the edge arrives at. The value of each key in this dictionary is the information to be printed on the edge when it is represented to the user.

The graph class also contains some additional attributes that are designed to make the accessing of certain pieces of information faster. There is an attribute called ‘neighbours’, which is a dictionary with the key being the initial state, but its value containing a list of the possible final states instead of a dictionary that also contains information about the transitions. This attribute is useful for instantiating the objects used for the nodes in the graph, as there is no need to use the information for the labels at that stage, and spending time removing them from a list is unnecessary at that stage.

The other attribute in the graph class is ‘statesHalting’. This is a list that stores all of the halting states in the Finite State Machine. This list list is especially useful when creating the nodes for the graph, as this list can be used to easily decide if the image for the node needs to be changed to reflect the fact that it is a halting state.

Example Graph

**graph:**

“state1”: [“state2”: “1 | 0 >”,

“state3”: “0 | 1 >”,

“state1”: “\* | \* >”]

“state2”: [“state2”: “1 | 1 >”,

“halt”: “\* | \* -”]

“state3”: [“state3”: “0 | 0 >”,

“halt”: “\* | \* -”]

**neighbours:**

“state1”: [“state2”, “state3”, “state1”]

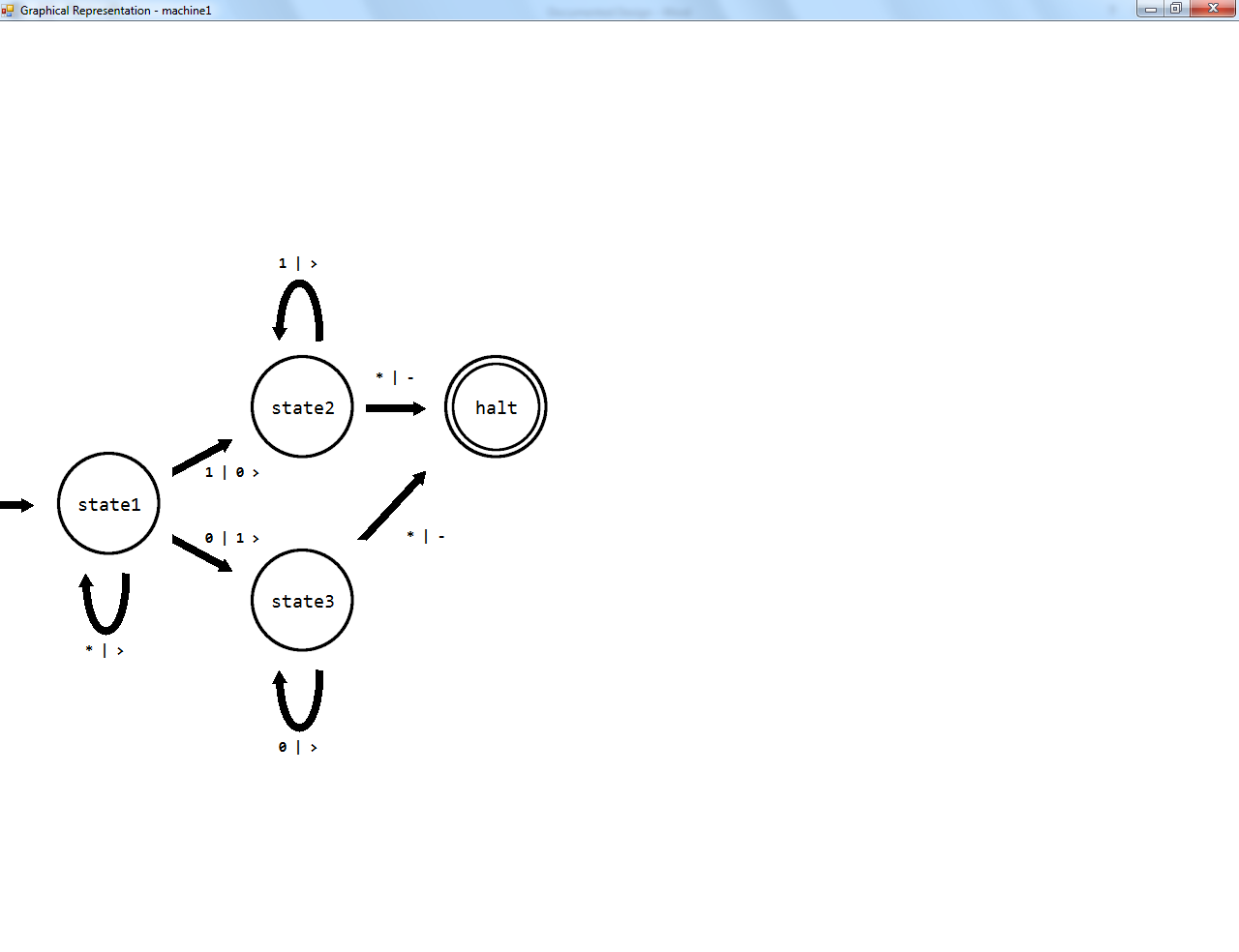
“state2”: [“state2”, “halt”]

“state3”: [“state3”, “halt”]

**statesHalting:**

[“halt”]

This graph is visually represented as:



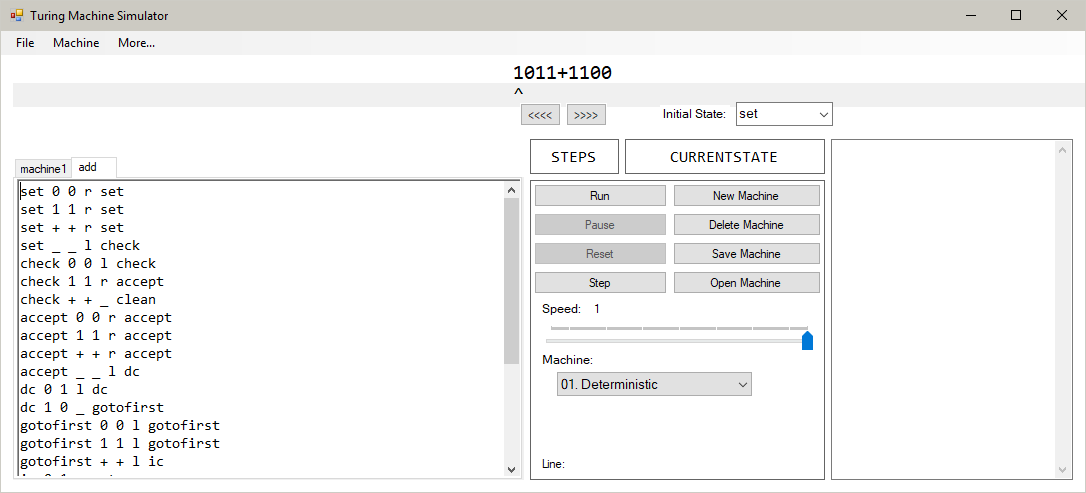
Use of Classes Inherited from System

A number of my classes are inherited from the System.Windows.Form namespace. These classes were used to create a certain UI element with set properties. The reason for creating an inherited class for these objects was to reduce the repetition in my program. For example, the TabObj class is used to create a new TabPage containing a rich text box when opening a new Turing machine. This can happen in three different subroutines. Initially, I copied and pasted the program code into all three subroutines. However, when maintaining the code, I found it very inefficient. Therefore, I decided to make a class to instantiate whenever I need to open a new tab which I could reference easily and only need to maintain in one place.

Another reason was for complexity. When creating the diagrammatic representation of the Turing machine, I need to create a number of PictureBox controls that all have the same dimensions, usually have the same image and additional similar properties. These are only added in one place in the program. For example, placing this code in the middle of Spawn(), an already long and complex method, would have made the method less easy to understand and maintain. Therefore, I moved this into its own class and file where it can be more easily tweaked independently of the rest of the program.

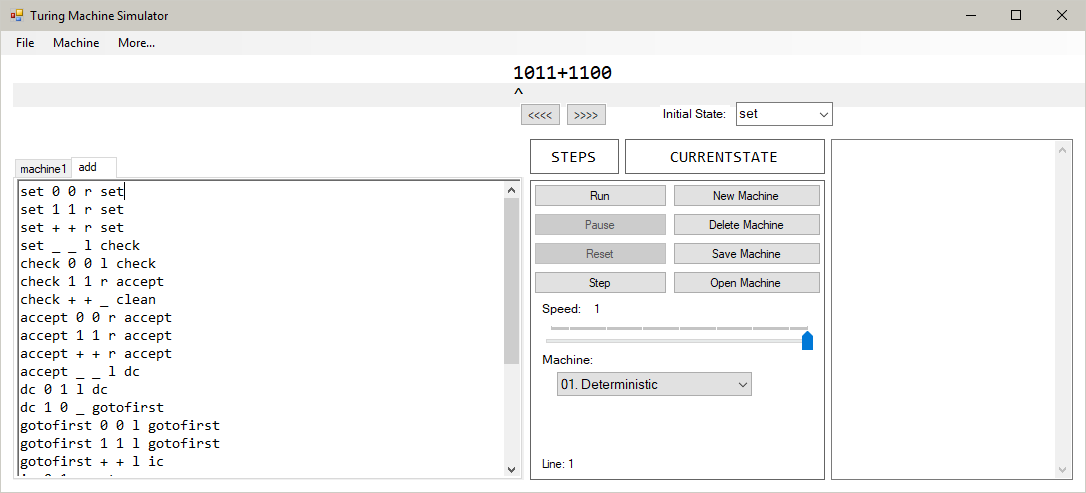
## Human-Computer Interface

Screenshots of the final layout all windows in the program can be found in the Appendix (pg. 301).



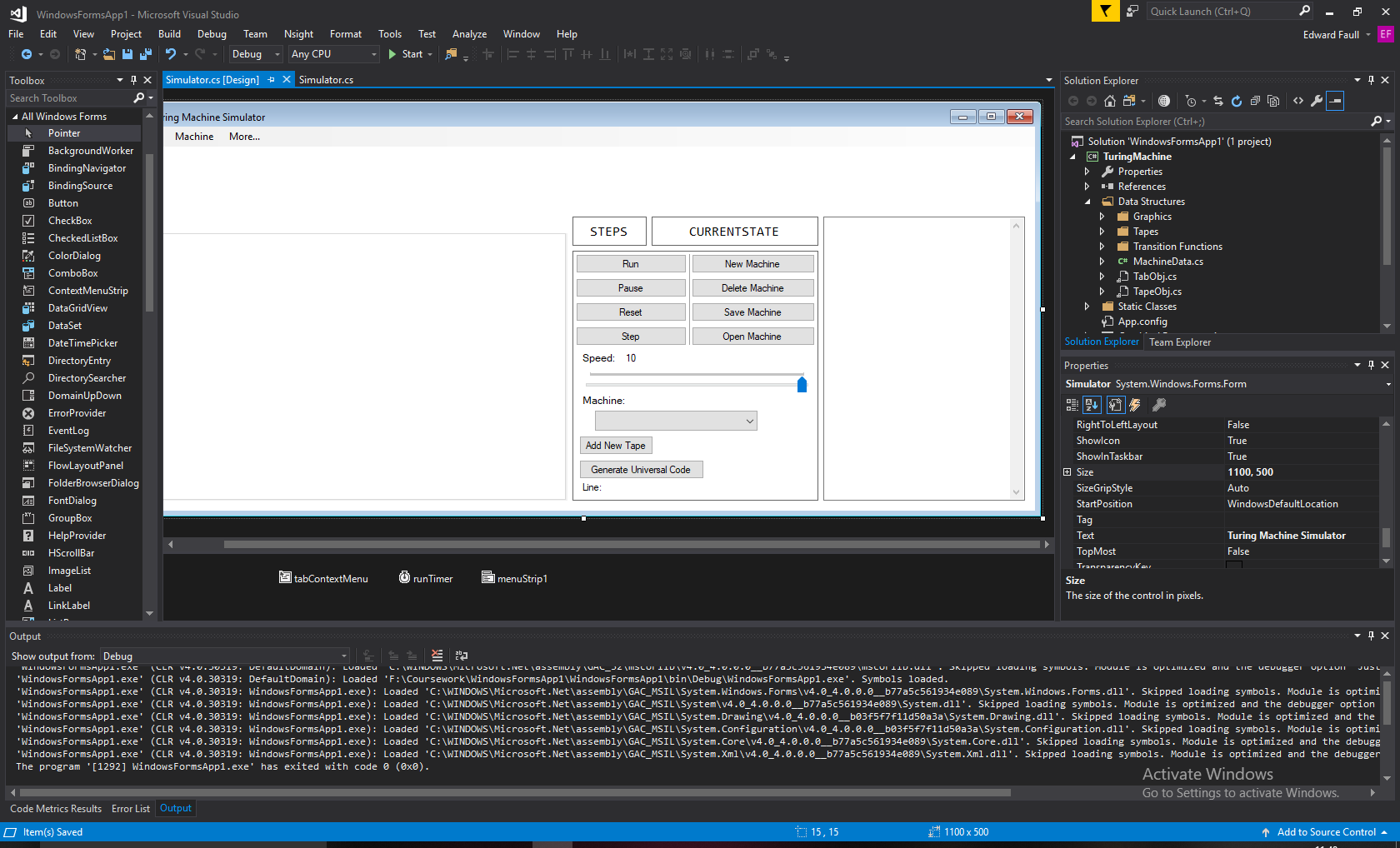
When designing the user interface for my program, I decided I wanted to keep the main section of the program (creating and running the machines) all in one window. This would ensure it would be easy for a user to access any tools they need to quickly. Because there are quite a few options to modify in this program, I decided to split the information into panels to make it easier to understand and navigate.

Code editor:



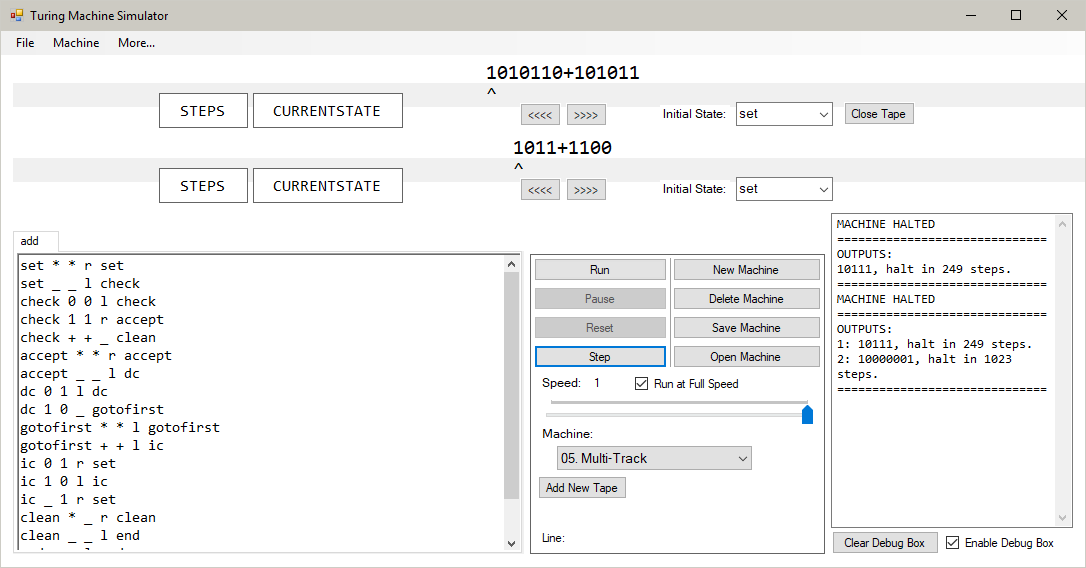
I decided to use a simple text box for the editing of the Turing machine’s transition functions. While this could have been performed using a series of text boxes for each function, adding more on when the user wants to make a new function, I thought it would be much more responsive when editing to be able to simply type in code, much like an IDE such as Visual Studio. A line counter was also included to tell the user what line they are selecting. This is useful, as when a syntax error message is given the line where the error is found is shown, so the user can easily find which line has an error in.

Options:



The options menu lets the user create new machines, run machines and configure certain aspects of its running. To make sure this interface is easy to use, I have split each option into areas. For example, the buttons associated with running the machine are important, so they are at the top, but are there is a clear vertical divide between them and the buttons to open and delete machines. Also, to make the pane easier to use, when certain options are not available, they will disappear. When using a single tape machine, the “Add New Tape” button will not appear. This makes sure certain options are not selected when it is not possible to do so. This is the same when trying to use “Generate Universal Code”; this option is only visible when using a Universal Turing machine. Some buttons when not available will be greyed out and disabled as well. For example, when a machine is not running, it cannot be paused. Therefore the “Pause” and “Reset” button will not be able to be used. The same can be said for the “Reset” and “Step” buttons when the machine is running, as the machine must be compiled but paused for these to be available.

Debug:



When running the machine, the user may wish to see the results of their machine after resetting it. The debug window allows for this by saving information about the results of the machine. This includes:

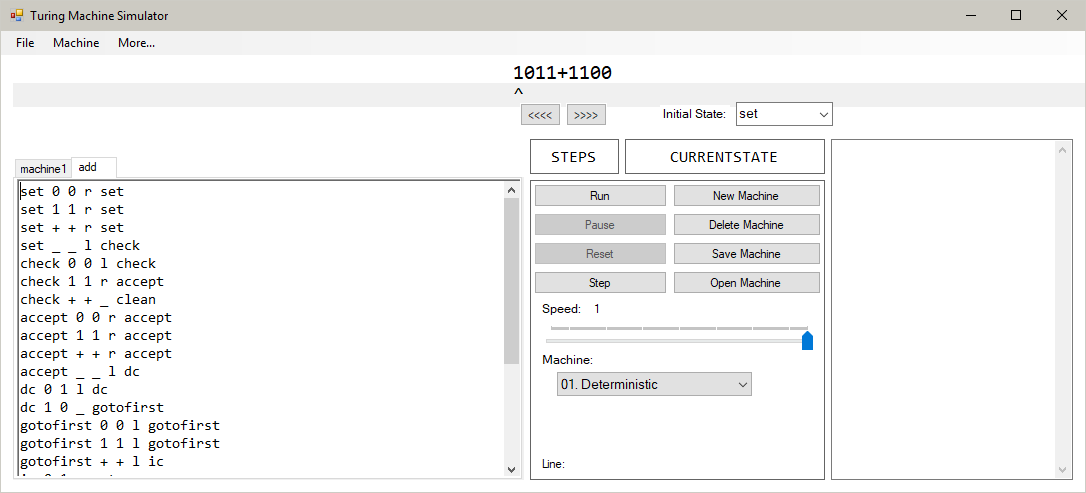
* Final contents of the tape
* Final state of the machine
* Number of steps completed

For the Universal Turing machine, as its purpose is slightly different from the other Turing machines available in the program, will give its results in a slightly different format to a standard multi-tape machine. The output will be given in the standard form for a Deterministic Turing machine, instead of a numbered list of the tapes followed by the final state and step count. The third tape in the UTM will become the tape value, the second, the final state, and the step count can be taken from the counter as usual.

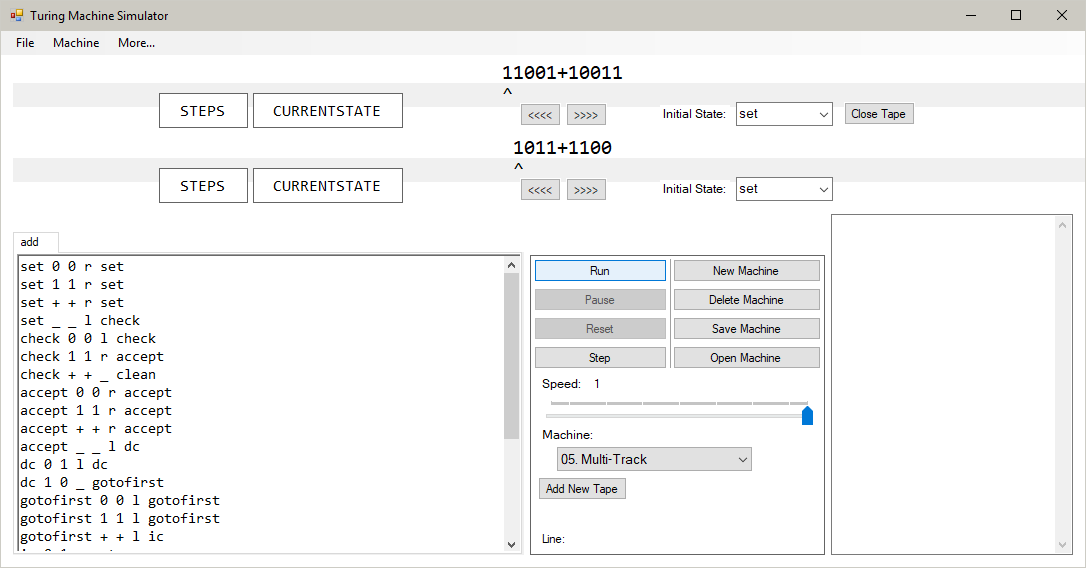
These pieces of information are repeated for each tape in the machine, and when more than one tape is present, such as in multi-tape and multi-track machines, each line will be numbered.

This information can help users compare methods for different algorithms if they are trying to find the most efficient way to complete a task.

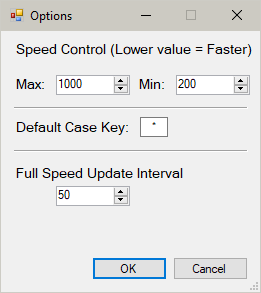
Tape:



The tape will move and change during the machine’s operation. As such, the tape and read/write head are text boxes, which allows them to be easily edited. In order to avoid the tape and pointers not aligning, the monospaced font Consolas is used for both objects. There are also buttons to adjust the initial position of the read/write head (not usable when running) and a drop-down box to choose the initial state of the machine. These initial state boxes can appear and disappear depending on the type of machine in use. For a multi-tape machine, as the state is persistent across all tapes, only the bottom one is visible. For a multi-track machine, as each tape works independently, each one has its own initial state setting, as well as its own current state box and step counter, which halts when its tape does:

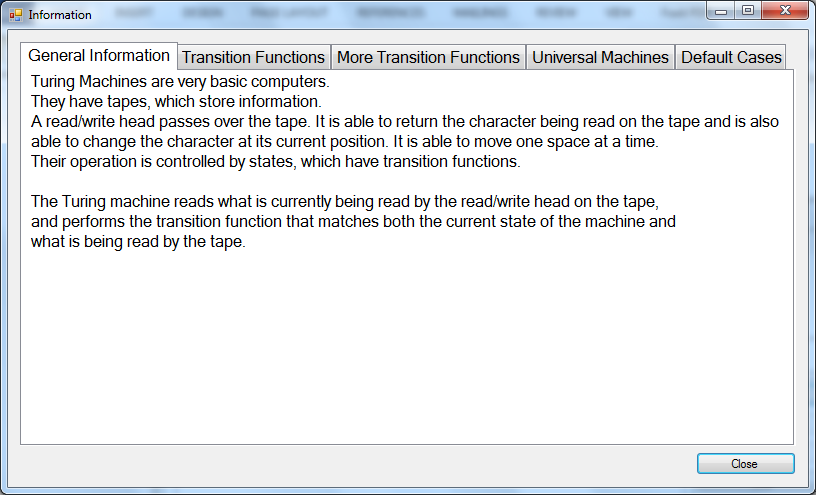


Settings:



This window allows the user to change certain aspects of the program. These aspects are parameters that are not strictly integral to the performance of the program, but provide additional utility for users that want to use different values from the default ones. Once a valid set of preferences have been set, they are saved to a text file (Format of External Files, pg. 34) so that they can be loaded into the program every time it opens. This means that the user does not need to manually set their preferences every time the program is opened.

Information:



This window provides information to the user on how to use the program. The information is split into categories that can be switched between easily using the tabs at the top. The layout of this window is as simple and straightforward as possible to make sure that the user understands what topic is being explained. The contents of the window is also laid out plainly to ensure the user does not get confused. Examples are also given when it helps the explanation of the subject:

