FINITE STATE MACHINES

INTRODUCTION TO FSM’S

STATE TRANSITION DIAGRAMS REVISITED

TYPES OF FSM

FSMs WITH OUTPUTS

FINITE STATE AUTOMATA (FSA)

FSM ACTIVITY

DETERMINISTIC AND NON DETERMINISTIC FSAs

TURING MACHINES

INTRODUCTION TO TMs

SIMPLE WORKED EXAMPLE FROM TEXTBOOK

TURING MACHINE SIMULATION ACTIVITY

HOW TO USE TURINGKARA

UNIVERSAL TURING MACHINES

SIMPLER TURING MACHINE USING KARA

BIBLIOGRAPHY
FINITE STATE MACHINES

INTRODUCTION TO FSM’S
As students will know from the AS Computing course, FSMs are used extensively in many branches of Computer Science. On the electronics / control side of the subject, they are used to model and design lifts, vending machines and traffic light controllers. Elsewhere, they are also used for specifying a language i.e. when given a string, an FSM is able to determine if that particular string is valid for that language as part of the syntax checking process. They are extensively used in spell checking, processing text containing HTML or XML and even in networking protocols that specify how computers communicate. However, as powerful as they are, FSMs cannot model English grammar.

FSMs can be used to match letter patterns in any piece of text involving any non-blank wildcard characters such as *.

One way of categorising FSMs is to consider them as being with or without outputs. In the context of the new A2 topics such as regular expressions and grammars, a FSM consists of a set of input symbols (this is simply the input symbol alphabet) and if it is a FSM that produces outputs it will have an output symbol alphabet. There will be a finite number of states and transitions that link these states together. This may generate an output if it is that type of FSM.

STATE TRANSITION DIAGRAMS REVISITED
The diagram below represents the state transition diagram for an FSM with outputs. Following on from the AS course and linking it into the new A2 topic on graph data structures, a state transition diagram is a directed graph with nodes that represent the states and transitions that link them.
The diagram can be interpreted in the following way: s and t are both nodes which represent the states. An edge such as the curved line from state s to state t is called an edge and is labelled with a symbolic code e.g. a ∣ b where a and b are symbols. The a part of the label is called the transition’s trigger and the b part (which is optional i.e. if it is a FSM without outputs, it is not needed) denotes the output symbol. Note that we have an arrow which appears from nowhere on the left side of the diagram which points to the initial state s.

Show your students this diagram and explain all about the transitions and the input and output alphabet. In this example we can write down:

input alphabet = \{a, b\}
output alphabet = \{a, b\}

The curly brackets are just used to enclose the members of the set. For this very restricted alphabet of just two symbols a and b, valid input strings can be any combination of a’s and b’s.

Having drawn a state transition diagram, you can represent this in tabular form.

<table>
<thead>
<tr>
<th>Current State</th>
<th>s</th>
<th>s</th>
<th>t</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Symbol</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>Next State</td>
<td>t</td>
<td>s</td>
<td>s</td>
<td>t</td>
</tr>
<tr>
<td>Output Symbol</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

Talk this table through with your students referring back to the diagram. What does this FSM do? It simply converts the symbol a to b and vice versa. If you were to input the input string abba, it would produce an output string baab for this FSM.

We next need to understand the difference between deterministic and non deterministic FSMs. The diagram drawn earlier has just one possible next state for each pair if state and input symbols. Thus it is said to be ‘deterministic’.

By contrast, if there were an ambiguity e.g. between the self transitions for state s and the exit state to state t because the same symbolic code a ∣ b were accidentally used to label both transitions, this would be said to be non-deterministic. A general definition for this would be to say that it is an FSM with several (i.e. more than 1) possible next states for each pair of input symbol and state.

There may also be halting states; these are when States have no outgoing transitions. If the FSM hasn’t processed all of the input string when it enters one of these will produce an error. FSMs can also be designed to carry out an action at the output stage instead of outputting a symbol.

There is a good non computing example for a state transition diagram and table for a digital watch that toggles to display the time and the date.
There are some good questions on FSMs in the textbook that you might like to use with your classes.

**TYPES OF FSM**

There are several different ways of categorising FSMs depending upon whether you are a Computer Scientist, Electronic Engineer etc. We will use the definition applicable to AQA GCE Computing here.

![Diagram of FSMs](image)

Traditionally, we teach Moore machines first in Electronics / Control Engineering. In this type of FSM, output is associated with a state. By contrast, in Mealy machines, output is associated with a transition.

A FSM without outputs is called a Finite State Automaton and their use is restricted to decision problems i.e. problems that are solved by answering YES or NO. Consequently they don’t write anything, they just assess if something is true or false. They are used as recognisers of valid strings or sentences in a language. Apart from syntax checking, they are also used in parity operations i.e. is the bit stream odd or even parity?

**FSMs WITH OUTPUTS**

These are the most common controllers of machines such as lifts, traffic lights, vending machines and washing machines. The traditional first steps in explaining about these would be to consider a pedestrian crossing which has lights to control the traffic and lights to control pedestrians.

You will recognise the following example from the A2 textbook. The traffic lights for vehicles are Red (R), Amber (A) and Green (G); the ones for pedestrians are red (r), green (g) and flashing green (fg).
The required FSM would have the following attributes:

- A finite set of states: Gr, Ar, Rg, RAfg as described in the table below.
- Designated start state: Gr (i.e. Traffic light on Green and Pedestrian Light on red).
- Input Alphabet: B for “button B pressed on Pedestrian lights and T for “Timer timed out” on Traffic light controller
- Transition function that assigns a next state to every state and input: i.e. if in state Gr, then go to state Ar.
- Output function that assigns to each state and inputs ab output action: e.g. Green off,

<table>
<thead>
<tr>
<th>Traffic Light</th>
<th>Pedestrian Light</th>
<th>State Codes</th>
<th>Explanation of State Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>red</td>
<td>Gr</td>
<td>Traffic light Green, Pedestrian light red.</td>
</tr>
<tr>
<td>Amber</td>
<td>red</td>
<td>Ar</td>
<td>Traffic light Amber, Pedestrian light red.</td>
</tr>
<tr>
<td>Amber</td>
<td>flashing green</td>
<td>RAfg</td>
<td>Traffic lights Red and Amber, Pedestrian light flashing green.</td>
</tr>
</tbody>
</table>

Amber on, start timer.

There will be two inputs to the TL controller:
- When Button B is pressed by the pedestrian, there will be a signal
- Then the time interval on the timer has elapsed i.e. the timer has timed out there will be a signal

There will be two types of outputs from the TL controller:
- Signals to the different coloured lights to switch them on and off
- A signal to start the timer when required.

This is a simple “on demand” system i.e. the controller reacts to the Pedestrian pressing button B only when it is in the Gr state (Traffic light Green and Pedestrian light red). Assuming the specified Gr state and looking at the Moore machine representation first. Remember, Moore machines are FSMs that determine its outputs only from the current state and not the actual inputs.

For a Moore FSM, the current state is determined by the sequence of inputs that it has processed so it is effectively remembering the input sequence. In this way, it can be considered to model RAM. Any electronic circuit with a memory (like a bistable Flip Flop) is a FSM. Computers can be considered as huge FSMs but I wouldn’t like to try to model them!!!
For every Moore Machine representation, there is an equivalent Mealy Machine representation but this time the outputs are determined not just by the current state but also by the inputs when it is in that state.

Moore Machine State Transition diagram with Outputs for Pedestrian Crossing system

For every Moore Machine representation, there is an equivalent Mealy Machine representation but this time the outputs are determined not just by the current state but also by the inputs when it is in that state.

Mealy Machine State Transition diagram with Outputs for Pedestrian Crossing system
FINITE STATE AUTOMATA (FSA)
This is simply a FSM that produces no output while processing the input. When it has finished processing the input, it just has to come up with a YES or NO answer. The state that causes it to produce a YES response is called an accepting state; all others by default must cause a NO answer. Accepting states are shown by drawing them with two concentric circles. A FSA always has one or more accepting states. Start states also need to be labelled.

The machine shown in the state transition diagram below begins in the start state S0 and will move to state S1 if the first symbol is an a or to state S2 if it is a b. From state S1 it moves to the accepting state S3 if the second symbol is a b. Alternatively, it will move to state S2 if another a is input. S2 is not an accepting state. The machine can leave the accepting state S3 to go to S2 if there is any further input i.e. an a or a b. However, from state S2 there is no possible transition to anywhere else because it has a self transition regardless of an a or a b being input. If the FSM is in state S3 when the input string is exhausted, the machine outputs a NO.

This only accepts the string ‘ab’, all others are not valid

FSM ACTIVITY
Use a FSM simulator to represent the FSM as shown and try different combinations of a’s and b’s. At least one combination should be ab but try strings like ba and abba.

I will demonstrate this using the bespoke AQA A2 FSM simulator marketed by educational-computing.co.uk. Unfortunately, the excellent free FSM simulator Boole Deusto can only simulate Moore and Mealy Machines, not FSA.

There are a lot more examples in the textbook that can be worked on theoretically or using an appropriate simulator.

DETERMINISTIC AND NON DETERMINISTIC FSAs
All of the examples considered to date are deterministic (DFAs) i.e. for each pair of state and input symbol there is just one next state. Sometimes we need to use non-deterministic types (NFAs). A particular example of these is in pattern searches in word processing and searching through documents. Any problem that can be solved with a non-deterministic FSA can be converted into a deterministic form but many more states may be needed.

The FSA simulator demonstrated earlier actually checks for determinism but has basic support for NFAs.
TURING MACHINES

INTRODUCTION TO TMs

Turing Machines are basic abstract symbol-manipulating devices which, despite their simplicity, can be adapted to simulate the logic of any computer algorithm. They were described in 1936 by Alan Turing. Turing Machines are not intended as a practical computing technology, but a thought experiment about the limits of mechanical computation. Thus they were not actually constructed. Studying their abstract properties yields many insights into computer science and complexity theory. A Turing Machine is a FSM that controls one or more tapes where at least one tape is of infinite length.

In 1934, Turing tried to produce an abstract model of computation that would tell him if the solution to Hilbert’s Tenth problem i.e. if $x^2 + y^2 = 1$ has roots that are whole numbers. He conceptualised a human computer and a piece of paper which was divided into squares and was hence 2 dimensional. The A2 textbook shows how this could be done to add two simple binary numbers. He showed that he didn’t need to use 2 D squares and that a piece of 1 D paper i.e a tape could be used instead. He further limited the symbols that could be written on the tape and for the purpose of A2, we will just consider 0’s and 1’s.

He published a major mathematical paper in 1937 called “On computable numbers with an application to the Entscheidungsproblem”. As far as Turing was concerned, a “computer problem” was a precise set of instructions given to the human computer relating actions to observations. A simple example would be if you read a 1 in a square on the tape (this is the observation), replace it by a zero and move one square to the left (these are the actions).

His fascinating article stated that simple operations included the following:

- A change of symbol on one of the observed squares
- Change of one of the squared observed to another square within a certain number of squares of one of the previously observed squares.

This is the level at which we need to work for A2 GCE Computing.

SIMPLE WORKED EXAMPLE FROM TEXTBOOK

Consider a Turing Machine that tests if a given character string consists of n zeroes followed by n ones e.g. the valid string 0 0 1 1 which is sandwiched between the delimiting symbols #. The “observation window” is the larger square on the leftmost # symbol.
If the input string is valid, the Turing Machine will halt with only empty squares between the # symbols. If it is not valid, it will halt with 0 or 1 symbols on the tape or it will report an error.

Using a special set of symbols as listed below, you can draw the state transition diagram for this simple Turing Machine.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Separates input from output</td>
</tr>
<tr>
<td>□</td>
<td>Blank Square</td>
</tr>
<tr>
<td>#</td>
<td>Delimiting symbol</td>
</tr>
<tr>
<td>→</td>
<td>Move the observation window one square to the right</td>
</tr>
<tr>
<td>←</td>
<td>Move the observation window one square to the left</td>
</tr>
</tbody>
</table>

You will probably also need the other table below to understand the transition diagram.

<table>
<thead>
<tr>
<th>Transition Label</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 0</td>
<td>←</td>
</tr>
<tr>
<td>#, □</td>
<td>→</td>
</tr>
<tr>
<td>0</td>
<td>□</td>
</tr>
<tr>
<td>1</td>
<td>□</td>
</tr>
<tr>
<td>0, #, □</td>
<td>#</td>
</tr>
</tbody>
</table>

State Transition diagram with outputs for the Turing Machine to validate that a given string contains n zeroes and n ones e.g. 0011
The Turing Machine example given here reads the symbol i.e. the input that the observation window is currently on and if necessary, replaces this symbol with another i.e. the output. The Turing Machine may move the observation window one square to the left or one square to the right. Probably the best way to illustrate this topic is now to use the TuringKara simulator.

First, you will need to know the transition rules to be able to program it.

<table>
<thead>
<tr>
<th>Current state</th>
<th>New State</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go Left</td>
<td>Go Left</td>
<td>1, 0</td>
<td>←</td>
</tr>
<tr>
<td>Go Left</td>
<td>Read 0</td>
<td>#, □</td>
<td>□ →</td>
</tr>
<tr>
<td>Read 0</td>
<td>Go Right</td>
<td>0</td>
<td>□ →</td>
</tr>
<tr>
<td>Read 0</td>
<td>Stop</td>
<td>1, #, □</td>
<td>nothing</td>
</tr>
<tr>
<td>Go right</td>
<td>Go Right</td>
<td>1, 0</td>
<td>→</td>
</tr>
<tr>
<td>Go right</td>
<td>Read 1</td>
<td>#, □</td>
<td>□ ←</td>
</tr>
<tr>
<td>Read 1</td>
<td>Go Left</td>
<td>1</td>
<td>□ ←</td>
</tr>
<tr>
<td>Read 1</td>
<td>Stop</td>
<td>0, #, □</td>
<td>#</td>
</tr>
</tbody>
</table>

TURING MACHINE SIMULATION ACTIVITY

Unless you have recently downloaded software to be able to run Java applets on your laptops, you will almost certainly need to download the current version of the Java Runtime Environment (jre-6u14-windows-i586-p-s.exe) from the Sun Java site BEFORE downloading and running the turingkara application.

Next, visit the URL http://www.swisseduc.ch/informatik/karatojava/download.html and download the program: turingkara-en.jar (this is the English version with solutions but other versions are also available). One way to download this is to move the cursor over the filename turingkara-en.jar then right click Save Target As … In the Save As dialog box at the bottom of the form, you need to select Save as type All Files and add the file extension .jar to the filename to create turingkara-en.jar in the File name box. If you have been successful, you should see the usual java type icon which you need to double left click to run the executable jar file.

Apparently you can install and run both the Sun JRE and TuringKara from a memory stick so as not to upset your Network Manager!! TuringKara is actually a 2-D Turing Machine simulator but we can just use it in a 1-D way. In this simulation, the other restriction is that the “tape” is not infinite in both directions but this won’t matter here.

HOW TO USE TURINGKARA

Assuming that you have downloaded the JRE software, you should just be able to click on the executable turingkara-en jar file and it should produce a screen as below:

The first thing to do is to set the Karaworld to something manageable. Move the Cursor to the box on the bottom left and click once. Set the height and width to height = 1 and width = 8.
Now just select the tick. This will give you an “infinite” One Dimensional tape of just 8 cells or squares.

Using the toolbar on the right-hand side, drag all the required symbols into position not forgetting to set the start cell to the leftmost # symbol so that it is boxed in red.

It might be a good idea to save this project now.

Call it something like OnesAndZeros and you can choose where to store it. A file extension of .world will be attached to it.

It is probably a good idea to set the preference using the Tool symbols to a white background for later work with the transition table.

**World objects**

In TuringKara's world, the following symbols can exist in addition to the read-write head:

```
0 1 # ← → ↓
```

- The symbols:
- The read-write head of TuringKara.

The symbols have no meaning for TuringKara and can therefore be used for any purpose. For example, a program can add two numbers which are written in binary notation using 0 and 1, or the numbers could be written using left- and right-arrows to represent the 0s and 1s.
WORLD EDITOR

- World: new, open, reload, save, save as
- Show exercises window
- Show preferences dialog
- Control TuringKara directly (write commands only)
- Put objects in world:
  - click & place, or
  - drag & drop
- Change world size
- Zoom world view
- Speed of execution
- Control program execution

PROGRAM EDITOR

- Program: new, open, reload, save, save as
- States overview diagram
- State table view
Next, you need to click on Programming in the top left corner which will produce a blank canvas for you to enter the state transition diagram. The Stop or Halt state which is a double ring will already be there for you.

You will probably need to maximize the window and also drag the boundary between the diagram and the table downwards. You now need to add the other states i.e. Go Left, Read 1, Go Right and Read 0 as well as indicating the Start State.

Adding the transitions is easy. Just hold the left mouse / touchpad button down and drag. Self-transitions are also pretty easy to set up. Don’t forget to keep saving every time you alter something, editing is primitive and you might have to start again!

With a bit of luck and patience, you should have created a screen like that below.
Now comes the challenging part. We have to set up the state table for the four states whose tabs will have been created underneath this state transition diagram.

I will just demonstrate how to do this here for the start state that we have called Go Left and is the highlighted state above. In the bottom half of the screen, this is where you set up the states.

The above state table view for the current state can be interpreted in the following way:

- If we are in the state **Go Left** and either a 1 or a 0 are sensed, the “output” is a move of the tape one space to the Left but we remain in the **Go Left** state.
- If we are in the state **Go Left** and either a # or a □ (i.e. a space symbol) are sensed, the “output” is a space and a move of the tape one space to the Right and we change state to **Read 0**.

You have to complete the state table for all of the other states except for the halting state called Stop. You then save the model again. Do not forget that you set up the world with the chosen bit pattern in the first part of the exercise.

Since the Turing Machine does not have internal memory to count the symbols, one solution is to erase ones and zeroes in pairs. This can be done in four steps.

1. Go to the Left end.
2. Read and erase symbol (halt if is not a zero).
3. Go to the Right end.
4. Read and erase symbol (halt if it is not a one).
These four steps are executed by the FSM Simulator until all symbols have been erased. If the Read / write head is not on a 1 in step 4, a # symbol is written before the Turing Machine halts. Otherwise it would incorrectly halt in a Kara World for strings that have exactly one zero more than they have ones.

The KaraTuring Simulator is incredibly powerful, especially when it used in its normal 2-D mode. There are inbuilt exercises for several interesting situations including several Boolean Logic tasks, Binary addition, subtraction and multiplication as well as Modulo operations and Binary to Unary transformation. Another activity would be to investigate bitwise logical operations using the 2-D capabilities. It is much more difficult to solve the same problem with a 1-D tape because of the much greater movements of the Read / Write head.

Alternative Turing Machine simulators are available. You could download the Java Applet from http://www.ironphoenix.org/tril/tm which has a number of examples but although it is a 1-D TM, it uses a different diagramming method than the A2 Textbook.

There is a lot more material about Turing Machines in the A2 textbook which we don’t have time to cover. They can be used to accept strings of characters but also to compute some function i.e. they can calculate. Anything that a real computer can compute can also be computed by a Turing Machine. It can simulate any type of subprogram found on programming languages.

UNIVERSAL TURING MACHINES
In his series of thought experiments soon after his major publication in 1936, he had a startling revelation. He had invented the model of the Universal Turing Machine (UTM)! This is a Turing Machine that can simulate the behaviour of any other Turing Machine. It is a computing machine that could process any arbitrary (but well-formed) sequence of instructions. This model is considered by some to be the origin of the stored program computer -- used by John von Neumann (1946) for his "Electronic Computing Instrument" that now bears von Neumann's name: the von Neumann architecture.

In a UTM that uses a single 1-D tape, the instructions of the TM being simulated are placed on a tape and followed by the data to be processed. One of the consequences of the UTM is that programs and data are really the same thing. A program is just a series of symbols that look like any other input. Anyone who has looked a low-level program code will rapidly come to the same conclusion. However, a UTM acts as an interpreter and begins to compute as it realizes what to do.
SIMPLER TURING MACHINE USING KARA

First Click on the tools symbol to set the background as white.

Next, click on the size symbol to set a height of 1 cell and a width of 19. You will also have to click on the Zoom symbol underneath and select “Fit to”.

You can now Set up the tape. Drag and drop the #, 0 and 1 symbols from the vertical toolbar on the right hand side and finally drag and drop the red lined observation window on the leftmost 1 on the tape.

Now you need to move to the programming phase of the activity. Just click on the Top Left programming button.

Create the states $S_0$ and $S_1$. $S_2$ is the “stop” state. You will also need to create the transitions between $S_0$ and $S_1$ and $S_1$ and $S_2$ (Stop)
Now you have reached stage 3. This is where you program in the transition information in the lower half of the programming window.

Now programming the final two transitions

We can now run the simulation. Minimise the programming window first. Click on the Run symbol in the Execution of program section on the bottom right. First you may want to move the Speed of execution slider to the “slow” position.

Has become:

```
# 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 #
```

i.e. the UNARY representation of 3 \((1 1 1 1)\) has become \(1 1 1 1 1\). In other words, it has added a ‘1’ to the unary representation for \(3_{10}\).
BIBLIOGRAPHY


http://en.wikipedia.org/wiki/Finite_state_machine

http://en.wikipedia.org/wiki/Turing_machine


http://scoter3.union.edu/~hannayd/csc350/simulators/FiniteStateMachine/fsm.htm

http://www.cs.binghamton.edu/~software/fsm/fsmdoc.html

http://www.ironphoenix.org/tril/tm


http://www.swisseduc.ch/informatik/karatojava/download.html

http://www.jflap.org/