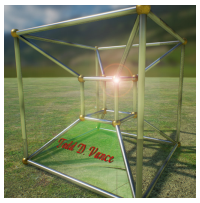


Game State Machine

HowTo

Build a framework for games both simple and complex

Table of Contents

1 Introduction

A Finite State Machine, more technically called a Deterministic Finite Automaton is a theoretical concept that models a simple robot that follows simple rules: it is in a state; it receives an input; Based on the current state and the input, it does something, and it goes into a new state (or the same state again). It has been "invented" many times in many different forms over the centuries (Euclid's straightedge and compass constructions are probably the earliest), but the most common model used in computer programming, similar to the one used here (we actually use a variant, called the Mealy model), was first described in detail in the

mid 20th century scholarly paper: *Moore, Edward F (1956). "Gedanken-experiments on Sequential Machines". Automata Studies, Annals of Math. Studies. Princeton, N.J.: Princeton University Press (34): 129–153.* (A "Gedanken-experiment" is a thought experiment, done in the head instead of in a laboratory).

In real life, finite state machines occur in lots of everyday products, nowadays controlled by computers but in earlier times, controlled by a mechanical process. An example would be a typical vending machine.

1.1 Mouse Trap Model

A very simple, mechanical example of a state machine is the standard mousetrap (not the modern techy humane ones—I don't know how they work): it has two states, set, and unset. The inputs are mouse or no mouse, or "set the trap". The behaviors are spring, or do nothing. If it is in the "unset" state, it does nothing, whether or not there is a mouse, and stays in the unset state. If it is in the "set" state, it springs if a mouse is present and goes to the unset state. If there is no mouse, it does nothing and remains in the set state. If it is unset and a human does "set the trap", it goes to the set state. If it is set and the human does "set the trap", this is really an error condition, and what actually happens might be that the trap springs and it goes to the unset state and the human learns his lesson.

1.2 Vending Machine Model

A typical vending machine spends most of its time in the "Ready" state, with lights on and (at least for soft drinks) refrigeration running.

The possible inputs are: press a product button (of which there are several), activate the "coin return" (though nowadays it returns bills too), or insert coins (or bills).

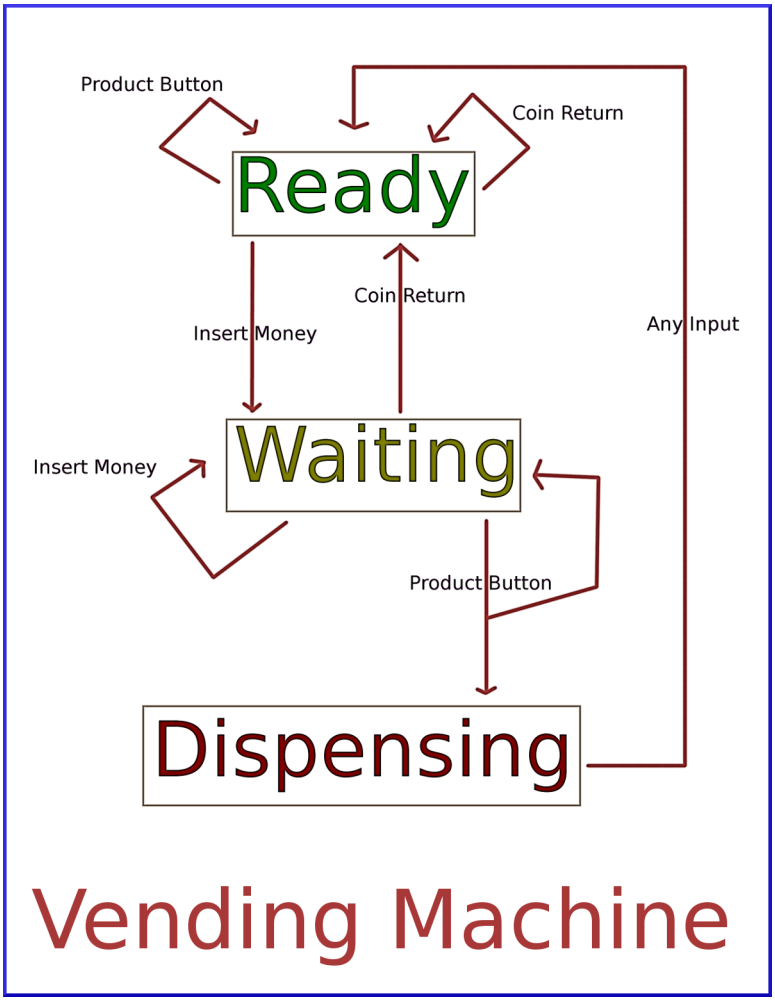
When in the ready state, if you press a product button, a typical machine would show the price of the product on a small display. Note that what makes a state machine a "state machine" is that the behavior depends not just on the input, but the state it is in. If the machine were in another state, it might dispense a product instead of just showing the price when the product button is pressed.

When in the ready state, if you press the coin return, nothing happens. (If something happens, such as money being returned, it means a previous customer walked away when the machine was in a different state and lost their money).

When in the ready state, if you start inserting money, now something useful happens. It changes to the "Waiting for Selection" state. (technically, there is additional state information here: the amount of money inserted so far).

When in the waiting for selection state, if you insert money, the amount of money inserted increases but nothing else happens.

When in the waiting for selection state, if you press the coin return, you get your money back (if the machine is working… strange how many are not working…) and the machine returns to the Ready state.



When in the waiting for selection state, if you press a product button, there are two possibilities: you have inserted enough money for the product or you didn't. If you inserted enough money, it goes to the Dispense state. Otherwise, you get some kind of error message and it stays in the Waiting for Selection state.

When in the dispense state: none of the inputs do anything. It just dispenses the product, returns any change if any, and goes back to the Ready state.

Note that this description of the Vending Machine state machine is a bit simplified. To make it a more accurate model, there would be additional states like "enough money", "not enough money", "return change state", and so on. That a state and input can have two possible results (enough money to buy product or not

enough) means more states are needed to be truly accurate, since this is not allowed with the technical definition of a pure deterministic automaton. But this should be enough to illustrate what a state machine is: a simple robot that follows simple rules.

1.3 What is a State Machine

A state machine has a finite set of "states" and a set of possible "inputs" and "behaviors". The state machine is always in one state. If it receives an input, then depending on which input it receives and which state it is in, it will perform one or more behaviors, and then either stay in the same state or transition to another state. Then it waits again for input and continues in the same way.

1.4 Exercise: Draw Your Own State Machine

Consider these exercises to be "additional fun," but also, a way of solidifying knowledge and enhancing future experience with state machines. Even a good but unsuccessful attempt will benefit the reader. Some exercises have Google-able solutions (or partial or even complete solutions may occur later in this document...), but the author suggests making a good try before looking. Remember, there are no grades—the goal of these exercises is to "think," and to gain some knowledge as a result.

The first exercise is: think of some system you are familiar with (like the Vending Machine example) and write up a model of it as a state machine. Possible ideas include: other kinds of vending machines, an Automated Teller Machine (ATM), an espresso machine, or even some games can be modeled as state machines. Note text adventure games are really large state machines.

1.5 Exercise: Make it Precise

Make your knowledge of state machines more precise. For example, if you are a mathematician, come up with a precise mathematical definition of state machine, say, around the level of precision of the definitions in Euclid's Elements. If you are a programmer, write code that implements a state machine. For example, you could try to beat Roller Coaster Simulator or The Sims or Flight Simulator or Sim City and create Vending Machine Simulator! The prototype would, in your favorite programming language, take input as entered text or button presses, and

behaviors (and which state the machine is in) would be shown by printing text. If this is too much, write Mouse Trap Simulator.

1.6 Exercise: Do Some Research

Using Google, Wikipedia, or even textbooks you have access to, read up on state machines. If you can, try to find the paper by Edward F. Moore. If not, google what you can find on his design (excerpts of the paper and writings about the paper are there). Find out things like, what other kinds of state machines there are and how they differ and how many of them are also essentially the same thing, why the Vending Machine "breaks the rules" of a deterministic automaton, and answer to your own satisfaction the question: "Is my computer a (really big) state machine?" Note there are very good reasons to answer either Yes or No to that one! Look up Turing Machines (which came before Moore's paper—Turing's paper I think is available online, full text, and is a good read, including the reasons he designed the Turing Machine the way he did—because of this paper, Alan Turing is considered to be the inventor of computer science) to find some good reasons the answer should be "No". For a bonus, answer to your own satisfaction, "Is the physical universe a (really, really big) state machine?". Note that Isaac Newton thought the answer to that was, "Yes," (but possibly an infinite state machine) but modern physicists tend toward "No". The idea is to find out why (Newton and modern physicists both have good reasons for their choices). And even though Newton was religious, the reasons don't even consider the religious element, which makes the question more complex…. Many of these ideas would make good term papers for certain courses—check with your teacher.

2 Game State Machine

The particular state machine this paper is about is the overall game state machine. This doesn't have much to do with the details of game play, but of the organization of levels, cutscenes, title screens, and so on in a game. Let us begin with an example.

2.1 Classic Arcade State Machine

Consider the arcade game Pac-Man, the kind that was in a wooden box at an arcade that one stood in front of and played after having fed it quarters. We shall discuss the state machine for it, noting that many classic arcade games had state machines that followed a similar pattern. When you walked up to the game, it was most likely in "attract" mode, the mode that tries to get you to spend a quarter. This is a big state that will be divided into smaller states (title, demo, and so on—note Namco/Bally probably have their own names for the states; I'm making up names that are reasonably descriptive). But before the attract mode, there is another mode not seen often unless you pull the plug and plug it back in: booting up.

1. **Boot**: this is the initial state, the start state of the game. It takes a certain amount of time to finish, and when it does, it immediately transitions to the Title state. Input is not noticed (although, for all I know, there might be special codes to do certain tests, etc. that the arcade operator could do at this point).

2. **Title**: the screen that shows the title of the game and some graphics to make someone come closer for a better look. After a certain amount of time has passed, it goes to the Description state. If a quarter is spent and the Play button is pressed, it instead goes to the Start Game state.

3. **Description**: here is a screen describing game play and points for various activities. After a certain amount of time has passed, it goes to the High Scores state (I might have Description and High Scores in the wrong order; it has been decades since I've played Pac-Man at an arcade). If a quarter is spent and the Play button is pressed, it instead goes to the Start Game state.
4. **High Scores**: the top ten scores, with initials, are displayed here. After a certain amount of time has passed, it goes to the Demo state. If a quarter is spent and the Play button is pressed, it instead goes to the Start Game state.
5. **Demo**: A sample game is shown, without sounds and possibility for player input. It doesn't last long before Pac-Man is eaten and the game returns to the Title state. If a quarter is spent and the Play button is pressed, it instead goes to the Start Game state.
6. **Start Game**: the Level 1 board is displayed, the intro music is played, Pac Man and the ghosts appear on the board, and then the game transitions to the Play Game state. Input is ignored during the Start Game state.
7. **Play Game**: now the input is connected to the game and a user can play. Depending on the events of the game, the next state is Level Up or Game Over (recall, this is just overall game state, we are not modeling the fine points of game play at his level).
8. **Level Up**: when the board is cleared, this state is entered. The board flashes, and there may be a cutscene. When this is done, the next level is loaded and the game goes back to Play Game.
9. **Game Over**: when the player runs out of lives, the game is over. A big Game Over message appears over top the game board while the ghosts dance in glee at having earned another quarter. After a time, one of two things happen. Either it returns directly to the Title state, or if the score achieved is in the top ten, it goes to the New High Score state.
10. **New High Score**: the player is given the opportunity to enter his initials to appear on the high score board. After the initials are entered (or if it times out with it stuck on AAA, since some players just walk away), the game returns to the High Score state--or rather, a duplicate of the High

Score state (I'll call it High Scores Reprise) that transitions to the Title rather than to the Demo.

11. **High Scores Reprise**: the top ten scores, with initials, are displayed here, with the just-made high score highlighted. After a certain amount of time has passed, it goes to the Title state. If a quarter is spent and the Play button is pressed, it instead goes to the Start Game state.

Note that one can "factor" this state machine. The outer state machine has for major states: Boot, Attract, Play, and Post Game. Within each major state is a refinement into more states.

Note also that because the Pac-Man state machine is such that each major state can be entered from another major state at only one inner state (namely, Boot, Title, Start Game, and New High Score, respectively), one could actually program this with 5 state machines: the outermost major state machine with 4 major states, and within each major state, start up a minor state machine (the Boot one is trivial, just one state) for the inner minor states. Even if it is not set up so that each major state can be entered at only one inner state, it could be done with some additional code. In real life programming, state machines are often not "pure" but require additional code for special cases.

2.2 Exercise: Draw Your Own Game State Machine

Pick a computer game, whether arcade, mobile, desktop, console, or even hand-held (like GameBoy or the old single-game machines like Football), and draw a game state diagram. Remember, this is the top level state machine for the game and doesn't include finer game play details.

2.3 Exercise: Code the Framework of Pac-Man

Code up, in your favorite language, the Pac-Man state machine. Either use the full 11-state machine or factor it into 5 state machines, the outermost one being the controlling one having 4 states. Keep it simple, accepting input like "play game" and printing out how it responds and what state it is now in. This code could then be used as a framework for the full game.

2.4 Exercise: Code the Framework of Any Game

For the state machine you made in Exercise 2.2, code up a simulator in your favorite programming language. Keep it simple, accepting input like "play game" and printing out how it responds and what state it is now in. This code could then be used as a framework for the full game.

3 Coding a State Machine in Unity3d C#

Unity3d is a popular, general purpose game engine that is free to use for hobbyists, though some premium pay features are also available. The online Udemy course is not free, but is written for Unity Version 4; with a Version 5 course in the works (to be sold at a discount to those who purchased Version 4). In addition, the Unity3d site has some tutorials and manuals and a forum for learning Unity.

A basic state machine can be coded as a C# class called StateMachine whose main responsibility is to keep track of which state the machine is in. A state is then an instance of another C# class, called State, which maintains its name and a list of states it can transition to. The simplest possible state machine code would thus look something like this, in a single file called SimpleStateMachine.cs. This code is specific to Unity3d, but it could easily be adapted to other systems and languages. In Unity, one can fill in state machine details using the Inspector. In other languages, it might be necessary to write code to fill in the details of a specific state machine. This version uses strings to label states, as well as a string array to list states that can be transitioned to, making it easy to fill in using plain text. In addition to the previous link, the state machine code occurs in the first Appendix[Page 14].

To use the simple state machine in Unity:

1. Create an empty GameObject in the Hierarchy, and rename it to StateMachine
2. Attach the SimpleStateMachine.cs script to the game object as a component
3. Select the StateMachine game object just created in the Hierarchy and look at it in the Inspector
4. Add new states at will, and transitions out of each state to other states.

5. Run the game. Note how the state can be changed by changing the value of the currentState field in the SimpleStateMachine script component.
6. Write an additional script that calls the Transition method of the SimpleStateMachine.cs script with an integer argument to select which state to transition to.

There are several problems with this simple state machine. First of all, all it does is show what state it is in. There is no additional behavior (though that can easily be added) and no means for transitioning in response to events. The second is a small issue since one only need put the StateMachine method Transition into an event (such as a button click) to cause the transition to happen upon that event.

A more serious problem is it is currently impossible to load a new scene without losing the state machine instance. This means that to be an overall game state machine, the state machine should be a Singleton class. That means, any game has only one state machine (other state machine classes could be used for smaller state machines that don't persist between scenes) and it remains active across scene changes.

3.1 Exercise: Fix the State Machine Class

At a minimum, think of how you would do this, Googling for "Unity DontDestroyOnLoad" to find out how to make a Singleton. If able, modify the code to make it robust and useful and test it well. As a bonus, redo some of the previous exercises with this new code. As an extended bonus, actually create a complete, playable game using this new code. The reader may wish to compare what they wrote with the author's State Machine code.

3.2 "Boss Level" Exercise: Design and Build a Better State Machine Asset for Unity 3D.

Start with a list of features it should have (for example, automatically transition after a certain amount of time has elapsed, or—this is very advanced—create a graphical Editor interface to build the state machine). Prioritize them into important, very useful, nice to have, and optional.

Using the priorities as a guide, built a state machine asset having as many of those features as you are able to do, understanding that what looks easy at the design stage might turn out to be hard when you actually do it.

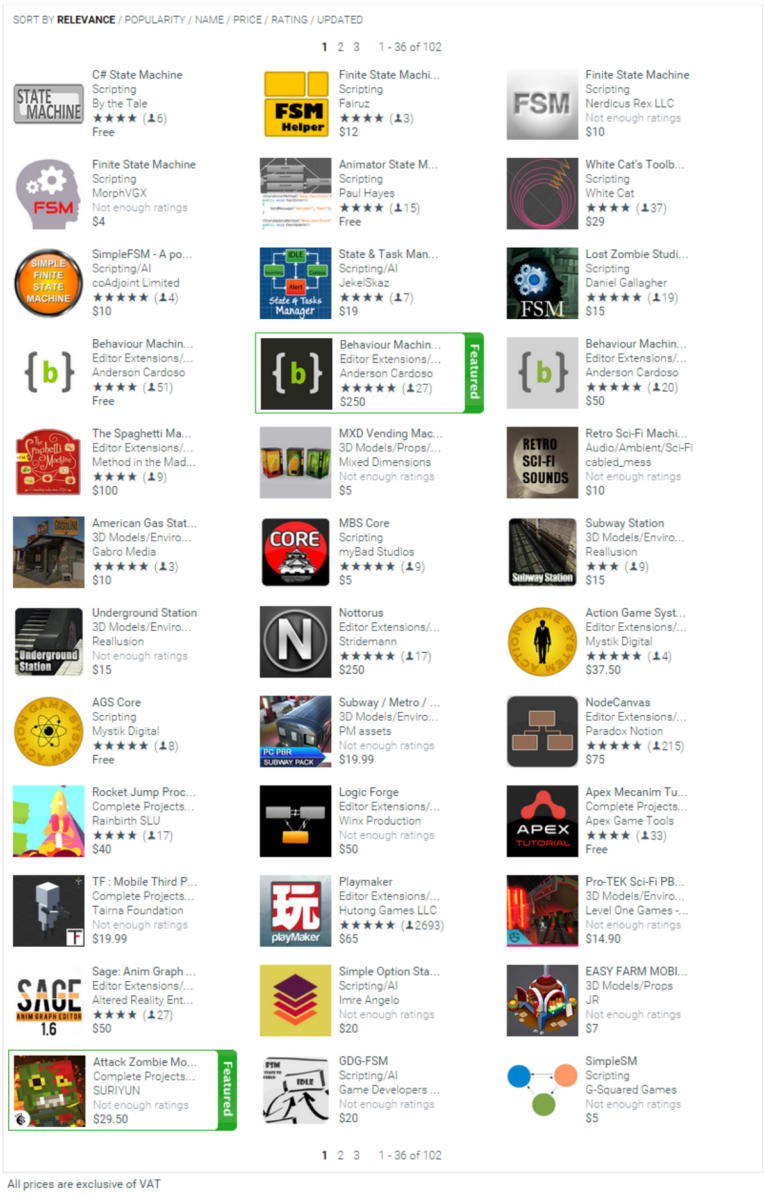
Test the result. Fix any bugs you find. Are there non-bugs that are still annoyances? Are there things that could be made to work better? Are there new features that appear useful in retrospect not in the original list? Amend the design and create a better version.

Test again. Be paranoid and test for anything that could possibly go wrong. Give it the acid test: build some games using the state machine and notice what goes wrong or what slows you down and think of how it can be improved.

Test again.

When all is working magnificently, go to the Unity3d website and research how to submit an asset to the Asset Store—the procedure is not trivial! But it can be done. The author has done it. Good luck and hope it gets accepted.

Also look at comparable assets in the store. If yours is significantly better than the others available, consider asking for money! That requires even more work and set up. If it isn't, can you amend the design and improve it? Some of the ones there have some advanced, hard-to-code features.



4 Using the GameStateMachine asset

The author's State Machine code can be used as an overall state machine. The StateMachine.cs class is found in the Assets/StateMachine folder. In addition are Unity3d scenes for testing: ClassicArcadeStart and MobileStyleStart. The first is similar (but not exactly like) the Pac-Man example

above. The second is a style similar to what might be found on a mobile game like Angry Birds. To use these, all scenes in subfolders must be in the Build Settings. The two scenes mentioned give two examples of how to set up the state machine for a game. The key object is the StateMachine prefab (in the Assets/StateMachine folder). Put this in the start scene of a game. It is recommended the start scene be a special scene that only loads Singletons (like the StateMachine) and is never returned to again.

Use the inspector to add new states and fill in details, such as transitions, auto transitions (after a certain number of seconds), scenes to load, and so on. Three events are defined for each state: entering, exiting, and loading the scene. The DefaultActions.cs file is a simple class that simply logs the events. A game programmer will likely provide their own classes for actions to be invoked upon events happening.

4.1 Exercise: Set up the GameStateMachine project in Unity3D

Download the entire folder structure from GitHub from the link given. Also ensure you have Unity version 5.5 or higher (though it should still be Unity 5). Then open the base folder (which contains the Assets directory and other files) in Unity. It should import everything. Make sure all scenes are in the Build Settings. Test-run the ClassicArcadeStart and the MobileStyleStart scenes and make sure all the possible button combinations work. Read through the StateMachine in the Inspector and try to understand how it works.

4.2 Exercise: Build a Simple GameStateMachine in Unity3D

Now, start with a new scene. Put the StateMachine prefab in it and build the Vending Machine simulator that was done in a previous exercise. Use UI buttons for simulating the various inputs.

4.3 Exercise: Build the Pac-Man GameStateMachine in Unity3D

Again, start with a new scene and build the Pac-Man system described above. Compare this with the ClassicArcadeStart.

4.4 Exercise: Build a Game

Create a complete, playable game of your choice using the GameStateMachine assets.

4.5 Exercise: Improve the GameStateMachine assets

The author left plenty of room for improvement. What can you improve? Redo the exercise[11] from the last section using this asset package.

5 Appendix: Simple State Machine code

SimpleStateMachine.cs file is shown here:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Maintain the current state that the state machine is in
/// </summary>
public class SimpleStateMachine : MonoBehaviour {

    #region Public interface

    /// <summary>
    /// A state in the state machine
    /// </summary>
    public struct State {
        [Tooltip("Name of the State")]
        public string name;

        [Tooltip("List of states this state can transition to")]
   public string[] canTransitionTo;

        //constructor for convenience: needed only to create a default "Start"
state.
        public State(string name) {
            this.name = name;
        }
    }

    [Tooltip("Changing this changes the state the machine is in")]
    public string currentState = startState;

    [Tooltip("All the available states")]
    public State[] states = { new State(startState) };

    public void Transition(int which) {
        currentState = activeState.canTransitionTo[which];
    }
    #endregion


    //the name of the start state, what the machine is in upon awakening
    private static string startState = "Start";

    //the currently active state
```

```
    State activeState;

    // Use this for initialization
    void Start() {
        //set the current state
        activeState = FindState(currentState);
    }

    // Update is called once per frame
    void Update() {
        if (currentState != activeState.name) {
            //if state changes, reset the current state
            activeState = FindState(currentState);
        }
    }

    //find state having given name
    State FindState(string name) {
        foreach (State state in states) {
            if (state.name == name) {
                return state;
            }
        }
        Debug.LogError("Missing state: " + name);
        return null;
    }
}
```

6 Appendix: Mathematical Definition of a Deterministic Finite Automaton

The key object in a DFA is a set, namely the state set S. Because it is a finite automaton, S must be finite. Other than that, there is no real restriction for what is in the set S. The elements can be thought of as labels for the states. In the vending machine example, S = {Ready, Waiting, Dispensing}, a set with three elements.

To make the definition precise, we need a second set, I, the set of inputs. For the vending machine example, I = {InsertMoney, PressCoinReturn, and PressProductButton}. So we have two sets so far.

In addition to the set, the mathematical definition must model the transitions, and this is best done with a function. A function in mathematics is a little different from a function in computer programming. Given two sets S and T, we say a function f maps elements of S to elements of T if for any element s we choose from S, we have f(s) = t for a *unique* element of T. For example, let S = {Ready, Waiting, Dispensing} and let T (temporarily) be the set of positive

integers {1, 2, 3, 4, 5, …}. Let f(s) be defined as "the number of letters in s". Then f is a function from S to T: f(Ready) = 5, f(Waiting) = 7, and f(Dispensing) = 10. It satisfies the requirements of a function: it maps each element of S to a unique element of T. There are 3 elements of S, and each one is mapped to something in T. And it is not the case that some element is mapped to two different values in T. Note that it is *not* required that all of T be used in the map. For example, 1 is not the result of mapping any element of S by the function f.

But the transition function is a different type of function from what was just described. It takes two inputs. So we modify the function a bit. If S and I are sets, and T is a set, then a two-argument function f maps a pair (s,i) consisting of an element s of S and an element i of I, to a unique element t of T.

For an example, again take S = {Ready, Waiting, Dispensing} and I = {InsertMoney, PressCoinReturn, and PressProductButton}, and take T to be the set of positive integers again: {1, 2, 3, 4, 5, …}. Let us define a two-argument function f whose first argument is from S and whose second argument is from I, and which sends these pairs of values to elements of T. Suppose f(s,i) is defined to be "the number of letters in s times the number of letters in i". Then, f(Ready,InsertMoney) = 5*1 = 55. f(Dispensing, PressProductButton) = 10 * 18 = 180. In fact, for each pair, the first item from S and the second item from I, f produces a unique element of T. Note that again not all elements of T are used. It's also ok for a function to produce the same element of T from different inputs.

Now we can specify the transition function. The transition function is a function t having two arguments, a state from S and an input from I. The result of the transition function is a state from S: it could be the same state as was input, or a different state.

Thus, for the vending machine example, take S = {Ready, Waiting, Dispensing} and I = {InsertMoney, PressCoinReturn, and PressProductButton}, and the transition function t is as follows. Note there are a few issues: I mentioned before the vending machine example isn't technically correct!

- t(Ready, InsertMoney) = Waiting
- t(Waiting, InsertMoney) = Waiting
- t(Dispensing, InsertMoney) = ??? (here is a place the vending machine model is incorrect; let is make the answer Waiting for this case to fix it)
- t(Ready, PressCoinReturn) = Ready

- t(Waiting, PressCoinReturn) = Ready
- t(Dispensing, PressCoinReturn) = Ready (this is another place the original model was broken.)
- t(Ready, PressProductButton) = Ready
- t(Waiting, PressProductButton) = Dispensing (this is where the original model was really broken. We need states to take into consideration enough or not enough money!)
- t(Dispensing, PressProductButton) = Ready (another place the original model was inaccurate).

Finally, there are outputs, which we called "behaviors" above. The outputs of a DFA are a finite set O. In the case of the vending machine, O = {AdjustInsertedAmount, ReturnMoneyAndResetInsertedAmount DispenseProductAndReturnMoneyAndResetInsertedAmount, ShowPrice, Nothing}

The output map is kind of like the transition map: it is a function of two arguments, the first being a state and the second being an input, and it maps the pair (s,i) with s a state and I an input, to an output o in the set O. For the vending machine example, the output map looks like this (with some fixes already applied):

- o(Ready, InsertMoney) = AdjustInsertedAmount
- o(Waiting, InsertMoney) = AdjustInsertedAmount
- o(Dispensing, InsertMoney) = DispenseProductAndReturnMoneyAndResetInsertedAmount
- o(Ready, PressCoinReturn) = Nothing
- o(Waiting, PressCoinReturn) = ReturnMoneyAndResetInsertedAmount
- o(Dispensing, PressCoinReturn) = DispenseProductAndReturnMoneyAndResetInsertedAmount
- o(Ready, PressProductButton) = ShowPrice
- o(Waiting, PressProductButton) = Nothing
- o(Dispensing, PressProductButton) = DispenseProductAndReturnMoneyAndResetInsertedAmount

To make the physical machine fit the mathematical description, it was necessary to make some of the behaviors "big" and do several things. To design this in the real world, one would actually use more states. Actually, it would not be a pure state machine (under the mathematical description) because the money counter and the possibility of different products having different prices means it is better to make some of the transitions depend on the result of executing code that

checks money received against product prices. So in reality, a vending machine is *almost* a state machine. I believe some of the old vending machines used code written in COBOL. There are a lot of devices out there that still run COBOL! However, languages like Java are replacing it, slowly.

So, putting it all together:

Definition(Discrete Finite Automaton): A DFA, or Finite State Machine (of the Mealy type), consists of three finite sets S, I, and O, whose elements are called, respectively, the states, the inputs, and the outputs (or sometimes behaviors), and two functions: a transition function t of two arguments mapping ordered pairs from S and I to elements of S, and an output (or behavior) function o of two arguments mapping ordered pairs from S and I to elements of O.