1 Overview

There are many ways of modeling the behavior of systems, and the use of *state machines* is one of the oldest and best known. State machines allow us to think about the "state" of a system at a particular point in time and characterize the behavior of the system based on that state. The use of this modeling technique is not limited to the development of software systems. In fact, the idea of state-based behavior can be traced back to the earliest considerations of physical matter. For example, H_2O can exist in 3 different states easily observable in nature: solid (ice), liquid (water), and gaseous (steam, fog, clouds). In each of these states, the behavior of H_2O is different. The means of forcing transitions between the 3 states is also well-defined. Many other natural and artificial systems may also be modeled by defining (1) the possible states the system can occupy, (2) the behavior in each of those states, and (3) how the system transitions between the states, including which states are "connected" and which are not.

We can use a similar technique to model and design software systems by identifying what states the system can be in, what inputs or events trigger state transitions, and how the system will behave in each state. In this model, we view the execution of the software as a sequence of transitions that move the system through its various states. In these notes, we will develop strategies and techniques to create these state machine models for software systems, and describe some of the ways they can be implemented. Section 2 will introduce several notational forms used to specify a state machine model. In section 3, we consider how to identify system states, state transitions, and transition triggers. Implementation methods are then covered in Section 4.

2 State Machines

We begin our discussion of state machines with a simple example. Consider a turnstile entry control at a subway station: a rotating gate with a coin receiver. The gate is initially 'locked" and will not rotate to allow a passenger to pass through. When a coin or token is deposited into the receiver, the gate is "unlocked" but does not turn until the person actually pushes on it and passes through the turnstile, at which time the gate locks until the next coin is deposited. From this description, it should be clear that the turnstile is always in exactly 1 of 2 possible states: Locked or Unlocked. There are also only 2 transition triggers: depositing a Coin and Passing through the gate. We can model the system graphically, using rounded rectangles to represent states and arrows to represent transitions between states, as shown in Figure 1 below.

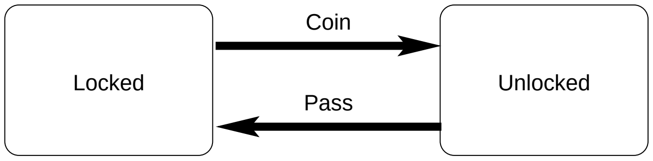

Figure 1: Simple State Machine Model of a Turnstile

While this is a good approximation of how the system functions, we can see that certain information about the turntile's behavior is not clearly defined. What happens when a coin is deposited before someone passes through the gate? Similarly, what happens if a person tries to pass through the gate without depositing a coin? In both cases, our simple model should remain in its current state. We show this behavior with "loop" transitions in Figure 2. We also add an indication of the initial state of the system using a solid circle and an arrow pointing to the start state.

We could continue to refine this model, but instead we will begin developing a conceptual definition of a finite state machine (FSM).

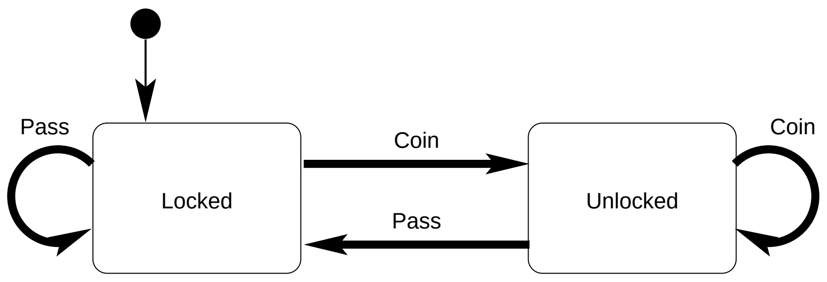

Figure 2: Improved State Machine Model of a Turnstile

2.1 Conceptual Definition of a Finite State Machine

From the discussion thus far, we can identify several key characteristics of a system that can be modeled with a finite state machine:

- The system must be describable by a finite set of states.
- The system must have a finite set of inputs and/or events that can trigger transitions between states.
- The behavior of the system at a given point in time depends upon the current state and the input or event that occur at that time.
- For each state the system may be in, behavior is defined for each possible input or event.
- The system has a particular initial state.

Let us consider another example of a system that can be described by a finite state machine — an old, mechanical soda machine. This machine dispenses only 1 kind of soft drink at the price of $0.30 each. It accepts only nickels, dimes, and quarters and automatically returns change for any amount deposited over $0.30. The machine has 2 buttons — one for dispensing a drink; the other to return the deposited coins. Because this is a purely mechanical system, it does not have a memory system, nor can it directly perform arithmetic operations like addition or subtraction.

We begin our analysis of this system by identifying the possible inputs to the soda machine: nickel (N), dime (D), quarter (Q), dispense button (DB), and coin return (R). To identify possible states for the machine, start with the states we can get to by depositing a single coin. We can label these states with the value, in cents, of the coins deposited so far: 5, 10, and 25. We illustrate our model thus far in Figure 3.

Next, we can think about what happens when we deposit a nickel in addition to the first coin. The resulting model is shown in Figure 4. We continue this expansion until we have identified the minimal set of states that lead to a total of at least $0.30, but are less than $0.55 (since depositing a quarter after 30 cents has been deposited will return the quarter immediately without changing the state of the machine). We show this version of the FSM in Figure 5.

The FSM in Figure 5 covers all possible inputs as long as the sum of the coins previously deposited is less than $0.30. Obviously, the graphical representation of this FSM is quickly becoming hard to handle, so we will introduce an alternative: a *tabular* representation. We list the states down the left side, and the inputs across the top. The table cell at the intersection of a particular row and column indicates the destination state of the FSM when the column's input is received when the machine is in the row's state.

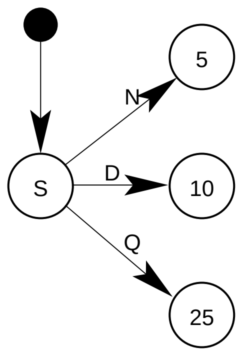

Figure 3: Start of the Soda Machine FSM Model

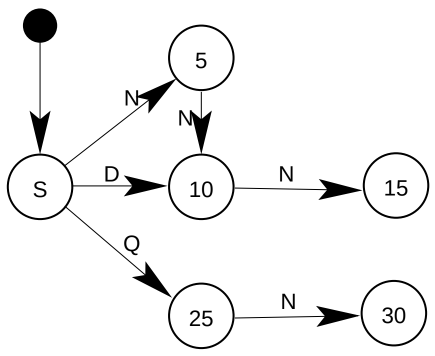

Figure 4: Second Phase of the Soda Machine FSM Model

However, our FSM is still far from complete! Looking back at the description of the soda machine, we see that the machine will "automatically" return the correct change for and coin deposit that exceeds the price of a soda. To accommodate this behavior, we need to add a few things to our models. First, we need to introduce an input that is really "no input" — in other words, we need to be able to create a trigger so the machine will perform some action without actually receiving an input from the user. We will call this a *null* input, and will designate it with **NULL** in the FSM table. To help keep our graphs simple, we will not indicate **NULL**-input transitions when there is no state change or action performed.

Next, we need to define some notation to indicate an action performed by the FSM. In the graphical representation, we will denote such an action that occurs during a transition be following the input symbol with a '/', followed in turn by the action that occurs. This notation is illustrated in Figure 6. In this example, when the FSM is in state **S1** and receives input **X**, the FSM will move to state **S2** and perform $action(y)$. In the tabular representation, we will add the action performed in parentheses after the destination state.

Table 2 is our tabular representation of the soda machine FSM incorporating our new notation for null inputs and transition actions. However, if we look carefully at this table carefully, we can use our new transition action notation to make a significant simplification of the FSM. Consider the states where coin deposits exceed the price of a soda - our initial design uses extra states and null inputs to create the "automatic" change return functionality required by the description. However, if we use our new action notation, we can make these transitions directly, eliminating 4 states as well as the **NULL** input symbol. We describe our new FSM graphically in Figure 7 as well as in tabular form in Table 3. (Note that in Figure 7, 4 transitions are labeled with lowercase letters corresponding to the descriptions in the legend below the diagram.)

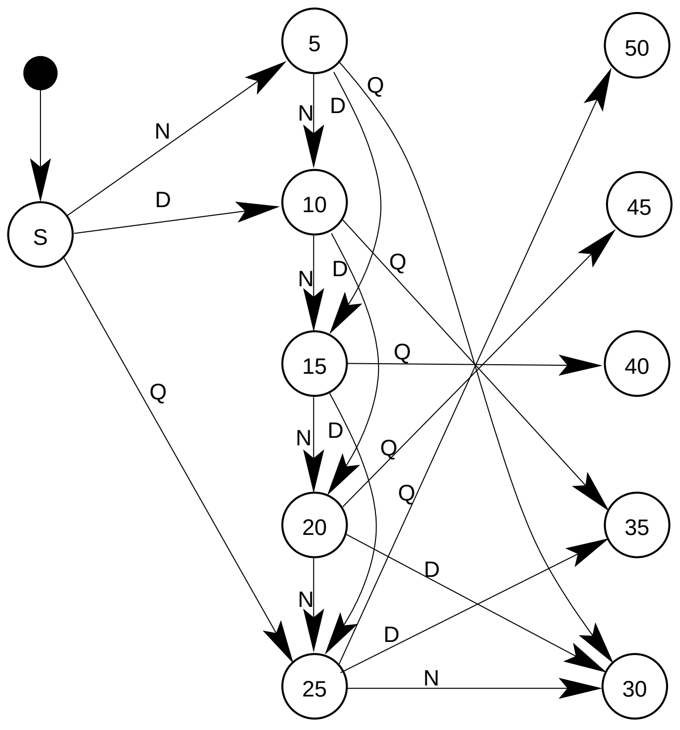

Figure 5: Soda Machine FSM Considering All Valid Coin Inputs

Table 1: Tabular Representation of the FSM in Figure 5

	N	D	Q
S	5	10	25
5	10	15	30
10	15	20	35
15	20	25	40
20	25	30	45
25	30	35	50
30			
35			
40			
45			
50			

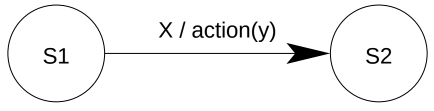

Figure 6: Denoting a Transition Action in a Graphical FSM Representation

Now we've established how the soda machine FSM behaves with respect to coin inputs, but we still haven't modeled how a user actually gets a drink out of the machine! First, however, we will consider the

Table 2: Updated Tabular Representation of the Soda Machine FSM

	N	**D**	**Q**	**NULL**
S	**5**	**10**	**25**	**S**
5	**10**	**15**	**30**	**5**
10	**15**	**20**	**35**	**10**
15	**20**	**25**	**40**	**15**
20	**25**	**30**	**45**	**20**
25	**30**	**35**	**50**	**25**
30	—	—	—	**30**
35	—	—	—	**30** (return(**N**))
40	—	—	—	**30** (return(**D**))
45	—	—	—	**30** (return(**ND**))
50	—	—	—	**30** (return(**DD**))

Table 3: Minimized Tabular Representation of the Soda Machine FSM

	N	**D**	**Q**
S	**5**	**10**	**25**
5	**10**	**15**	**30**
10	**15**	**20**	**30** (return(**N**))
15	**20**	**25**	**30** (return(**D**))
20	**25**	**30**	**30** (return(**ND**))
25	**30**	**30** (return(**N**))	**30** (return(**DD**))
30	**30** (return(**N**))	**30** (return(**D**))	**30** (return(**Q**))

coin-return button, which is supposed to return all coins the user has deposited, or at least a set of coins with the exact same value. Since the machine knows the value of the deposited coins (based on the state it is in), it is straightforward to define the appropriate behavior for the FSM. We simply return a pre-defined combination of coins corresponding to the state the machine is in. This behavior is incorporated into Table 4.

Table 4: Final Tabular Representation of the Soda Machine FSM

	N	**D**	**Q**	**R**	**DB**
S	**5**	**10**	**25**	**S**	**S**
5	**10**	**15**	**30**	**S** (return(**N**))	**5**
10	**15**	**20**	**30** (return(**N**))	**S** (return(**D**))	**10**
15	**20**	**25**	**30** (return(**D**))	**S** (return(**ND**))	**15**
20	**25**	**30**	**30** (return(**ND**))	**S** (return(**DD**))	**20**
25	**30**	**30** (return(**N**))	**30** (return(**DD**))	**S** (return(**Q**))	**25**
30	**30** (return(**N**))	**30** (return(**D**))	**30** (return(**Q**))	**S** (return(**QN**))	**S** (dispense())

Finally, we add the transitions that specify the behavior of the soda machine when the dispense button is pressed. From the description, this button does nothing unless the required amount of money has been deposited into the machine, so our FSM remains in the current state unless it is in the state labeled '30' (the state where 30 cents has been deposited), which triggers the dispensing of a drink, and returns the FSM to

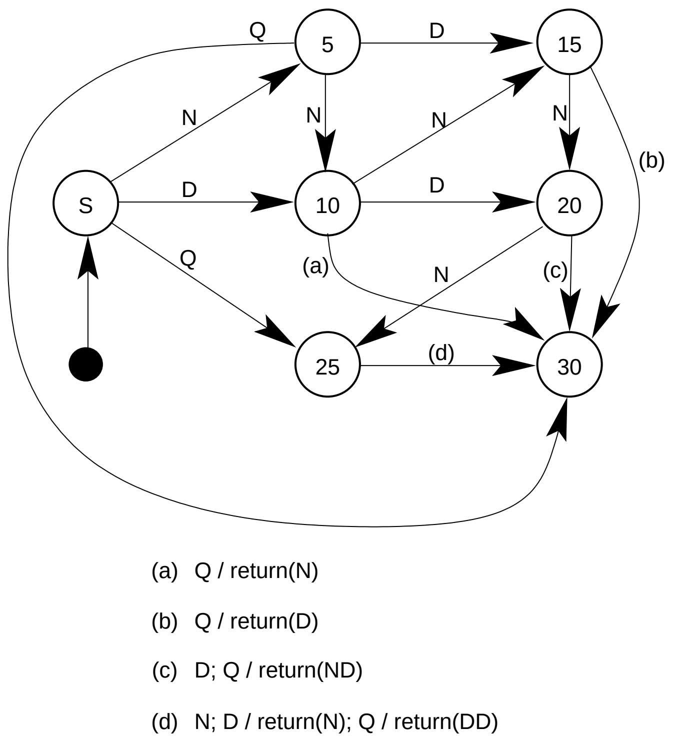

Figure 7: Minimized Graphical Representation of the Soda Machine FSM

the starting state. Table 4 describes the final design of our FSM for the soda machine.

2.2 Mathematical Definition of a Finite State Machine

As computer scientists, we often need to describe systems in a formal, mathematical way. This allows us to reason about systems both in general as well as in specific instances. Formal notation also helps to eliminate ambiguity when we communicate with others about system designs. We provide a brief introduction to the formal notation of FSMs or *finite automata* (FA), beginning with Definition 1.

Definition 1. *A finite automaton M is defined by a 5-tuple $(\Sigma, Q, q_0, F, \delta)$, where*

- *Σ is the set of symbols representing input to M*
- *Q is the set of states of M*
- *$q_0 \in Q$ is the start state of M*
- *$F \subseteq Q$ is the set of final states of M*
- *$\delta : Q \times \Sigma \to Q$ is the transition function*

Definition 2. *A string over an alphabet Σ is a finite length sequence of symbols from Σ.*

Definition 3. *A language over an alphabet Σ is a set of strings over the alphabet Σ.*

Definition 4. *A transition function is a mapping of a set of states and a set of symbols onto the original set of states. The interpretation of $\delta(q, x) = p$ is that for a finite automata M in state q with input x will be in state p after processing an input string x. The notation $\delta(q, x, p)$ is equivalent to $\delta(q, x) = p$ for brevity.*

Definition 5. *A string x is accepted by M if $\delta(q, x) = p$ for some $p \in M$.*

Definition 6. *The language accepted or recognized by M, denoted $L(M)$, is the set of all strings x accepted by M. Using set notation, we can write $L(M) = \{x \mid \delta(q_0, x) \in F\}$*

Definition 7. *The set of languages recognized by the set of finite automata is called the set of regular languages.*

Definition 8. *To denote zero or more sequential repetitions of a symbol in a string, a superscripted '*' is used. For example, 1^* means "zero or more repetitions of the symbol '1"' and includes the subset $\{\emptyset, 1, 11, 111, 1111, 11111\}$. This notation may be applied to a sequence of symbols by parenthesizing the symbols as in $(101)^*$. Similarly, one or more sequential repetitions of a symbol or sequence of symbols, a superscripted '+' is used, as in 1^+ and $(101)^+$. Note that the empty string (a string with length 0) is denoted by $\emptyset$.*

We begin this discussion by considering another simple example:

$$L = \{x \in \{0, 1\}^* \mid x \text{ ends in 1 and does not contain the substring } 00\}$$

We know from the language definition that the inputs are the set of all strings over $\{0, 1\}$, and begin our analysis by identifying the possible states of the FA that recognizes this language. Since a string x in L does not contain the substring **00**, if our FA identifies this substring, we know for certain that the string containing it *is not* in L. We call this state **N**. We should also consider the state where the FA has read in the empty string, $\emptyset$, noting that this is also the initial state of the FA.

The remaining states indicate the last symbol read by the FA, which we label **0** and **1**. If the last symbol read was a **0**, and the next symbol read is also a **0**, the machine must then be in state **N**, since a substring 00 has been read. If the last symbol read was a **1**, we can consider that state to be a final state, since if there are no more symbols in the input, the string will have ended in a **1** and did not contain the substring 00. On the other hand, if there are more symbols to be read, the FA will transition to another state based on that input. We can use the formal notation defined above to describe this FA:

Let $M = (\Sigma, Q, q_0, F, \delta)$ define a FA such that

$$L(M) = \{x \in \{0, 1\}^* \mid x \text{ ends in 1 and does not contain the substring } 00\}$$

where

$$\begin{aligned}
\Sigma &= \{0, 1\} \\
Q &= \{\emptyset, 0, 1, N\} \\
q_0 &= \emptyset \\
F &= \{1\} \\
\delta &= \{\delta(\emptyset, 0, 0), \delta(\emptyset, 1, 1), \delta(0, 0, N), \delta(0, 1, 1), \delta(1, 0, 0), \delta(1, 1, 1), \delta(N, 0, N), \delta(N, 1, N)\}
\end{aligned}$$

We can easily transform this definition to the "bubble diagram" form used previously in these notes. The result is shown in Figure 8 below.

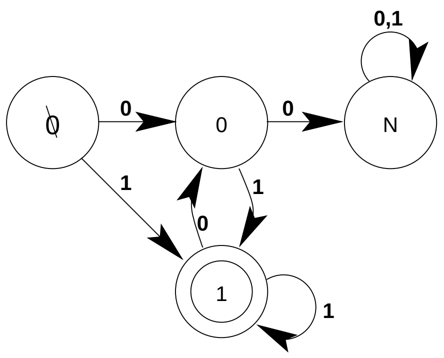

Figure 8: Simple Finite Automata

One question that may occur is: "Why worry about FAs that recognize strings or 0s and 1s? That is too simple!" There are 2 parts to the answer. First, remember that the digital computers we use do *everything* in binary numbers: 0s and 1s. FAs are very commonly used to model the hardware logic circuits that make up a wide variety of components in digital computers, including keyboard interpreters, communication interfaces (modems and network cards), etc. Devices like these must process sequences of binary information and pick out special sequences that carry control commands. The second reason builds on the first — we can build composite FAs to solve more complex problems. For example, consider a FA recognizing the language defined by the ASCII character set. Each character is a string over $\{0,1\}^*$ that is accepted by the FA. But we want to recognize a language over the ASCII characters, so we design a second FA that uses the output of the ASCII automaton as input. We can continue composing more complex structures from simpler components. But for learning and illustrative purposes, it is easier to work with simple alphabets and languages for now.

There are 4 operations that we can use to build more complex FAs: *union*, *intersection*, *complement*, and *concatenation*. The example in Figure 8 also illustrates all of these operations. Consider the language specification again:

$$L = \{x \in \{0,1\}^* \,|\, x \text{ ends in 1 and does not contain the substring 00}\}$$

.

Note the *and* in the definition — this means that we have an intersection of 2 languages. We'll call them L_1 and L_2 and redefine L:

$$\begin{aligned} L_1 &= \{x \in \{0,1\}^* \,|\, x \text{ ends in } 1\} \\ L_2 &= \{x \in \{0,1\}^* \,|\, x \text{ does not contain the substring } 00\} \\ L &= L_1 \cap L_2 \end{aligned}$$

Figure 9 shows the FA that recognizes the language of all strings over $\{0,1\}^*$ that end with a 1, $L_!$ in our decomposition. We can think of it as the *union* of two languages - those with strings that end in 0 and those with strings that end in 1.Since we are interested in accepting strings that end with 1, state **1** is the only accepting state. FAs that are the result of a union of 2 languages generally have a parallel structure when we represent them graphically, with each of the parallel paths corresponding to one of the languages in the union.

Next, we consider L_2, the language of strings that do not contain 00 as a substring. A starting point is to design a FA that accepts strings that *do* contain the substring 00. We do this in Figure 10.

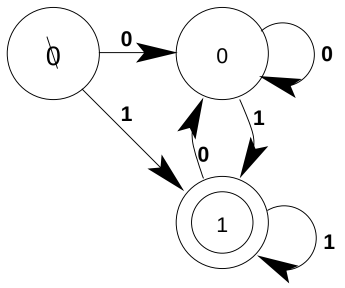

Figure 9: FA recognizing $L_1 = \{x \in \{0,1\}^* \mid x \text{ ends in } 1\}$

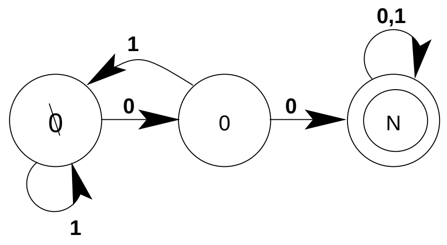

Figure 10: FA recognizing $\overline{L_2} = \{x \in \{0,1\}^* \mid x \text{ contains the substring } 00\}$

If this FA ever reads in 2 consecutive 0s, it goes to the accepting state and stays there. Now, to complement this FA, we make the non-accepting states accepting, and vice-versa. The result is shown in Figure 11.

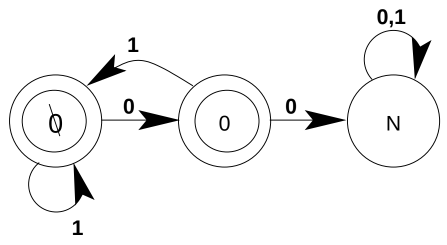

Figure 11: FA recognizing $L_2 = \{x \in \{0,1\}^* \mid x \text{ does not contain the substring } 00\}$

Since the original definition of the language L used the word *and*, it must contain strings that meet the criteria for both L_1 *and* L_2, hence the use of set intersection. We must also modify the accepting states to reflect the fact that *both* conditions must be true. The result, as intended, is the FA shown in Figure 8.

An additional comment on the FA in Figure 8 concerns state **N**. Often in the construction of a state machine model it is necessary to define a state that represents the detection of an error condition from which the system cannot accept any input sequence. State **N** is such a state. This follows logically from the description of the languages: if the substring 00 is ever detected, the input string cannot be accepted, regardless of the remaining input. Therefore, the FA should enter a state that that is not on a path to an accepting state.

2.3 Example FSM Construction

We conclude this section with a final example that will illustrate the construction of a finite state machine model. We first define 2 languages:

$$\begin{aligned} L_1 &= \{x \in \{0,1\}^* \,|\, x \text{ contains the substring } 10\} \\ L_2 &= \{x \in \{0,1\}^* \,|\, x \text{ ends with } 01\} \end{aligned}$$

Figure 12 traces the construction of a finite state machine for the language L_1 which defines a set of strings that all contain the substring "10" somewhere within the string. The task of the FA that recognizes or accepts strings in this languages it to detect this particular sequence of symbols. Since the first symbol of interest is a '1' we can wait in the start state (state **A** in Figure 12(a)) as long as there are 0s in the input. As soon as a 1 is read, however, the machine must transition to a new state, since it must now look for a 0 as the next character. This behavior is different from the initial state of the machine, which is an indication that we must add a state as shown in (b). Note that we have defined the behavior of state **A** for all of the possible inputs, so we can now concentrate on state **B**.

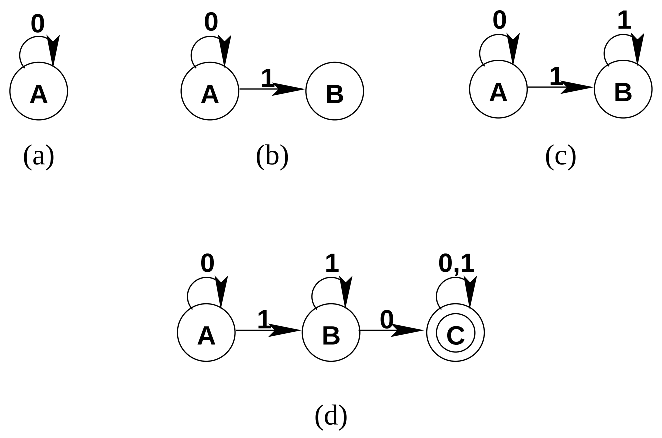

Figure 12: Constructing a Finite State Machine for L_1

In state **B**, the machine "knows" that the last symbol read was a 1. It doesn't know (or care) how many 1s have been read, only that the last symbol was a 1, and that to accept a string, it must now read a 0. The loop transition on state **B**, shown in Figure 12(c) indicates this state's character as "remembering" that 1 was the last symbol read. If the machine reads a 0 in this state, the string contains the substring 10, and thus should be accepted. Figure 12(d) shows this with a transition from state **B** to state **C** on input 0, with **C** denoted as an accepting state by the concentric circle. Finally, we consider the behavior of the machine in state **C**. Since the string has already been accepted as a member of L_1, whatever follows the identified "10" substring does not matter. We indicate this with a loop transition on state **C** as shown in Figure 12(d).

Figure 13 shows the FSM constructed for L_2. Note that since the language definition requires that member strings *end* with a particular sequence of symbols, the FSM does not remain in the accepting state once it is reached, as it did with the FSM for L_1. Instead, it must move to a state where is can again try to find the desired sequence at the end of a string. Strings are recognized if and only if the machine is in the accepting state when the input ends.

Next, we want to consider 3 possible combinations of these languages, defined as L_3, L_4, and L_5 below.

$$L_3 = \{x \in \{0,1\}^* \,|\, x \text{ contains the substring } 10 \text{ or } x \text{ ends with } 01\}$$

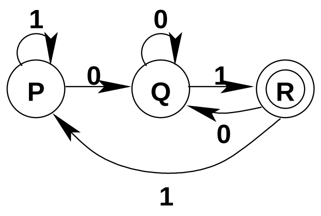

Figure 13: Constructing a Finite State Machine for L_2

$$\begin{aligned} L_4 &= \{x \in \{0,1\}^* \,|\, x \text{ contains the substring 10 and } x \text{ ends with 01}\} \\ L_5 &= \{x \in \{0,1\}^* \,|\, x \text{ does not contain the substring 10 and } x \text{ ends with 01}\} \end{aligned}$$

In order to recognize a language that is some combination of 2 other languages, we must keep track of where we are in *both* FSMs simultaneously. We could imagine using 2 fingers, keeping one on the current state in Figure 12(d), and the other on the current state in Figure 13. When the machines read an input symbol (they both get the same symbol at the same time), each transitions based upon their own definitions, and we keep track of both machines' states with out fingers. Of course, this can get to be very difficult in practice, and does not lend itself well to implementation.

One way to do this is to create a new set of states by taking the *cross product* of the states of M_1 and M_2, as shown in Figure 14. Note that each state of $M-1$ has been paired up with each state of M_2. Then, if M_1 is in state **B**, for example, and M_2 is in state **Q**, our new machine will be in state **BQ**. This allows us to "keep our fingers" on the current state of each machine in the construction of out new machine.

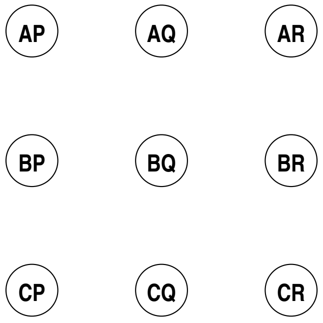

Figure 14: Constructing a FSM by Combining M_1 & M_2 - The Set of States

We must first identify the initial state of our new machine, which will be the state corresponding to the initial states of M_1 and M_2, which is labeled **AP** in the new FSM. From **A** in M_1 on a 0 input, we remain in **A**, while from **P** in M_2 with the same input, we transition to **Q**. Since we now have our fingers on states **A** and **Q**, we are in state **AQ** of the new machine. We repeat this process for an input of 1 in the initial state and get the intermediate FSM shown in Figure 15.

Next, we will consider the transitions from **AQ**, the state that corresponds to having our fingers on **A** in M_1 and **Q** in M_2. In both machines, an input of 0 causes the machine to remain in the same state, so we add a loop transition on **AQ**. With an input of 1, M_1 moves to **B** and M_2 moves to **R**, so we add a transition to state **BR** in the new FSM, as shown in Figure 16.

The next state we consider is **BP**, corresponding to states **B** and **P** in M_1 and M_2. In both machines, an input of 1 does not cause a state change, so we add a loop transition for a 1 input. A 0 input in state **B**

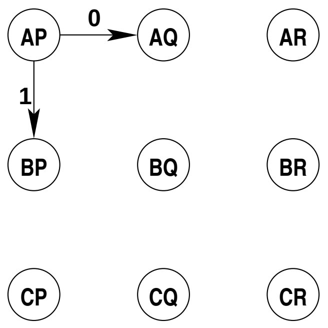

Figure 15: Constructing a FSM by Combining M_1 & M_2 - Transitions from the Initial State

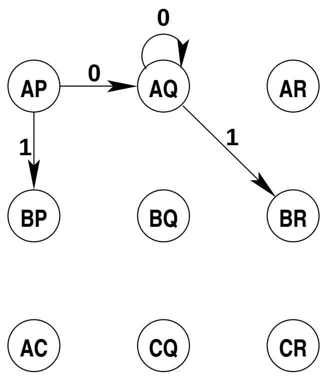

Figure 16: Constructing a FSM by Combining M_1 & M_2 - Transitions From State **AQ**

causes a transition to **C**, while the same input in **P** moves M_2 to state **Q**, giving us a state label of **CQ** in our new machine. Figure 17 shows the result.

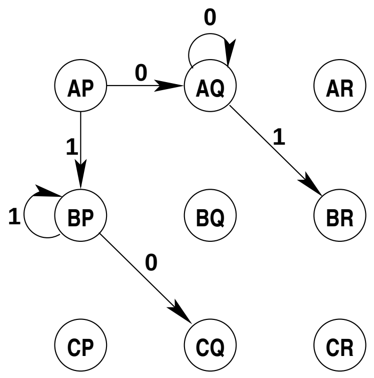

Figure 17: Constructing a FSM by Combining M_1 & M_2 - Transitions from State **BP**

We continue this process for all states we have already reached, considering all of the possible inputs for each state along with the behavior of both of the original machines to determine the resulting states in the new FSM. Figure 18 illustrates the diagram that results from following previously drawn transitions. The method we've used here is to consider the transitions out of states in the order that those states are reached as other states are considered, but we could have just as easily taken them in any other order. The key ingredient is to account for the machine's behavior for all of the possible inputs in each state.

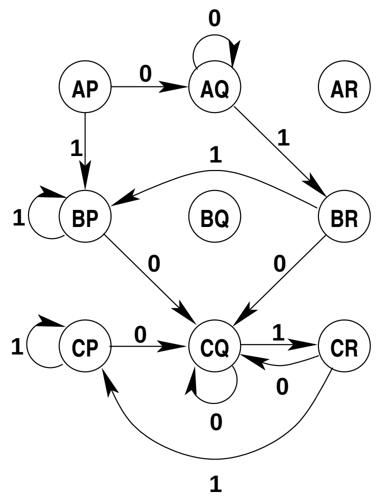

Figure 18: Constructing a FSM by Combining M_1 & M_2 - Step 7

When we've drawn the diagram shown in Figure 18, we run out of "connected" states, but there are 2 states remaining, **AR** and **BQ**. We call these states *unreachable* because there is no path from the start state to either of these 2 states. For completeness in illustrating the FSM construction, we include the transitions *out* of these states in our construction, as shown in Figure 19.

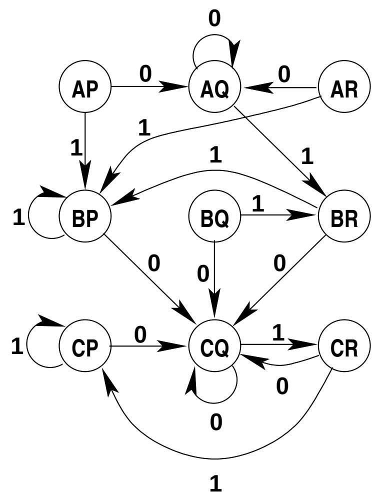

Figure 19: Constructing a FSM by Combining M_1 & M_2 - All Transitions Complete

Maintaining the unreachable states in our model can become cumbersome as well as detracting from the clarity of the model. Since there is no way to reach either state **AR** or **BQ**, we can safely remove them from the model without altering the actual behavior of the FSM. This diagram is shown in Figure 20. We also present the transition function in tabular form in Table 5.

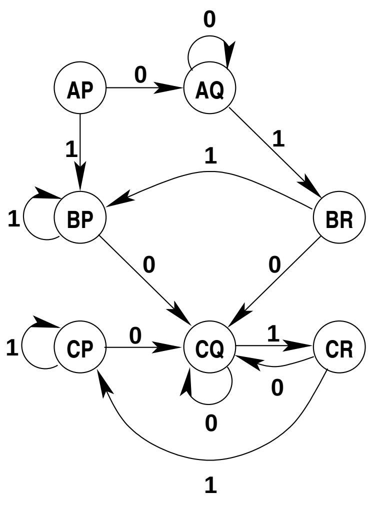

Figure 20: Constructing a FSM by Combining M_1 & M_2 - Unreachable States Removed

Table 5: Transition Function (δ) for the FSM of Figure 20

State	$\delta(p,0)$	$\delta(p,1)$
AP	**AQ**	**BP**
AQ	**AQ**	**BR**
BP	**CQ**	**BP**
BR	**CQ**	**BP**
CP	**CQ**	**CP**
CQ	**CQ**	**CR**
CR	**CQ**	**CP**

Now we can begin to consider the FSMs that will recognize the languages we defined earlier:

$$\begin{aligned}
L_3 &= \{x \in \{0,1\}^* \mid x \text{ contains the substring 10 or } x \text{ ends with } 01\} \\
L_4 &= \{x \in \{0,1\}^* \mid x \text{ contains the substring 10 and } x \text{ ends with } 01\} \\
L_5 &= \{x \in \{0,1\}^* \mid x \text{ does not contain the substring 10 and } x \text{ ends with } 01\}
\end{aligned}$$

We start with L_3, noting that strings in this language are those that are in either of L_1 of L_2. This means that L_3 is the *union* of these 2 languages, and so the set of final states in our FSM is the union of the sets of final states in the original FSMs. In the new machine, this set will include all of the states whose labels contain the names of final states in the original machines. We define this machine formally:

Let $M_3 = (\Sigma, Q, q_0, F, \delta)$ be a finite state machine recognizing L_3, where

$$\begin{aligned}
\Sigma &= \{0,1\} \\
Q &= \{\mathsf{AP}, \mathsf{AQ}, \mathsf{BP}, \mathsf{BR}, \mathsf{CP}, \mathsf{CQ}, \mathsf{CR}\} \\
q_0 &= \mathsf{AP} \\
F &= F_1 \cup F_2 = \{\mathsf{BR}, \mathsf{CP}, \mathsf{CQ}, \mathsf{CR}\} \\
\delta &= \text{Given in Table 5}
\end{aligned}$$

Note that this FSM can be simplified significantly be exploiting the structure of the original machines, although this is not always the case. We leave this optimization as an exercise for the student.

Next, we consider the language L_4. This language specifies that all member strings contain the substring 10 and end with 01. The word "and" in the definition implies that strings in this languages must be a member of *both* of the original languages. We could also express this as $L_1 \cap L_2$, the intersection of the 2 languages. We define the machine formally as:

Let $M_4 = (\Sigma, Q, q_0, F, \delta)$ be a finite state machine recognizing L_4, where

$$\begin{aligned}
\Sigma &= \{0,1\} \\
Q &= \{\mathsf{AP}, \mathsf{AQ}, \mathsf{BP}, \mathsf{BR}, \mathsf{CP}, \mathsf{CQ}, \mathsf{CR}\} \\
q_0 &= \mathsf{AP} \\
F &= F_1 \cap F_2 = \{\mathsf{CR}\} \\
\delta &= \text{Given in Table 5}
\end{aligned}$$

Finally, we consider L_5, the definition of which also contains the word "and" as well as the word "not" that indicates the complement of L_1. This language is the set of strings that are *not* in L_1 and are also in L_2. Note the description of the set of final states in the machine definition below:

Let $M_5 = (\Sigma, Q, q_0, F, \delta)$ be a finite state machine recognizing L_5, where

$$\begin{aligned}
\Sigma &= \{0,1\} \\
Q &= \{\mathsf{AP}, \mathsf{AQ}, \mathsf{BP}, \mathsf{BR}, \mathsf{CP}, \mathsf{CQ}, \mathsf{CR}\} \\
q_0 &= \mathsf{AP} \\
F &= (Q_1 - F_1) \cap F_2 \\
&= (\{\mathsf{AP}, \mathsf{AQ}, \mathsf{BP}, \mathsf{BR}, \mathsf{CP}, \mathsf{CQ}, \mathsf{CR}\} - \{\mathsf{CP}, \mathsf{CQ}, \mathsf{CR}\}) \cap \{\mathsf{BR}, \mathsf{CR}\} \\
&= \{\mathsf{AP}, \mathsf{AQ}, \mathsf{BP}, \mathsf{BR}\} \cap \{\mathsf{BR}, \mathsf{CR}\} \\
&= \{\mathsf{BR}\} \\
\delta &= \text{Given in Table 5}
\end{aligned}$$

3 State Machines as an Object Oriented Modeling Tool

State machines have long been a popular and effective way of modeling the dynamic behavior of software systems at many different levels. The previous section explicitly deals with a very low-level representation of a system, with the understanding that the binary representation can be scaled up to more complex languages and representations. This section will introduce the UML notation for *state transition diagrams* and describe how design and develop them.

3.1 UML Notation

Figure 21 illustrates the 3 basic elements in the UML State Transition Diagram notation. States are represented by a rectangle with rounded corners, and must contain a state name unique to the scope of the diagram. If there are activities that are dependent upon or essential to an object in this state, or that occur when the object enters or exits the state, they are included below a line that separates them from the state name. Activities represent computations that may occur while the system is in a particular state, but that do not directly cause a transition out of that state. Figure 21(a) shows a typical *State* icon. Note that every state must be named, but may not include activities.

Part (b) of Figure 21 shows a UML *Transition* icon, which is used to indicate the movement of the modeled system from one state to another. The transition label has 3 parts: *Event*, *Guard* (enclosed in square brackets), and *Action*. An **event** is an occurrence that causes a state change. When a transition does not have an event label, it is assumed that the state change occurs as soon as any activity associated with that state is completed. A **guard** is a logical condition that can control whether or not a transition actually occurs. A guarded transition allows a state change only when the condition is "true"; otherwise, the system remains in the current state. Finally, an **action** is some computation that occurs in the process of moving from one state to another.

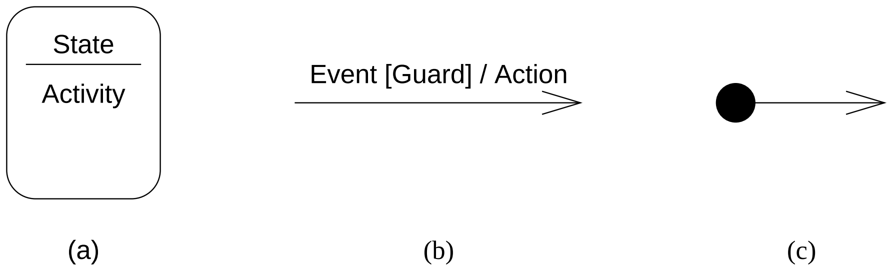

Figure 21: UML Notation for State Transition Diagrams

A solid circle with an arrow is used to indicate the initial or starting state of a model. Figure 21(c) shows a typical *start state icon*. UML also provides notation for entities called *superstates*, which encapsulate a group of related states with common entry and/or exit transitions. We will see an example of a superstate later in this section.

Actions are assumed to (effectively) take zero time, while activities may run for indeterminate time. Action labels generally take one of 3 basic forms:

- An abstract method call, e.g., `heater.start()`
- The name of an event that the transition triggers, e.g., `HardwareFailure`
- Start or stop some activity, e.g., **start Heating**

Activities may be interrupted by an event which triggers a state change. A frequent use of the activity label is to denote an activity that starts (or stops) when a state is entered (or exited).

There is a specific ordering for the evaluation of conditional state transitions. Given a state S with transition T upon event E with condition C and action A, the following order is defined:

- Event E occurs
- Condition C is evaluated
- If C evaluates `true`, then T is triggered and action A is invoked
- If C evaluates `false`, the state transition may not be triggered until the event occurs again, and the condition is re-evaluated

This ordering ensures that an action that occurs in the process of a state transition does not affect the transition triggering. For example, suppose that when an event occurs, the condition evaluates to `true`, and the transition is initiated. During the transition, the *exit* action of the current state is executed, changing the environment such that the condition is no longer true. We still want the transition to proceed, since the condition was true when the event occurred. There is no problem with the truth of the condition changing after the transition process begins, but such a change should not affect the transition in progress.

3.2 Identifying States, Transitions, & Events

As we have discussed in class, objects have *state*, *behavior*, and *identity*, but not all classes have behavior that is dependent upon it's state. The objects we choose to incorporate into a state-transition model depend upon what kind of system behavior we are modeling. An object whose state is "interesting" in one context may not be so in another context.

Often it is easier to first identify the events that may occur in the context we are interested in. The example FSMs in the previous section have fairly simple events: read a 0, read a 1, and reach the end of the input. The object whose state you are concerned with is the "language recognizer" or the FSM itself, and the state of that object is dependent upon the events that occur. Other situations may not be quite so clear.

As an example, let's return to the subway turnstile model introduced at the beginning of these notes, with one additional behavior: if a rider attempts to pass through the turnstile while it is locked, an alarm is triggered, and remains on until it is reset by the operator in the station booth. While the alarm is activated, the turnstile remains locked, and ignores (by rejecting) any coins that are deposited.

From this (and the earlier) description, we can identify 3 distinct events in this system: a coin deposit, a rider passing through the turnstile, and an alarm reset. We can also see 3 possible states the system can be in: locked, unlocked, and alarm active. The system description also identifies the state transitions the events trigger. We can put this information together to generate the updated model of the turnstile system shown in Figure 22.

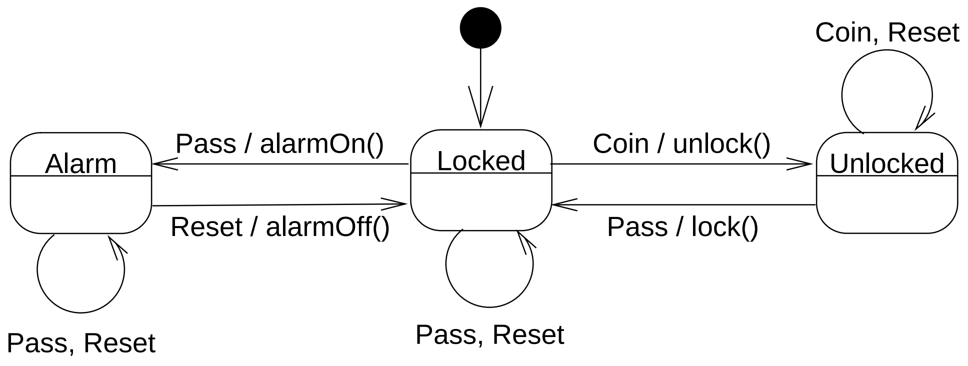

Figure 22: Revised Turnstile State Transition Diagram

Closer examination of this model reveals that there are still 2 "primary" states of the turnstile — *locked* and *unlocked*. What has been added is additional functionality in the locked state. This is a situation where the UML *superstate* notation can add clarity to the model. We can combine the **Alarm** and **Locked** states into a superstate that is entered and exited through a single location, while maintaining the 2 distinct behaviors that are evident when the turnstile is locked. This revised State Transition model is shown in

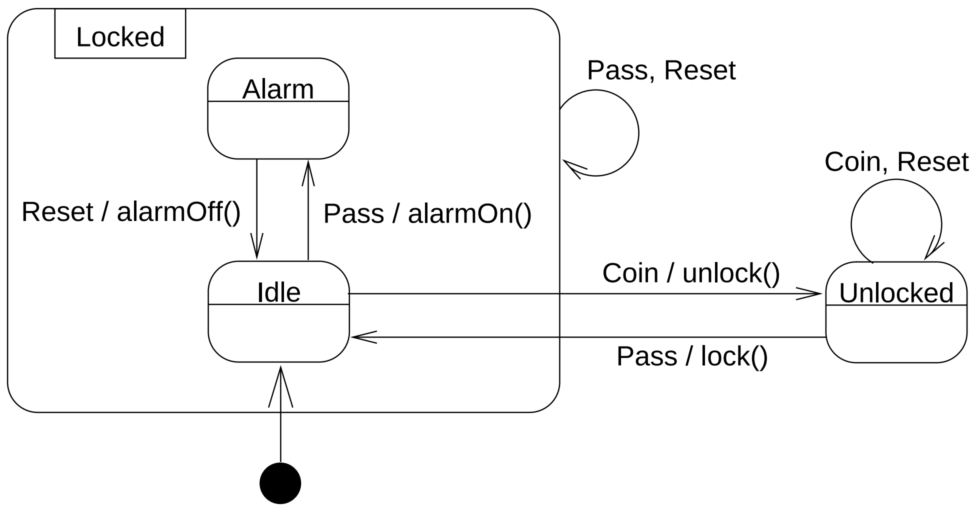

Figure 23: Revised Turnstile State Transition Diagram with Superstate

Figure 23. Superstates are indicated with a rounded-corner box enclosing 2 or more other states, and with the superstate name boxed off along the top edge of the enclosing box.

UML does not place any restrictions on how many levels of superstate nesting may occur in a single model. However, if you reach 2 or 3 levels of nesting, you may want to separate the nestings into distinct diagrams, indicating only the highest-level superstates in each. Then, each superstate at one level becomes a separate diagram at the next lower level.

4 Implementations

This section will describe 2 different approaches to implementing state machine models in Java. While these examples are typical of their situations, they are certainly not exhaustive with regard to how a state machine model can be coded. In fact, it is often the case that the model is not implemented directly; rather the behavior specified by the model is incorporated throughout a subsystem within the program.

An example of this might be the event processing model for a user interface. GUIs change state as keyboard focus and mouse click event occur, and the interface must react differently in each state. For example, if you are in a “paint” program and click on a paintbrush, the drawing behavior of the mouse cursor will be like you are moving a brush on a canvas. But you cannot erase part of the drawing while you are in this mode or state. Many aspects of modern GUI systems are very state-dependent (and valid user input sequences are often specified as a language or set of symbol sequences), but they are rarely implemented using either of the methods shown below. However, deep inside the operating system you will commonly find very large state machines, commonly implemented using some variation of the *while-switch* idiom described below. These FSMs often have approximately 1,000 states, and handle not only keyboard and mouse inputs, but hardware and software interrupts and other inputs to the system.

Programming language compilers and interpreters use large FSMs to ”tokenize” source files, picking out keywords, symbols, and literal values. Web browsers use similar code to tokenize HTML documents before converting them to an abstract document model structure that supports visual rendering in the browser as well as programmatic manipulation with scripting languages like JavaScript, VBScript, etc. The interpreters for these scripting languages also incorporate state machine-based tokenizers. Word processors and program editors use state machines to check what you type, letter by letter, for spelling or syntax highlighting. Hardware drivers use state machines to pick out control sequences and other information from the data that pass through them on the way to the hardware.

4.1 State Machine Implementation Using the *while-switch* Idiom

The *while-switch* idiom is a programming technique that predates object-oriented programming languages, and is often used to implement state machines that read binary- or character-based input. For our example, we will implement a class that accepts the language L_5 defined in the previous section. Here again is the definition of that language:

$$L_5 = \{x \in \{0,1\}^* \mid x \text{ does not contain the substring 10 and } x \text{ ends with 01}\}$$

We also defined a finite state machine that accepts strings in this language:

Let $M_5 = (\Sigma, Q, q_0, F, \delta)$ be a finite state machine recognizing L_5, where

$$\begin{aligned} \Sigma &= \{0,1\} \\ Q &= \{\mathsf{AP}, \mathsf{AQ}, \mathsf{BP}, \mathsf{BR}, \mathsf{CP}, \mathsf{CQ}, \mathsf{CR}\} \\ q_0 &= \mathsf{AP} \\ F &= \{\mathsf{BR}\} \end{aligned}$$

δ =

State	$\delta(p,0)$	$\delta(p,1)$
AP	**AQ**	**BP**
AQ	**AQ**	**BR**
BP	**CQ**	**BP**
BR	**CQ**	**BP**
CP	**CQ**	**CP**
CQ	**CQ**	**CR**
CR	**CQ**	**CP**

Now we can begin designing a class that implements this FSM. The first thing we can observe, when considering how this FSM class might interact with other parts of a program, is that all it needs to do is read input from a character-oriented source and indicate whether or not the input represents a string in the desired language. Users of this class will not be interested in the intermediate state while the input is being processed, only in the result that is determined after all of the input is complete. Since this is the case, we might consider implementing the functionality as a `static` method, much like the methods in the `java.lang.Math` class. This has the added advantage of not requiring memory allocation for an instance whenever the FSM is used.

Next, we must decide upon the interface for our class. Since the role of this class is to provide the functionality of a finite state machine to accept or reject strings in a language, we might consider calling the method that encapsulates this functionality "accept" and have it return a boolean value representing the status of the input with respect to the L_5. We must also decide what parameters will be needed for this method. We would like to have as much flexibility as possible in terms of an input source, so an InputStream-based parameter type might be appropriate. This would allow input to come from a variety of input sources, as the Java API defines many different wrapper classes on InputStreams to allow programmers to customize the input functionality. However, this could also complicate the logic of our state machine. If we used a

Reader-type parameter, the input will be restricted to character-based data, which is closer to what we want to read in. Furthermore, we could also enhance the efficiency of the program by using a BufferedReader.

We do not need to expose the internal state of the FSM, since our method will return `true` or `false` based on its acceptance or rejection of an input as a string in the language. The basic public interface for our class is shown in Figure 24. Note that the return statement is added to the method body to allow this code to compile.

```
public class FSM {

   public static boolean accept(java.io.BufferedReader input) {
      return false;
   }
}
```

Figure 24: Public Interface for the FSM Class

Now we can get on with implementation details. The *while-switch* idiom refers to a programming construct that uses a `switch` statement within a `while` loop. At the beginning of each iteration of the loop, one element of input is read in, and is then processed by the switch. When used to implement a finite state machine, each `case` clause in the switch statement encapsulates the behavior of the FSM while in a particular state. Since the Java switch statement must have an integer value as the switch-condition, we need to represent our states with integer values. Since our method is static, we need our state variables to be static as well. Furthermore, since these are constant values, we make them `final` as well, as shown in Figure 25.

```
private static final int AP = 1;
private static final int AQ = 2;
private static final int BP = 3;
private static final int BR = 4;
private static final int CP = 5;
private static final int CQ = 6;
private static final int CR = 7;
private static final int ERROR = 8;
```

Figure 25: Defining the FSM States

Note the addition of an `ERROR` state to our state variables that was not a state in the original FSM. The reason for this will become clear as we develop the code, but we can say at this point that it will be used to indicate the input of a symbol (character) that is not in the alphabet L_5 is defined over. It will also be necessary to maintain the state of the FSM at all times while it is processing input. This raises the question: Should this state variable be a class variable, or would it be better to make it local to the method that encapsulates the FSM itself? One hint to an answer is that if the variable has class scope, every time the accept method is invoked, the variable *must* be reset to represent the initial state of the machine. Since we need to reinitialize the variable each time the accept method is called, it is probably better to declare it locally within the method, along with an initialization.

Another decision we need to consider is that of exception handling. From the API Documentation, we know that most methods on instances of Streams and Readers may generate exceptions when problems occur, and we need to decide if our accept method should try to handle such exceptions, or merely pass them on to the client. Generally, we will handle an exception only if we are able to recover from the error and continue the computation that was ongoing when the error occurred. In the case of our FSM, an error reading the input is not something that we could reliable recover from, since the error might have involved the loss of one or more characters from the received input. Our FSM will not have the capability to "replace" those lost characters, so there will be no way to know how they might have affected the acceptance or rejection of the input. Thus we cannot reliably decide if the string is in the language or not, so it will be better to allow the user of our class to handle the exception at that level.

As mentioned earlier in this section, the *while-switch* idiom uses a `while` loop around a `switch` statement. This loop should iterate as long as there is input, or until the FSM enters the `ERROR` state, indicating an irrecoverable error in the input characters that does not trigger an `IOException`. A new character should be read from the input on each iteration of the loop. Figure 26 shows our progress in expanding the stub of the accept method. Note that the state of the FSM is initialized to the initial state of the state machine M_5 as we defined it earlier.

```
public static boolean accept(java.io.BufferedReader input)
      throws java.io.IOException {
   int state = AP;
   while(input.ready() && (ERROR != state)) {
      int ch = input.read();
   }
   return false;
}
```

Figure 26: Expanding the `accept` Method Stub

The behavior of our FSM is dependent upon the current state of the machine. We use the `switch` statement to "select" a segment of code to execute based on the value stored in the state variable. Each state in the FSM is represented with a separate `case` clause. It is also good programming practice to *always* include a `default` case to take care of situations when the machine's state becomes undefined. While this often can be prevented programmatically, it is still a good habit to get into.

From our FSM definition, we know that the valid input symbols are the '0' and '1' characters. Also, from the API documentation for the `BufferedReader`'s `read()` method, we must also account for another important value that may be returned. The `read()` method returns -1 when the end of the input has been reached, so we should use this as a trigger to make the final decision about the acceptance or rejection of the input as a string in the language L_5. To be safe, we should also account for input characters that are not one of these 3 values, and since these characters are not in the alphabet we are interested in, they should put the FSM into an unrecoverable error state. Figure 27 shows the addition of the stubbed switch statement, along with the implementation of the behavior for state **AP**.

Note that the behavior implemented is directly from the FSM definition. Inputs cause state changes. An input that indicate the end of the input sequence means that the FSM should stop, and if it is in an accepting state, it should return `true`. If it is not in an accepting state, as with state **AP**, it should return `false`. Invalid input characters will put the machine into an error state, which will terminate the loop. The reader should also not the presence of a `break` statement at the end of the case clause. Neglecting this

```
public static boolean accept(java.io.BufferedReader input)
      throws java.io.IOException {
   int state = AP;
   while(input.ready() && (ERROR != state)) {
      int ch = input.read();
      switch(state) {
         case AP:
            if('0' == ch)       { state = AQ; }
            else if('1' == ch) { state = BP; }
            else if(-1 == ch)  { return false; }
            else                { state = ERROR; }
            break;
         case AQ:
         case BP:
         case BR:
         case CP:
         case CQ:
         case CR:
         default:
            return false;
   }
   return false;
}
```

Figure 27: Adding the First `case` of the `switch` statement

statement will allow execution to cascade to the next `case` clause, which can produce decidedly undesirable behavior, as well as significant debugging frustration. The full implementation of the FSM class, including a main method that tests the FSM with 3 different inputs, is included in Appendix A.

4.2 Object-Oriented State Machine Implementation

As noted earlier, the while-switch idiom for state machine implementation predates the development of Object-Oriented programming languages. This does not mean, however, that we cannot implement a state machine model in an OO fashion. In fact, we can make the implementation match the model even more closely than is possible in a non-OO language. In the process, we can make the implementation more flexible and maintainable.

The key idea behind an OO FSM implementation is to encapsulate the behavior of the FSM within a classes representing the states of the FSM. This technique make use of *inheritance* and *polymorphism*, which we will cover soon, and refer back to this example in that discussion. For now, imagine a generic “State” class that is written so that we can create derived classes that share the common behavior of that class. These classes implement a method for each possible input to the machine. While this may seem inefficient from a coding standpoint, it is often the case that many of the possible inputs will cause a behavior that is common across many states. This common behavior is factored up to the generic “State” class, and the classes derived from it inherit that common behavior. This is the essence of the inheritance relation. Derived classes need only implement that functionality that is not common to other classes.

The FSM class only needs to remember its current state, in this case a reference to a State-derived object. When an input is received, it calls the corresponding method on this object, and the behavior implemented for that state is executed. The runtime system handles making the connection with the proper class implementation.

As an example, we consider the turnstile model developed earlier. The state transition diagram we created is shown again in Figure 28. The code for this example can be found in Appendix B. The classes representing the 3 states of the turnstile are implemented as *nested* or *inner* classes within the definition of the `TurnstileFSM` class. This is done to hide these classes completely from any users. Additionally, the `TurnstileFSM` class itself is implemented in the same file as the `Turnstile` class, thus hiding it from users as well.

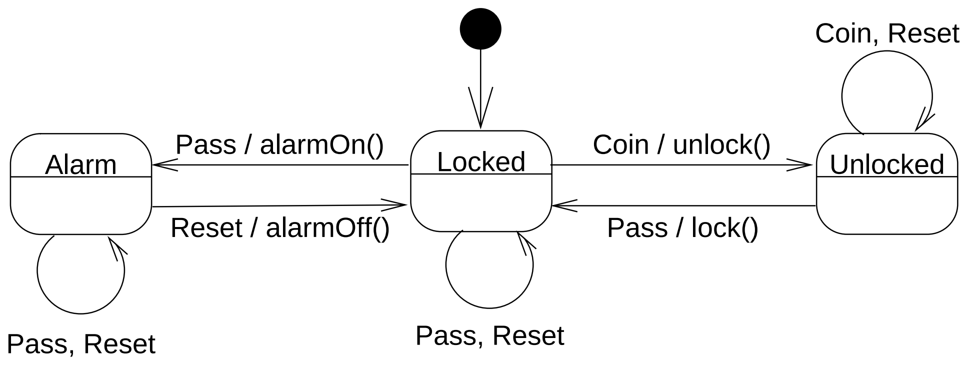

Figure 28: Turnstile State Transition Diagram

This code only implements the transition functionality of the model — it does not include the possible method calls necessary to toggle the locking mechanism or trigger the alarm. These calls, if needed, would be coded into the methods that perform the corresponding state changes.

Also included in the source code for this example is the class `TurnstileTest`, which provides a series of inputs to the turnstile to check its operation. The student is encouraged to read and trace this code in advance of our upcoming discussion of inheritance and polymorphism.

5 Appendix A. Source Code for the while-switch FSM Example

```
// File FSM.java
import java.io.*;

public class FSM {
   private static final int AP = 1;
   private static final int AQ = 2;
   private static final int BP = 3;
   private static final int BR = 4;
   private static final int CP = 5;
   private static final int CQ = 6;
   private static final int CR = 7;
   private static final int ERROR = 8;

   public static boolean accept(BufferedReader input) throws IOException {
      int state = AP;
      while(input.ready() && (ERROR != state)) {
         int ch = input.read();
         switch(state) {
            case AP:
               if('0' == ch)      { state = AQ; }
               else if('1' == ch) { state = BP; }
               else if(-1 == ch)  { return false; }
               else               { state = ERROR; }
               break;
            case AQ:
               if('0' == ch)      { state = AQ; }
               else if('1' == ch) { state = BR; }
               else if(-1 == ch)  { return false; }
               else               { state = ERROR; }
               break;
            case BP:
               if('0' == ch)      { state = CQ; }
               else if('1' == ch) { state = BP; }
               else if(-1 == ch)  { return false; }
               else               { state = ERROR; }
               break;
            case BR:
               if('0' == ch)      { state = CQ; }
               else if('1' == ch) { state = BP; }
               else if(-1 == ch)  { return true; }
               else               { state = ERROR; }
               break;
            case CP:
               if('0' == ch)      { state = CQ; }
               else if('1' == ch) { state = CP; }
               else if(-1 == ch)  { return false; }
               else               { state = ERROR; }
               break;
            case CQ:
               if('0' == ch)      { state = CQ; }
               else if('1' == ch) { state = CR; }
               else if(-1 == ch)  { return false; }
               else               { state = ERROR; }
               break;
            case CR:
               if('0' == ch)      { state = CQ; }
               else if('1' == ch) { state = CP; }
               else if(-1 == ch)  { return false; }
               else               { state = ERROR; }
               break;
            default:
               return false;
         }
      }
      return false;
   }

   public static void main(String[] args) {
      String t1 = "00001";
      String t2 = "00011001";
      String t3 = "11111";
      BufferedReader in1 = new BufferedReader(new StringReader(t1));
      BufferedReader in2 = new BufferedReader(new StringReader(t2));
      BufferedReader in3 = new BufferedReader(new StringReader(t3));
      boolean check = true;
      try {
         if(!FSM.accept(in1)) {
            System.out.println(t1 + " was not accepted but is a valid string.");
            check = false;
         }
         if(FSM.accept(in2)) {
            System.out.println(t2 + " was accepted but is not a valid string.");
            check = false;
         }
         if(FSM.accept(in3)) {
            System.out.println(t3 + " was accepted but is not a valid string.");
            check = false;
         }
      } catch (IOException ex) { System.err.println("Error reading input"); }
      if(check) { System.out.println("All tests passed!"); }
   }
}
```

6 Appendix B. Source Code for the OO FSM Example

```
// File Turnstile.java
public class Turnstile {
   private TurnstileFSM fsm;

   public Turnstile() {
      fsm = new TurnstileFSM();
   }

   public void depositToken() { fsm.token(); }
   public void passGate() { fsm.pass(); }
   public void resetAlarm() { fsm.reset(); }
   public boolean isAlarmOn() { return fsm.isAlarmOn(); }
   public boolean isLocked() { return fsm.isLocked(); }
}

class TurnstileFSM {
   private TurnstileState state;

   TurnstileFSM() {
      state = new LockedState();
   }

   void setState(TurnstileState s) { state = s; }
   void token() { state.token(this); }
   void pass() { state.pass(this); }
   void reset() { state.reset(this); }
   boolean isAlarmOn() { return state.alarmOn(); }
   boolean isLocked() { return state.locked(); }

   abstract class TurnstileState {
      abstract void token(TurnstileFSM fsm);
      abstract void pass(TurnstileFSM fsm);
      abstract void reset(TurnstileFSM fsm);
      boolean alarmOn() { return false; }
      boolean locked() { return true; }
   }
   class LockedState extends TurnstileState {
      void token(TurnstileFSM fsm) { fsm.setState(new UnlockedState()); }
      void pass(TurnstileFSM fsm) { fsm.setState(new AlarmState()); }
      void reset(TurnstileFSM fsm) { }
   }
   class UnlockedState extends TurnstileState {
      void token(TurnstileFSM fsm) { }
      void pass(TurnstileFSM fsm) { fsm.setState(new LockedState()); }
      void reset(TurnstileFSM fsm) { }
      boolean locked() { return false; }
   }
   class AlarmState extends TurnstileState {
      void token(TurnstileFSM fsm) { }
      void pass(TurnstileFSM fsm) { }
      void reset(TurnstileFSM fsm) { fsm.setState(new LockedState());   }
      boolean alarmOn() { return true; }
   }
}
```

```
// File TurnstileTest.java
public class TurnstileTest {
   public static void main(String[] args) {
      Turnstile t = new Turnstile();
      System.out.println("Initial state: " + turnstileState(t));
      testTokenDeposit(t, false, false);
      testPassGate(t, true, false);
      testPassGate(t, true, true);
      testResetAlarm(t, true, false);
      testTokenDeposit(t, false, false);
      testPassGate(t, true, false);
   }
   private static void testTokenDeposit(Turnstile t, boolean lock,
                                        boolean alarm) {
      t.depositToken();
      if(doChecks(t, "testTokenDeposit", lock, alarm)) {
         printPassMsg("testTokenDeposit");
      }
   }
   private static void testPassGate(Turnstile t, boolean lock, boolean alarm) {
      t.passGate();
      if(doChecks(t, "testPassGate", lock, alarm)) {
         printPassMsg("testPassGate");
      }
   }
   private static void testResetAlarm(Turnstile t, boolean lock, boolean alarm) {
      t.resetAlarm();
      if(doChecks(t, "testResetAlarm", lock, alarm)) {
         printPassMsg("testResetAlarm");
      }
   }
   private static boolean doChecks(Turnstile t, String tname,
                                   boolean lock, boolean alarm) {
      boolean passed = true;
      if(lock != t.isLocked()) {
         printFailMsg(tname, "isLocked", turnstileState(t), lock);
         passed = false;
      }
      if(alarm != t.isAlarmOn()) {
         printFailMsg(tname, "isAlarmOn", turnstileState(t), alarm);
         passed = false;
      }
      return passed;
   }
   private static void printFailMsg(String tname, String mname,
                                    String tstate, boolean expect) {
      System.out.println(tname + " failed testing " + mname +
                         " expecting " + expect);
      System.out.println("Turnstile state: " + tstate);
   }
   private static void printPassMsg(String tname) {
      System.out.println(tname + " passed all tests");
   }
   private static String turnstileState(Turnstile t) {
      return "Locked: " + t.isLocked() + "\tAlarm: " + t.isAlarmOn();
   }
}
```

This Page Intentionally Blank