Finite State Machines and Graph-Based Testing in C++ Games Engineering II Lecture Notes

Dr. Omer Ali

6 October 2025

1 Introduction to Finite State Machines (FSMs)

Finite State Machines (FSMs) are a fundamental model of computation and system behavior. Informally, an FSM consists of a finite set of **states** and a set of **transitions** between states that are triggered by events or conditions. A classic real-world example is a **traffic light controller**, which cycles through states *Green*, *Yellow*, and *Red* in response to a timer. Another everyday example is an **elevator**: it has states like *Idle*, *Moving Up*, *Moving Down*, *Door Open*, etc., and transitions triggered by events (button presses, reaching a floor) with conditions (doors only open when elevator is stationary). These analogies provide an intuition: an FSM represents "behavioral states" of a system and how the system moves between them. Formally, an FSM can be described as a directed graph: nodes correspond to states and edges correspond to transitions between states. Each transition typically occurs instantaneously (zero time) and represents a change in one or more state variables. The FSM may also designate an **initial state** (where the system starts) and possibly one or more accepting or final states (not always needed for testing, but used in formal definitions).

FSM transitions may include **guards** (conditions that must be true for the transition to fire) and are often triggered by explicit **events**. For example, in an elevator FSM, a transition from "Closed" to "Open" might have a guard *(elevatorSpeed == 0)* and be triggered by the event *openButtonPressed*; only if the elevator is stationary and the open button is pressed will the door open. Transitions can also carry out **actions** that occur when the transition fires (or on entering/exiting a state). In the elevator example, when transitioning to the "Open" state, the action might be to turn on the door motor to open the doors.

In a vending machine (which we'll see later), a transition for dispensing an item might have an action to release the item.

An FSM can be deterministic (each state and event leads to at most one defined transition) or non-deterministic (an event might result in one of several transitions). In this lecture, we focus on deterministic FSMs for simplicity and because they are easier to implement and test with predictable outcomes. We also constrain ourselves to FSMs with a manageable number of states (since an explosion in the number of states can make modeling impractical).

FSMs are incredibly prevalent in both software and hardware systems. Ammann & Offutt note that an FSM model can describe the state behavior of a wide variety of software systems. Indeed, many real-world software components can be modeled as FSMs: *traffic signals*, *elevator controllers*, *vending machines*, *video game AI characters*, *communication protocols*, and numerous *embedded systems* (like remote controls, appliance controllers, etc.) all exhibit behavior that can be captured in terms of states and transitions [1]. Modeling a system as an FSM not only clarifies its behavior but also sets the stage for systematic software testing, as we will explore. Before diving into testing, let us first solidify our understanding with concrete examples.

2 FSMs and Graph-Based Software Testing

One of the key reasons FSMs are important in software engineering is their close relationship with graph-based testing techniques. An FSM, viewed as a state transition graph, can be directly subjected to **graph coverage criteria** for test design. In other words, because an FSM is essentially a graph model of software behavior, the testing criteria defined for graphs can be applied to derive test cases for FSM-based software. This insight is emphasized by Ammann & Offutt [1]: FSMs (as graphs) enable us to leverage standard graph coverage criteria (like node, edge, and path coverage) to design tests.

Graph coverage criteria provide a rigorous way to ensure thorough traversal of the state graph in testing. The main criteria we consider in this lecture are:

Node Coverage (also called *State Coverage* in the context of FSMs): Every state in the FSM must be visited by at least one test case. This ensures each state's behavior is observed.

Edge Coverage (also called *Transition Coverage*): Every transition (edge) in the FSM must be taken by at least one test. This ensures each allowed state-

to-state movement is executed during testing. In code terms, each "trigger-condition-action" combination is executed at least once. Edge coverage implicitly includes node coverage (since traversing every edge necessitates visiting its source and target states).

Edge-Pair Coverage (also known as *transition-pair* or "two-trip" coverage): Every sequence of two consecutive transitions is covered by some test. Formally, this means for every reachable path of length 2 in the state graph, there is at least one test case that traverses that path [1]. The idea is to ensure not just individual transitions, but also particular combinations of transitions in sequence, are tested. This criterion begins to expose issues that might occur in the interaction between transitions (for example, taking transition A followed by B might reveal a bug even if A and B individually appeared fine). Edge-pair coverage is a weaker form of ensuring path coverage; it's a step beyond single edges.

Prime Path Coverage: Every *prime path* in the graph should be exercised by at least one test. A prime path is defined as a simple path (no repeating nodes other than possibly the start/end being the same) that is not a subpath of any longer simple path. Intuitively, these are the maximal simple paths through the graph. Prime path coverage is a strong criterion that subsumes node, edge, and edge-pair coverage. By covering all prime paths, we are testing all fundamental cycles and sequences in the FSM [1]. This usually results in very thorough tests, although it can lead to a large number of test cases if the FSM has many loops or paths. (In practice, one might select a subset of longer paths if exhaustive prime path coverage is too extensive or if some paths are infeasible.)

These criteria stem from general graph theory and are discussed in detail in software testing literature [1] (Also covered in previous lectures). By applying them to an FSM, we are systematically designing tests that (a) visit every state, (b) traverse every transition, (c) exercise pairs of transitions, and (d) cover significant cycles and sequences of transitions. The end goal is to uncover faults related to incorrect state transitions or missing states, which might not be caught by simpler testing methods. FSM-based testing is particularly useful for catching integration bugs in system behaviors, such as "after doing X then Y, the system should go to state Z, but it doesn't."

In the context of this module, FSMs also naturally arise in game development, especially for game AI and game logic. Enemies, NPCs, or game objects often have behaviors that can be modeled as states (patrolling, chasing/following, attacking, etc. in AI; or menu screens states in UI; or levels in game

progression). By applying graph coverage criteria to these FSM models, we can derive robust test suites for game logic. This ensures, for example, that an enemy AI is tested in all its modes and transitions (patrol→chase, chase→attack, etc.), or that all menu navigation paths function correctly. The following sections will present progressively complex FSM examples and demonstrate how to implement and test them in C++. Each example will include a state diagram, code implementation, and discussion of testing (with a focus on graph coverage where applicable).

3 Example 1: Traffic Light Controller FSM

Our first example is a simple traffic light controller. This FSM manages a traffic light that cycles through three states: **Green**, **Yellow**, and **Red**. The transitions are triggered by a timer expiring, which we'll simply call a **"timer tick"** event. Figure 1 illustrates the state diagram for this traffic light FSM, including the cyclic transitions.

Figure 1: Traffic Light FSM.

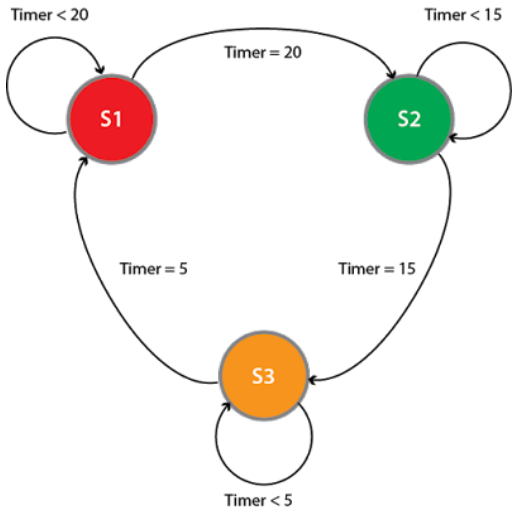

Figure 1: Traffic Light finite state machine. It has three states: *Red*, *Green*, and *Yellow*. A timer-triggered transition causes the sequence Red → Green → Yellow → Red (cycling indefinitely).

In the traffic light FSM (Figure 1), each state corresponds to a light that is actively lit. On each timer event (e.g., a fixed interval passes), the controller

transitions to the next state: from *Red* to *Green*, Green to Yellow, and Yellow back to Red, forming a cycle. This FSM is deterministic: at any given state, the timer event leads to exactly one next state. There are no additional guard conditions here besides the implicit **"timer expired"** trigger, and actions could include signals to the hardware (e.g., turning specific lights on/off) when entering a state.

To implement this in C++, we can use an `enum` to represent the state and a simple class to encapsulate the FSM behavior. The code below shows the implementation of a traffic light FSM with a method to advance to the next state on a **timer tick**. We also include a brief demonstration of its usage and a couple of assertions (using `cassert`) to verify the transitions: **Note:** We will use more advance testing frameworks such as Google Test as we move along. For now, since last week we focused only on `cassert`, it would be best to use this and understand FSM logic.

```
1  #include <cassert>
2  #include <iostream>
3  enum LightState { Red, Green, Yellow };
4
5  class TrafficLight {
6  public:
7  TrafficLight() : state(Red) {} // Initial state is Red
8  void timerTick() {
9  // Advance to next state in cycle
10 switch(state) {
11 case Red: state = Green; break;
12 case Green: state = Yellow; break;
13 case Yellow: state = Red; break;
14 }
15 }
16 LightState getState() const { return state; }
17 private:
18 LightState state;
19 };
20
21 int main() {
22 TrafficLight tl;
```

```
23 assert(tl.getState() == Red); // Initially Red
24 tl.timerTick();
25 assert(tl.getState() == Green); // Red -> Green on timer
26 tl.timerTick();
27 assert(tl.getState() == Yellow); // Green -> Yellow on
       timer
28 tl.timerTick();
29 assert(tl.getState() == Red); // Yellow -> Red on timer (
       cycle complete)
30 std::cout << "TrafficLight FSM test passed\n";
31 }
```

Listing 1: Traffic Light FSM Implementation in C++

In the code above (Listing 1), the `TrafficLight` class holds the current state and has a method `timerTick()` that updates the state according to the FSM diagram. We use a simple `switch` to cycle through the states. The `main` function then simulates the timer ticks and uses `assert` to verify that after each tick, the state is as expected. We tested one full cycle: Red → Green → Yellow → Red, and indeed the assertions passed (if any assertion failed, the program would terminate, indicating a bug in our FSM logic).

Takeaway: This traffic light FSM example demonstrates a straightforward FSM implementation. Even with such a simple model, we can think in terms of coverage: node coverage here means every light (state) was reached by some timer events (our test above achieved that by cycling through). Edge coverage means each transition (Red→Green, Green→Yellow, Yellow→Red) was executed at least once (again achieved by the test). In this simple cycle, node and edge coverage are satisfied by one full cycle test. Edge-pair coverage in this case is trivial since any two-step sequence in the cycle is also covered by the continuous cycle (e.g., Red→Green→Yellow was covered as part of the cycle). Prime path coverage would require covering the cycle as a whole (which we did) – in fact, the cycle itself is the only prime path aside from subpaths. Thus, even this basic example can be related to testing criteria. In practice, such a simple FSM might not need elaborate testing beyond ensuring the cycle is correct, but it sets the stage for more complex cases.

4 Example 2: Vending Machine FSM

Our second example is a more complex FSM: a simplified **vending machine**. This vending machine sells a product for a certain cost (say $1 for simplicity) and has the following behavior:

It starts in an *Idle* state, waiting for a coin.

When a coin is inserted, it transitions to *HasCoin* state (coin has been inserted, awaiting selection).

If the user selects a product and the product is in stock, it transitions to *Dispensing* state (deliver the product). After dispensing, it returns to Idle.

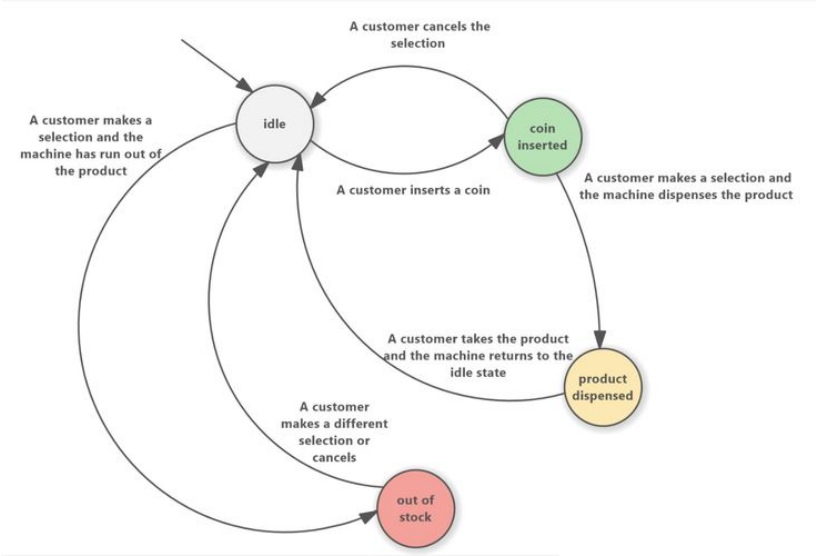

Figure 2: Vending Machine FSM. States: *Idle*, *HasCoin*, *Dispensing*. Transitions: inserting a coin (Idle→HasCoin), cancelling (HasCoin→Idle), selecting a product (HasCoin→Dispensing if item in stock; otherwise HasCoin→Idle if out of stock), and completion of dispensing (Dispensing→Idle).

If the user decides to cancel the transaction (coin return), it transitions from HasCoin back to Idle (and would return the coin, as an action).

If a selection is made but the product is out of stock, the machine returns to Idle (and in a real machine, it would return the coin since it can't dispense). We will model this by a transition from HasCoin back to Idle on a select event when out-of-stock.

The vending machine FSM has more interesting transitions with guard conditions (e.g., only dispense if stock > 0) and multiple events (insert coin, select

item, cancel). Figure 2 shows a state diagram of the vending machine FSM, including these guarded transitions.

In Figure 2, the Idle state represents "waiting for coin," HasCoin means "coin inserted (credit available), waiting for selection or cancellation," and Dispensing represents the machine delivering the item. From HasCoin, there are two possible outcomes when the customer presses the selection button: if the item is available (stock > 0), the machine enters Dispensing; if the item is not available (stock $= 0$), the machine immediately goes back to Idle (perhaps returning the coin – which would be an action on that transition). We have thus modeled a guard condition on the "select" transition. This is a **deterministic** FSM in the extended sense: given the state and the conditions, the next state is determined (if stock>0 we have one path, if stock=0 we have an alternate path). After dispensing, the machine automatically goes back to Idle (we assume the machine dispenses one item per coin).

Now, let's implement this FSM in C++. We will create a `VendingMachine` class with methods to represent events: `insertCoin()`, `selectItem()`, and `cancel()`. We'll also maintain a simple inventory count (stock) for the item, to demonstrate guard conditions. We use `cassert` in a test routine to verify the correct transitions for various scenarios (including dispensing and out-of-stock behavior).

```
1  #include <cassert>
2  #include <iostream>
3  enum VMState { Idle, HasCoin, Dispensing };
4
5  class VendingMachine {
6  public:
7  VendingMachine(int initialStock) : state(Idle), stock(
       initialStock) {}
8  VMState getState() const { return state; }
9  int getStock() const { return stock; }
10 void insertCoin() {
11 if (state == Idle) {
12 state = HasCoin;
13 }
14 // If coin inserted in HasCoin or Dispensing, we might
       reject or ignore (not modeled here)
```

```
15 }
16 void selectItem() {
17 if (state == HasCoin) {
18 if (stock > 0) {
19 // Item available: dispense it
20 stock--;
21 state = Dispensing;
22 // After dispensing, automatically return to Idle
23 state = Idle;
24 } else {
25 // Out of stock: cannot dispense, return coin
26 state = Idle;
27 }
28 }
29 }
30 void cancel() {
31 if (state == HasCoin) {
32 // Coin returned
33 state = Idle;
34 }
35 }
36 private:
37 VMState state;
38 int stock;
39 };
40
41 int main() {
42 // Test 1: Normal purchase scenario
43 VendingMachine vm(5); // start with 5 items in stock
44 assert(vm.getState() == Idle);
45 vm.insertCoin();
46 assert(vm.getState() == HasCoin);
47 vm.selectItem();
48 assert(vm.getState() == Idle); // should dispense and
    return to Idle
49 assert(vm.getStock() == 4); // stock should decrement by
```

```
    1
50
51  // Test 2: Cancel scenario
52  vm.insertCoin();
53  assert(vm.getState() == HasCoin);
54  vm.cancel();
55  assert(vm.getState() == Idle);            // coin returned,
        back to Idle
56  assert(vm.getStock() == 4);               // stock unchanged
57
58  // Test 3: Out-of-stock scenario
59  VendingMachine vm2(0);                    // empty stock
60  vm2.insertCoin();
61  assert(vm2.getState() == HasCoin);
62  vm2.selectItem();
63  // Item not dispensed due to stock=0, should return to
        Idle
64  assert(vm2.getState() == Idle);
65  assert(vm2.getStock() == 0);              // stock remains 0
66
67  std::cout << "VendingMachine FSM all tests passed!\n";
68
69
70  }
```

Listing 2: Vending Machine FSM Implementation and Tests in C++

Let's walk through the code in Listing 2. The `VendingMachine` class holds the current state (as a `VMState` enum) and the remaining stock of the product. The methods implement the FSM transitions:

`insertCoin()` moves the state from Idle to HasCoin (if the machine was Idle). We've not modeled the behavior of inserting additional coins in other states – typically, inserting a coin when already in HasCoin might be disallowed or queued, but for simplicity, our model ignores that or would treat it as no state change.

`selectItem()` is only meaningful in HasCoin. In that state, we check the guard condition `stock > 0`. If true, we simulate dispensing: decrement the stock, set state to Dispensing, then immediately set state to Idle (because dis-

pensing completes and the machine resets to Idle). The intermediate Dispensing state is brief in this model. If `stock == 0`, we simply set state back to Idle (representing the coin return due to out-of-stock).

`cancel()` moves the state from HasCoin back to Idle, representing the user aborting the purchase and getting their coin back.

The `main` function then creates instances and runs three sets of tests:

Test 1 (Normal purchase): We start with stock 5. Insert a coin (Idle→HasCoin), then select an item. We expect the machine to go to Dispensing then immediately to Idle, and the stock to decrement to 4. The asserts confirm these.

Test 2 (Cancel): Using the same machine (which now has stock 4 and is Idle), insert a coin to go to HasCoin, then cancel. The state should return to Idle and stock should remain 4 (since no item was dispensed).

Test 3 (Out-of-stock): We test a machine initialized with 0 stock. Insert coin (Idle→HasCoin), then select item. The guard will detect no stock and thus go back to Idle without dispensing. The state ends up Idle and stock remains 0.

All tests print a confirmation message if passed. We've thus systematically tested different transitions: coin insertion (Idle→HasCoin), cancellation (HasCoin→Idle), item selection in two scenarios: with stock (HasCoin→Dispensing→Idle) and without stock (HasCoin→Idle). These tests achieve at least edge coverage for the vending machine FSM: every transition in Figure 2 has been taken by some test:

Idle → HasCoin (Test 1, 2, 3 all cover this via insertCoin).

HasCoin → Idle on cancel (Test 2).

HasCoin → Dispensing (Test 1 covers this, with stock ¿ 0).

Dispensing → Idle (Test 1, right after dispensing).

HasCoin → Idle on out-of-stock (Test 3 covers this, with stock = 0 on select).

We also achieved node coverage easily (Idle, HasCoin, Dispensing all visited across tests). What about edge-pair coverage? To satisfy edge-pair (length-2 paths) in this FSM, consider sequences of two transitions: for example, Idle → HasCoin → Idle (via cancel) – covered in Test 2; Idle → HasCoin → Dispensing (Test 1 covers Idle→HasCoin→Dispensing, though we immediately then went Idle, we can consider the pair Idle→HasCoin and HasCoin→Dispensing as adjacent in the sequence of events in Test 1); HasCoin → Dispensing → Idle (Test 1 covers this sequence explicitly during the selectItem call); Idle → HasCoin → Idle (via out-of-stock select in Test 3). It turns out our tests collectively cover all pairs of adjacent transitions in the FSM's transition graph. For instance, one possible two-step path we didn't explicitly do as one continuous run

is HasCoin→Idle (cancel) followed by Idle→HasCoin (another coin) – but we did Idle→HasCoin then HasCoin→Idle in Test 2, which is indeed that sequence. Thus, edge-pair coverage is achieved.

For prime path coverage, we would consider maximal paths through this FSM. One prime path is Idle→HasCoin→Idle (through cancel or out-of-stock) – we have tests for both variants of that. Another prime path is Idle→HasCoin → Dispensing→Idle (the loop of a successful purchase) – **Test 1 covers that** . There are no longer simple paths because any further repetition would revisit states (making it non-simple). So in effect, our tests also covered the prime paths of this FSM. Systematically, we have designed tests corresponding to all meaningful usage scenarios of the vending machine model. This illustrates how thinking in terms of graph coverage criteria helps enumerate test cases: we ensured every state and transition and their combinations were accounted for in our testing.

Notes: Our C++ implementation is simplistic. In a real vending machine, the `Dispensing` state might involve a delay and an asynchronous event when dispensing is complete, rather than immediately transitioning to Idle. We might also handle invalid events (e.g., pressing cancel when no coin is present might do nothing, which in our model it would since state doesn't change). These aspects could be modeled by additional states or guard logic, but we kept it simple to focus on core FSM concepts. Despite simplifications, the test design approach remains valid – one would extend the test cases to cover any new transitions or states added in a more realistic model.

5 Example 3: Basic Enemy AI State Machine (Game AI)

In games, a common pattern for AI is to use an FSM to manage an enemy's behavior states. Our third example models a basic enemy AI with three states: *Patrolling*, *Chasing/Following*, and *Attacking*. The enemy starts patrolling an area. If the player is spotted, it transitions to a chasing state to follow the player. When the enemy is close enough to the player, it transitions to attacking. If at any point the enemy loses sight of the player or the player escapes far enough, the enemy either goes back to patrolling (if the player is completely lost) or from attacking back to chasing (if the player is still potentially visible but not in range). We will keep this model simple with a few key transitions:

Patrol → *Chase* on event `seesPlayer`.

Chase → *Attack* on event `inRange` (player is within attack range).

Attack → *Chase* on event `targetOutOfRange` (player moved out of melee range but still visible).

Chase → *Patrol* on event `lostTarget` (player is no longer visible at all – e.g., escaped or hid).

Attack → *Patrol* on event `lostTarget` as well (player disappeared during attack, so enemy gives up and returns to patrol).

This FSM has a loop between Chase and Attack (an enemy might repeatedly toggle between chasing and attacking as distance changes), and eventual transitions back to Patrol when the target is gone. Figure 3 depicts this enemy AI FSM.

Figure 3: Enemy AI FSM for a basic game enemy.

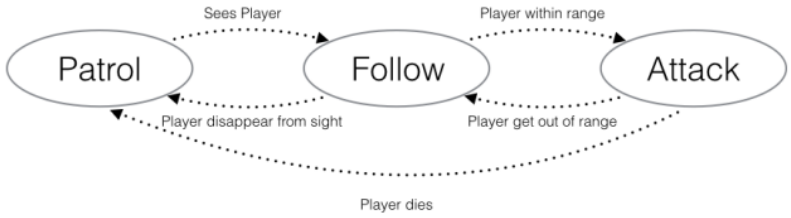

Figure 3: Enemy AI FSM with states *Patrol*, *Chase*, *Attack*. Transitions: seeing the player (`seesPlayer`: Patrol→Chase), reaching attack range (`inRange`: Chase→Attack), target moving out of range (`targetOutOfRange`: Attack→Chase), and losing sight of target (`lostTarget`: Chase→Patrol or Attack→Patrol).

This FSM is non-trivial: it includes multiple transitions out of some states based on different events, and a couple of transitions converge to the same state (both Chase and Attack can end up back at Patrol on a "target lost" event). However, it's still deterministic—given the current state and event, the next state is uniquely determined. We also have implicit guard conditions: for example, the `inRange` event would only happen if the AI was already chasing and then got close enough, and `lostTarget` might be triggered only if the target is truly gone (so one would not expect `lostTarget` and `targetOutOfRange` at the same time, they're mutually exclusive scenarios). For our implementation, we will assume events are raised appropriately by the game engine (we'll just call methods to simulate them).

Below is a C++ snippet implementing this enemy AI FSM. We'll focus on the state transitions logic. We won't include a full test suite for this as we did for the vending machine, but we will illustrate usage with a sequence of events. One could design tests similar to the previous examples to cover all transitions.

```
1  #include <iostream>
2  #include <cassert>
3  enum AIState { Patrol, Chase, Attack };
4
5  class EnemyAI {
6  public:
7  EnemyAI() : state(Patrol) {}
8  AIState getState() const { return state; }
9  void seesPlayer() {
10 if (state == Patrol) {
11 state = Chase;
12 }
13 }
14 void lostTarget() {
15 // Lost sight of player, go back to patrol from Chase or
       Attack
16 if (state == Chase || state == Attack) {
17 state = Patrol;
18 }
19 }
20 void inRange() {
21 if (state == Chase) {
22 state = Attack;
23 }
24 }
25 void targetOutOfRange() {
26 if (state == Attack) {
27 state = Chase;
28 }
29 }
30 private:
31 AIState state;
```

```
};

int main() {
EnemyAI enemy;
assert(enemy.getState() == Patrol);
enemy.seesPlayer();
assert(enemy.getState() == Chase); // Patrol -> Chase
enemy.inRange();
assert(enemy.getState() == Attack); // Chase -> Attack
enemy.targetOutOfRange();
assert(enemy.getState() == Chase); // Attack -> Chase (
    player backed up)
enemy.lostTarget();
assert(enemy.getState() == Patrol); // Chase -> Patrol (
    player disappeared)
std::cout << "EnemyAI FSM test sequence passed!\n";
}
```

Listing 3: Enemy AI FSM Implementation in C++ (Basic)

In the code above (Listing 3), the `EnemyAI` class holds the current AI state. We have methods corresponding to the key events: `seesPlayer()`, `lostTarget()`, `inRange()`, `targetOutOfRange()`. The logic in each method follows the FSM: e.g., if the enemy is patrolling and sees the player, it switches to Chase; if it's chasing and the player is in range, switch to Attack; if attacking and the player goes out of range, switch back to Chase; if either chasing or attacking and the player is lost, go back to Patrol. The main function simulates one possible scenario: the enemy sees the player (Patrol→Chase), then gets in range (Chase→Attack), then the player moves away (Attack→Chase), then the player is lost completely (Chase→Patrol). We use asserts to verify each state transition. This sequence covered a loop (Patrol→Chase→Attack→Chase→Patrol). We didn't explicitly cover Attack-¿Patrol directly, but one could simulate that by starting an attack and then losing target immediately. A comprehensive test suite would try all transitions: e.g., Attack→Patrol scenario, or perhaps repeated oscillations between chase and attack.

Discussion: From a graph coverage perspective, designing tests for this EnemyAI FSM means ensuring each state and transition is exercised. Node coverage is straightforward (Patrol, Chase, Attack should appear in some test). Edge

coverage would require tests for: Patrol→Chase, Chase→Attack, Attack→Chase, Chase→Patrol, Attack→Patrol. Our single scenario above hit Patrol→Chase, Chase→Attack, Attack→Chase, and Chase→ Patrol. We missed Attack→Patrol in that specific run, which could be another test case. Edge-pair coverage would involve sequences like Patrol→Chase→Attack (we did that), Chase → Attack→Chase (did that), Attack→Chase→ Patrol (did that at the end), Patrol →Chase→Patrol (which would imply **seesPlayer** then **lostTarget** without attacking – a test case we could add), etc. Designing these systematically ensures the AI's state transitions are all functioning. In a game, such testing might catch scenarios where, say, an enemy fails to stop attacking when it should, or gets stuck in a wrong state.

This enemy AI example also highlights the use of guard-like logic in code (we only transition Attack → Chase if currently in Attack, etc.), which corresponds to the guarded transitions in the FSM diagram. We did not incorporate any actions (like "shoot at player" action in Attack state, or "move toward player" in Chase) since those would be side effects rather than state transitions. In a full implementation, those actions would be executed in these state methods or on state entry, but for testing the FSM structure, we concentrate on state changes.

Key Point: Modeling game AI with FSMs not only clarifies the behavior for the developer, but it also provides a basis for testing complex AI logic in a systematic way. By covering various paths through the AI state machine, testers can gain confidence that the AI will respond correctly to different gameplay situations. This approach scales to larger FSMs too (though one must manage the number of test cases as state count grows).

6 Applying Graph Coverage Criteria to FSM Testing

Throughout the examples above, we implicitly applied graph coverage criteria to design test cases. Let us summarize how each criterion maps to testing the FSMs, and emphasize why this approach is effective:

State (Node) Coverage: Ensure each state is reached by some test. This uncovers problems where certain modes of the system might never be entered correctly. For instance, in the vending machine, if we never tested the Dispensing state, we might miss a bug in the dispensing logic. In the enemy AI, if we never

test the Attack state, the game might crash the first time an enemy actually attacks in a real play-through. Node coverage is a minimum baseline – every "mode" of the system gets some exposure.

Transition (Edge) Coverage: Ensure each transition is taken by some test. This is stronger than node coverage and can reveal issues in the conditions or triggers for transitions. For example, we tested the transition for coin insertion, selection (with and without stock), cancellation, etc., each at least once. If a transition is never tested, the code handling that transition (or the conditions enabling it) might contain faults. In practice, achieving edge coverage often requires carefully selecting input sequences or event sequences to invoke those transitions, as we did by constructing specific scenarios (like the out-of-stock selection).

Edge-Pair Coverage: Ensure sequences of two transitions are covered. This can catch bugs that arise from particular orderings of transitions. For instance, an edge-pair in the vending machine FSM is HasCoin-¿Dispensing followed by Dispensing-¿Idle. While each alone might work, the combination could fail if, say, the system doesn't properly reset some variable when going through Dispensing back to Idle. By executing transition pairs, we test the interface between consecutive transitions. Edge-pair coverage is especially useful in FSMs with loops or where certain state transitions can occur back-to-back. It was originally defined specifically for FSMs (as "two-trip" coverage) to systematically test sequences in state machines.

Prime Path Coverage: Ensure every prime path (maximal simple path) is tested. This is a powerful criterion because it means testing not just short sequences but also full loops and longer sequences of actions. In the traffic light, the full cycle Red-¿Green-¿Yellow-¿Red is a prime path that we tested. In the enemy AI, a prime path would be Patrol-¿Chase-¿Attack-¿Chase-¿Patrol (the loop including one attack cycle and losing the target). We executed most of that path in our test. Prime path coverage, in effect, pushes testers to consider "round-trip" scenarios and cycle behavior, ensuring the system can go out and back to a state correctly. Ammann & Offutt discuss round-trip coverage as a special case of path coverage in FSMs, where a path starts and ends in the same state. Our tests for the enemy AI, for example, included a round trip from Patrol back to Patrol after a series of events. Testing such cycles can reveal issues in systems that are supposed to return to a baseline idle state after some activity.

Table 1 summarizes these coverage criteria in the context of FSM testing:

Table 1: Graph Coverage Criteria Applied to FSMs

Coverage Criterion	**Description (FSM Context)**
Node Coverage (All States)	Each state is visited by at least one test case.
Edge Coverage (All Transitions)	Each transition is taken by at least one test case.
Edge-Pair Coverage (Transition-Pair)	Each sequence of two adjacent transitions is taken by a test.
Prime Path Coverage (All Prime Paths)	Each prime path (maximal simple path in state graph) is exercised.

By designing tests to meet these criteria, we achieve a scientifically rigorous testing approach for state-based software. Instead of random or ad-hoc event sequences, we have a clear criterion for what constitutes a thorough test suite. This approach is grounded in software testing research [1] and helps avoid gaps in testing. It's worth noting that achieving full prime path coverage can be labor-intensive for complex FSMs (the number of paths can grow rapidly), so testers often balance between coverage and effort, perhaps stopping at edge-pair or focusing on a subset of prime paths that are most critical. Nonetheless, the criteria provide a goal and a measure of thoroughness.

Finally, a practical note: when implementing FSMs in C++ (especially in larger systems), it's common to separate the logic from side effects. For instance, our examples directly changed state and sometimes stock variables. In a real system, one might have the FSM logic decide "next state" and trigger actions (like dispensing item or firing weapon) via separate functions or event dispatch. When testing, it can be helpful to have a method to query the current state (as we used `getState()`) or even to simulate events in a test mode so that you can drive the FSM without the full hardware or game engine. This allows the kind of unit testing we demonstrated.

7 Conclusion

In this lecture, we introduced Finite State Machines and illustrated their power in modeling software behavior and guiding software testing. We started with intuitive analogies (traffic lights and elevators) to ground the concept of FSMs in everyday systems. We then established how FSMs can be represented as graphs and how this opens the door to applying graph-based coverage criteria

to systematically design tests – a principle highlighted by Ammann and Offutt's work on software testing [1].

Through progressively richer C++ examples (traffic light, vending machine, and enemy AI), we saw how to implement FSMs using basic language features (enums, classes, and control structures) without advanced libraries. We included assertions (`cassert`) to verify that the state transitions occur as expected, effectively embedding test cases into our examples. Each example reinforced key FSM concepts: the traffic light showed a simple cycle (deterministic loop), the vending machine introduced guards and actions (dispense or refund), and the enemy AI showed a branched structure with loops and multiple events relevant to games. For each FSM, we discussed how to achieve test coverage – from covering all states and transitions to covering sequences of transitions and cycles. We demonstrated a complete test script for the vending machine FSM, which systematically covered normal operation, cancellation, and out-of-stock scenarios, thereby achieving all the main graph coverage criteria for that FSM.

The emphasis on graph coverage criteria (node, edge, edge-pair, prime path coverage) in FSM testing provides a rigorous methodology to validate software. Rather than guessing test scenarios, we use the FSM model to derive them. This is scientifically rigorous and aligns with established testing principles [1]. Moreover, it helps reveal subtle bugs: for example, missing transitions, incorrect guards, or improper state resets often surface when you insist on traversing every edge or cycle in the state graph. In a games engineering context, this means more robust game logic and AI – players are less likely to encounter AI stuck in a wrong state or game objects not responding because some state was never tested.

In conclusion, finite state machines offer a clear way to think about dynamic behavior, and when combined with graph-based test design, they become a powerful tool to improve software reliability. As you build games or other interactive systems, consider drawing an FSM for complex objects or AI agents. Use it to reason about the completeness of your implementation and as a guide for writing tests. By covering the FSM's graph in your tests, you can approach the testing problem with confidence and thoroughness. This approach exemplifies how theoretical computer science concepts (like state machines and graph theory) directly benefit practical software engineering by enhancing clarity and quality. We encourage you to refer to Ammann Offutt's textbook [1] for deeper insights into coverage criteria and to explore further resources like game AI textbooks [2] to see how FSMs are applied in different domains. Happy state

modeling and testing!

References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed., Cambridge University Press, 2017.

[2] M. Buckland, *Programming Game AI by Example*, Jones Bartlett Learning, 2004.