FINITE STATE MACHINES IN SOFTWARE ENGINEERING:

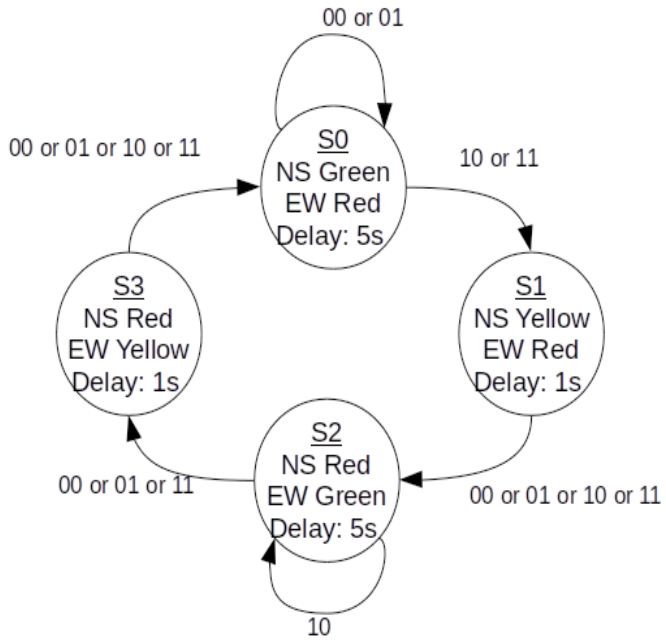
Simplifying Complex Systems with States and Transitions

ECC 811 – SOFTWARE ENGINEERING

Monday, July 2, 2025

WHAT IS A FINITE STATE MACHINE?

1. **Finite State Machine(FSM)** is a mathematical model describing a system with finite states, transitions triggered by events, and associated actions.
2. **Real-World Examples of FSM:**
   - **Traffic lights** (Red → Yellow → Green)
   - **Vending machines** (Idle → Payment → Dispense)
3. **Purpose:** FSMs are powerful tools for modeling complex systems, especially those exhibiting sequential logic and distinct states.



WHY FSMS MATTER IN SOFTWARE ENGINEERING

The benefits of using FSMs in Software engineering are:

1. Clarity: Visualize complex workflows (e.g., UI flows, payment gateways).

2. Debugging: Predictable behavior → easier error tracing.

3. Maintainability: Isolate state-specific logic.

CORE FSM COMPONENTS

FSMs have four main components:

1. **Visual Diagram:**

   [States] ←(Transitions)→ [States]

2. **States:** System configurations

Example: Locked, Unlocked).

3. **Transitions:** State changes triggered by events

4. **Actions:** Operations during transitions (e.g., validate_password()).

TYPES OF FINITE STATE MACHINES

TYPE	KEY RULE	USE CASE
Mealy	Action on transition	Network controllers
Moore	Action on state entry	Hardware systems
DFA/NFA	Single/Multiple paths per event	Parsers, compilers

MODELING FSMS: STATE DIAGRAMS

1. **Example Diagram:**

- [Locked] -- insert_coin --> [Unlocked] / unlock_door()
- [Unlocked] -- timeout --> [Locked] / lock_door()

2. **Best Practices:**

- Use ➔ for transitions.
- Label: Event [Guard] / Action.
- Tool Suggestion: Draw.io, PlantUML.

MODELING FSMS: STATE TRANSITION TABLES

Current State	**Event**	**Next State**	**Action**
Locked	insert_coin	Unlocked	unlock_door
Unlocked	timeout	Locked	lock_door

When to Use: State transition tables are used in complex FSMs with many states.

IMPLEMENTING FSMS: STATE PATTERN (OOP)

Java:

```
interface State {
    void handleEvent(Context context);
}
class LockedState implements State {
    public void handleEvent(Context ctx) {
        unlockDoor();
        ctx.setState(new UnlockedState());
    }
}
```

Pros: Encapsulation, extensibility .

IMPLEMENTING FSMs: STATE TABLES (DATA-DRIVEN)

Code Snippet (Python):

```
fsm = {
    ("Locked", "insert_coin"): ("Unlocked",
unlock_door),
    ("Unlocked", "timeout"): ("Locked", lock_door),
}
# Runtime engine:
current_state, event = "Locked", "insert_coin"
next_state, action = fsm[(current_state, event)]
action()
```

Pros: Decouples logic from code; easy to modify.

FSM: TOOLS & LIBRARIES

1. Popular Tools:
   - JavaScript: XState
   - Python: Transitions
   - C#: Stateless

2. Why Use Them:
   - Built-in guards/hierarchical states.
   - Visual debugging.

REAL-WORLD EXAMPLE: LOGIN WORKFLOW

1. **States:**

   INITIAL → INPUT → VALIDATING → SUCCESS/ERROR

2. **Events:**

   submit_form(), validation_success(), validation_failed()

3. **Diagram:**

   Linear flow with error recovery.

REAL-WORLD EXAMPLE: PAYMENT GATEWAY

1. **States:**

PENDING → PROCESSING → COMPLETED/FAILED → REFUND

2. **Critical Events:**

payment_received, timeout, refund_requested

ADVANTAGES OF FSM

The advantages of FSM in Software Engineering are:

1. **Modularity:** Isolate state logic.
2. **Testability:** States/transitions are unit-testable.
3. **Scalability:** Handle new states without rewriting core logic.

CHALLENGES & SOLUTIONS

Challenges:

- State explosion (too many transitions).
- Concurrency limitations.

Solutions:

- Hierarchical FSMs: Nest states (e.g., PAUSED within GAME_RUNNING).
- Statecharts: Advanced modeling (parallel states, history).

SUMMARY

1. FSMs simplify event-driven systems.
2. Choose between
   - Mealy (transition actions) or
   - Moore (state actions).
3. Implement via
   - State Pattern (OOP) or
   - State Tables (data-driven).