OOP

Spring 2024

Assignment 4 (FSM) Report

Students:

Norhan Alatyr 900203525

Ahmed Badr 900202868

I. General Flow of the Project

The project simulates a Finite State Machine (FSM) with the following main components: states, transitions, actions, and variables. The FSM reads a configuration file describing these components. It then starts execution from the initial state, processing actions and transitioning between states based on the provided input. The FSM supports various actions, including printing messages, assigning values to variables, sleeping, waiting for user input, and jumping to different states or FSMs.

II. Short Description of Each Class and Basic Functionality

1. StateMachine

Manages the overall state machine, including states, transitions, and variables.

Key Functions:

- addVariable: Adds a new variable with an initial value of 0.
- addState: Adds a new state to the state machine.
- addTransition: Adds a new transition between states.
- setStartState: Sets the initial state of the FSM.
- start: Begins the execution of the FSM.
- end: Ends the execution of the FSM.
- setCurrentState: Sets the current state based on the state name.
- transition: Processes a state transition based on user input.
- getStateNames: Retrieves the names of all states in the FSM.
- getTransitions: Retrieves all transitions as strings.
- clearTransitions: Clears all transitions in the FSM.
- getMachine: Static method to get an FSM instance by name for handling JMP0.
- addMachine: Static method to register an FSM instance, for JMP0.

2. State

Represents a state within the FSM.

Key Functions:

- getName: Returns the name of the state.
- addAction: Adds an action to the state.
- execute: Executes all actions associated with the state using a lambda expression to encapsulate the logic.

3. Transition

Represents a transition between states.

Key Functions:

- getInput: Returns the input that triggers the transition.
- getSource: Returns the source state of the transition.
- getDestination: Returns the destination state of the transition.

4. **Action (Abstract Base Class)**

Base class for all actions that can be performed in a state.

Key Functions:

- execute: Pure virtual function to execute the action.

5. **PrintAction**

Prints a message or the value of a variable.

Key Functions:

- execute: Prints the message or variable value.

6. **SleepAction**

Pauses execution for a specified duration.

Key Functions:

- execute: Pauses execution using std::this_thread::sleep_for.

7. **AssignAction**

Assigns a value to a variable based on an operation and operands.

Key Functions:

- execute: Performs the assignment operation and updates the variable.

8. **JMPAction**

Jumps to a different state within the same FSM.

Key Functions:

- execute: Changes the current state of the FSM.

9. **WAITAction**

Waits for user input to transition to the next state.

Key Functions:

- execute: Prompts for user input and processes the transition.

10. ENDACTION

Ends the execution of the FSM.

Key Functions:

- execute: Sets the FSM to the end state.

11. JMP0Action

Jumps to the start state of another FSM.

Key Functions:

- execute: Ends the current FSM and starts the specified FSM.

12. Utils

Provides utility functions.

Key Functions:

- stripFSMExtension: Removes the file extension from the FSM filename.

III. Reasonable Restrictions

- **Input Validation:** The input file must adhere strictly to the specified format. Any deviation may result in parsing errors.
- **Variable Initialization:** Variables are initialized to 0 unless explicitly modified. Uninitialized variables may cause unexpected behavior.
- **Circular Dependencies:** The FSM should avoid circular dependencies where states endlessly transition between each other without a condition to break the cycle. However, the use can input "END" to stop the machine and exit the program.
- **Thread Safety:** The current implementation is not thread-safe and assumes single-threaded execution. This happens because of the sleep action.
- **Error Handling:** Errors in the input file or during execution will terminate the FSM with an error message. Proper error handling mechanisms are placed to ensure nice termination.
- **File Dependencies:** The FSM relies on configuration files to define its structure. Missing or incorrectly named files will cause runtime errors.

IV. Usage of OOP tools:

- ✓ **Inheritance**
  - **Action** is a base class from which specific action classes (PrintAction, SleepAction, AssignAction, etc.) are derived.
- ✓ **Polymorphism**
  - The **Action** class defines a common interface for all actions.
  - Polymorphism is used to call the appropriate **execute** method of the derived action classes at runtime.
- ✓ **Abstract Class**
  - **Action** is an abstract class that defines the interface for different types of actions.
- ✓ **Pure Virtual Functions**
  - Action class declares a **pure virtual function execute**, which must be implemented by all derived action classes.

```
class Action {
public:
    virtual ~Action() = default;
    virtual void execute(std::map<std::string, int>& variables, StateMachine& fsm) const = 0;
};
```

- ✓ **Smart Pointers**
  - Smart pointers are used throughout the project to manage memory for State and Action instances.
  - **StateMachine** class uses std::shared_ptr to manage State objects.
  - **State** class uses std::shared_ptr to manage Action objects.
- ✓ **Design Patterns**

State Pattern

- It represents different states and their behaviors within the FSM.
- **StateMachine** class manages the current state and transitions between states.
- **State** class encapsulates the behavior associated with each state.

Strategy Pattern

- It encapsulates different actions as interchangeable strategies.
- **Action** interface is implemented by various action classes (PrintAction, SleepAction, AssignAction, etc.).
- **State** class uses these actions as strategies that can be executed in sequence.

- ✓ **Lambda Expression**

- In the State class, the execute method uses a lambda expression to iterate over and execute each action in the state's action list.

```cpp
void State::execute(std::map<std::string, int>& variables, StateMachine& fsm) const {
    // Using a lambda expression to execute each action
    auto executeAction = [&](const std::shared_ptr<Action>& action) {
        action->execute(variables, fsm);
    };

    for (const auto& action : actions) {
        executeAction(action);
    }
}
```

V. Sample test FSMs running on the simulator

The given two sample fsms were used for the initial testing; However, the had some missing transitions that are important to test the program. Here are two sample test FSMs used to test the simulator.

Testing on the given **fsm2.fsm**(but with a slight change):

fsm2.fsm is as follows:

fsm2.fsm

```
1   FSM fsm2
2   VAR W, X
3   STATES:
4   [A] PRINT (State A); X = *(X, 2); WAIT;
5   [B] PRINT (State B); W = +(3, W, 4); WAIT;
6   [C] PRINT (Thank you for using fsm2); PRINT (X); PRINT (W); END;
7   [D] PRINT (State D); JMP (C);
8   [E] PRINT (Jumping to another FSM); JMP0 (fsm1.fsm);
9   TRANSITIONS:
10  [7] A >> D
11  [1] A >> B
12  [2] B >> A
13  [2] A >> A
14  [1] B >> B
15  [3] B >> C
16  [4] B >> E
```

The program first prints the states in the machine and the transitions. For the sake of verifying correct parsing, and also for the user readability.

```
States in fsm2.fsm:
A
B
C
D
E
Transitions in fsm2.fsm:
[7] A >> D
[1] A >> B
[2] B >> A
[2] A >> A
[1] B >> B
[3] B >> C
[4] B >> E
fsm2: State A
fsm2: Executing PRINT(State A)
State A
fsm2: Executing X = *(X, 2)
Updated variable X to 2
fsm2: Executing WAIT
```

The program starts at the start state which is state A in this case.

As per the recommendation,

“Recommended to print the name of active state and currently executed action preceded by the active machine name on the display.”

the machine name, the active state, and the action being executed are always displayed.

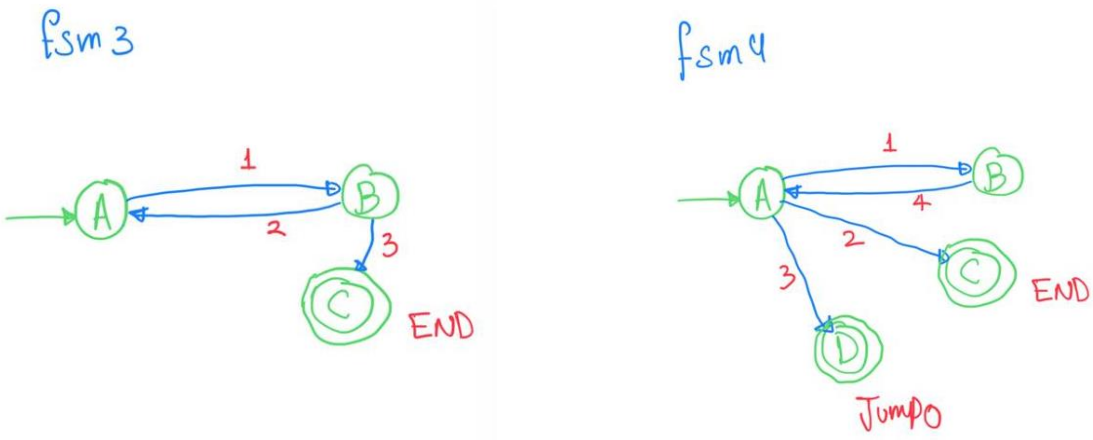
In the following example, the transitions are executed properly based on the user input. Additonally, the values of the variables are displayed once they are updated, for the sake of following up on their values.

```
fsm2: Executing PRINT(State A)
State A
fsm2: Executing X = *(X, 2)
Updated variable X to 2
fsm2: Executing WAIT
1
fsm2: State B
fsm2: Executing PRINT(State B)
State B
fsm2: Executing W = +(3, W, 4)
Updated variable W to 7
fsm2: Executing WAIT
2
fsm2: State A
fsm2: Executing PRINT(State A)
State A
fsm2: Executing X = *(X, 2)
Updated variable X to 4
fsm2: Executing WAIT
7
fsm2: State D
fsm2: Executing PRINT(State D)
State D
fsm2: Executing JMP(C)
fsm2: Executing JMP action to state C
fsm2: State C
fsm2: Executing PRINT(Thank you for using fsm2)
Thank you for using fsm2
fsm2: Executing PRINT(X)
4
fsm2: Executing PRINT(W)
7
fsm2: Executing END
PS D:\Academic\Semester 8 - Spring 2024\OOP\Assignment\A4 - Copy>
```

Finally, to test JMP, “[7] D >> A” in the given fsm was changed to “[7] A >> D”. When the current state is A and the user entered 7, the current state jumps to state C and then state C is executed, which prints “Thank you for using fsm2”, the values of the variables X and W and then, ends.

Testing on fsm4.fsm (that has a jump to machine fsm3.fsm)

Here is an illustration for both machines.



This is fsm4.fsm:

fsm4.fsm

```
1   FSM fsm4
2   VAR X, Y
3   STATES:
4   [A] PRINT(State A); X = *(X, 3); WAIT;
5   [B] PRINT(State B); Y = +(Y, 2, 3); WAIT;
6   [C] PRINT(Thank you for using fsm4); PRINT(X); PRINT(Y); END;
7   [D] PRINT(Jumping to another FSM); JMP0(fsm3.fsm);
8   TRANSITIONS:
9   [1] A >> B
10  [2] A >> C
11  [3] A >> D
12  [4] B >> A
```

fsm3.fsm:

fsm3.fsm

```
1   FSM fsm3
2   VAR X
3   STATES:
4   [A] PRINT(State A); X = +(X, 1); WAIT;
5   [B] PRINT(State B); X = +(X, 2); WAIT;
6   [C] PRINT(End of fsm1); PRINT(X); END;
7   TRANSITIONS:
8   [1] A >> B
9   [2] B >> A
10  [3] B >> C
11
```

Running fsm4.fsm:

Similar to the previous example, the states and transitions are displayed. The program is waiting for the user input to transition from state A.

```
States in fsm4.fsm:
A
B
C
D
Transitions in fsm4.fsm:
[1] A >> B
[2] A >> C
[3] A >> D
[4] B >> A
fsm4: State A
fsm4: Executing PRINT(State A)
State A
fsm4: Executing X = *(X, 3)
Updated variable X to 3
fsm4: Executing WAIT
```

Then, the user enters 1 which transition to state B, executes the actions in it then waits for user input again.

```
1
fsm4: State B
fsm4: Executing PRINT(State B)
State B
fsm4: Executing Y = +(Y, 2, 3)
Updated variable Y to 5
fsm4: Executing WAIT
```

Then, the user enters 4 which transition to state A back, executes the actions in it then waits for user input.

```
fsm4: Executing WAIT
4
fsm4: State A
fsm4: Executing PRINT(State A)
State A
fsm4: Executing X = *(X, 3)
Updated variable X to 9
fsm4: Executing WAIT
```

Now, from state A, let's transition to state D to test JMP0 which jumps to another machine. The user inputs 3 which transitions to state D, in which there is a jump to machine fsm3.fsm. Then, fsm3.fsm starts as expected.

```
State A
fsm4: Executing X = *(X, 3)
Updated variable X to 9
fsm4: Executing WAIT
3
fsm4: State D
fsm4: Executing PRINT(Jumping to another FSM)
Jumping to another FSM
fsm4: Executing JMP0(fsm3.fsm)
fsm3: State A
fsm3: Executing PRINT(State A)
State A
fsm3: Executing X = +(X, 1)
Updated variable X to 1
fsm3: Executing WAIT
```

Finally, in machine fsm3.fsm, the transitions and the action work properly as expected.

```
State A
fsm3: Executing X = +(X, 1)
Updated variable X to 1
fsm3: Executing WAIT
1
fsm3: State B
fsm3: Executing PRINT(State B)
State B
fsm3: Executing X = +(X, 2)
Updated variable X to 3
fsm3: Executing WAIT
3
fsm3: State C
fsm3: Executing PRINT(End of fsm1)
End of fsm1
fsm3: Executing PRINT(X)
3
fsm3: Executing END
```