User Guide

General Flow and Functionality

The project is designed to parse a Finite State Machine (FSM) file and construct a corresponding FSM object. The FSM file contains information about variables, states, transitions, and actions associated with each state. The main entry point of the program is the `main` function, which typically calls the `parseFSMFile` function of the `Parser` class to read and process the FSM file. The `Parser` class is responsible for parsing the file and constructing the FSM object.

The FSM object consists of states, transitions, and variables. Each state represents a unique state of the FSM, and transitions define the flow between states based on input conditions. The variables hold values that can be manipulated during the execution of the FSM.

The project follows a general flow where:

1. The program reads a text file containing the FSM definition, including variables, states, transitions, and actions.
2. The FSM parser analyzes the file line by line, extracting relevant information and constructing the FSM object.
3. Once the FSM is constructed, you can perform various operations on it, such as executing the FSM, accessing variables, and retrieving information about states and transitions.

Throughout the process, the project allows you to define states, transitions, and actions for each state in a text file and then execute the FSM based on the provided input. The constructed FSM transitions between states according to the defined transitions and actions, enabling you to implement and utilize a functional Finite State Machine.

Class Descriptions and Basic Functionality

1. `Parser` class:

   Description: The `Parser` class is responsible for parsing the FSM file and constructing the FSM object.

   Functionality: It contains the `parseFSMFile` function, which takes the filename and an FSM reference as parameters. This function parses the file line by line, identifies sections such as variables, states, and transitions, and constructs the FSM accordingly.

   Restrictions: Assumes the FSM file follows a specific format with sections for variables, states, and transitions.

2. `FSM` class:

Description: The `FSM` class represents the Finite State Machine.

Functionality: It stores states, transitions, and variables defined in the FSM file. The class provides functions to add states, transitions, and variables to the FSM. Additionally, it allows executing the FSM, accessing variables, and retrieving information about states and transitions.

3. `State` class:

   Description: The `State` class represents a state in the FSM.

   Functionality: It contains a name and a collection of actions associated with the state. The class provides functions to add actions to the state.

4. `Transition` class:

   Description: The `Transition` class represents a transition between two states in the FSM.

   Functionality: It contains the source state, destination state, and input condition required for the transition. The class provides functions to access the source state, destination state, and input condition.

5. `Variable` class template:

   Description: The `Variable` class template represents a variable used in the FSM.

   Functionality: The template parameter `T` defines the type of the variable, such as `int`, `float`, or `string`. The class provides functions to set and get the value of the variable, as well as retrieve the variable's name.

   Restrictions: The template parameter `T` can be any valid data type.

6. `Action` classes (`PrintAction`, `JumpAction`, `SleepAction`, `WaitAction`, `EndAction`):

   Description: Represents different types of actions that can be performed in a state.

   Functionality: Each action class defines its specific behavior. For example, `PrintAction` prints an expression, `JumpAction` transitions to a specified state, `SleepAction` adds a delay, `WaitAction` pauses execution, and `EndAction` terminates the FSM.

Other Syntax Used

- Templates: The `Variable` class is defined as a template class, allowing it to work with different data types.

- Standard Library Containers: The program utilizes `std::vector`, `std::string`, `std::shared_ptr`, and `std::istringstream` for data storage, string manipulation, and stream parsing.
- File I/O: The program reads the FSM definition from a text file using `std::ifstream`.
- Error Handling: The program checks for errors such as file opening failure and handles them appropriately.

Design Patterns Used

State Pattern

The State Pattern is utilized in two parts of the project: within the `FSM` class and in the `Action` interface and its derived classes.

`FSM` Class:

The State Pattern is used in the `FSM` class within the `setCurrentState` method. This method sets the current state of the FSM by iterating through the available states and finding a match based on the provided state name. Once the matching state is found, the current state is set, and the corresponding state action is executed.

This implementation adheres to the State Pattern by encapsulating the behavior of each state within its own class (`State`), allowing the `FSM` class to interact with states through a common interface. The `FSM` class delegates the execution of state actions to the current state object, following the principle of encapsulating state-specific behavior.

`Action` Interface and Derived Classes:

The State Pattern is also applied to the `Action` interface and its derived classes (`JumpAction`, `WaitAction`, `SleepAction`, `PrintAction`, and `EndAction`). The `Action` interface defines a common set of methods that each action class implements.

By employing the State Pattern, the `FSM` class can treat different types of actions uniformly through the common interface. During execution, the `FSM` class calls the appropriate methods on the current action object, regardless of its specific type. This design allows for flexible addition or modification of actions without impacting the `FSM`'s core implementation.

Overall, the usage of the State Pattern enhances the maintainability and extensibility of the FSM implementation. It promotes encapsulation and separation of concerns by associating state-specific behavior with respective state classes and providing a unified interface for actions.

Method Factory Pattern

The Method Factory Pattern was used in the project within the `Parser` class to dynamically create instances of different action classes based on the provided action type. Here's an overview of how the pattern was applied:

Method Factory Method: The `create_action` method within the `Parser` class serves as the factory method. It takes as input the action type (`action`) and a reference to an `std::istringstream` (`tokenIss`) containing the action-specific parameters. The factory method is responsible for creating the appropriate action object based on the action type.

Delegation: The responsibility of creating actions is delegated to the `create_action` factory method. Instead of directly creating action objects within the parsing logic, the factory method is invoked to handle the creation process. The factory method encapsulates the creation logic and returns a shared pointer to the created action.

Dynamic Object Creation: Inside the `create_action` factory method, a series of conditional statements are used to determine the action type and instantiate the corresponding action object. The factory method utilizes the `std::istringstream` to extract any additional parameters required for action construction. The factory method constructs the action object using the appropriate constructor and returns it as a shared pointer to the base `Action` class.

By employing the Method Factory Pattern, the project achieves flexibility and extensibility in creating different action objects without modifying the parsing logic directly. New action types can be added by implementing the corresponding action class and updating the factory method to handle the new type. This promotes code reusability and maintainability.

Class Diagram

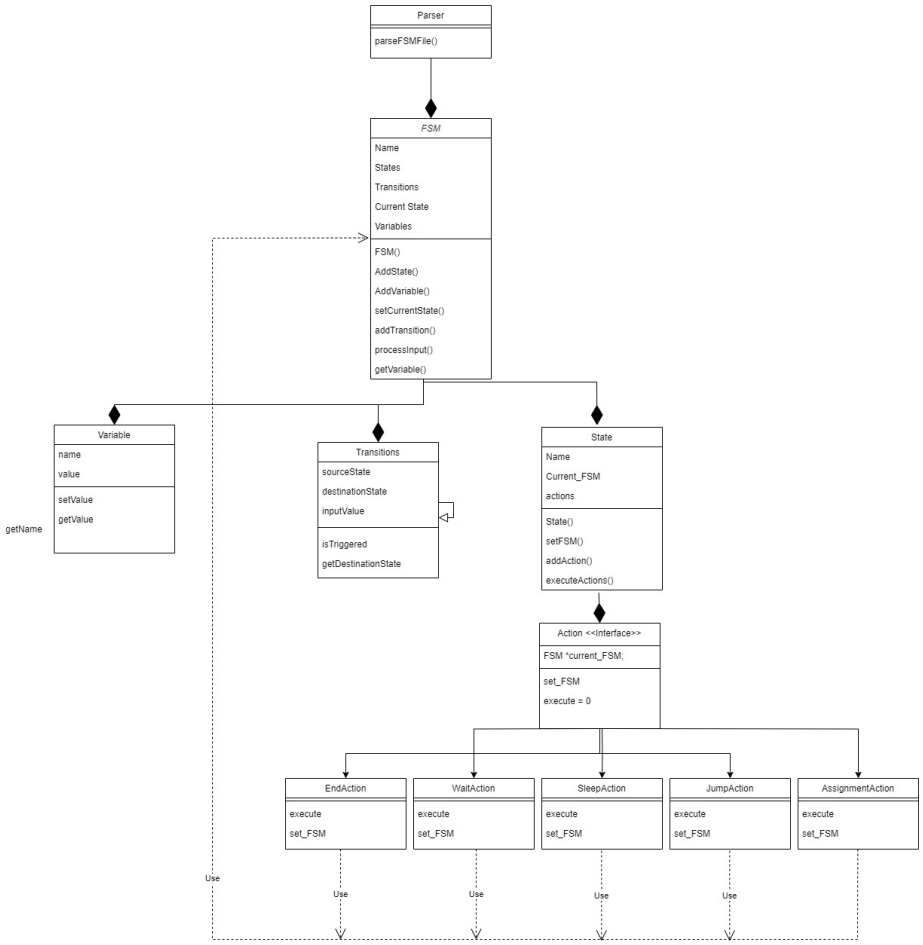

Figure 1: Class Diagram