# Finite State Machine (FSM) Application

Conceptual Design Document

Hugo Bohácsek, Filip Jenis, Eliška Křeménková

May 11, 2025

## 1 Introduction

This document presents the conceptual design of a Finite State Machine (FSM) editor and simulator application. The application allows users to create, edit, and simulate Moore machines with extended features such as variables, expressions, and timed transitions. The design emphasizes a clear separation between the model (backend) and the view/controller (frontend) components.

## 2 Architecture Overview

The application follows the Model-View-Controller (MVC) architectural pattern:

- **Model:** Core FSM components (MooreMachine, State, Transition, etc.)
- **View:** Qt-based UI components (AutomatonEditor, dialogs)
- **Controller:** Bridge between model and view (FSMBridge)

This separation allows for:

- Independent testing of the FSM logic
- Potential future changes to the UI without affecting the core FSM functionality
- Clear responsibility boundaries between components

## 3 Class Diagram

The following class diagram shows the main classes of the application and their relationships:

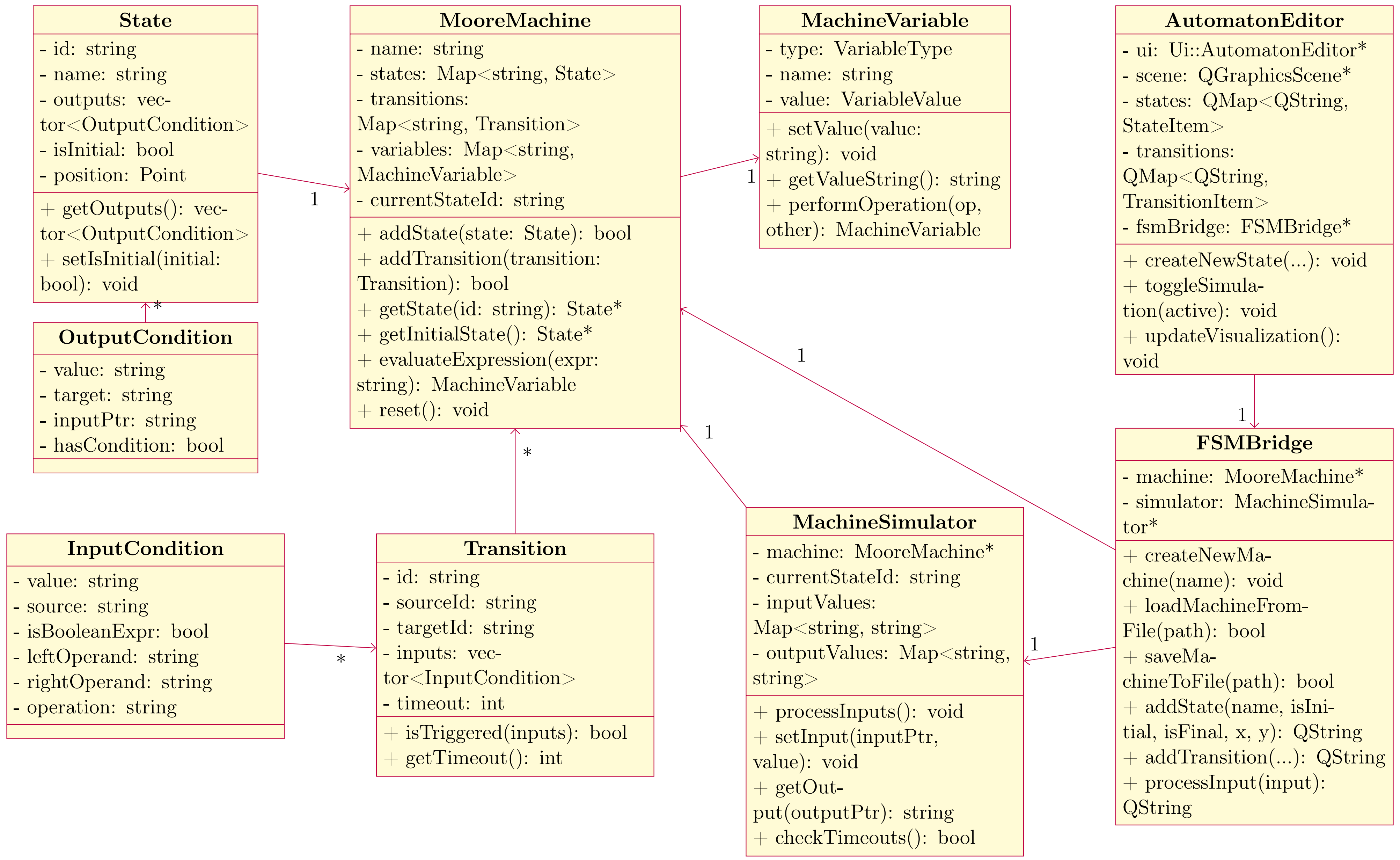
State
- id: string
- name: string
- outputs: vector<OutputCondition>
- isInitial: bool
- position: Point
+ getOutputs(): vector<OutputCondition>
+ setIsInitial(initial: bool): void
MooreMachine
- name: string
- states: Map<string, State>
- transitions: Map<string, Transition>
- variables: Map<string, MachineVariable>
- currentStateId: string
+ addState(state: State): bool
+ addTransition(transition: Transition): bool
+ getState(id: string): State*
+ getInitialState(): State*
+ evaluateExpression(expr: string): MachineVariable
+ reset(): void
MachineVariable
- type: VariableType
- name: string
- value: VariableValue
+ setValue(value: string): void
+ getValueString(): string
+ performOperation(op, other): MachineVariable
AutomatonEditor
- ui: Ui::AutomatonEditor*
- scene: QGraphicsScene*
- states: QMap<QString, StateItem>
- transitions: QMap<QString, TransitionItem>
- fsmBridge: FSMBridge*
+ createNewState(...): void
+ toggleSimulation(active): void
+ updateVisualization(): void
OutputCondition
- value: string
- target: string
- inputPtr: string
- hasCondition: bool
InputCondition
- value: string
- source: string
- isBooleanExpr: bool
- leftOperand: string
- rightOperand: string
- operation: string
Transition
- id: string
- sourceId: string
- targetId: string
- inputs: vector<InputCondition>
- timeout: int
+ isTriggered(inputs): bool
+ getTimeout(): int
MachineSimulator
- machine: MooreMachine*
- currentStateId: string
- inputValues: Map<string, string>
- outputValues: Map<string, string>
+ processInputs(): void
+ setInput(inputPtr, value): void
+ getOutput(outputPtr): string
+ checkTimeouts(): bool
FSMBridge
- machine: MooreMachine*
- simulator: MachineSimulator*
+ createNewMachine(name): void
+ loadMachineFromFile(path): bool
+ saveMachineToFile(path): bool
+ addState(name, isInitial, isFinal, x, y): QString
+ addTransition(...): QString
+ processInput(input): QString
1
1
1
1
1
1
*
*
*

Figure 1: Main Class Diagram of FSM Application

## 4 Component Diagram

The following component diagram illustrates the high-level structure of the application:

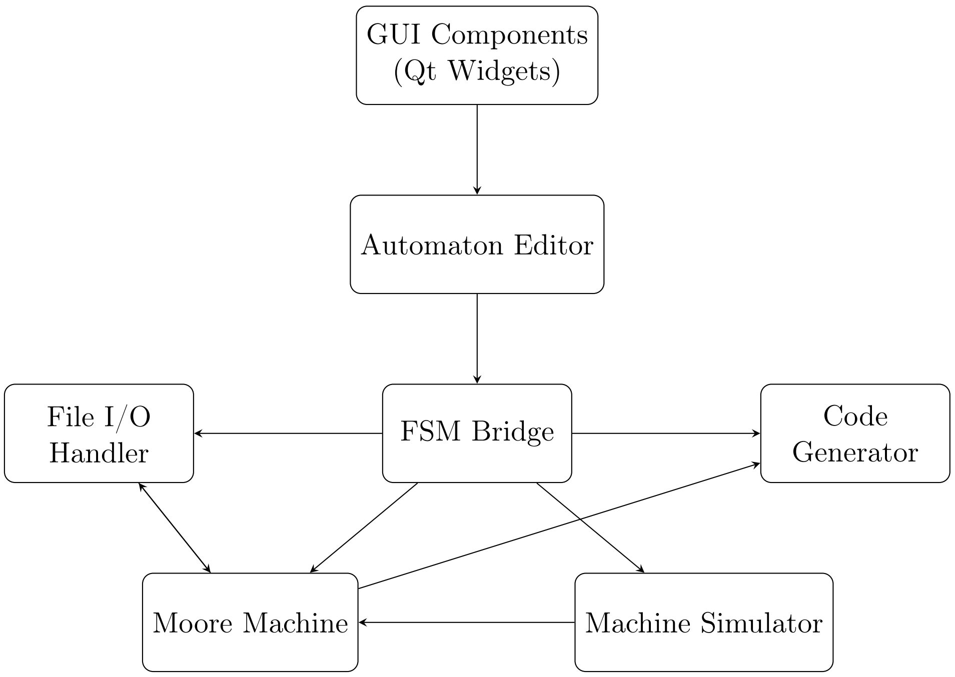

Figure 2: Component Diagram of FSM Application

## 5 Core Classes Description

### 5.1 Model Components

- **MooreMachine:** The central class that represents a Moore finite state machine with extended features. It manages states, transitions, variables, and provides methods for evaluating expressions and validating the machine.
- **State:** Represents a state in the FSM with attributes such as ID, name, position, and outputs. Can be marked as initial or final.
- **Transition:** Represents a transition between states, with associated input conditions and optional timeout.
- **InputCondition:** Defines a condition that can trigger a transition, either based on input values or boolean expressions involving variables.
- **OutputCondition:** Defines an output produced by a state, which can be unconditional or conditional based on input values.
- **MachineVariable:** Represents a variable in the machine with support for different types (int, float, string) and operations.
- **MachineSimulator:** Handles the runtime execution of the machine, processing inputs, checking timeouts, and generating outputs.

### 5.2 Controller Components

- **FSMBridge:** Acts as a bridge between the UI and the model components, providing methods for creating, manipulating, and simulating the FSM.
- **MachineFileHandler:** Handles saving and loading machines to/from files.

- **ExpressionParser:** Parses and evaluates expressions used in the FSM, including variable assignments, output expressions, and boolean conditions.
- **CodeGenerator:** Generates executable C++ code from the machine definition.
- **IncludableGenerator:** Generates reusable header and source files from the machine definition for inclusion in other projects.

### 5.3 View Components

- **AutomatonEditor:** The main UI class that provides the editor interface for creating, editing, and simulating FSMs.
- **DialogAddState:** Dialog for adding a new state to the machine.
- **DialogAddTransition:** Dialog for adding a new transition between states.
- **DialogAddBooleanTransition:** Dialog for adding a transition with a boolean expression condition.
- **DialogAddVariable:** Dialog for adding a new variable to the machine.
- **DialogEditTransition:** Dialog for editing an existing transition.

## 6 Design Patterns

The application incorporates several design patterns:

- **Model-View-Controller (MVC):** The overall architectural pattern, separating the data model (FSM) from its presentation and control.
- **Bridge Pattern:** The FSMBridge class acts as a bridge between the high-level UI and the low-level FSM implementation.
- **Factory Method:** Used for creating states, transitions, and variables with proper initialization.
- **Observer Pattern:** Used for state changes during simulation, where the UI observes and reacts to changes in the machine state.
- **Command Pattern:** Used for implementing undo/redo functionality in the editor.
- **Visitor Pattern:** Used in the code generation process to traverse the machine structure.

## 7 Implementation Constraints

The implementation will adhere to the following constraints:

- The application will be implemented in C++ using the Qt framework for the UI.
- The code will follow object-oriented design principles with clear separation of concerns.
- The model components will be independent of the UI framework to allow for reuse.
- The application will support saving/loading machines to/from files in a well-defined format.
- The generated code will be compliant with C++17 standard.

## 8 Conclusion

This conceptual design document outlines the architecture and design of the FSM application. The implementation will follow the design principles described here, with a focus on modularity, extensibility, and usability.

The class diagrams provide a high-level overview of the system structure, while the sequence diagrams illustrate the dynamic behavior during key operations. The design emphasizes a clear separation between the FSM model and its presentation, allowing for independent development and testing of these components.