Assingment 1 - Internal DSL

Marc Bertelsen
berte20@student.sdu.dk

February 28, 2024

1 Desing

Designing the API for representing state machines, firstly requires abstracting the commonalities present for all state machines, this can be accomplished by creating a metamodel.

1.1 Metamodel

Noting what parts, a state machine is made of: the overall state machine itself, the individual states, and the transitions between the states. The state machine has a current state, as well as the possible other states that it can transition to. States have transitions that can happen based on external factors like input and conditional logic. Transition can result in four different outcomes:

1. the state machine goes to a new state.
2. some parameter is updated.
3. the state machine goes to a new state and parameter update.
4. the state machine comes to an end.

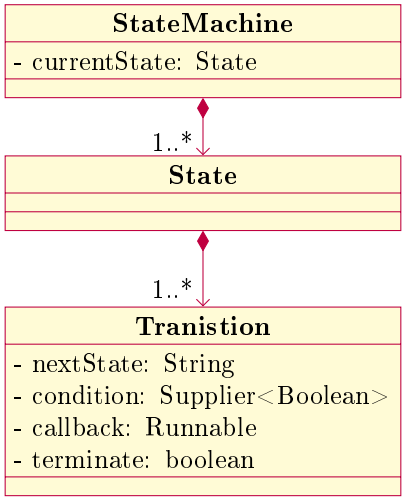

Figure 1: State Machine Metamodel

1.2 Syntax

The intent is it create a syntax that will read like:
"**Given** the model is in SOME_STATE, **when** we give it SOME_INPUT **and** SOME_CONDITION is met/true, **then** trasition to SOME_OTHER_STATE

and/or EXECUTE_CODE."

Ideally the example above will translate into code looking like the example seen on fig. 1.

```
statemachine
    .given("SOME_STATE")
        .when("SOME_INPUT")
        .and(() -> true) //SOME_CONDITOIN
        .then("SOME_OTHER_STATE")
        .then(() -> {}) //EXCUTE_CODE
```

Listing 1: Example of Intented API Usage

given - Specify the state the model should be in.

```
T gien(String state);
```

when - Specify the input that will cause a transition.

```
T when(String input);
```

and - Add an additional constraint that must be met to cause the transition.

```
T and(Supplier<Boolean> condition);
```

then - Specify the next state for the transition.

```
T then(String nextState);
```

then - Specify some code that will be run with the transition.

```
T then(Runnable callback);
```

end - Specify that a transition will lead to an end state.

```
T end();
```

start - Specify the initial state.

```
T start(String initialState);
```

1.2.1 Interfaces

The syntax design results in the **IStateMachine** interface. Making the interface generic allows for the implementation to specify the return type as itself, which in turn will allow for chaining the method calls.

```
public interface IStateMachine<T> {
    T given(String state);
    T when(String input);
    T and(Supplier<Boolean> condition);
    T then(String nextState);
    T then(Runnable callback);
```

```
    T end();
    T start(String initialState);
}
```

Listing 2: State Machine Interface

2 Implmentation

Repo: `https://github.com/Waf197/MDSD/tree/main/A1/internal_dsl`

2.1 State Machines

This subsection features the code representation of the three state machines: CD Player, Microwave Oven, and Cooking Hood. All the code listings featured are simplified from the actual implementations, as to only show the usage of the API.

2.1.1 CD Player

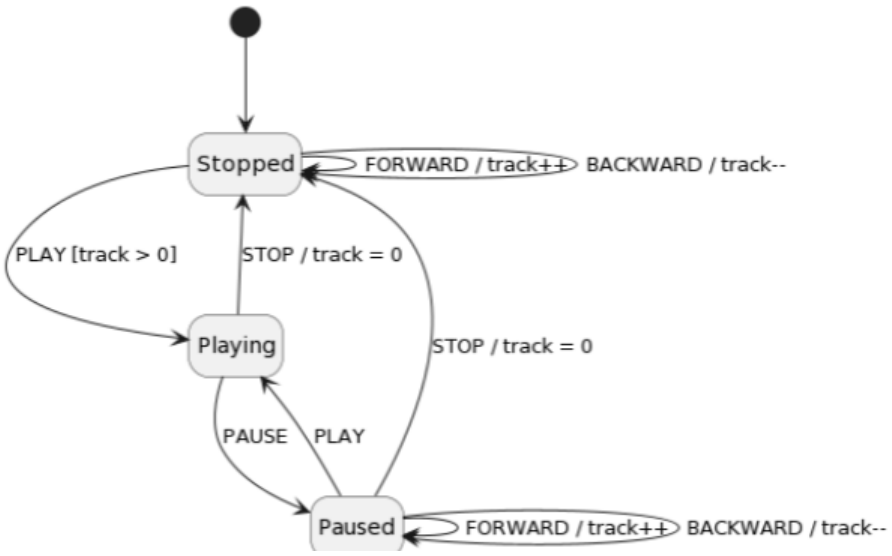

Figure 2: CD Player State Machine Diagram

```
new StateMachine("CD_PLAYER")

    .given("STOPPED")
        .when("PLAY")
        .and(() -> !trackQueue.isEmpty())
```

```
        .then("PLAYING")

        .when("FORWARD")
        .then(() -> trackIndex++)

        .when("BACKWARD")
        .then(() -> trackIndex--)

    .given("PLAYING")
        .when("STOP")
        .then("STOPPED")
        .then(() -> trackIndex = 0)

        .when("PAUSE")
        .then("PAUSED")

    .given("PAUSED")
        .when("PLAY")
        .then("PLAYING")

        .when("STOP")
        .then("STOPPED")
        .then(() -> trackIndex = 0)

        .when("FORWARD")
        .then(() -> trackIndex++)

        .when("BACKWARD")
        .then(() -> trackIndex--)

    .start("STOPPED");
```

Listing 3: CD Player Implmentation

2.1.2 Micowave Oven

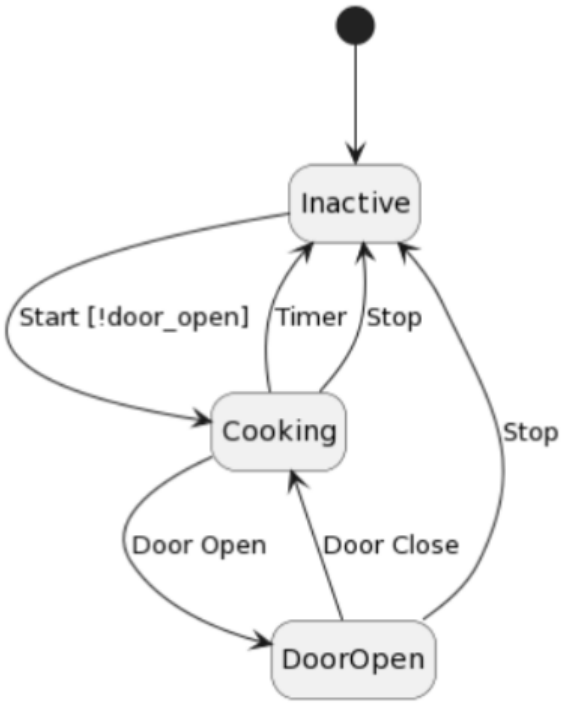

Figure 3: Microwave Oven State Machine Diagram

```
new StateMachine("MICROWAVE_OVEN")

    .given("INACTIVE")
        .when("START")
        .and(() -> !doorOpen)
        .then("COOKING")

    .given("COOKING")
        .when("TIMER")
        .then("INACTIVE")

        .when("STOP")
        .then("INACTIVE")

        .when("OPEN_DOOR")
        .then("DOOR_OPEN")
```

```
        .then(() -> doorOpen = true)

    .given("DOOR_OPEN")
        .when("CLOSE_DOOR")
        .then(() -> doorOpen = true)
        .then("COOKING")

        .when("STOP")
        .then("INACTIVE")

    .start("INACTIVE");
```

Listing 4: Microwave Oven Implemntation

2.1.3 Cooking Hood

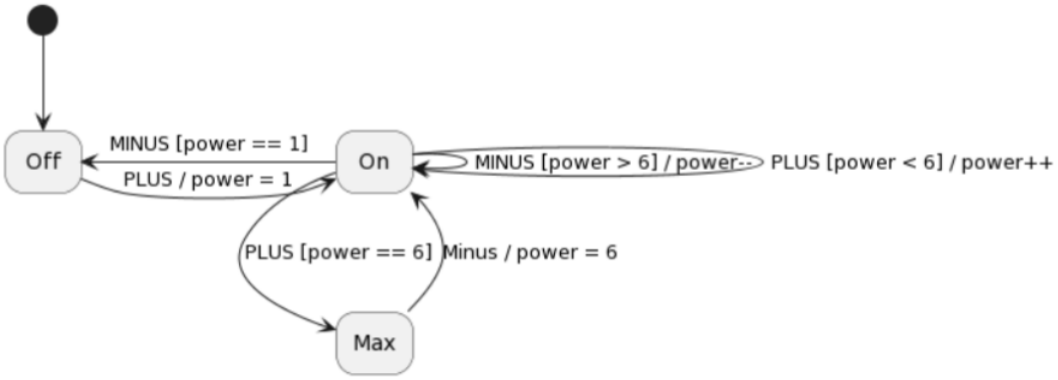

Figure 4: Cooking Hood State Machine Diagram

```
new StateMachine("COOKING_HOOD")

    .given("OFF")
        .when("PLUS")
        .then("ON")
        .then(() -> power = 1)

    .given("ON")
        .when("MINUS")
        .and(() -> power == 1)
        .then("OFF")

        .when("PLUS")
```

```
            .and(() -> power == 6)
            .then("MAX")

            .when("PLUS")
            .and(() -> power < 6)
            .then(() -> power++)

            .when("MINUS")
            .and(() -> power == 1)
            .then("OFF")

            .when("MINUS")
            .and(() -> power > 0)
            .then(() -> power--)

        .given("MAX")
            .when("MINUS")
            .then("ON")
            .then(() -> power = 6)

        .start("OFF");
```

Listing 5: Cooking Hood Implmentation

3 Test

For the purpose of testing, three additional methods are added to the state machine implementation: i(input), o(output), and printMode. These allow for configuring the input and output for the state machine, as well as how much info it outputs. Default for input and output is **System.in** and **System.out**, respectively, with printMode defaulting to **NORMAL**.

3.1 Simulating Input and Output

A queue is used to simulate the input for the state machines in the various tests. This way it will be possible to structure and ensure the order of each command to the state machine. For the output, a list is used to collect everything from the state machine. This output list is then compared with a predefined list with the expected outputs. Additionally, the print mode is set to **TESTING**, so only the state and errors are given as output.

```
final Queue<String> simulatedInputs = new LinkedList<>()
{{
    add("A");
    // structure of inputs
    // ...
```

```
}};
final Queue<String> simulatedOutputs = new LinkedList<>();
final Queue<String> expectedOutputs = new LinkedList<>()
{{
    add("STATE_A");
    // structure of expeced output
    // ...
}};

new StateMachine("GENERIC_MODEL")
        .printMode(PrintMode.TESTING)
        .i(simulatedInputs::poll)
        .o(simulatedOutputs::add)
        // state machine logic
        // ...

// state machine has terminated
assertEquals(
    expectedOutputs.size(), simulatedOutputs.size());

assertArrayEquals(
    expectedOutputs.toArray(), simulatedOutputs.toArray());
```

Listing 6: Input and Output Test Setup

Different tests have been written to test the flow from state to state, as well as testing for bad/wrong input. On fig. 5 the total coverage of the tests can be seen, with the only remaining code being print statements never reached while the printMode is **TESTING**.

Element	Class, %	Method, %	Line, %
wafl	85% (6/7)	88% (37/42)	72% (142/195)
dsl	100% (6/6)	90% (37/41)	88% (142/160)
Transition	100% (2/2)	100% (13/13)	100% (25/25)
IStateMachine	100% (0/0)	100% (0/0)	100% (0/0)
State	100% (2/2)	80% (8/10)	95% (45/47)
StateMachine	100% (2/2)	88% (16/18)	81% (72/88)
Main	0% (0/1)	0% (0/1)	0% (0/35)

Figure 5: Test Coverage

4 Conclusion

The resulting API is able to capture the logic from all three state machine diagrams, in addition to the main program having a state machine for selecting each of the implementations to run. So, the API is usable for its intended purpose.
Creating an API based on the commonalities of the different state machines, made implementing each trivial, with the added benefit of easily being able to add and change the logic without huge refactors.