Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

# COMP6255 - Coursework 2

# Finite State Machine Framework in Rust

Kristian Ivanov - kii1u20@soton.ac.uk

16/05/2024

# Implementation Details

The framework consists of two traits and five macros, four of which are user accessible and one used internally in the implementation of the state machine. The framework supports executing user-defined code when a transition completes successfully, or executing user-defined error handling code when an unknown event is received. It also supports hierarchical state definitions, allowing the nesting of states and the creation of sub-state machines. The hierarchy can be infinitely deep (as long as the computer has enough memory to allocate to the program).

---

## Traits

- **StateMachine**
  This trait represents a complete state machine and defines the types of **States**, **Events**, **Actions**, and a type for **error handling**. It also supports two generic functions: **"new"** and **"transition"**. The function **"new"** is responsible for creating a new state machine and takes as input an initial state of the State type, a transitions array (of type vector in Rust), whose elements must implement the Transition trait (explained below), and a parent HashMap, which is used to map each state to its parent, allowing for Hierarchical State Machines. The **"transition"** function takes an Event as input and moves the state machine from one state to another using the transitions vector.
- **Transition**
  This trait is used to represent a transition within the state machine and defines the type of the states, the type of the events, a special **Action** type that must be a function, representing the action to be executed when the transition occurs, and a special **HandleError** type that must also be a function, which represents the function called when the state machine reads an unknown event, allowing the user to define custom error handling. It includes a **"new"** function that defines a transition's **begin_state**, **end_state**, the **event** that initiates the transition, the **action** function to be called after the transition is completed, and the **handle_error** function to be executed if an unknown event occurs. The type for the states and events must be the same as the ones supplied to the StateMachine struct.
  Both the **action** and **handle_error** functions' type definitions include Rc (Reference Counting). This allows for the same action function and error handling function to be shared between various instances of the StateTransition struct (more on the struct below) by passing a pointer, which meets the Rust Borrow Checker's requirements. This requires the user to use **.clone()** when calling their action and error handling functions, but it just copies the pointer, not the data. Furthermore, both functions utilise the Option enum, which enables the handling of absent values. This was necessary because the user might choose not to define an action function or an error handling function. Currently, there is no way to check if a value passed to a macro is empty, hence it is impossible to create the None value automatically. The trait also defines generic getters: **begin_state**, **end_state**, **event**, **action**, and **handle_error**, which return the corresponding variable from the implementing struct.

## Structs

- **StateTransition**
  This struct defines the necessary variables for each transition. It provides concrete types for the implementation, in this case String for the states and events. The action and handle_error type is **Option<Rc<Box<dyn Fn() -> ()>>>**, which means "an optional reference-counted boxed function pointer". This allows it to hold a reference to a function, or it might not, in which case it would be **None**. The same function can be used by multiple instances of the struct (hence the use of Rc).
- **Machine**
  This struct implements the StateMachine trait, and wraps the necessary variables for the state machine: the **states**, **parent** HashMap, and **current_state**. The **current_state** variable is used for keeping track of the state the machine is currently in. This struct also provides the type for the parent HashMap, in this case from String to String (because the states are of type String). **States** represents the vector of all transitions in the state machine.

## Macros

- **extract_state_name**
  This macro is used internally by the create_states macro (more on this macro bellow), and its only function is to return a vector of the identifiers of all children states of the given parent state. This is needed for the setup of the parent HashMap.
- **create_states**
  This macro is used for populating the states vector as well as the parents HashMap. This macro calls itself recursively, traversing through the structure of the state machine, allowing the creation of hierarchical states. It creates an instance of the StateTransition struct for each transition that it matches.
- **create_state_machine**
  This macro is where it all comes together. It generates the struct **Machine**, and implements the **StateMachine** trait for that struct.
  The **"new"** function accepts the initial state, the states vector and the parent HashMap, both created by the create_states macro. It then returns the Machine instance, allowing the user to assign it to a variable.
  The implementations of the transition function loops over the vector of states (instances of the StateTransition struct), and remembers the handle_error action defined for the current_state of the machine. The reason for this is because if the event received by the state machine cannot be executed in the current state, the state machine will try to execute it on the parent state, if one exists. If this fails as well, it should execute the handle_error action from the state that tried to execute the event initially, instead of executing the handle_error action assigned to the parent state.
  After saving the handle_event action, it will populate a new vector of possible end states, if multiple exist. If that vector is not empty, i.e. an end state exists, it will try and execute the transition, executing the transition action defined by the user as well. If more than one end states are found, it will randomly select one, simulating non-determinism. If the end state of the transition doesn't exist in the states vector, the function will print an error message to the console ("The goal state {} of event {} for state {} is not defined!"), and the transition will be halted, not executing the user-defined action or moving onto a new state. The last if statement in the implementation is responsible for traversing the parent-child structure, in case the current state doesn't have a defined response to the input event. This will continue

until no more parent states exist, and if the event is not executed by that point, the user-defined handle_error action will get executed, and the machine won't transition onto a new state.

- **define_action**
  This macro is used for defining actions to be executed when a transition completes, or when an error occurs. It accepts a name for the action, followed by a block expression. A block expression in Rust Macros represents a block of code with its own scope. For example, a user can define **action1** as:
  ```
  define_action!(action1, {
      println!("Hello, this is action 1!");
  });
  ```
  and the action can be called as **action1.clone();** as both a transition action or a handle_error action. The decision to implement this macro instead of allowing the user to directly type the code into the correct field was taken because an action might be used more than once, and for different states, which could create a lot of code repetition. Furthermore, the macro generates the correct type of code block, allowing it to be saved to a variable, and passed to the StateTransition struct. This makes it easier for the user compared to if they had to define the function themselves.
- **define_state_and_parent**
  This macro returns the states vector and parent HashMap without the user having to manually define them. It is called like this:
  ```
  let (mut states, mut parent) = define_state_and_parent!();
  ```
  and returns the states and parent variables, which then need to be passed to the create_states macro and later on to the create_state_machine macro. A complete guide on how these are used is provided in Usage section below.

## Design Decisions

Initially, there was only one trait ,the StateMachine one, which defined transitions, state management, action handling, etc. This works for a simple implementation of the framework, but adding the ability for users to define their own actions, error handling, and implementing Hierarchical States, meant that the trait was becoming too complicated, and should be broken down. This is why I implemented the Transition trait.
Before the Transition trait was implemented, actions were defined as just a stream of tokens, which would get put at the correct place by the macro, and executed at runtime. Implementing the trait meant that now there needed to be a way to capture all actions during state creation, which is done at compile time, and store them in a variable. Because of this, actions now have the type Fn() -> (), which allows the user to define a function, and a pointer to that function can be saved at compile time. Furthermore, it is not possible to save a token tree to a variable, expect for converting it to a String. But after doing so, it is not possible to convert the String to a token tree.
Currently, the framework doesn't allow for a value to be passed to the state, and captured by the action. I tried implementing the create action macro like this:

```
macro_rules! define_action {
   ($name:ident, $($tokens:tt)*) => {
      macro_rules! $name {
         ($param:expr) => {
            let input = $param;
           $($tokens)*
         };
      }
   };
}
```

In this version, the define_action macro accepts a name and a token tree, and then returns the code for creating a new macro, which accepts a parameter. This allowed the action to be invoked as:

**action1(someValue);**

But the input variable was not visible to the code that comes after it. Running cargo expand shows that they appear to be in the same scope, but that is not the case. Reading the Rust documentation revealed that when a macro is expanded, every identifier introduced by the macro is intentionally hidden from the caller code, in order to prevent name clashes between the user-defined variables and the ones used in the macro. This is due to the macros in Rust being hygienic. The same holds true when I tried passing a value to the functions currently used for defining actions. This is because in Rust, these functions are captures as closures, and closures capture the environment they are defined in, not executed in. The action is expanded at compile time, and won't have access to the runtime variables of the transition function. There is one "hacky" way to have the actions interact with variables outside of the macro scope: define a function outside of the main that does whatever you want to do with the variable, and pass this function to the action. An example of this is provided in the submission (ModifyingExternalVariables.rs).

In the implementation of the traits, I have defined the type for the states and events as String. The only way currently available to allow "anything" to be a state or an event is to use an enum, but these have a lot of limitations. Most importantly, they don't act as vectors and cannot be extended after their creation. Because the macro state_machine is recursively calling itself, it is impossible to define one enum, and add all the states to it. Furthermore, enums can't store the structs used to represent Transitions, necessitating the use of the vector type. Vectors require a concrete type when used, and cannot hold more than one type of elements. For this reason, the current implementation restricts states and events to the String type. The traits all have generic types, allowing another developer to implement the state machine in a different way, if they wish to do so.

The reason for defining multiple types in the traits, as opposed to defining them as **trait traitName<T>** is to allow for the states and events to have different types from each other. For example, the states could be Strings, and the events integers, if a developer was to use the traits for their own implementation of the state machine framework.

This implementation requires that all states have a unique name. That is, no two states can have the same identifier, no matter if they are child states or parent states. Repeating identifiers are not supported even for child states of different parent states. This is because in the parent HashMap, there can be only one key with a given identifier. Further to that, if the states vector has multiple states with the same identifier, this can lead to the state machine transitioning to the wrong state, rendering it useless.

## Usage

To use the library, some imports need to be added at the top of the main.rs file:

```
mod state_machine_library;
use crate::state_machine_library::StateMachine;
use crate::state_machine_library::StateTransition;
use crate::state_machine_library::Transition;
use std::collections::HashMap;
use std::rc::Rc;
use rand::Rng;
```

The rand library needs to be included as a dependency in the Cargo.toml file. This is necessary so that all components of the framework are available. Then, the user will need to call the **define_state_and_parent** macro, and save the output to some variables:

```
let (mut states, mut parent) = define_state_and_parent!();
```

If the user wants to define custom actions, they can do so using the **define_action!** macro. For example, to define **action1**, the macro can be called as follows:

```
define_action!(action1, {
    println!(“Hello, this is action 1!”);
});
```

Then, **create_states**! needs to be called, passing it the output variables from the **define_state_and_parent**! macro, followed by the state machine definition.

The structure of the state machines is as follows:

```
create_states!(states, parent,
    StateName => {
      EventName => [EndState1, EndState2 …] => {ActionWhenEventExecutes},
      …
   } => {ActionToBeExecutedOnUnknownEventReceived}
      + [
        ChildStateName => {
          EventName => [EndState, …] => {ActionWhenEventExecutes},
        } => {ActionToBeExecutedOnUnknownEventReceived}
       ],
      …
);
```

The simple example of a state machine given in the coursework specifications can be expressed as:

```
create_states!(states, parent,
    Locked => {
      Coin => [Unlocked] => None,
      Push => [Locked] => None
    } => {None},
    Unlocked => {
      Coin => [Unlocked] => None,
      Push => [Locked] => None
    } => {None}
);
```

This state machine has two states: Locked and Unlocked. Each state can respond to two events: Coin and Push. There are no actions defined to be executed when the transitions complete, and no error handling actions either. To specify an action to be executed, the user can pass an action defined earlier using the define action macro. Here is an example on how the Coin event can be modified:

```
Coin => [Unlocked] => {action1.clone()}
```

Now, if the event executes successfully, the state machine will transition to the Unlocked state, and execute action1, as defined above.
To execute an action if an error occurs, the user can pass an action defined earlier using the define action macro. For example:

```
Locked => {
      Coin => [Unlocked] => None,
      Push => [Locked] => None
    } => {action1.clone()}
```

In this example, when the state machine receives an event that cannot be handled, action1 will be called.
The framework also supports hierarchical states, meaning that child states and sub-state machines can be defined. It allows for infinitely many levels, and will traverse up the structure to find a parent capable of executing an event in case the current state cannot. Child states are defined using + **[]**, and inside the brackets a state definition is written as before. For example, if the user wants to define the Unlocked state a child of Locked, it can be done as follows:

```
create_states!(states, parent,
    Locked => {
      Coin => [Unlocked] => None,
      Push => [Locked] => None,
      TestEvent => [Unlocked] => None
    } => {None} + [
          Unlocked => {
            Coin => [Unlocked] => None,
            Push => [Locked] => None
          } => {None}
      ]
);
```

Here, Unlocked is a child state of Locked, and I have added an extra event to Locked: TestEvent. If machine is in the Unlocked state, and receives “TestEvent” as input, it will check the parent state, and will execute the transition in the parent state.

The state machine framework supports non-deterministic automatons as well. It will randomly select a state from the vector of end states, simulating the behaviour of non-deterministic state machines.

If an event has an end state that is not defined in the state machine, an error will be printed to the console, telling the user exactly which state is undefined. For the example above, if I made the Push event in Unlocked lead to a state called Hello, I will get this error in the console:

```
The goal state Hello of event Push for state Unlocked is not defined!
```

## Quick explanation of the example applications

The main.rs file provides a complete example of a state machine that has custom events, custom error handling, hierarchical states, and non-determinism. The example demonstrates a weird vending machine that chooses a random drink each time the user puts coins in. After a user puts a coin in, it will transition to the Unlocked state. After pushing a button, it will randomly select one of three drinks. This will transition it to one of the given states (Pressed1, Pressed2, or Pressed3, referred to as slot states below). After a confirmation from the user, the machine will start dispensing the drink, transitioning to the DispensingItem1, DispensingItem2, or DispensingItem3 event, depending on the state selected before. After the dispensing is complete, the vending machine will send a Complete event, and this will transition the machine to the Locked state. If at any point the user puts more coins in, the machine will remove their selection, transitioning back to the Unlocked state. If at any point an unknown event is received, the machine will execute the handleErrorAction. Action1, Action2, and Action3 are defined for each of the item slot states.

The HandleErrorAction.rs file provides a simple example of how custom error handling works.
The StateNotDefinedError.rs file provides an example of the error shown when a goal state is not defined.
The ChildStatesAndNondeterminism.rs file provides an example of a state machine with multiple child states, transitioning to one of them non-deterministically.
The ModifyingExternalVariables.rs file provides an example of interacting with outside variables. For this example, the lazy_static crate is required as a dependency in the Cargo.toml file.
The LockedUnlockedSimple.rs file implements the basic example from the coursework specification.