TDD4FSM (test driven development for finite state machines)

Content

Preface

Primary motivation for FSM (Finite State Machine) test driven development and its very closely related sibling “finite state machine exploitability” is simply commerce. Sometimes automata get so complex, that it’s impossible to develop them further and at the same time it’s not possible to let them go wide for much audience without protection - the machine has its value! The aim of this document is very simple, to provide to any audience (with few previous knowhow) insight into possibilities that start to exist at the moment, when you modify automaton main function to accept callbacks, that may bypass the definition of finite state automaton.

Imagine you have 50 nodes in a FSM graph definition and you want to test transition between node 38 and 39; the only traditional way is to move via XX transitions from starting transition/node to node 38 and then do the test... aim of my document is to say nope, we exploit directly to node 38 which is against FSM definition and rules

What is the graph

Graph is a set of two subsets: vertices/vertexes (in this document typically “nodes”) and edges (in this special case of bi-directional strongly connected graph “transitions”). Definition of graph in a programming language is a set (array) of vertices/nodes and a set (array) of transitions - tuples (two vertexes) where direction matters.

What is the automaton (machine, FSM)

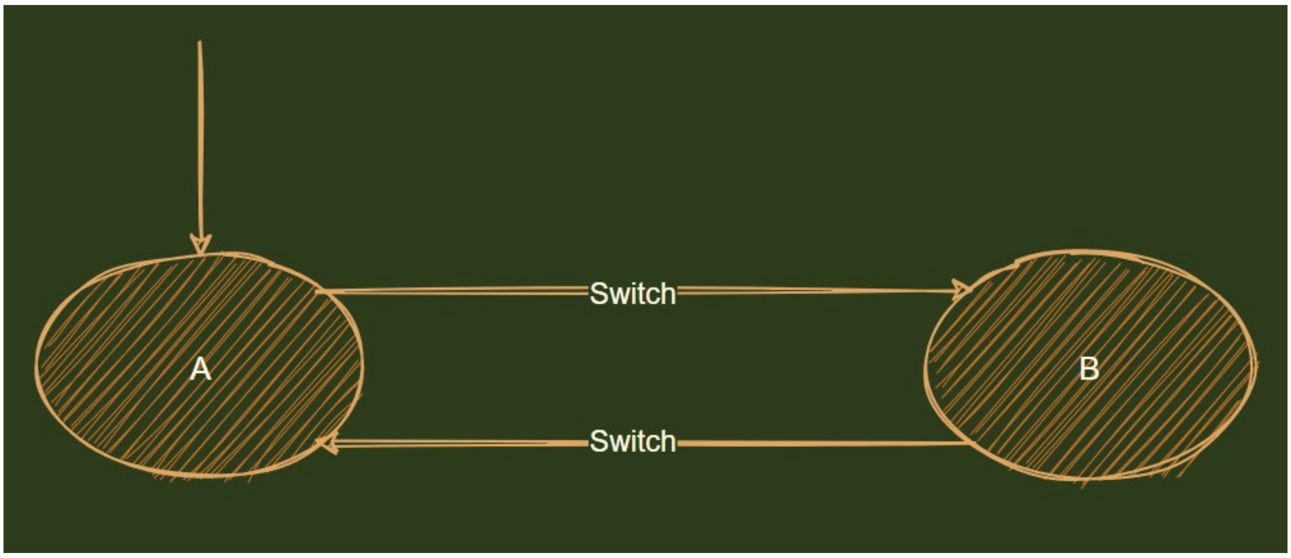
Automaton is after all a function (feel free to read infinity of info to find out that ... yeah, it’s really just a function). This function on some action (input, signal, …) resolves from some node via transition based on input/signal to some ending node.
Example: street lights have 4 possible states: green exclusive on, green and orange on, red exclusive on, red and orange / just orange on, this is automaton with 4 vertices and 4 transitions between each vertex that looks in visual representation like a circle.

What is test driven development

Test driven development is a computer software methodology that advocates writing of simple and small tests before actual implementation of code, that leads to positive passing of those tests. This approach is typically mixed with implementation itself before tests and it's really not easy to switch to mindset of TDD programmer from non-TDD one. Advantage of TDD is in complete coverage of project with tests, which leads to 0 bugs if everything goes smoothly. It is sometimes thought that tests have bigger value than the code they cover has.

Javascript minimal automaton

Core function accepts signal string, on execution iterates over transitions, on match of signal with transition moves state from current state to target state based on transition and executes new state's function. That's all. Please read 5 times. This is the core business of finite state automaton. Below is minimal implementation of this function.



```
var nodes = [
        {"name":"a","value":"A","fun":"alert"},
        {"name":"b","value":"B","fun":"alert"},
    ];
    var connections = [
        {"name":"switch","from":"a","to":"b"},
        {"name":"switch","from":"b","to":"a"},
    ];
    var functions = [
        {"name":"alert","xec":function(p1){alert(p1);}}
    ];
    var getFunByName = function(name) {
        for (var ff in functions) {
            if (functions[ff]["name"]===name) {
                return functions[ff]["xec"];
            }
        }
        return function(){};
    };
    var getNodeValueByNodeName = function(nodeName) {
        for (var nd in nodes) {
            if (nodes[nd]["name"]===nodeName) {
                return nodes[nd]["value"];
            }
        }
        return "";
    }

    /**
     * minimal main Finite State Machine function
     ***/
    var doAction = function(signal){
        var conRef = null;
        for (var con in connections) {
            var connection = connections[con];
            if (connection["name"] === signal && state["node"] === connection["from"]) {
                conRef = connection;
                break;
            }
        }
        var transitionFound = false;
        for (var nod in nodes) {
            var node = nodes[nod];
            if (state["node"] === node["name"] && node["name"] === conRef["from"] && signal ===
conRef["name"]) {
                state["node"] = conRef["to"];

                var fuun = getFunByName(node["fun"]);
                transitionFound = true;
                break;
            }
        }
        if (transitionFound){
            state["value"] = getNodeValueByNodeName(state["node"]);
            fuun(state["value"]);
        }
    };

    var state = {"node":"a","value":"A"};
    alert(JSON.stringify(state));
    doAction("switch");
    alert(JSON.stringify(state));
    doAction("switch");
    alert(JSON.stringify(state));
```

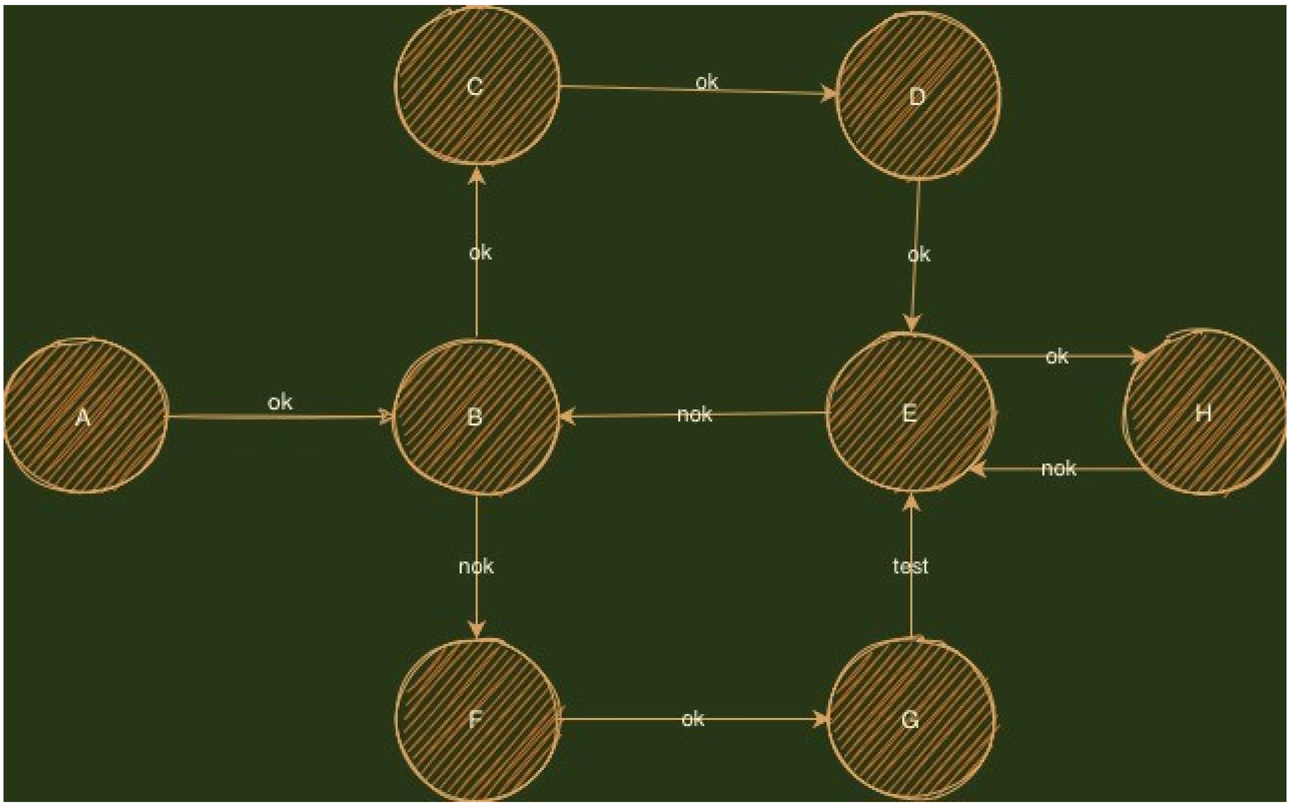
In this example initial state is A and the transition “switch” sets the state to B with function call (alert). Next call to function doAction sets the state back to A and calls function for node A. This is a very small automaton and you may imagine for example the bulb and the button as real life counterpart. State A is turned off, state B is turned on and the signal from button is “switch”.

Paste your code to .html file with this content:

```
<html><body><script>
/*paste here*/
</script></body></html>
```

… and open in browser, see 5 alerts (last five rows)

Larger machine



```
var globalState = {"x":1};
var nodes = [
        {"name":"a","value":"A","fun":"console"},
        {"name":"b","value":"B","fun":"console"},
        {"name":"c","value":"C","fun":"prompt"},
        {"name":"d","value":"D","fun":"console"},
        {"name":"e","value":"E","fun":"alert"},
        {"name":"f","value":"F","fun":"alert"},
        {"name":"g","value":"G","fun":"prompt"},
        {"name":"h","value":"H","fun":"report"}
    ];
    var connections = [
        {"name":"ok","from":"a","to":"b"},
        {"name":"ok","from":"b","to":"c"},
        {"name":"ok","from":"c","to":"d"},
        {"name":"ok","from":"d","to":"e"},
        {"name":"nok","from":"b","to":"f"},
        {"name":"ok","from":"f","to":"g"},
        {"name":"test","from":"g","to":"e"},
        {"name":"nok","from":"e","to":"b"},
        {"name":"ok","from":"e","to":"h"},
        {"name":"nok","from":"h","to":"e"},
    ];
    var functions = [
        {"name":"alert","xec":function(p1){alert(p1);}},
        {"name":"console","xec":function(p1){console.log(p1);}},
        {"name":"prompt","xec":function(p1){globalState["x"] = prompt("Enter result", "result"); }},
        {"name":"report","xec":function(p1){alert(“Machine state: ”+globalState["x"]);}}
    ];
```

Now we extend our FSM definition. Imagine this FSM is a product processing multi-industry-line setup where stations check the product and halt/pass/redirect the product. Graph now contains 8 nodes, there are 10 transitions between them, 3 signals exist and 4 functions are called upon arrival to some state. I will not go into detail, but let me ask a very simple question: how would you test the function, that is called upon arrival of FSM to node H from node E taking into consideration all previous states that the FSM could enter on its journey to node H systematically? Is it really necessary to create all combination of all walkthroughs through the graph? How about isolated set of tests that matter? In this example we need just tests for C, G and H to cover all possibilities for state H. Traditional FSM implementation doesn’t provide all necessary tools…

First hack

Let’s add a first callback to main automaton function.

```
    /**
    * main Finite State Machine function #2
    * accepts:
    * signal (mandatory)
    * callback1 (non-mandatory function, that accepts signal and returns {virtualNode:"x", virtualSig-
nal:"y",bypassExecution:true/false})
     ***/
    var doAction = function(signal, callback1){


            var bypassExecution = false;
            if (typeof callback1 === "function") {
                    var callback1result = callback1(signal);
                    if (typeof callback1result["virtualNode"] !== "undefined") {
                            state["node"] = callback1result["virtualNode"];
                    }
                    if (typeof callback1result["virtualSignal"] !== "undefined") {
                            signal = callback1result["virtualSignal"];
                    }
                    if (typeof callback1result["bypassExecution"] !== "undefined" && callback1result["bypas-
sExecution"] === true) {
                            bypassExecution = true;
                    }
            }
        var conRef = null;
        for (var con in connections) {
           var connection = connections[con];
           if (connection["name"] === signal && state["node"] === connection["from"]) {
              conRef = connection;
              break;
           }
        }
        var transitionFound = false;
        for (var nod in nodes) {
           var node = nodes[nod];
           if (state["node"] === node["name"] && node["name"] === conRef["from"] && signal ===
conRef["name"]) {
              state["node"] = conRef["to"];

              var fuun = getFunByName(node["fun"]);
```

```
                transitionFound = true;
                break;
            }
        }
        if (transitionFound){
            state["value"] = getNodeValueByNodeName(state["node"]);
            if(!bypassExecution) {
                    fuun(state["value"]);
            }

        }
    };
```

Now the automaton function without callback function works exactly the same as previous implementation, but when we add a callback function, we can modify the starting node and signal. Example usage:

```
    var globalState = {x:1};
    var state = {"node":"a","value":"A"};
    alert(JSON.stringify(state));
    //normal execution of automaton ends here
    doAction("nok",function(signal){return{"virtualNode":"e","virtualSignal":"ok"}});
    alert(JSON.stringify(state));
```

In this example we skip the whole finite state machine and position ourselves to node "e", signaling "ok", which will bring us to the tested node "h" directly. Let’s expand and test everything that matters for globalState["x"]:

```
    var globalState = {x:1};
    var state = {"node":"a","value":"A"};
    alert(JSON.stringify(state));
    //normal execution of automaton ends here
    doAction("nok",function(signal){return{"virtualNode":"b","virtualSignal":"ok"}});
    alert(JSON.stringify(state));
    doAction("nok",function(signal){return{"virtualNode":"e","virtualSignal":"ok"}});
    alert(JSON.stringify(state));
```

...and...

```
    var globalState = {x:1};
    var state = {"node":"a","value":"A"};
    alert(JSON.stringify(state));
    //normal execution of automaton ends here
    doAction("nok",function(signal){return{"virtualNode":"f","virtualSignal":"ok"}});
    alert(JSON.stringify(state));
    doAction("nok",function(signal){return{"virtualNode":"e","virtualSignal":"ok"}});
    alert(JSON.stringify(state));
```

These two test artificially set starting node to "b" or "e" and enter the tested prompt node from these nodes and then artificially enter the last tested node. Please note we didn’t walk through the whole FSM and still we tested everything. These tests are just for demonstration, prompt and seeing value populated via this prompt is a really extremely simple test scenario...

Full hacky code

In previous example, we run callback function BEFORE the FSM takes it’s action. Let’s add a callback, that gets run AFTER the FSM enters node and runs it’s action.

```
  /**
  * main Finite State Machine function #3
  * accepts:
  * signal (mandatory)
  * callback1 (non-mandatory function, that accepts signal and returns {virtualNode:"x", virtualSig-
nal:"y",bypassExecution:true/false})
  * callback2 (non-mandatory function, that accepts node after execution and signal that led to it’s state)
   ***/
  var doAction = function(signal, callback1, callback2){


        var bypassExecution = false;
        if (typeof callback1 === "function") {
                var callback1result = callback1(signal);
                if (typeof callback1result["virtualNode"] !== "undefined") {
                        state["node"] = callback1result["virtualNode"];
                }
                if (typeof callback1result["virtualSignal"] !== "undefined") {
                        signal = callback1result["virtualSignal"];
                }
                if (typeof callback1result["bypassExecution"] !== "undefined" && callback1result["bypas-
sExecution"] === true) {
                        bypassExecution = true;
                }
        }
     var conRef = null;
     for (var con in connections) {
        var connection = connections[con];
        if (connection["name"] === signal && state["node"] === connection["from"]) {
           conRef = connection;
           break;
        }
     }
     var transitionFound = false;
     for (var nod in nodes) {
        var node = nodes[nod];
        if (state["node"] === node["name"] && node["name"] === conRef["from"] && signal ===
conRef["name"]) {
           state["node"] = conRef["to"];

           var fuun = getFunByName(node["fun"]);
           transitionFound = true;
           break;
        }
     }
     if (transitionFound){
        state["value"] = getNodeValueByNodeName(state["node"]);
         if(!bypassExecution) {
               fuun(state["value"]);
         }
```

```
    }
        if (typeof callback2 === "function") {
            callback2(state["node"],signal);
        }
    };
```

Now we have a very simple framework for reverting tests we created to test our FSM.

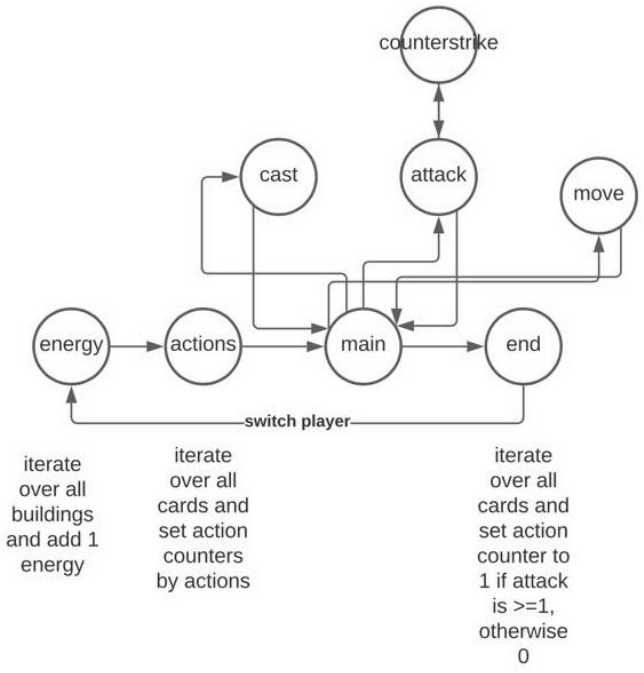
From small machine to the monster using TDD

Two primitive assertions exist in FSM: the node exists and the transition/connection exists. Using our extended action function, we can not test the fact that node exists and we can not easily test the transition exists. To test the node and transition existence we use two helper functions.

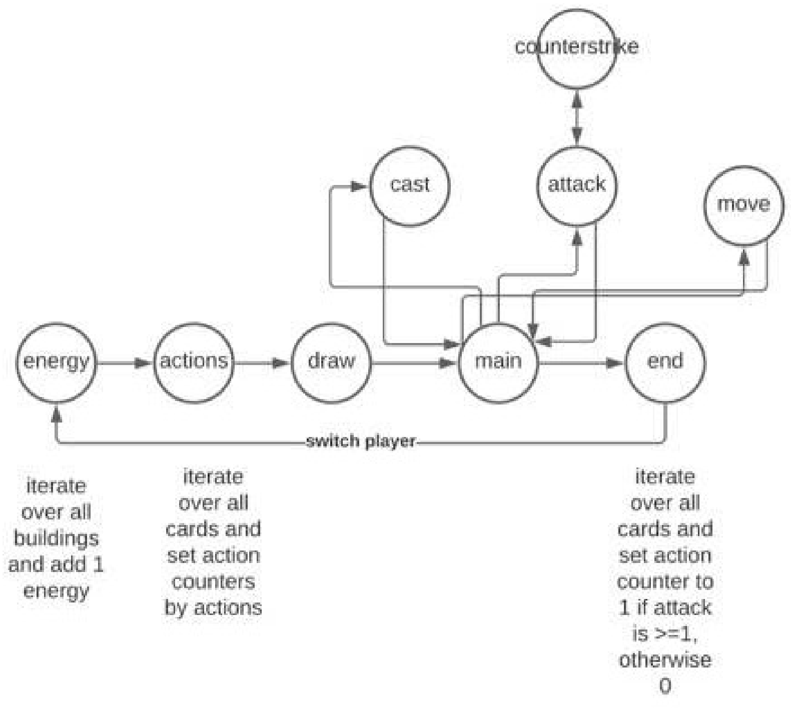
```
var getNodeByNodeName = function(nodeName) {
        for (var node in nodes) {
                if (nodes[node]["name"]===nodeName) {
                        return nodes[node];
                }
        }
        return false;
}

var getTransitionByFromTo = function(from, to) {
        for (var transition in transitions) {
                if (transitions[transition]["from"]===from && transitions[transition]["to"]===to ) {
                        return transitions[transition];
                }
        }
        return false;
}
```

We will demonstrate one step in the machine, that is already developed to some advanced state. The original state is this:


counterstrike
cast
attack
move
energy
actions
main
end
switch player
iterate over all buildings and add 1 energy
iterate over all cards and set action counters by actions
iterate over all cards and set action counter to 1 if attack is >=1, otherwise 0

The new state is about adding draw phase to the game.



counterstrike
cast
attack
move
energy
actions
draw
main
end
switch player
iterate over all buildings and add 1 energy
iterate over all cards and set action counters by actions
iterate over all cards and set action counter to 1 if attack is >=1, otherwise 0

First, we have to cancel the transition between actions and main, we'll write a test for it. Then we add node draw, add transition between actions and draw, add transition between draw and main and test transition from actions to draw. Tests first, implementation later. So let's start. I don't use any assertion framework to keep things VERY simple.

First, we can see some global state:
var globalState = {"cards":3};

First test driven step cancels transition from actions to main:

```
function test1(){
        console.log("actions→main transition should not exist")
        if (getTransitionByFromTo("actions","main")) {
                console.log("failed");
        } else {
                console.log("passed");
        }
}
```

Second test driven step establishes node draw:

```
function test2(){
        console.log("node draw should exist");
        if (getNodeByName("draw")) {
                console.log("passed");
        } else {
                console.log("failed");
        }
}
```

Third and fourth test driven step establish transitions from actions to draw and from draw to main:

```
function test3(){
	console.log("actions→draw transition should exist");
	if (getTransitionByFromTo("actions","draw")) {
		console.log("passed");
	} else {
		console.log("failed");
	}
}
function test4(){
	console.log("draw→main transition should exist");
	if (getTransitionByFromTo("draw","main")) {
		console.log("passed");
	} else {
		console.log("failed");
	}
}
```

Now the interesting part: let’s say we have globalState number of cards 3. Upon entering of node draw we increment (we keep things very simple). We want to test that:

```
function test5(){
	console.log("draw should increment number of cards in hand");
	//set starting node of action to actions, send ok signal
	var startingCards = globalState["cards"];
	doAction("ok",function(signal){
		return{"virtualNode":"actions","virtualSignal":"ok"}
	});
	if (globalState["cards"] === (startingCards +1)) {
		console.log("passed");
	} else {
		console.log("failed");
	}
	//revert action
	doAction("ok",function(signal){
		return{"virtualNode":"actions","virtualSignal":"ok","bypassExecution":true
	}}, function(endNode,signal){
		--globalState["cards"];
	});

}
```

Now when we run those 5 tests, they all fail. So one after the other we implement what is required for those tests to pass and we have a well tested and in a way documented upgrade of the project. I’ll leave playing with these things as example and will not provide full code of the game.

What is submachine

Submachine is a finite state machine, that starts on entering of some node for which it is defined. We will demonstrate how to handle this architecture later.

Full concept overview

We have created some machines and skipped a lot of functionality to test just what’s common to states spread across the FSM. We demonstrated how to use small TDD steps to grow a big system.

Conclusion

With callbacks, we may easily create tests for any automaton by functionality that’s spread across any FSM states, skipping states, that don’t matter.

References

Unfortunately I’m not able to find any reference from which I would take info. This all was written from memory gathered for several years something like 5-20 years ago.

Document version: 1.1.2

©7/2021 Čeněk Svoboda, svobo.c@gmail.com