Embedded Systems Course Project Report

Hierarchical Finite State Machine (HFSM)

Team :
Gautam Kumar (B19EE031)
Guvvala Sujitha (B19EE033)

BASIC TERMINOLOGY:

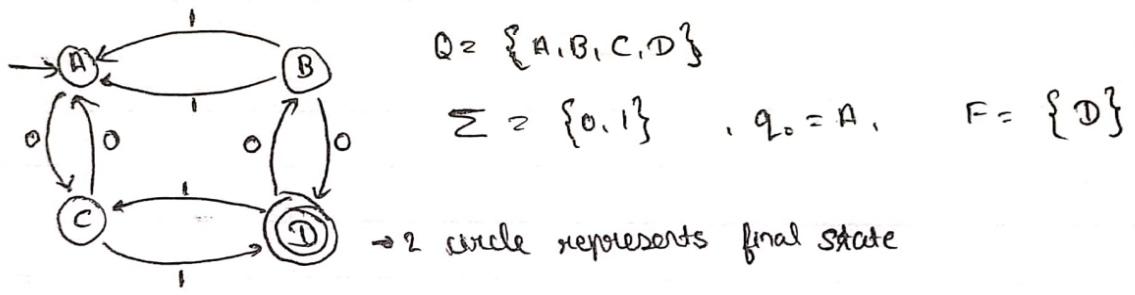
Actions :
An action is executed upon entering or exiting a state. Actions are the mechanism by which a finite state machine (FSM) gets work done.

Guards :
A guard constrains transitions by allowing transitions to occur only under specific circumstances. A good example of how guards are useful can be found in the security domain. In this domain a guard can be implemented that only allows a transition to occur if the user has previously authenticated with the system.

States :
A state is considered to be the value of all the variables in an FSM at some point in time. All the states of interest as well as the initial state must be declared while initializing the FSM in order for it to build it's internal transition table.

Transitions :
A transition is the action of moving an FSM from an initial state to a desired state. All the possible transitions must be declared while initializing the FSM in order to constrain how states are traversed.

APPLICATION-1: Implementation of a simple FSM
We have implemented a simple machine with 4 states: A, B, C, D. We have A as the starting state and D is the terminating state. We have state transitions based on the provided input, as shown in the figure.



Here Q is set of all all states, ∑ = inputs, q0 = initial state, F = set of final states.

APPLICATION-2: 2D player movement

This programme displays deterministic state machines, which implies that if you are in a specific state and a specific input is provided, you will be able to predict which transition state will occur and which active state you will end up in. Move, jump, and idle are the states performed.
Each state of FSM has 3 main entry points , they are :

Enter() —> Update() —> Exit()
(Initialisation) (Inputs) (clean up)

This application was created in Unity Hub, a game development platform for 2D and 3D games. The code is developed in the C# programming language. This application comprises a two-dimensional square and a ground floor (also a two-dimensional square) that collides with the square. The base.cs code mentioned in the github repository contains all of the application's basestates. We have declared them merely to note that they are present in this application, and the actual implementation was done in child classes (in sm.cs), so the states written in base.cs script will not do all of the behavior defining .

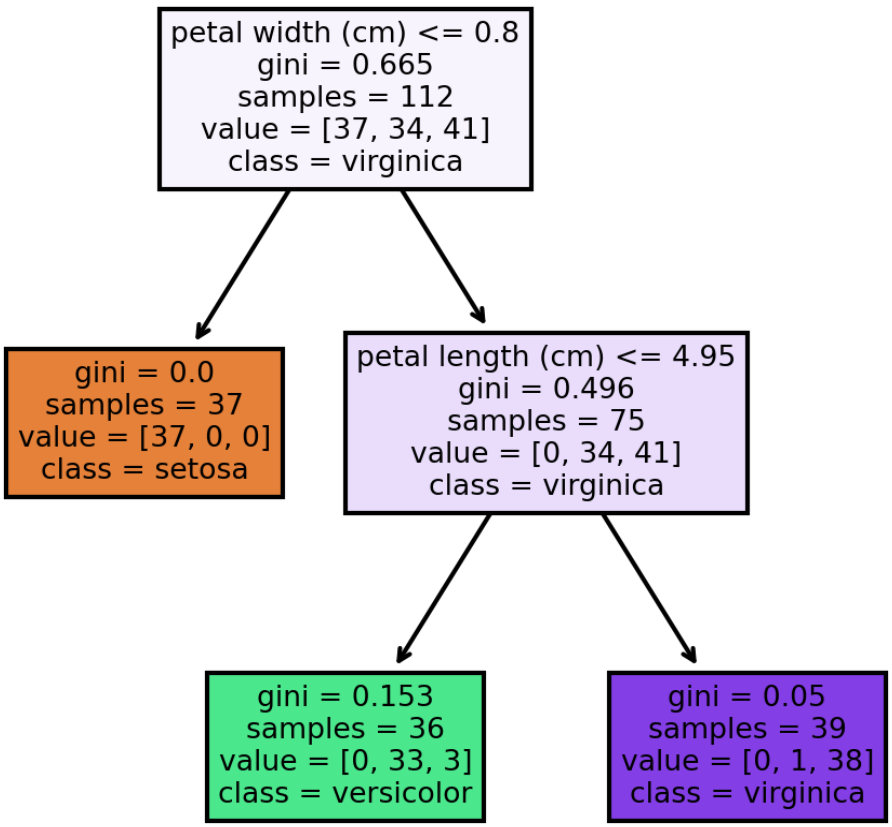
The move_sm.cs has all the states that are idle, move and jump which inherits from sm.cs scripts. Idle.cs is a script for the idle state that is fully integrated with move_sm.cs. A moving state was also established in the same way. If the input is not equal to zero, the transition from idle to movement will occur. As we travel from one state to the next, the color of the square changes as well. The property Sprite Renderer, which is present in UnityHub, was used to make this color change.

By converting this basic FSM into a hierarchical FSM, a jumping state was introduced. For this, a grounded state was designed that includes idle and moving states and can handle the

transition from grounded to jumping. When the spacebar is pressed, the state changes to jumping state.

APPLICATION-3: HFSM using decision tree

The hierarchical finite state machine (HFSM) not only divides your system into separate states, it puts them into a hierarchy of sub-states, which themselves can be state machines. We have used decision trees to implement one such application. A hierarchy is decisions are made at every node until we reach the root node. In this application, we have used decision trees to classify 3 different flowers (setosa, virginica and versicolor) by using their features (sepal length,sepal width, petal length, petal width).



Decision Tree representing hierarchical flow of decisions at different nodes.

APPLICATION-4 :

We used the run, move, and jump states in this application. We built the methods OnEnter(), OnUpdate(), and OnExit() to run these states (). We organized these states in a hierarchy after they were produced. Substates and transitions were used to accomplish this. The LoadSubState() method, which is described in the code, was used to add substates: For example, run state is a substate of move state by default.

The EnterStateMachine() method on the root state was used to start the State Machine. Hierarchically, the OnEnter() functions were invoked. The OnUpdate() method was also invoked in a hierarchical manner.The OnExit() methods are called from the bottom up, starting with the lowest current substate, as they are for ExitStateMachine(). The code's AddTransition() method was used to create transitions from one sub state to the next. A trigger must be communicated to the statemachine in order for the transitions to occur.

This trigger will act globally, regardless of the hierarchical level to which it is transmitted. The trigger will go from the root state to the current substates until it locates a consumer. Once a state has responded to the trigger, it will be consumed (states will be modified) and will not transmit further.

CONCLUSION

Hierarchy is a useful construct in enhancing the expressive power of finite state machines and facilitating the modeling of large systems. We summarized recent work on the theory of hierarchical finite state machines and related specification mechanisms (HMSC's). We discussed the effect of concurrency and nondeterminism, the classification of their expressive power, and the complexity of various algorithmic problems. The picture that emerged is rather comprehensive. One specific problem that remains open is the equivalence of deterministic HSM's. More broadly, we did not touch on the interaction with variables (extended HSM's). Also, as we mentioned more work remains to be done on the testing of hierarchical FSM's.

	Ordinary FSM's	Hierarchical FSM's
Reachability	$O(\|M\|)$	$O(\|M\|)$
Automata-checking	$O(\|M\| \cdot \|A\|)$	$O(\|M\| \cdot \|A\|^2)$
LTL model checking	$O(\|M\| \cdot 2^{\|\phi\|})$	$O(\|M\| \cdot 4^{\|\phi\|})$
CTL model checking	$O(\|M\| \cdot \|\phi\|)$	$O(\|M\| \cdot 2^{\|\phi\| d})$

Fig. 2. Summary of FSM and HSM model checking

Another issue concerns design problems, such as the structuring and modularization of the behavioral aspects of systems, to form a suitable hierarchical model. For example, suppose that we have a legacy system which we want to capture formally in a requirement model like uBET. We have available or can generate a large number of executions of the system to obtain a library of MSC scenarios. How do we go from this collection of MSC's to a more higher level, structured view of the system, in the form of a hierarchical use case graph built from basic MSC's?
Similarly, suppose that we have a test model of a system in the form of a large FSM; for example, we produced recently automatically such a model for a large Lucent enterprise switch from the test platform in use for the testing of the switch [EY00]; the FSM model has hundreds of thousands of states. The model can be used for targeted test generation to meet a variety of criteria.

Codes for all the applications are uploaded in the below github repository :
https://github.com/gautamHCSCV/HFSM-Python

(OUTPUTS FOR EACH APPLICATION ARE MENTIONED IN THE RESPECTIVE READ.ME FILES PRESENT IN THE GITHUB REPOSITORY)