Journal of Network Intelligence ©2021 ISSN 2414-8105 (Online)
Taiwan Ubiquitous Information Volume **6**, Number **2**, May **2021**

A Generalized Framework for FSM Implementation

Mao-Hsiung Hung*
Fujian Provincial Key Laboratory of Big Data Mining and Applications, Fujian University of Technology
School of Computer Science and Mathematics, Fujian University of Technology
no. 33, Xuefunan Road, University Town, Minhou, Fuzhou, 350118, China
*Corresponding author: mhhun0502@qq.com

Chaur-Heh Hsieh
College of Artificial Intelligence, Yango University, Fuzhou, 350015, China
chaoho1204@qq.com

Jinshui Wang
Research School of Computer Science, Australian National University, ACT, 2601, Australia
ymkscom@gmail.com

Received January 2021; revised April 2021

Abstract. *Finite State Machine (FSM) is a useful and powerful tool to model a dynamic system. A traditional implementation of FSM using nested switch-case statement exists problems of poor reusability and maintainability for programming. This paper presents a generalized framework of algorithmic implementation to improve these disadvantages of nested switch-case statement. A lookup table and an event handle function are used to construct various FSMs in the proposed framework. In the experiments, we apply the proposed framework to implement unconditional/conditional FSMs and hierarchical/concurrent-hierarchical FSMs. The experimental results demonstrate the framework's good generalization.*
Keywords: Finite State Machines, FSM Implementation, Generalized Framework

1. **Introduction.** Finite State Machine (FSM) provides a useful and powerful tool to model a dynamic system. Particularly, FSMs can effectively and efficiently describe complex logics of systems, so that engineers and programmers apply FSM to avoid heuristic design for complex systems, for examples of network protocols [1]-[3] and control systems [4]. Therefore, FSMs have receiving a largely wide applications and developments in many engineering and scientific domains.

A traditional and simple programming implementation uses nested switch-case statement to construct FSMs [5]. However, the implementation framework of nested switch-case statement has poor reusability and its codes are required to maintain because of the simple structure. [6]-[9] applied an object-oriented programming to encapsulate the nested switch-case statement and enhance FSM's code structure, so that the reusability and maintainability of FSM implementation framework is obtained improvement. However, the core part of the nested switch-case statement is still needed rewriting to adapt various FSMs in the object-oriented framework. Several works developed drawing and visualization tools to plot and describe state transition diagrams of FSMs such as Finite State Machine Editor [10] and then these tools can automatically generate FSM codes. However, when the modification of the state diagrams are demanded, these FSM codes are still again generated and deployed to the implementation framework.

In this work, we developed an implementation framework for FSMs, featured by high generalization. A lookup table is applied to represent state transitions of a FSM in our proposed framework. Cooperating with the lookup table, we propose event handle algorithms to perform state transition, output action and condition examination of FSMs. The proposed implementation framework achieves that the configuration of the lookup table is only required to construct various FSMs without changing the core codes. Moreover, our proposed framework can be extended to apply hierarchical FSMs. In the experiments, we implemented unconditional/conditional FSMs and hierarchical/concurrent-hierarchical FSMs to demonstrate that the core part can keep unchanged between these FSMs' implementations.

The remainder of this paper is organized as follows. Section 2 reviews a FSM implemented by nested switch-case statements. Section 3 describes the proposed algorithm. Section 4 demonstrates and discusses experimental results of the proposed methods. The conclusions are drawn in Section 5.

Algorithm I: Event handle of FSM by nested switch-case statement

```
1   Input: an event of e
2   function EventHandle(e)
3   switch(curStateFsm)
4   case S1: switch(e)
5                case I1: ... ns ←Sx
6                case I2: ... ns ←Sx
7                ⋮
8                case In: ... ns ←Sx
9            end switch
10  case S2: switch(e)
11               case I1: ... ns ←Sx
12               case I2: ... ns ←Sx
13               ⋮
14               case In: ... ns ←Sx
15           end switch
16  ⋮
17  case Sn: switch(e)
18               case I1: ... ns ←Sx
19               case I2: ... ns ←Sx
20               ⋮
21               case In: ... ns ←Sx
22           end switch
23  end switch
24  curStateFsm ← ns
```

2. **Related Work.** Nested switch-case statement is one of the most common methods to implement FSM. Algorithm I lists pseudo codes of event handle of nested switch-case statement, where EventHandle(e) function processes a transition caused by an event of e. The current state of a FSM is pre-stored in a variable of curStateFsm before the function call. The statement of switch($curStateFsm$) in Line 3 selects one of S1, S2,..., and Sn branches according to $curStateFsm$'s value. Then, following S1, S2,..., or Sn cases, the selected switch(e) statement chooses one of branches of I1, I2,..., and In according to the event. Then, the statement of ns ←Sx performs to assign the next state of Sx. Finally, the statement of $curStateFsm \leftarrow ns$ completes Sx assignment to $curStateFsm$.

Using the nested switch-case structure, we can construct a various of FSMs. However, because the case values of the switch statements have to be programmed into constants, the most disadvantage of the nested switch-case structure is that its program codes are required to modify for every FSM's changes. As a result, the programming codes of the nested switch-case structure become very poor in reusability and generalization. Therefore, how to parameterize the case values of the switch statements is the key issue to design a universal program for FSMs.

3. **Proposed Method.**

3.1. **FSM without and with condition.** To represent a state transition diagram of a FSM, we apply a lookup table to store the configuration of all state transitions of the FSM. Each row of the lookup table is used to represent a state transition. We designed a data structure in a row, which contains five fields to record a transition. The five fields are denoted by fromState, event, toState, act and comp. The fromState field records the starting state of a transition. The event field records a specific event to drive the state transition. The toState field records the ending state of the transition. The act field stores a runner of an action which is performed once the state transition happens. The comp filed stores a comparator to judge whether a specific condition required by the state transition. The conditional state transition is also regarded an extended FSM, EFSM [11].

FIGURE 1. (a) Single transition without condition (b) Single transition with condition

The first four fields are enough to deal with a state transition without condition. If a state transition executes with a condition, then it needs the five fields to represent. Fig.1(a) shows a single state transition from a state of X to a state of Y driven by a event of I to perform a action of O with no condition. Fig.1(b) shows the single state transition of Fig.1(a) but with a condition of C. Table 1 is a lookup table of Tab and its rows of Tab[0], Tab[1],..., Tab[$N-1$] are used to represent N state transitions of a FSM.

TABLE 1. Data structure of lookup table

	fromState	event	toState	act	comp
Tab[0]	...	...	...	...	...
Tab[1]	...	...	...	...	...
⋮					
Tab[$N-1$]	...	...	...	...	...

After the construction of the lookup table of Tab, we propose the event handle of the FSM to process the state transition caused by event input. We separate into two algorithms to describe the two event handles without condition and with condition. Algorithm II is proposed to process the event handle without condition which is written by a function

of EventHandle(e) inputting an event of e.

Algorithm II: Event handle of FSM without condition

```
1   Input: an event of e
2   function EventHandle(e)
3   cs ← curStateFsm, ns ← cs, act ←null, flag ←false
4   for i ←0 to N − 1 do
5      if cs =Tab[i].fromState and e =Tab[i].event then
6         ns ←Tab[i].toState, act ←Tab[i].act, flag ←true
7         exit for-loop
8      end if
9   end for
10
11  if flag then
12     if act ≠null
13        then call act.run()
14        else do nothing
15     end if
16     curStateFsm ← ns
17  else
18     do nothing
19  end if
```

In the algorithm, we first define the variable of $curStateFsm$ to store the current state of the FSM. The beginning of the FSM execution initializes $curStateFsm$ variable to one of state. Then, one of events happens to trig a state transition and an action performance. In Line 3, the four local variables of cs, ns, act and $flag$ are initialized. cs and ns are assigned by $curStateFsm$ and act is assigned by null. The null value makes the variable not to refer any object. flag is assigned by false that means not yet matching for the current state and the event input before the table is looked up.

After the variable initialization, we travel all rows of the lookup table of Tab using a for-loop in Line 4-9. And then, we match cs and e with Tab[i].fromState and Tab[i].event for i=0, 1,..., $N-1$ in Line 5. In Line 6, once the match of cs and e hits, ns is assigned by Tab[i].toState and act is assigned by Tab[i].act. As a result, the next state of the trigged state transition is ready in ns and act refers to the corresponding action runner. The flag changes to true that it means a successful matching for the current state and the event input, and then we exits the for-loop. If cs and e have no matching with Tab[0], Tab[1],...Tab[$N-1$], $flag$ will keep false.

After the matching of cs and e, we check whether $flag$ is true in Line 11. If yes, then we check whether $act \neq null$. If $act \neq null$, then an action is required to execute in the state transition, so we call act.run(), otherwise nothing performs, as the described in Line 12-15. Then in Line 16, $curStateFsm$ is assigned by ns i.e. the next state, and the state transition finishes. If $flag$ is not true, that means any state transition and any action will not perform.

To process the event handle with a condition, we propose Algorithm III which is written by a function of EventHandle(e,c) inputting an event of e and a condition of c. Algorithm III is the partially same as Algorithm II. The different part between two algorithms locates in Line 6-11. The for-loop iteratively compares cs and e respectively with Tab[i].fromState and Tab[i].event. When the match of cs and e hits in the for-loop, a variable of cp is assigned by Tab[i].comp, a condition comparator of the state transition. The condition

comparator is used to check whether the inputting condition is equals to a specific condition.

```
Algorithm III: Event handle of FSM with condition
 1  Input: an event of e and a condition of c
 2  function EventHandle(e,c)
 3  cs ← curStateFsm, ns ← cs, act ←null, flag ←false
 4  for i ←0 to N − 1 do
 5     if cs =Tab[i].fromState and e =Tab[i].event then
 6        cp ←Tab[i].comp
 7        if cp =null then
 8           ns ←Tab[i].toState, act ←Tab[i].act, flag ←true
 9        elseif cp.equal(c) then
10           ns ←Tab[i].toState, act ←Tab[i].act, flag ←true
11        end if
12        exit for-loop
13     end if
14  end for
15
16  if flag then
17     if act ≠ null
18        then call act.run()
19        else do nothing
20     end if
21     curStateFsm ← ns
22  else
23     do nothing
24  end if
```

Then, we check whether *cp* = null. If yes, that means the selected transition with no condition. Then, we assign *ns*, *act* and *flag* respectively to Tab[*i*].toState, Tab[*i*].act and true in Line 8. If *cp* is not equal to null, that means a specific condition is required to perform the selected transition. Therefore, we call a function of *cp*.equal(*c*) to check whether *c* is equals to a specific condition. If yes, then we assign *ns*, *act* and *flag* respectively to Tab[*i*].toState, Tab[*i*].act and true in Line 10, otherwise *flag* keeps false. As a result, the selected state transition and its corresponding action are able to perform according to a specific set of condition and event.

3.2. **Hierarchical FSM.** Hierarchical structures are common used to represent the relationships between main FSMs and their sub FSMs in FSM designs. A sub FSMs is contained in one of states of a main FSM, which is called by superstate. The superstate becomes the entrance and exit between the main FSM and the sub FSM. As shown in Fig.2, a main FSM transfers the current state from W to X and its sub FSM is contained in X superstate. When the machine entries X superstate, and we need to enable the sub FSM and then perform an entrance action of O1. By contrary, when the machine exits the X superstate, and we need to perform an exit action of O2 and then disable the sub FSM. Moreover, once the sub FSM enables, its current state is assigned to an initial state of Y.

To join entry/exit actions of the superstate and enable/disable of the sub FSM into our proposed method, we add four rows our proposed lookup, including entryNs, exitCs, enSub and deSub, as shown in Table 2. The entryNs field is used to store an action object

for the calling during an entrance of toState, which is specified by a state transition. The exitCs field also stores an action object for an exit of a fromState. The enSub and deSub fields are used to set whether the sub FSM enables or disables.

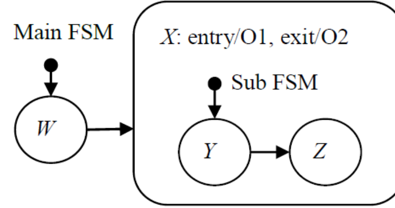

FIGURE 2. An example of hierarchical FSM

TABLE 2. Lockup table for hierarchical FSM

	fromState	event	toState	act	comp	entryNs	exitCs	enSub	deSub
Tab[0]	...	...	...	...	...	...	...	...	...
Tab[1]	...	...	...	...	...	...	...	...	...
⋮									
Tab[$N-1$]	...	...	...	...	...	...	...	...	...

Algorithm IV: Event handle of hierarchical FSM

1 Input: an event of e
2 **function** EventHandle(e)
3 $cs \leftarrow curStateFsm$, $ns \leftarrow cs$, $act \leftarrow$null, $flag \leftarrow$false
4 $actEntryNs \leftarrow$null, $actExitCs \leftarrow$null, $enSub \leftarrow$false, $deSub \leftarrow$false
5 **for** $i \leftarrow 0$ to $N-1$ **do**
6   **if** $cs =$Tab[i].fromState **and** $e =$Tab[i].event **then**
7     $ns \leftarrow$Tab[i].toState, $act \leftarrow$Tab[i].act, $flag \leftarrow$true
8     $actEntryNs \leftarrow$ Tab[i].entryNs, $actExitCs \leftarrow$ Tab[i].entryCs
9     $enSub \leftarrow$Tab[i].enSub, $deSub \leftarrow$Tab[i].deSub
10   **exit** for-loop
11   **end if**
12 **end for**
13
14 **if** $flag$ **then**
15   **if** $deSub$ **then call** $subFsm$.disable()
16   **if** $actExitCs \neq$null **then call** $actExitCs$.run()
17   **if** $act \neq$null **then call** act.run() **else do** nothing
18   **if** $enSub$ **then call** $subFsm$.enable()
19   **if** $actEntryNs \neq$null **then call** $actEntryNs$.run()
20   $curStateFsm \leftarrow ns$
21 **else**
22   **do** nothing
23 **end if**

After the extension of the lookup table, we modify the proposed event handle from Algorithm II to Algorithm IV for hierarchical FSM. In Algorithm IV, four variables of $actEntryNs$, $actExitNs$, $enSub$ and $deSub$ are initialized in Line 4. When the current state and the input event match fromState and event fields in a row of the lookup tables, the four variables load value from the responding fields in the row, as written in Line 8-9. If $flag$ variable is true, before an action object starts to run, we call a disable function of the sub-FSM object ($subFsm$) depending on $deSub$ value and the exit action for fromState starts to run if any, as written in Line 15-16. Similarly, after an action object starts to run, the calling of an enable function of $subFsm$ and the calling the entry action's run function for toState perform, as written in Line 18-19. The $subFsm$ variable is used to store a sub-FSM object belonging to a main FSM and the $subFsm$ configuration is needed to be done during the main FSM's creation. In addition, $subFsm$ variable can be programmed into a list structure of FSM to store two and more sub-FSM objects.

4. **Experimental result.** In our proposed method, the lookup table structure and the event handle function are the core parts of FSM framework. The core part is also basically unchanged. We first define symbols of states, action and conditions, and then configure the lookup table to build several of FSMs, so that it makes the framework to achieve generalized purposes.

4.1. **Unconditional FSM.** To demonstrate the generalized ability, we applied our proposed framework to two typical FSMs. One is an FSM of elevator door controlling and it belongs to event handle without condition. The other one is an FSM of stack operation and it belongs to event handle with conditions. In the meantime, we inputted event and condition sequences to simulate of the two FSMs.

The FSM of elevator door controlling has four states of the door including opened, closing, closed and opening. The two events for door's button are "request to open" and "request to close". Another two events for door's sensor are "sensor closed" and "sensor opened". The three actions of the door are "move to close", "move to open", and "stop moving". For the convenience of representation, we define symbols before the construction of the state transitions of FSMs. Table 3 lists the meanings and symbols of elevator door controlling FSM. These symbols include states of S1, S2, S3 and S4, events of I1, I2, I3 and I4, and actions of O1, O2 and O3. Based on these symbols, the state transition diagram of the FSM is as shown in Fig.3.

Table 3. Meaning and symbol of FSM of elevator door controlling

	Meaning	Symbol
State	Door is opened	S1
	Door is closing	S2
	Door is closed	S3
	Door is opening	S4
Event	Request to close	I1
	Sensor closed	I2
	Request to open	I3
	Sensor opened	I4
Action	Door moves to close	O1
	Door moves to open	O2
	Door stops moving	O3

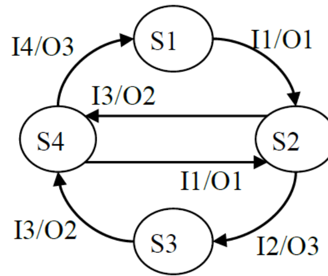

FIGURE 3. State transition diagram of elevator door controlling

TABLE 4. Configuration of Lookup table of FSM of elevator door controlling

	fromState	event	toState	act	comp
Tab[0]	S1	I1	S2	ActO1	null
Tab[1]	S2	I2	S3	ActO3	null
Tab[2]	S3	I3	S4	ActO2	null
Tab[3]	S4	I4	S1	ActO3	null
Tab[4]	S2	I3	S4	ActO2	null
Tab[5]	S4	I1	S2	ActO1	null

Listing 1: Simulation result of elevator door controlling FSM

Line	Output	Line	Output
1	Start to test an elevator door FSM.	12	I1(Request to close) is coming.
2	Current state is S1(Opened)	13	ActO1: Door moves to close.
3	I1(Request to close) is coming.	14	Current state is S2(Closing)
4	ActO1: Door moves to close.	15	I3(Request to open) is coming.
5	Current state is S2(Closing)	16	ActO2: Door moves to open.
6	I2(Sensor closed) is coming.	17	Current state is S4(Opening)
7	ActO3: Door stops moving.	18	I4(Sensor opened) is coming.
8	Current state is S3(Closed)	19	ActO3: Door stops moving.
9	I3(Request to open) is coming.	20	Current state is S1(Opened)
10	ActO2: Door moves to open.	21	Finish testing.
11	Current state is S4(Opening)		

According to the state transition diagram, we configure the lookup table as listed in Table 4. The six rows of Tab[0], Tab[1],..., Tab[5] represent the six transition in the diagram. For an example, the transition of I1/O1 from S1 to S2 is assigned by S1, I1, S2, ActO1 and null respectively in fromState, event, toState, act and comp fields of Tab[0]. In the act field, ActO1, ActO2 and ActO3 means action runners respectively corresponding to O1, O2 and O3. Each action runner contains a run() function to perform a specific operation. For an example, the action runner of ActO1 contains ActO1.run() function to perform O1 action. Due to the FSM of event handle without condition, we assigned comp fields to null in all rows.

During simulation phrase of the FSM, we first assigned the initial state to S1, and then inputted an event sequence of I1, I2, I3, I1, I3, I4 and tested the function of EventHandle(e) in one at a time way. The testing results displayed that the FSM transferred state by a sequence of S1, S2, S3, S4, S2, S4, S1 and performed a series of actions

of ActO1, ActO3, ActO2, ActO1, ActO2, ActO3, as shown in Listing 1.

4.2. **Conditional FSM.** The FSM of stack operation has three states of the stack including empty, full and "not empty and not full". The two events are "request to push into stack" and "request to pop from stack". The four actions of the stack operation are "push", "pop", "overflow" and "underflow". The four conditions are "only one space left in the stack", "only one element left in the stack" and their inverts. To implement the FSM of stack operation, we first define symbols and their meanings, as listed in Table 5. These symbols include states of S1, S2, and S3, events of I1 and I2, actions of O1, O2, O3 and O4 and conditions of Nc, C1, C2, C1b and C2b. For the convenience of simulation, we additionally define a condition symbol of Nc which means no condition. /C1 and /C2 are respectively inverts of C1 and C2. We rewrite /C1 and /C2 respectively to C1b and C2b for the convenience of expression. Based on these symbols, the state transition diagram of the FSM is as shown in Fig.4.

According to the state transition diagram, we set the lookup table as listed in Table 6. The eight rows of Tab[0], Tab[1],..., Tab[5] represent the eight transitions in the diagram. For an example, the transition of I1[C1]/O1 from S2 to S3 is assigned by S2, I1, S2, ActO1 and CompC1 respectively in fromState, event, toState, act and comp fields of Tab[0]. In the comp field, CompC1, CompC2, CompC1b and CompC2b mean condition comparators respectively corresponding to C1, C2, C1b and C2b. Each condition comparator contains an equal(c) function to perform a comparing operation which checks c whether equals to an expected condition. For an example, the condition comparator of CompC1 contains CompC1.equal(c) function to check c whether equals to C1 condition.

We first assigned an initial state of S1, input an event sequence of I1, I1, I1, I1, I2, I2, I2, I2 with a condition sequence of Nc, C1b, C1, Nc, Nc, C2b, C2, Nc and tested the function of EventHandle(e,c) with one pair of event and condition in a time. The testing result displayed that the FSM transferred state by a sequence of S1, S2, S2, S3, S3, S2, S2, S1, S1 and performed a series of actions of ActO1, ActO1, ActO1, ActO3, ActO2, ActO2, ActO2, ActO4, as shown in Listing 2.

Table 5. Meaning and symbol of FSM of stack algorithm

	Meaning	Symbol
State	Stack is empty	S1
	Stack is not empty and not full	S2
	Stack is full	S3
Event	Request to push into stack	I1
	Request to pop from stack	I2
Action	Push operation	O1
	Pop operation	O2
	Display overflow	O3
	Display underflow	O4
Condition	No condition	Nc
	Only one space left in the stack	C1
	Only one element left in the stack	C2
	Not only one space left in the stack	/C1(C1b)
	Not only one element left in the stack	/C2(C2b)

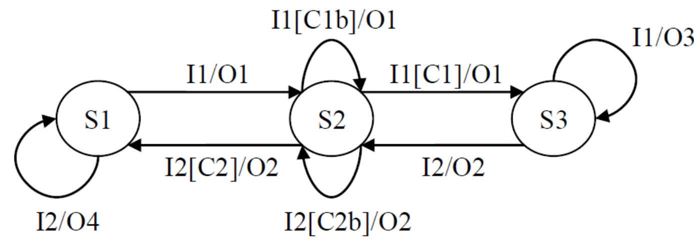

FIGURE 4. State transition diagram of stack operation

TABLE 6. Configuration of Lookup table of FSM of stack operation

	fromState	event	toState	act	comp
Tab[0]	S1	I1	S2	ActO1	null
Tab[1]	S2	I1	S2	ActO1	CompC1
Tab[2]	S3	I1	S3	ActO3	null
Tab[3]	S3	I2	S2	ActO2	null
Tab[4]	S2	I2	S1	ActO2	CompC2
Tab[5]	S1	I2	S1	ActO4	null
Tab[6]	S2	I1	S2	ActO4	CompC1b
Tab[7]	S2	I2	S2	ActO1	CompC2b

Listing 2: Simulation result of stack operation FSM

Line	Output	Line	Output
1	Start to test a stack FSM.	15	I2(Request to pop) is coming with Nc(No condition)
2	Current state is S1: Empty	16	ActO2: Pop operation
3	I1(Request to push) is coming with Nc(No condition)	17	Current state is S2: Not empty and not full
4	ActO1: Push operation	18	I2(Request to pop) is coming with C2b(Not only one element left)
5	Current state is S2: Not empty and not full	19	ActO2: Pop operation
6	I1(Request to push) is coming with C1b(Not only one space left)	20	Current state is S2: Not empty and not full
7	ActO1: Push operation	21	I2(Request to pop) is coming with C2(Only one element left)
8	Current state is S2: Not empty and not full	22	ActO2: Pop operation
9	I1(Request to push) is coming with C1(Only one space left)	23	Current state is S1: Empty
10	ActO1: Push operation	24	I2(Request to pop) is coming with Nc(No condition)
11	Current state is S3: Full	25	ActO4: Underflow
12	I1(Request to push) is coming with Nc(No condition)	26	Current state is S1: Empty
13	ActO3: Overflow	27	Finish testing.
14	Current state is S3: Full		

4.3. **Hierarchical FSM.** To demonstrate the application to hierarchical FSM using our method, we implemented an air-condition controller's FSM [12] based on the proposed lookup table. The FSM is design to control the actions of fan and condenser of an air-conditioner. Two buttons of power and AC mode trig events for the state transmissions in the FSM. The FSM is represented by a hierarchical structure of a main FSM and a sub FSM, as shown in Fig.5. The power button switches off state and running state each other in the main FSM, where the running state is a superstate which contains the sub FSM. In addition, the fan starts to work when the machine enters the running state. The AC button switches fan-only state and AC state each other in the sub FSM of AC mode. Meanwhile, the condenser starts and stops to work respectively in the entrance and exit

moments of AC states.

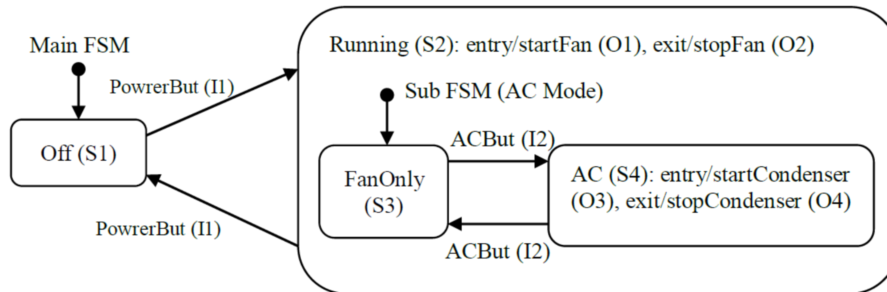

FIGURE 5. Hierarchical FSM diagram of air-condition controller

TABLE 7. Meaning and symbol of FSM of air-condition controller

	Meaning	Symbol
State	Air-conditioner is off	Off (S1)
	Air-conditioner is running	Running (S2)
	Only fan is on	FanOnly (S3)
	Condenser is on	AC (S4)
	Fan's speed is low	Low (S5)
	Fan's speed is high	High (S6)
Event	Press power button	PowrBut (I1)
	Press AC button	ACBut (I2)
	Press speed button	SpeedBut (I3)
Action	Turn on fan	startFan (O1)
	Turn off fan	stopFan (O2)
	Turn on condenser	startCondenser (O3)
	Turn off condenser	stopCondenserUp (O4)
	Speed up fan	speedUp (O5)
	Speed down fan	speedDown (O6)

Before the configuration of the lookup table for the FSM of air-condition controller, we need to define several symbols, as listed in Table 7. S1 and S2 are respectively off state and running state in the main FSM. S3 and S4 are respectively fan-only state and AC state in the sub FSM. I1 and I2 respectively represent the events caused by pressing power button and AC button. Two actions of turning-on and turning-off of fan are respectively O1 and O2. Turning-on and turning-off of condenser are respectively O3 and O4. Therefore, according to these symbols and the state transition diagram, we configured the two lookup tables for the main FSM and the sub FSM in the hierarchical structure, as listed in Table 8.

We input the event sequence of I1, I2, I2, I1, I1, I2, I1 to the hierarchical FSM of air-conditioner controller. As shown in Listing 3, the actions of startFan, startCondenser, stopCondenser, stopFan, startFan, startCondenser, stopCondenser and stopFan executed accordingly. In particular, when the I1 (powerbut) event is triggered during S4(AC) state, the stopCondenser action is needed to perform first, and then the stopfan action performs, as listed in Line 25-29. The stopCondenser action is programed in the disable() function of the sub FSM of ACMode. When the current state is demanded exiting from AC, our

proposed algorithm calls the sub FSM disable() and the stopCondenser action can executes reasonably before the entrance of the next state of S1(Off).

TABLE 8. Configuration of Lookup table of hierarchical FSM of air-condition controller

	fromState	event	toState	act	comp	entryNs	exitCs	enSub	deSub
Tab[0]	S1	I1	S2	null	null	O1	null	true	false
Tab[1]	S2	I1	S1	null	null	null	O2	false	true

(a) Main FSM

	fromState	event	toState	act	comp	entryNs	exitCs	enSub	deSub
Tab[0]	S3	I2	S4	null	null	O3	null	false	false
Tab[1]	S4	I2	S3	null	null	null	O4	false	false

(b) Sub FSM

Listing 3: Simulation result of hierarchical FSM of air-conditioner controller

Line	Output	Line	Output
1	Start to test a hierarchical Ac FSM	16	ActO2: stopFan
2	fsmMain's current state is S1(Off)	17	fsmMain's current state is S1(Off)
3	I1(PowerBut) is coming.	18	I1(PowerBut) is coming.
4	ActO1: startFan	19	ActO1: startFan
5	fsmMain's current state is S2(Running)	20	fsmMain's current state is S2(Running)
6	fsmSub's current state is S3(FanOnly)	21	fsmSub's current state is S3(FanOnly)
7	I2(ACBut) is coming.	22	I2(ACBut) is coming.
8	fsmMain's current state is S2(Running)	23	fsmMain's current state is S2(Running)
9	ActO3: startCondenser	24	ActO3: startCondenser
10	fsmSub's current state is S4(AC)	25	fsmSub's current state is S4(AC)
11	I2(ACBut) is coming.	26	I1(PowerBut) is coming.
12	fsmMain's current state is S2(Running)	27	fsmACMode's disable(): stopCondenser
13	ActO4: stopCondenser	28	ActO2: stopFan
14	fsmSub's current state is S3(FanOnly)	29	fsmMain's current state is S1(Off)
15	I1(PowerBut) is coming.	30	Finish testing

4.4. **Concurrent-hierarchical FSM.** We extended the above hierarchical FSM to a concurrent-hierarchical FSM, as shown in Fig.6. Two Sub FSMs of AC Mode and Speed are contained in a main FSM, and they can operate concurrently during the entrance of Running state. The Speed FSM is given to control fan's speed. To build Speed FSM, we added several symbols in Table 7, including Low(S5) and High(S6) states, SpeedBut(I3) event, and speedUp(O5) and speedDown(O6) actions. According to the diagram of the concurrent-hierarchical FSM, we defined three lookup tables to configure the main and two sub FSMs, as listed in Table 9.

We input the event sequence of I1, I2, I3, I3, I2, I1, I1, I2, I3, I1 to the concurrent-hierarchical FSM of air-conditioner controller. The simulation outputs for the event sequence are as listed in Listing 4. As a result, ACMode and Speed sub-FSMs can operate concurrently and independently. It is noted that we just apply the addition of new lookup tables to finish the implementation of the concurrent FSMs without the modification for our proposed event handle algorithm.

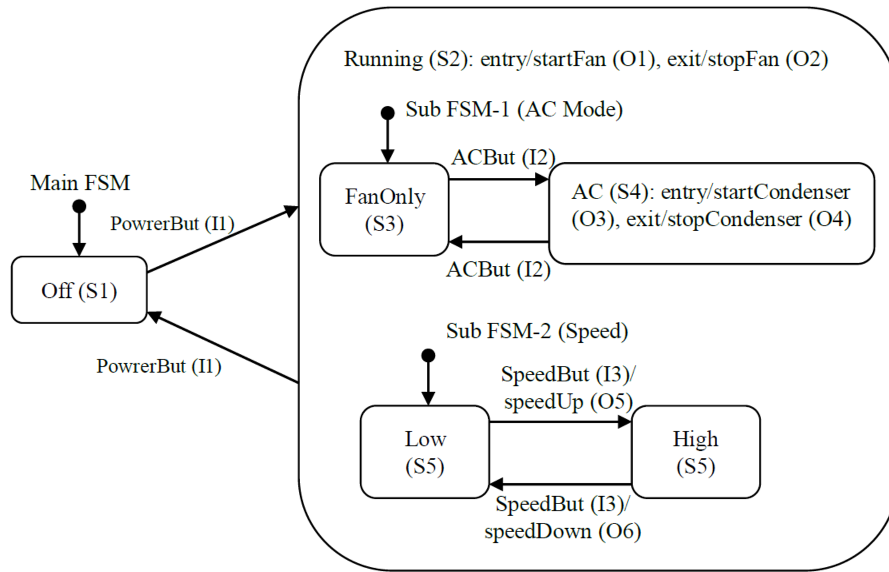

FIGURE 6. Hierarchical FSM diagram of air-condition controller

Listing 4: Simulation result of concurrent-hierarchical FSM of air-conditioner controller

Line	Output	Line	Output
1	Test a concourrent-hierachical Ac FSM	26	fsmSub1's current state is S3(FanOnly)
2	fsmMain's current state is S1:Off	27	fsmSub2's current state is S5(Low)
3	I1(PowerBut) is coming.	28	I1(PowerBut) is coming.
4	ActO1: startFan	29	ActO2: stopFan
5	fsmMain's current state is S2(Running)	30	fsmMain's current state is S1(Off)
6	fsmSub1's current state is S3(FanOnly)	31	I1(PowerBut) is coming.
7	fsmSub2's current state is S5(Low)	32	ActO1: startFan
8	I2(ACBut) is coming.	33	fsmMain's current state is S2(Running)
9	fsmMain's current state is S2(Running)	34	fsmSub1's current state is S3(FanOnly)
10	ActO3: startCondenser	35	fsmSub2's current state is S5(Low)
11	fsmSub1's current state is S4(AC)	36	I2(ACBut) is coming.
12	fsmSub2's current state is S5(Low)	37	fsmMain's current state is S2(Running)
13	I3(SpeedBut) is coming.	38	ActO3: startCondenser
14	fsmMain's current state is S2(Running)	39	fsmSub1's current state is S4(AC)
15	fsmSub1's current state is S4(AC)	40	fsmSub2's current state is S5(Low)
16	ActO5: speedUp	41	I3(SpeedBut) is coming.
17	fsmSub2's current state is S6(High)	42	fsmMain's current state is S2(Running)
18	I3(SpeedBut) is coming.	43	fsmSub1's current state is S4(AC)
19	fsmMain's current state is S2(Running)	44	ActO5: speedUp
20	fsmSub1's current state is S4(AC)	45	fsmSub2's current state is S6(High)
21	ActO6: speedDown	46	I1(PowerBut) is coming.
22	fsmSub2's current state is S5(Low)	47	FsmACMode's disable(): stopCondenser
23	I2(ACBut) is coming.	48	ActO2: stopFan
24	fsmMain's current state is S2(Running)	49	fsmMain's current state is S1(Off)
25	ActO4: stopCondenser	50	Finish testing

5. **Conclusions.** This paper has presented a generalized framework on algorithmic implementation for FSMs. Dou to poor generalization, the traditional implementations of nested switch-case statement is required to modification for different FSM applications.

Cooperating with different lookup tables, our proposed framework can be applied to various FSMs without changes. The experimental results indicate that good generalization is achieved by the proposed framework. Moreover, the proposed algorithmic implementation will become good reference on hardware design of FSMs [13].

Table 9. Configuration of Lookup table of concurrent-hierarchical FSM of air-condition controller

	fromState	event	toState	act	comp	entryNs	exitCs	enSub	deSub
Tab[0]	S1	I1	S2	null	null	O1	null	true	false
Tab[1]	S2	I1	S1	null	null	null	O2	false	true

(a) Main FSM

	fromState	event	toState	act	comp	entryNs	exitCs	enSub	deSub
Tab[0]	S3	I2	S4	null	null	O3	null	false	false
Tab[1]	S4	I2	S3	null	null	null	O4	false	false

(b) Sub FSM-1

	fromState	event	toState	act	comp	entryNs	exitCs	enSub	deSub
Tab[0]	S5	I3	S6	O5	null	null	null	false	false
Tab[1]	S6	I3	S5	O6	null	null	null	false	false

(c) Sub FSM-2

Acknowledgment. This work was supported in part by Fujian University of Technology, Granted KF-X18009 and GY-Z15087, and by Fujian Provincial Department of Science and Technology, Granted No.2017J01729.

REFERENCES

[1] J. Zhang, H. Nian, X. Ye, X. Ji and Y. He, *A Spatial Correlation Based Partial Coverage Scheduling Scheme in Wireless Sensor Networks*, Journal of Network Intelligence, vol. 5, no. 2, pp.34–43, 2020.

[2] J. N. Chen, Y. P. Zhou, Z. J. Hunag, T. Y. Wu, F. M. Zou and R. Tso *An Efficient Aggregate Signature Scheme for Healthcare Wireless Sensor Networks*, Journal of Network Intelligence, vol. 6, no. 1, pp.1–15, 2021.

[3] E. K Wang, C. M Chen, M. M. Hassan and A. Almogren *A deep learning based medical image segmentation technique in Internet-of-Medical-Things domain*, Future Generation Computer Systems, vol. 108, pp.135–144, 2020.

[4] E. K Wang, X Liu, C. M. Chen, S. Kumari, M. Shojafar and M. S. Hossain *Voice-Transfer Attacking on Industrial Voice Control Systems in 5G-Aided IIoT Domain*, IEEE Transactions on Industrial Informatics, DOI: 10.1109/TII.2020.3023677 , 2020.

[5] J. van Gurp and J. Bosch, *On the Implementation of Finite State Machines*, 3rd Annual IASTED International Conference on Software Engineering and Applications vol. 3, no. 1, pp.1–15, 1999.

[6] X. Xu, L. Wang and H. Zhou, *Implementation framework of finite state machines*, Journal of Engineering Design, vol. 10, no. 5, pp.251–255, 2003. (in Chinese)

[7] Z. Juhasz and A. Sipos, *Implementation of a Finite State Machine with Active Libraries in C++*, Proc. of the 7th International Conference on Applied Informatics, vol. 2, pp.247-255, 2007.

[8] M. H. Abidi, A. Jakimi, R. Alaoui, and E.H. EI Kinani, *An Object-Oriented Approach To Generate Java Code From Hierarchical-Concurrent and History States*, International Journal of Information and Network Security, vol. 2, no. 6, pp.429–440, 2013.

[9] V. Spinke, *An object-oriented implementation of concurrent and hierarchical state machines*, Information and Software Technology, vol. 55, no. 10, pp.1726–1740, 2013.

[10] *Finite State Machine Editor*, http://fsme.sourceforge.net/

[11] N. Almasri, L. Tahat and M. Alshraideh, *Maintenance-Oriented Classifications of EFSM Transitions*, Journal of Software, vol. 11, no. 1, pp.64–79, 2016.

[12] J. Ali, *Using Java Enums to Implement Concurrent-Hierarchical State Machines*, Journal of Software Engineering, vol. 4, no. 3, pp.215–230, 2010.

[13] Y. U. Chengjuei, W. U. Yihsin and S. Wang, *An Approach to the Design of Specific Hardware Circuits from C Programs*, Journal of Information Science and Engineering, vol. 34, no. 2, pp.337–351, 2018.