International Journal of Information & Network Security (IJINS)
Vol.2, No.6, December 2013, pp. 429~440
ISSN: 2089-3299

An Object-Oriented Approach To Generate Java Code From Hierarchical-Concurrent and HistoryStates

M.H. Aabidi**, **A.Jakimi****, **R. Alaoui**+* **and E.H. El Kinani***

* A.A Group, Mathematical Department, MoulayIsmaïl University, Faculty of Sciences and Technics Errachidia, Box 509 .Morocco.
+Computer Sciences Department, Mohammed Premier University, ENSA
Al Hoceima .Morocco
** R.O.I Group, Computer Sciences Department, Moulay Ismaïl University, Faculty of Sciences and Technics Errachidia, Box 509. Morocco

Article Info

Article history:

Received Oct 15th, 2013
Revised Nov 20th, 2013
Accepted Dec 26th, 2013

Keyword:

UML
State machine
Java Enumtype
Hierarchical/Concurrent state
History pseudo-state
Code generation

ABSTRACT

UML is widely accepted and practiced in industries for modeling and design of software systems. Software requirements and design are very important parts of software development. UML State Machine diagram is an important formalism to model the dynamic behavior of the system. In this paper, we present a new approach to generate compact and efficient Java code fromcomplex state machine involving hierarchical-concurrent states;in fact the proposed code can be used as a Java implementation patternfor automatic code generation for an object-oriented system. The main idea of our study is using Java Enum type with attributes for implementing complex state machines.

Corresponding Author:

E.H. El KINANI
A.A Group, Mathematical Department
MoulayIsmaïl University,
Faculty of Sciences and Technics, Box 509 Errachidia, Morocco
E-mail: elkinani_67@yahoo.com

1. INTRODUCTION

The Unified Modeling Language (UML) [1] has now become the de-facto industrystandard for object-oriented (OO) software development.UML provides a set of diagrams tomodel structural and behavioral aspects of an object-oriented system [2,3]. The UMLdiagrams created in the design phase to model an object-oriented system are later used in theimplementation phase [4]. UML diagrams are powerful enough to hold most of theimplementation details. However, manual translation of UML diagrams into object-orientedcode does not guarantee conformance of the code with the UML diagrams due to thepossibility of occurrence of human errors. Automatic translation of UML diagrams to objectoriented code is highly desirable because it eliminates the chances of introduction of humanerrors in the translation process.Automatic code generation is efficient which, in turn, helpsthe software engineers deliver the software on time.UML State machines are used to understand the dynamic aspects of classifiers such as classes; use cases; subsystems; entire systems. UML State machines supported the analyst in modeling the functional requirements of an evolvingsystem.UML State Machine Diagram is a variation of Harel's statechart ((Harel, 1987)[5], (Harel, 1988)[6]))that incorporates hierarchical states (OR-substates) and concurrent states (AND-substates) into the traditional state transition diagram.

The UML state machine is an improved version of finite state machine (FSM). The earliest technique to implement finite state machines is to use switch statement (Aho and Ullman, 1979) [7]. Based on the current active state, it performs a jump to the code for processing the event. States are represented as data values. This technique works well for classical flat state machines and is mostly used in non-object-oriented procedural languages.

In object-oriented systems, the behavior of a class of objects is implemented as a set of methods in the class. For classes having complex behavior, the methods will perform differently depending on the current state of the object. An object-oriented extension to the state machine has been done by (Coleman and al., 1992) [8]. They introduced inheritance into state machines and linked class methods to the transitions in the corresponding state machine.

Rumbaugh[9] proposes an object-oriented approach to implement state machines. He suggests using class-inheritance to represent state hierarchy. The State pattern [10] guides how to implement multi-state classes. Each state is implemented as a different object, which changes at runtime. It does not handle state hierarchy and concurrency. Sane and Campbell [11] say that states can be represented as classes and transitions as operations. They implement embedded states by making a table for the super state and do not consider concurrent states. MOODS [12] is a complex variant of the State Pattern. In this variant, the state class hierarchy uses multiple inheritance to model nested states.

Ali and Tanaka[13] use classes to represent individual states and methods to represent events/transitions. They implement state hierarchy with class inheritance. Their resulting code has too many classes and a class' behavior is not encapsulated within the class. Douglass [14] proposed the State Table Pattern to implement state machine diagrams. States and transitions are modeled as classes. Gurp and Bosch [15] developed a framework of few classes to instantiate and execute finite state machines (FSM). The FSM is not hard coded in the source code. Instead, it is read from an XML file and appropriate objects are created from the framework classes that together represent the FSM. Each state is represented as an object (not a class) and actions as attributes.

Kohler and al. [16] presented an approach for code generation from state machines. Their approach adapts the idea of generic array based state-table but uses object structure to represent state-table at runtime. They use objects to represent states of a state machine and attributes to hold the entry and exit actions. Knapp and Merz[17] described a set of tools called Hugo for the code generation of UML State Machine Diagram. A generic set of Java classes provides a standard runtime component for the state machine. Every state of a state machine is represented by a separate object, which provides methods for activation, deactivation, initialization and event handling. Events, guards and actions are also implemented as classes.

Chauvel and Jezequel[18] discuss different approaches to implement state machines. For more efficient code, they suggest to use enumerated values to represent states and events. To handle hierarchical states, they suggest first flattening the state hierarchy. For more flexible code, they suggest to use the State pattern [10].

There are many tools (Tiella and al. [19]; Jakimi and Elkoutbi [20]) that generate executable code from state machines. However, the papers do not give details of how states, events and transitions are represented in the generated code.Jauhar [21] proposed a new approach for implementing state machines using Java enum type, the generated code is encapsulated inside a single class. The disadvantage of this approach lies in the fact that this approach does not describe exactly the state of the studied system, because it just gives the name of the state and does not introduce the concept of attribute to better identify the state.

We present in this study a new approach for the implementation of state machine diagrams using Java Enumtype with attributes; these are used as parameters for a clear and comprehensive identification of each state of the system or the object class under study. Each state is represented as an enum-value in addition to the values of the attributes that allow a more detailed identification of the state. Hierarchical states and concurrent orthogonal regions within state machine diagrams are implemented by linking the related enum-values to each other. Our approach has several advantages. First, the use of Java enums makes the resulting code compact, efficient and easy to understand. Second, the structure of the state machine is obvious in the implementation code. Third, the whole state machine's behavior is encapsulated within a single class (called StateMachine). Fourth, each state is identified clearly and completely through the values of its attributes.

The rest of this paper is structured as follows: section 2 presents the main idea of our approach. Section 3, is devoted to the a demonstration of our approach on complex statemachine examples.Section 4 concludes the paper and give some prospective points for the future work of this research.

2. RESEARCH METHOD

Our approach combines existing appoaches for code generation, the first approach is that proposed by Jauhar[21] and the second is that of Jakimi and Elkoutbi [20]. The code generated by the first approach is encapsulated into a single class and it is well-structured, compact and easy to understand.

However, asa state is a set of valuesthat describes an object at a specific point in time, and it represents a point in anobject’s life in which it satisfies some condition, performs some action, or waits for somethingto happen and as an event is something that takes place at a certain point in time andchanges a value(s) that describes anobject, which, in turn, changes the object’s state and asobjects move from state to state, they undergo transitions;the generated code remains incomplete because the values of Java Enum type used to represent the system states does not describe exactly the studied system states, but just gives a name to each state of the system or the object class under study, and the second approach for a more detailed description of the state in which is the system or the object class studied. The fusion of the two approaches provides a better way to implement state machines leveraging the positive points of both approaches, state machine encapsulated within a single class, code well structured, clear, compact and easy to understand for the first approach and a better identification of the state for the second approach.

3. RESULTS AND ANALYSIS

In this section, we demonstrate our aproach on complex state machine involving hierarchical-concurrent states.

3.1. Hierarchical states with history pseudo-state

State machines may have hierarchical states where the substates inherit transitions from its superstate.Figure 1 shows the behavior of the the Ceiling-fan with a superstate (On). If the On state is active, the system will be either in Fan_stopped (default), either in Fan_slow_speed, either in Fan_medium_speed or in Fan_fast_speed state. In any of the four substates, if the wall swithis actuated, the state will change to Off.

To demonstrate our approach on a complex state machine diagram comprising hierarchical states with history pseudo-state, consider the example of a Ceiling-fan which works as follows:

The equipment has two commands a wall general switch and a pull tab for the rotation speed.

- ✓ The wall switch stops or turns on in the state where you stopped the aircraft.
- ✓ The speed pull tab passes in a circular fashion the speed of the aircraft:
  Stopped→ Slow → Medium → Fast →Stopped →…..
- ✓ Obtained without particular difficulty this state machine diagram as follows:

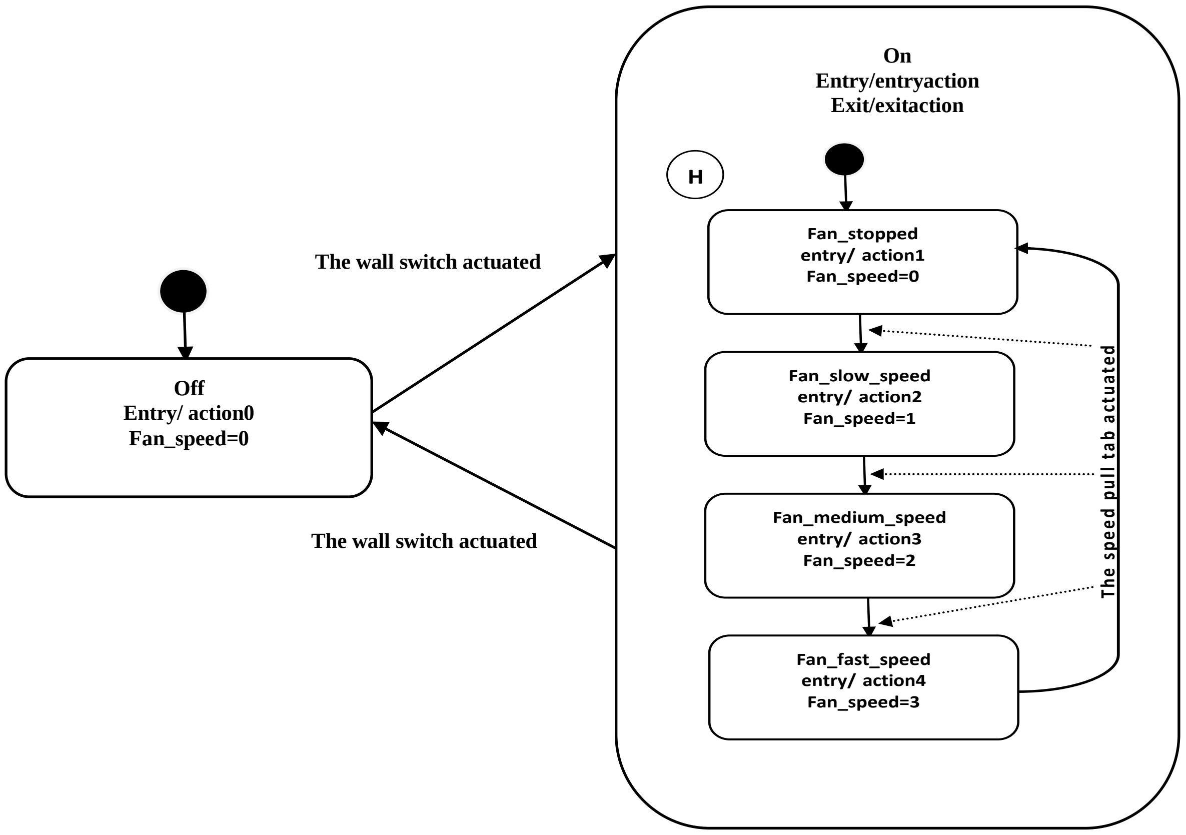

Figure 1. State machine diagram of the ceiling-fan (Version1)

The initial state of the ceiling-fan is the state "Off". When the wall switch is actuated the system switches into the super-state "On" having an entry action and exit action. The initial state of the super-state "On" is the substate "Fan_stopped" having an entry action into the substate named "action1". Once the speed pull tab is actuated the system switches to the substate "Fan_slow_speed" having an action entry into the substate named "action2". Actuate a second time the speed pull tab swiches the system to the substate "Fan_medium_speed" having an entry action into the substate named "action3." Actuate a third time the speed pull tab swiches the system to the substate "Fan_fast_speed" having an entry action named "action4".

Actuate a fourth time the speed pull tab switches the system to the substate "Fan_stopped" and the same cycle repeats. The system can memorize the last substate "history pseudo-state" in which was the system before leaving the super-state "On" from to the state "Off" after actuating again the wall switch.

The Java code generated from the state machine diagram is as follows:

```
public class Fan{
        StateMachinestateMachine;
        Fan () {stateMachine= new StateMachine(this);}
         //the action methods:  to be replaced with the appropriate code
        private void entryAction() { System.out.println("entry action executed"); }
        private void exitAction() { System.out.println("entry action executed");}
private void action0() {  System.out.println("action_0  executed");}
        private void action1()  {System.out.println("action_1  executed"); }
        private void action2() { System.out.println("action_2  executed"); }
        private void action3()  { System.out.println("action_3  executed"); }
        private void action4()  { System.out.println("action_4  executed"); }

//events delegated to StateMachine
        public void walSwitch(){stateMachine.walSwitch();}
        public void spdPulTab(){stateMachine.spdPulTab();}
  //the StateMachine class
        static class StateMachine{
                Fan context;
                State state;
                State history;
        StateMachine(Fan context){
                        this.context=context;
                        state=State.Off; //initial state
                        state.fan_speed=0;
                        history=null;// history pseudo-state
        }
        private void walSwitch () {state.process(this,Event.WalSwitch );}
        private void spdPulTab () {state.process(this,Event.SpdPulTab);}
        private void enterState(State... states){
                for(State s:states)  s.entry(this);
                state=states[states.length-1];
                if(state!=State.Off)  history=state;
        }
        private void exitAll(State child,State parent) {
                State s=child;
                while(1){
                        s.exit(this);
                        if(s==parent) break;
                         s=s.parent;
                }
}
 //all of the events
        enum Event {WalSwitch ,SpdPulTab}
// the events “the wall switch actuated” and  “the speed pull tab actuated”
 //all of the states
        enum State {
                Off(null){  //the state Off
                        void entry(StateMachinesm) {sm.context.action0();}
                        void process(StateMachinesm,Event e) {
                                                        switch(e){
```

```
                                                      caseWalSwitch :
                                                      this.exit(sm);
                                             if(sm.history==null)
                         {sm.enterState(On ,Fan_stopped); sm.state.fan_speed=0;}
      else  {sm.enterState(On ,sm.history); sm.state.fan_speed=sm.history.fan_speed;}

 }
                   }
 },
                  On(null){  //the super-state On
                           void entry(StateMachinesm) {sm.context.entryAction();}
                           void exit(StateMachinesm)    {sm.context.exitAction();}
                           void process(StateMachinesm,Event e) {
                                    switch(e){
                                    caseWalSwitch :
                                    sm.exitAll(sm.state,this);
                                    sm.enterState(Off);
                                             sm.state.fan_speed=0;
                   }
                           }
          },
                  Fan_stopped(On){ //the substateFan_stopped
                  void entry(StateMachinesm){sm.context.action1();}
                           void process(StateMachinesm,Event e) {
                                    switch(e){
                           caseSpdPulTab:
                           this.exit(sm);
                           sm.enterState(Fan_slow_speed);
                           sm.state.fan_speed=1;
                           break;
                           default:  parent.process(sm,e);
          }
                   }
                 },
                  Fan_slow_speed(On){ //the substateFan_slow_speed
                  void entry(StateMachinesm) {sm.context.action2();}
                           void process(StateMachinesm,Event e){
                                    switch(e){
                           caseSpdPulTab:
                           this.exit(sm);
                           sm.enterState(Fan_medium_speed);
                           sm.state.fan_speed=2;
                           break;
                           default:  parent.process(sm,e);
          }
}
 },
                  Fan_medium_speed(On){ //the substateFan_medium_speed
                  void entry(StateMachinesm) {sm.context.action3();}
                           void process(StateMachinesm,Event e){
                                    switch(e){
                                    caseSpdPulTab:
                                    this.exit(sm);
                                    sm.enterState(Fan_fast_speed);
                                    sm.state.fan_speed=3;
                           break;
                                    default:  parent.process(sm,e);
                   }
                   }
```

```
},
                    Fan_fast_speed(On){ //the substateFan_fast_speed
                    void entry(StateMachinesm)  {sm.context.action4();}
                              void process(StateMachinesm,Event e) {
                                        switch(e){
                                        caseSpdPulTab:
                                        this.exit(sm);
                                        sm.enterState(Fan_stopped);
                                        sm.state.fan_speed=0;
                                        break;
                                        default:parent.process(sm,e);
                              }
                    }
  };
intfan_speed;
   State parent;
State(State p) {parent=p;}
abstract void process(StateMachinesm,Event e);
void exit(StateMachinesm){}
void entry(StateMachinesm){}
     } //end of enum State
   } //end of class StateMachine
 }//end of class Fan
```

3.2. Concurrent states with historypseudo-state

State machines may have concurrent substates, which means that all the substates are active when their superstate is active. Figure 2 shows the Ceiling-fan’s behavior with concurrent states. When the system is inOn state, both LightMode and SpeedMode regions (concurrent substates) are active. In the LightMode region, either Light_off or Light_on will be active. Similarly, in the SpeedMode region, eitherFan_stopped, either Fan_slow_speed, either Fan_medium_speed or Fan_fast_speed will be active.

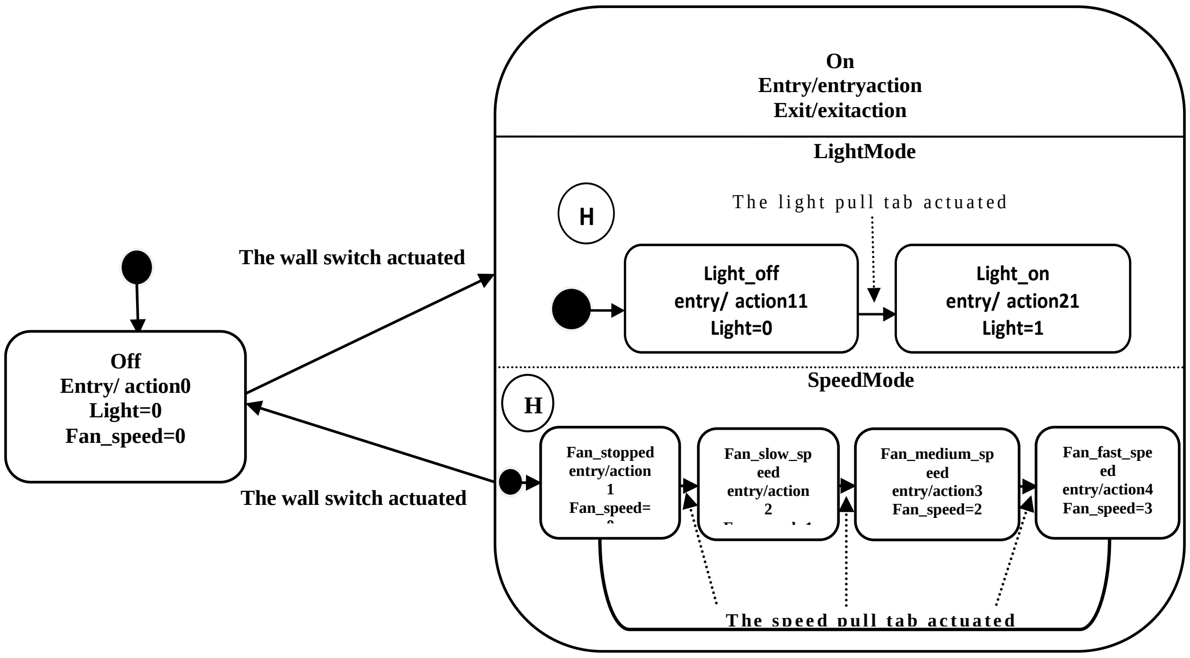

Figure 2.State machine diagram of the ceiling-fan (Version2)

To demonstrate our approach on a state machine diagram comprising concurrent states, consider the example of a Ceiling-fan equipped this time with a light pull tab and which works as follows:

The equipment has tree commands a wall general switch, a light pull tab and a pull tab for the rotation speed.

- ✓ The wall switch stops or turns on in the state where you stopped the aircraft.
- ✓ The light pull tab at each action turns on or off the bulb.
- ✓ The speed pull tab passes in a circular fashion the speed of the aircraft:

  Stopped→ Slow → Medium → Fast →Stopped →…..

The initial state of the ceiling-fan is the state "Off". When the wall switch is actuated the system switches into the super-state "On" having an entry action and exit action and two concurrent states "LightMode" and "SpeedMode". The initial state of the concurrent state "SpeedMode" is the substate "Fan_stopped" having an entry action into the substate named "action1". Once the speed pull tab is actuated the system switches to the substate "Fan_slow_speed" having an action entry into the substate named "action2". Actuate a second time the speed pull tab swiches the system to the substate "Fan_medium_speed" having an entry action into the substate named "action3." Actuate a third time the speed pull tab swiches the system to the substate "Fan_fast_speed" having an entry action named "action4". Actuate a fourth time the speed pull tab switches the system to the substate "Fan_stopped" and the same cycle repeats.

In parallel the initial state of the concurrent state "LightMode" is the substate "Light_off". Actuate the light pull tab switches the system to the substate "Light_on" and vice versa.

The system can memorize the two last substates "history pseudo-states" in which was the system before leaving the super-state "On" from to the state "Off" after actuating again the wall switch.

The Java code generated from the state machine diagram is as follows:

```
public class Fan1{
        StateMachinestateMachine;
        Fan1 () {stateMachine= new StateMachine(this);}
 //the action methods: to be replaced with the appropriate code
        private void entryAction(){System.out.println("entry action executed ");}
        private void exitAction(){System.out.println("exit action executed");}
        private void action0(){ System.out.println("action_0  executed");}
        private void action11(){System.out.println("action_11  executed"); }
        private void action21(){System.out.println("action_21  executed ");}
        private void action1() { System.out.println("action_1  executed");}
        private void action2(){ System.out.println("action_2  executed ");}
        private void action3() { System.out.println("action_3  executed ");}
        private void action4() { System.out.println("action_4  executed ");}

//events delegated to StateMachine
        public void walSwitch (){stateMachine.walSwitch ();}
        public void spdPulTab(){stateMachine.spdPulTab();}
        public void lightPulTab(){stateMachine.lightPulTab();}
 //the StateMachine class
        static class StateMachine{
                        Fan1 context;
                        State state;
                        State lightmode, speedmode;
                        State histo_light,histo_speed;
                        StateMachine(Fan1 context){
                                this.context=context;
                                state=State.Off; //initial state
                                state.light=false ; // attributes values of the initial state
                                state.fan_speed=0;
                                histo_light=null;
 //the history pseudo-state "histo_light" set to null before entering the super-state "On".
                                histo_speed=null;
 // the history pseudo-state "histo_speed" set to null before entering the super-state "On".
                        }
        private void walSwitch () {state.process(this,Event.WalSwitch );}
        private void spdPulTab () {state.process(this,Event.SpdPulTab);}
        private void lightPulTab () {state.process(this,Event.LightPulTab);}
        private void enterState(State... states){
                for(State s:states)  s.entry(this);
                state=states[states.length-1];
        }
        private void enterLightMode(State... states) {
                for(State s:states)  s.entry(this);
                lightmode=states[states.length-1];
                histo_light=lightmode;
        }
        private void enterSpeedMode(State... states){
                for(State s:states)  s.entry(this);
                speedmode=states[states.length-1];
                histo_speed=speedmode;
        }
        private void exitAll(State child,State parent) {
                State s=child;
                while(1){
                        s.exit(this);
                        if(s==parent) break;
                         s=s.parent;
  }
}
 // the events “the wall switch actuated”, “the speed pull tab actuated” and “the light pull tab actuated”
enum Event {WalSwitch ,SpdPulTab,LightPulTab}

 //all of the states
        enum State {
                Off(null){  //the state Off
                void entry(StateMachinesm) {sm.context.action0();}
                void process(StateMachinesm,Event e) {
                        switch(e){
                        caseWalSwitch :
                        this.exit(sm);
                        sm.enterState(On);
                        if(sm.histo_light==null &&sm.histo_speed==null){
                                sm.enterLightMode(LightMode ,Light_Off);
                        sm.enterSpeedMode(SpeedMode ,Fan_stopped);
                sm.lightmode.light=false;  sm.speedmode.fan_speed=0;
}
        else{
                                sm.enterLightMode(LightMode ,sm.histo_light);
                sm.enterSpeedMode(SpeedMode ,sm.histo_speed);
                sm.lightmode.light=sm.histo_light.light;
                sm.speedmode.fan_speed=sm.histo_speed.fan_speed;
                }
}
 }
  },
                On(null){   //the super-state “On”
                void entry(StateMachinesm) {sm.context.entryAction();}
                void exit(StateMachinesm){sm.context.exitAction();}
                void process(StateMachinesm,Event e){
                        switch(e){
                        caseWalSwitch :
                        sm.exitAll(sm.lightmode,LightMode);
                        sm.exitAll(sm.speedmode,this);
                        sm.enterState(Off); sm.state.light=false;
                        sm.state.fan_speed=0; break;
```

```
                                        caseLightPulTab:
                                        caseSpdPulTab:
                                        sm.lightmode.process(sm,e);
                                        sm.speedmode.process(sm,e);
                              }
}
 },
                    LightMode(On){   //the concurrent state LightMode
                    void process(StateMachinesm,Event e){}
},
                    Light_Off(LightMode){ //the substateLight_Off of the concurrent state LightMode
                    void entry(StateMachinesm) {sm.context.action11();}
                              void process(StateMachinesm,Event e){
                                        switch(e){
                                        caseLightPulTab:
                                        this.exit(sm);
                                        sm.enterLightMode(Light_On);
                                        sm.lightmode.light=true;
                                        break;
                                        default: parent.process(sm,e);
                      }
                      }
                   },
                    Light_On(LightMode){ // the substateLight_On of the concurrent state LightMode
                    void entry(StateMachinesm){sm.context.action21();}
                              void process(StateMachinesm,Event e) {
                                        switch(e){
                                        caseLightPulTab:
                                        this.exit(sm);
                                        sm.enterLightMode(Light_Off);
                                        sm.lightmode.light=false;
                                        break;
                                        default: parent.process(sm,e);
                                }
                     }
                   },
                    SpeedMode(On){ //the concurrent state SpeedMode
                    void process(StateMachinesm,Event e){}
          },
                    Fan_stopped(SpeedMode){ //the substateFan_stopped of the concurrent state SpeedMode
                    void entry(StateMachinesm){sm.context.action1();}
                              void process(StateMachinesm,Event e){
                                        switch(e){
                                        caseSpdPulTab:
                                        this.exit(sm);
                                        sm.enterSpeedMode(Fan_slow_speed);
                                        sm.speedmode.fan_speed=1;
                                        break;
                                        default: parent.process(sm,e);
                              }
                     }
},
          Fan_slow_speed(SpeedMode){ // the substateFan_slow_speed of the concurrent state SpeedMode
                    void entry(StateMachinesm) {sm.context.action2();}
                              void process(StateMachinesm,Event e){
                                        switch(e){
                                        caseSpdPulTab:
                                        this.exit(sm);
                                        sm.enterVitMmode (Fan_medium_speed);
                                        sm.speedmode.fan_speed=2;
                                        break;
                                        default:  parent.process(sm,e);
                                }
                    }
                   },
                    Fan_medium_speed(SpeedMode){
// the substateFan_medium_speed of the concurrent state SpeedMode
                    void entry(StateMachinesm){sm.context.action3();}
                    void process(StateMachinesm,Event e){
                              switch(e){
                              caseSpdPulTab:
                              this.exit(sm);
                              sm.enterSpeedMode (Fan_fast_speed);
                              sm.speedmode.fan_speed=3;
                              break;
                              default:   parent.process(sm,e);
                    }
             }
                   },
                    Fan_fast_speed(SpeedMode){
// the substateFan_fast_speed of the concurrent state SpeedMode
                    void entry(StateMachinesm) {sm.context.action4();}
                              void process(StateMachinesm,Event e){
                                        switch(e){
                                        caseSpdPulTab:
                                        this.exit(sm);
                                        sm.enterSpeedMode (Fan_stopped);
                                        sm.speedmode.fan_speed=0;
                                        break;
                                        default: parent.process(sm,e);
                                }
                    }
                   };
intfan_speed;
boolean light;
  State parent;
State(State p) {parent=p;}
abstract void process(StateMachinesm,Event e);
void exit(StateMachinesm){}
void entry(StateMachinesm){}
     } //end of enum State
   } //end of class StateMachine
 }//end of class Fan1
```

3.3. Discussion

We represent states as Java Enum type with attributesand eventsas Java Enumtype, which makes the code efficient and the statemachine more detailed. Java Enum type are loaded when the enclosing class is loaded. Its performance is comparable to primitives. State hierarchy is represented as Enum hierarchy and concurrent states are represented by using the concept of object composition. Java does not support enum inheritance explicitly. We used a parent reference in the State enum to handle state hierarchy.

As noted by Chauvel and Jezequel (2005)[18], representing states as enumerations is more efficient than using state classes. They suggest using enumeration for flattened state machines only. We use Java Enums type with attributesto implement all types of state machines including concurrent-hierarchical ones. We improve performance (by using Java Enum type with attributes) without compromising on flexibility (by using Enum-hierarchy).

Java Enums type, after compilation, are equivalent to objects. Therefore all the approaches (Rumbaugh, 1993[9]; Gamma and al., 1995[10]; Sane and Campbell, 1995[11]; Ran, 1996[12]; Ali and

Tanaka, 1998[13]; Douglass, 1998[14]; Gurp and Bosch, 1999[15]; Knapp and Merz, 2002[17]) which suggest representing states as classes (or objects) are in the support of our approach.

Among the many approaches we have reviewed, only few (Ran, 1996[12]; Ali and Tanaka, 1998[13]) support concurrent-hierarchical state machines. However the resulting code has too many classes, is less efficient and is not encapsulated in the owner class. Our proposed code is well-structured and the state machine details are obvious in the resulting source code. This makes it easy to reverse-engineer the code back to a state machine, if needed. Furthermore, validating the code against the corresponding state machine is straight-forward. In fact, the code can be generated automatically because there is almost a one-to-one correspondence between state machine elements and our proposed code.

In our proposed code, the behavior of a class or a system, represented by the corresponding state machine, is completely encapsulated inside the class and implemented as a nested class. In principle, the nested StateMachine class and all its members should be private. In the example listings, we did not declare them private so that the test codecan work properly. The StateMachine class is static because Java Enum type are static by default and they require a static environment.

4. CONCLUSION

In this paper, a new approach is proposed for an efficient implementation for complex state machine diagrams involving hierarchical-concurrent states. Our approach allows a complete identification of each state using attributes within Java enum type used to represent the different states of the studied system, in addition the whole state machine's behavior is encapsulated within a single class allowing a better structuring of the state machine and makes the code clear and easy to understand by exploiting the advantages offered by Java Enum type as well as adding attributes to Java enum type allowing a better identification of the states of the studied system. The proposed code can serve as a Java implementation pattern for state machine diagrams.

The proposed approachallowedreducing the gap between the analysis/design models and the implementation of a system by the generation of Java code from UML complex state machine.As perspectives, we will explore the possibilities of generating code for platforms using the following technologies: J2EE, Web Services, .Net and reverse engineering from UML diagrams.

Acknowledgments : The authors would like to thank the anonymous reviewer for his useful comments and suggestions .

REFERENCES

[1] Object Management Group (OMG), Unified Modeling Language Specification, Version 2.1.1, (2007-02-07).
[2] Booch G. Object Oriented Design with Applications, Benjamin/Cummings, Redwood, California, 1991.
ISBN: 0-8053-0091-0, ISSN: 0896-8438, 1991.
[3] Coad P, YoudonE.Object-Oriented Analysis, Prentice Hall, Eaglewood Cliffs, New Jersey, 1991.
ISBN: 0-13-630070-7.
[4] Jacobson I, BoochG, RumbaughJ. The Unified Software Development Process, Addison-Wesley, Reading, MA, 1999.
[5] HarelD.“Statecharts: A visual formalism for complex systems”, *Science of Computer Programming*, 1987; 8(3): 231-274.
[6] HarelD. "On Visual Formalisms", *Comm. Assoc. Comput. Mach*. 1988;31(5): 514-530.
[7] Aho A, UllmanJ. Principles of Computer Design. Addison Wesley, Massachusetts, 1979.
[8] Coleman D, Hayes F, BearS.. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Trans. Software Eng*. 1992; 18(1): 8-18.
[9] Rumbaugh J.Controlling code: How to implement dynamic models.*J. Object-Oriented Programming*. 1993; 6: 25-30.
[10] Gamma E, RichardH, Johnson R, VlissidesJ.Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Massachusetts, 1995.
[11] Sane A, CampbellR.Object-oriented state machines: Subclassing, composition, delegation and genericity. *ACM SIGPLAN Notices*. 1995; 30(10): 17-32.
[12] Ran A. MOODS: Models for Object-Oriented Design of State. In: Pattern Languages of Program Design 2, Vlissides J.M, Coplien J.O, Kerth N.L. (Eds.). Addison-Wesley, Boston, USA, 1996.
[13] Ali J, TanakaJ.An object oriented approach to generate executable code from the OMT-based dynamic model. *J. Integrated Des. Process Technol*. 1998; 2(4): 65-77.
[14] Douglass B.P. Real Time UML - Developing Efficient Objects for Embedded Systems.Addison-Wesley, Massachusetts, 1998.
[15] Gurp J.V, BoschJ. *On the implementation of finite state machines*. Proceedings of the 3rd Annual IASTED International Conference on Software Engineering and Applications, Oct. 6-8, Scottsdale, Arizona. 1999; 1-15.
[16] Kohler H,NickelU,Niere J, ZundorfA. *Integrating UML diagrams for production control systems*. Proceedings of the 22nd International Conference on Software Engineering, (ICSE'00), Limerick, Ireland. 2000: 241-251.

[17] Knapp A, MerzS. *Model checking and code generation for UML state machines and collaborations*. Proceedings of the 5th Workshop on Tools for System Design and Verification, (WTSDV'02), Reisenburg, Germany. 2002; 59-64.
[18] Chauvel F, JezequelJ. Code *generation from UML Models with semantic variation points*. Proceedings of the 8th International Conference MoDELS 2005, Oct. 2-7, Montego Bay, Jamaica.2005; 54-68.
[19] Tiella R, Villafiorita A, Tomasi S.*FSMC+, a tool for the generation of java code from statecharts*. Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java. Sept. 5-7,Lisboa, Portugal.2007;93-102.
[20] Jakimi A, ElkoutbiM. Automatic code generation from UML statechart. *Int. J. Eng. Technol.*2009; 1(2): 165-168.
[21] Jauhar Ali. Using Java Enums to Implement Concurrent-Hierarchical State Machines. *Journal of Software Engineering*. 2010; 4(3): 215-230.

BIOGRAPHY OF AUTHORS

Aabidi My Hafid, is a teacher of computer Science at the Centre for preparatory classes for engineering schools in Omar Ibn El-Khattab high school, Meknes; Morocco. His current research interests include requirements engineering, user interface prototyping, design transformations, scenario engineering, code generation and reverse engineering.

Abdeslam Jakimi, is a professor at Faculty of Science and Technics in Errachidia, My Ismail University; Morocco. His current research interests include requirements engineering, user interface prototyping, design transformations, scenario engineering, code generation and reverse engineering.

Rachid Alaoui, is a professor at ENSA Al Hoceima. His research interests include code generation, pattern recognition and computer vision, image retrieval by similarity, data clustering, and biometric systems.

El Hassan El Kinani received the Ph.D in mathematical physics in 1999 from Mohamed V University Rabat Morocco. He is full professor at department of mathematics in MoulayIsmaïl University, Faculty of Sciences and Technics, Errachidia, Morocco. He is interested in classical and quantum cryptography.