State Pattern supporting both composite States and extension/specialization of State Machines

BIRGER MØLLER-PEDERSEN, RAGNHILD KOBRO RUNDE, University of Oslo, Norway

---

Most modelling languages support full state machine modelling, including especially composite states. Existing approaches to programming with states (state design patterns) either represent composite states by means of inheritance between state classes, or do not support composite states and instead use inheritance for extension/specialization of state machines. In this paper, we present a state machine design pattern using method forwarding to support composite states and inheritance to support extensible state machines.

Categories and Subject Descriptors: D.2.11 [Software Architectures] Patterns; D.2.2 State diagrams

General Terms: Design

Additional Key Words and Phrases: State Machine, Composite State

---

1. INTRODUCTION

In order to avoid inconsistent model and program artefacts when using both modelling and programming languages during software development, (Madsen and Møller-Pedersen 2010) proposed the definition of a combined modelling and programming language. The definition of such a language should be based on an analysis of how important modelling concepts can be supported by programming languages. Before embarking upon making new language constructs, it is regarded a good idea to implement the constructs in some existing language in order to get experiences with the constructs. This paper describes a state machine design pattern for programming that supports important elements of state machine modelling.

We require all of the most commonly supported mechanisms in modelling languages: composite states (with history, entry and exit actions), and specialization of state machines (all of this supported by e.g. SDL (ITU 2011) and UML (OMG 2015)).

As already introduced in 1987 (Harel 1987) a composite state is a state with sub states (contained states) such that all events and corresponding transitions that apply to the composite state by default apply to all of the sub states, unless specified differently. The original state design pattern (Gamma, Helm et al. 1995) represents states as subclasses of a general class State, redefining the event methods of State in the different state subclasses. This readily supports simple states. Composite states are usually represented by further subclasses (for the sub states) of the classes for the composite states. The event methods of the composite state classes are therefore inherited, and event methods may be overridden for the sub states in cases where the default behaviour specified for the composite state shall not apply.

In 2008, (Chin and Millstein 2008) demonstrated the need for specialization of state machines (by adding states and events methods, and by extending states), and how to do that by means of a state design pattern. Their approach uses subclassing in order to specialize state machines, so therefore they do not support composite states, as this is usually also done by subclassing.

In order to support both composite states and specialization of state machines we therefore pursue the idea of representing composite states by contained state objects being linked to their enclosing state and by forwarding event method call via these links. With the link from a sub state object to the enclosing state object, an event method call on a sub state that does not define this event method will be forwarded to the enclosing state. Subclassing may then be used for specialization of state machines.

2. EXAMPLE

Figure 1 is the simple state machine of a media switch. It specifies that the initial state of the media switch will be `Off` (indicated by the black dot with arrow). When powered on it will enter the state `On` with its initial state `CD`. The mode is changed by the `mode` event. The state `On` has an `entry` action that is executed whenever `On` is entered, turning on the display backlight, and an `exit` action that is executed whenever `On` is exited, turning off the backlight.

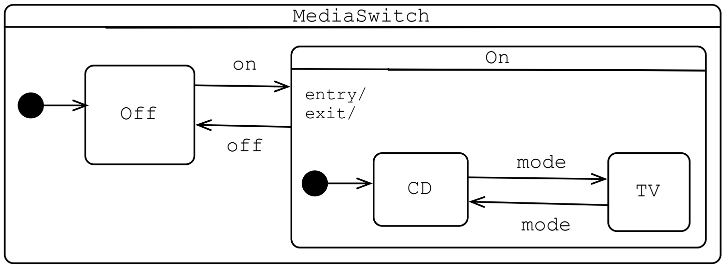

Figure 1 State Machine of a Media Switch

2.1 Composite States

State `On` is a composite state. The main property of a composite state is that transitions defined for the composite state apply to its contained states, if not specified otherwise. For the `MediaSwitch` state machine this means that the event `off` will make the machine enter the state `Off`, from any of the states in `On`. The entry/exit actions defined for `On` do not apply to the contained states; they may have their own entry/exit actions. However, exiting e.g. `CD` with the transition to `Off` will include the execution of an eventual exit action of `CD` followed by execution of the exit action of `On`.

Using the state design pattern, the sub states `CD` and `TV` would be represented by subclasses of the class representing the composite state `On`. The subclasses for `CD` and `TV` would then inherit the event method for `off` from the class representing `On`, with the possibility to override it. In addition, the subclasses would implement suitable event methods for the `mode` event.

2.2 Specialization of State Machines

Recently, (Chin and Millstein 2008) demonstrated the need for specialization of state machines (called *extensible* state machines) and how to support that by a design pattern. This is achieved by not using subclassing to specify composite states, but rather use subclassing to specify extension. The implication is that their state pattern only covers state machines with simple states and not composite states (as these are usually covered by subclassing).

Specialization of state machines is illustrated by defining the `MediaSwitch` state machine as a specialization of a simpler and more general `Switch` state machine, see Figure 2.

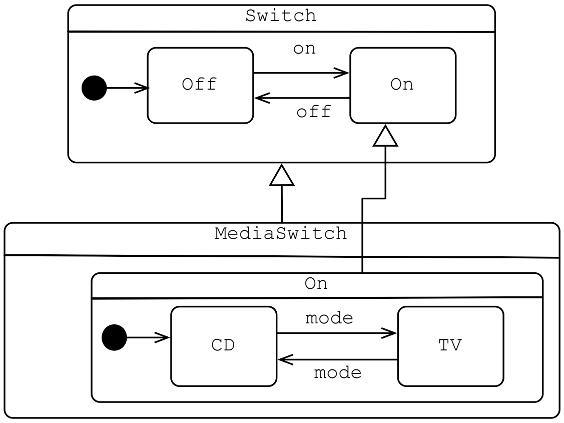

Figure 2 MediaSwitch as a specialization of Switch

With the extensible state machine design pattern, state classes are defined as inner classes to a state machine class. A specialization is specified by defining a subclass of the enclosing state machine class. In this example, `MediaSwitch` would be represented by a subclass of the class representing `Switch`. Extending a state (e.g. in order to handle additional events or to have inner states) is done by making a subclass of the state class from the general state machine class. This is illustrated in Figure 2 with the `On` state class, as this is extended to become a composite state and to handle the new event `mode`. The class for `On` in `MediaSwitch` would be a subclass of the class for `On` in `Switch`.

2.3 Problem

There are two problems with the common practice of using subclassing to cover composite states. First of all, subclassing is the obvious mechanism to use for covering specialization of state machines as described above. Composite states must then by supported by another mechanism in order to be able to distinguish between the two.

Another problem with the use of subclassing for composite states is that entry/exit action methods then will be inherited by contained states, and as described above this is not the semantics of entry/exit actions of composite states according to UML (OMG 2015). In our example, if the states `CD` and `TV` inherit the entry/exit actions of the enclosing state `On`, then changing back and forth between the states `CD` and `TV` (by the event `mode`) would imply that the display backlight would be turned on and off for each state change.

3. COMPOSITE AND EXTENSIBLE STATE MACHINE PATTERN

3.1 Pattern name

Composite and Extensible State Machine

3.2 Context

Whenever the simple state design pattern is not enough, and there is a need for composite states *and* specialization of state machines. Even in situations where only one of composition or specialization is needed, the pattern is useful as it supports further development and in particular history and entry/exit actions.

3.3 Problem

How can composite states and specialization of state machines be combined when programming state machines?

3.4 Forces

Subclassing is a powerful object-oriented mechanism, and it may be use for both composite states and specialization of state machines. However, it is usually not a good idea to use the same language mechanism for two different things. If using subclassing for composite states, then things like entry/exit actions defined for the composite state class will be inherited by the contained states that are represented by subclasses of the composite state class, and that is not according to state machine semantics. One would have to override of these in all state subclasses, and that would be both cumbersome and error-prone.

State machines may later be extended into more complex ones, so the solution should be scalable without being unnecessary complicated.

3.5 Solution

The solution is to use method forwarding for composite states and inheritance for specialization of state machines. A template state machine diagram containing both composite states and specialization is shown in Figure 3. The original state machine `SM1` contains a number of states, including `A` and `B`. The state machine `SM2` is a specialization of `SM1`, where the composite state `` B` `` (consisting of a number of states including `B1` and `B2`) is a specialization of `B`.

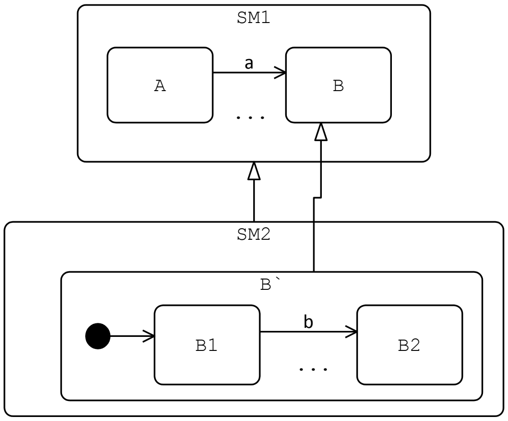

Figure 3 SM2 is a specialization of SM1, the composite state B` is a specialization of B

A state machine like the one in Figure 3 can be implemented as illustrated by the class and object diagrams in Figure 4. In order to use a combination of method forwarding and subclassing for making extensible state machines with composite states, there is a need for a general class `StateMachine`, with a general class `State` as an inner class. The `enclState` association between instances of `State` is used to support composite states.

A specific state machine (e.g. `SM1`) is defined by a subclass of the `StateMachine`, defining the special states for this machine (`A` and `B`) as subclasses of the inherited class `State`. Further specializations (`SM2`) are made by making subclasses of `SM1` and further add new states and/or extend inherited states (e.g. `B` extended into `` B` ``).

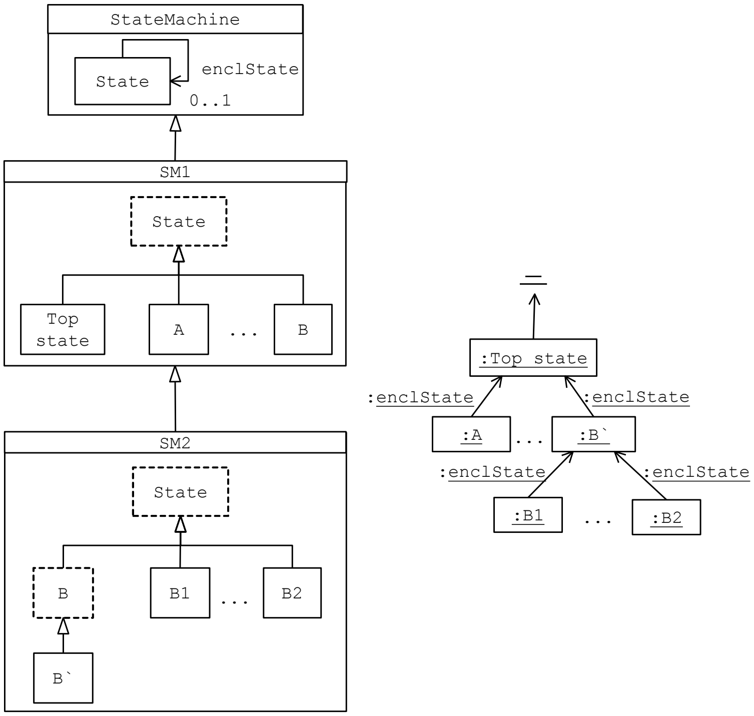

Figure 4 Composite and extensible state machine implemented using inheritance and method forwarding

Using method forwarding, an event method call to a state is forwarded to its enclosing (composite) state in case the event method is not defined specifically for the current state. Each sub state object will have a link (`enclState`) to its composite state object, as illustrated in the object diagram in the right part of Figure 4. The `Top state` root state is reached when event method calls are forwarded to the root (i.e. not handled in any of the other states). The composite state structure is set up as part of the constructors for the state classes. Each constructor has a reference to the enclosing state object as parameter.

Example. The `MediaSwitch` state machine from Figure 1 uses composite states, but not specialization. Applying the current pattern gives that `MediaSwitch` should be implemented as given in Figure 5.

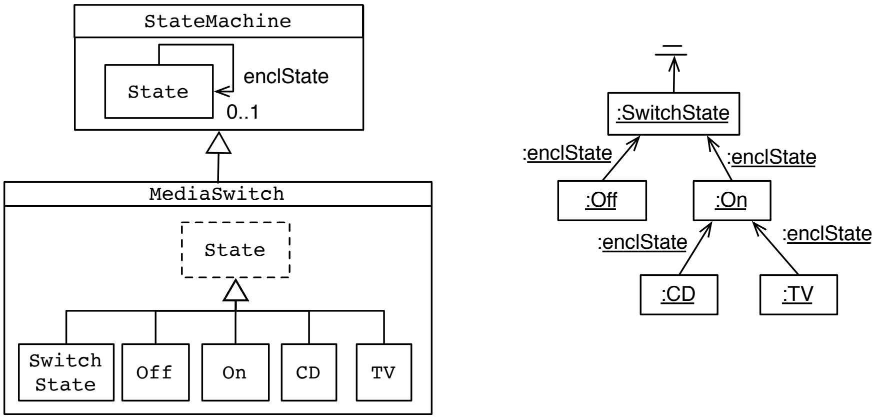

Figure 5 Class diagram and object diagram with method forwarding links for the MediaSwitch state machine

The way in which inherited state classes may be extended depends upon the programming language; see the following section on *Implementation*.

Implementation. The following describes how the combined pattern may be implemented in Java. The forwarding approach as illustrated in Figure 4 is the simplest alternative, with all states being objects of subclasses of `State`, and the state hierarchy being maintained by the link `enclState`.

Figure 6 defines the general classes of `StateMachine` and `State`. In addition, we have included the interface `IState`, which is used as the type for state links. For a given state machine, an interface that extends `IState` should be defined, with the event methods for that state machine. In case the design pattern should be extended to become a framework, then predefined code for states would be in the class `State`. The context object (the object that has a state machine) will also implement `IState` and forward any event method call to the corresponding method of its `StateMachine` object which in turn will call the method on the current state object. In order not to generate new `State` objects for each next state, the constructor for the state machine should generate state objects and set the links to their enclosing state.

```
interface IState {}

class StateMachine {
  IState cS;                        // current state
  void changeState(Class ns){}

  class State implements IState {
    IState enclState;               // enclosing state
  }
}
```

Figure 6 StateMachine and State

When it comes to specialization of state machines, (Chin and Millstein 2008) provides a detailed description of what is required to make a design pattern in Java, and show how it is possible, by means of subclassing, to add new states and extend states with new states and new events. The following is a simplified description of this. A final combined design pattern using method forwarding for composite states and subclassing for specialization of state machines will have to include all the details from (Chin and Millstein 2008).

The idea behind (Chin and Millstein 2008) is that states of a general state machine are extended in specialized state machines. Java does not provide extension of inner, inherited classes, so that has to be done in this way:

- For states that shall be extended, subclasses of these state classes are defined in the special `StateMachine` class.
- Instead of denoting state objects by references, state objects are referenced by 'reference methods' (much like factory methods), and these may then be overridden to reference state objects according to the state subclasses.

Using `IState`, `StateMachine` and `State` from Figure 5, the generic state machine `SM1` in Figure 4 can be implemented as shown in Figure 7. For illustration purposes, it is assumed that the state machine has a transition from state `A` to state `B` with event `a` as seen in Figure 3. In order to keep it simple, we have not included the constructors of the state classes, and we have not specified what should be the behaviour of `a` (and other event methods) in case they are delegated to the root `TopState`. Implementation of these methods will tell whether it is an error or simply a no-op to get such events in states where these events do not define any transition.

```
interface ISM1 extends IState {
  public void a();
}
class SM1 extends StateMachine implements ISM1 {
  SM1(){
    // generate state objects stateTopState, stateA, stateB
  }
  public void a(){(ISM1)cS.a()};

  class TopState extends State implements ISM1 {}

  class A extends State implements ISM1 {
    public void a(){changeState(stateB());}
  }

  class B extends State implements ISM1 {
    public void a(){}
  }

  // referencing methods for A and B
  ISM1 stateA(){ return stateA; }
  ISM1 stateB(){ return stateB; }
}
```

Figure 7 Generic implementation of a state machine

Using the same principles as above for specialization, and using method forwarding for composition, the state machine SM2 in Figure 4 can be implemented as shown in Figure 8. Here, the state `B` has to be extended in order to become a composite state (with `B1` and `B2` as contained states). Java does not support extension of classes, so we define ``B` `` as a subclass of the inherited `B`, and then override the corresponding referencing method for `B` to yield the state object for ``B` ``.

```
interface ISM2 extends ISM1 {
  public void b()
}
class SM2 extends SM1 implements ISM2 {
  SM2(){
    // generates state objects stateB`, stateB1, stateB2
  }
  public void b(){(ISM2)cS.b()};

  class B` extends B implements ISM2 {
    public void b(){enclState.b();}
  }

  class B1 extends State implements ISM2 {
    public void a(){ (ISM2)enclState.a(); }
    public void b(){ changeState(stateB2); }
  }

  class B2 extends State implements ISM2 {
    public void a(){ (ISM2)enclState.a(); }
    public void b(){}
  }

  //overriding referencing method for B
  IState stateB(){ return stateB`; }
}
```

Figure 8 Generic implementation of an extensible state machine with composite states

Example. Following the implementation design for MediaSwitch given in Figure 5, this can be implemented in Java as given in Figure 9.

The ‘`...`’ in the event methods represent the actions of the transition, followed by a specification of the next state. In case there is no method forwarding, then programmers of the event method simply use the `changeState` method, see e.g. `changeState(stateOn)` in the event method `on` in state `Off`. In case of method forwarding, programmers have to call the corresponding method on the enclosing state, see e.g. `(SwitchState)enclState.off()` in the event methods `off` in state `CD` and `TV`.

```
interface IMedia extends IState{
  public void on();
  public void off();
  public void mode();
}

class MediaSwitch extends StateMachine implements IMedia {
  MediaSwitch(){
    // constructor setting up state objects and their enclosing state object
    // state objects: stateOn, stateOff, stateCD, stateTV
  }
  public void on(){(IMedia)cS.on();}
  public void off(){(IMedia)cS.off();}
  public void mode(){(IMedia)cS.mode();}

  class SwitchState extends State implements IMedia {
    public void on(){} public void off(){} public void mode(){}
  }
  class On extends State implements IMedia {
    public void on(){}
    public void off(){ ...; changeState(stateOff); }
    public void mode(){}
  }
  class Off extends State implements IMedia {
    public void on(){ ...; changeState(stateOn); }
    public void off(){}
    public void mode(){}
  }
  class CD extends State implements IMedia {
    public void on(){}
    public void off(){ ...; (IMedia)enclState.off(); }
    public void mode(){ ...; changeState(stateTV); }
  }
  class TV extends State implements IMedia {
    public void on(){}
    public void off(){ ...; (IMedia)enclState.off(); }
    public void mode(){ ...; changeState(stateCD); }
  }
}
```

Figure 9 MediaSwitch by method forwarding

3.6 Resulting context

While a subclassing solution to composite states creates the composite states by making the state class/subclass hierarchy, the method forwarding solution specifies the state structure by a structure of state objects. The benefit of using forwarding in addition to subclassing for state machine specialization is that it is a well-known mechanism. Subclassing is a mechanism for specifying specialization and therefore a relationship between classes, while forwarding is a relationship between objects.

The above design pattern is based upon a more elaborate framework (Andresen, Møller-Pedersen et al. 2015). In order to support entry/exit actions and history, the design pattern above has to be combined with such a framework of predefined classes for `StateMachine` and the inner `State`. Entry/exit actions are in the framework defined as methods in the class `State`; these may then be overridden in specific states, and the framework will ensure that they are called in the right order when states are entered/exited. In order to support transition to history states, the framework ensures that each time a state is entered, the state is set as the shallow history of its immediate enclosing composite state. In order to support transition to deep history states, each time the current state changes, one will have to traverse the state hierarchy from the current state and up to the root state, and for all composite states on the path store current state as their deep history.

3.7 Related Patterns and related work

According to the classification scheme of (Noble 1998) our design pattern is an extension of the original state design pattern, in that it supports both composite states and specialization of state machines. In addition it may also easily become a framework (and thereby a partial implementation of state machines and states) by extending the class `StateMachine` with its inner class `State` with code to provide the support for history and for calling the entry/exit actions in the right order. Doing this just by applying a design pattern would be cumbersome and error prone.

(Dyson and Anderson 1998) present seven refinements/extensions of the original state design pattern. Our design pattern is also based upon the original state design pattern, so all of these seven refinements/extensions may be applied as well. Among the seven there is no extension for specialization of state machines.

In (Henney 2003), states are not represented by objects according to state classes, but by methods, or in fact by references to methods. This requires the language to support references to methods, and the approach will have difficulties in supporting specialization of state machines.

The only approach in the survey (Adamczyk 2003) that is similar to ours is the one called Subclassing State Machines (Sane and Cambell 1995). It describes how to specialize a state machine, and it composes state machines, but it does not support hierarchical state machine by means of composite states. They have the notion of composite state, but that is rather a state that stems from a composed state machine and therefore is e.g. a pair of states from each of the composed state machines.

The Pattern Language of Statecharts (Jacoub and Ammar 1998) is also similar to our approach in that it devise hierarchical statecharts by means of references, one from leaf states (state being part of another state) to its container states, and another (currentState) from a container state to the current state with the container. This last reference also facilitates History. However, specialization of state machines is not supported.

As described in the introduction, the original state design pattern does not cover composite states. Existing state machine APIs in various programming languages support full state machines.

Among the approaches that are integrated with existing language mechanisms (i.e. not design patterns), the Actor model (Hewitt, Bishop et al. 1973) was the first approach. Actors can change description (class) explicitly and thereby accepting a new set of messages. The Modes approach (Taivalsaari 1993) also belongs to the well-integrated approaches, and it is directed towards supporting state-oriented programming in that an object does not have to change its class, only its virtual method dispatch pointer.

State-Oriented Programming (Sterkin 2008) is very similar to our approach. It recognizes that states have to be defined by objects that are linked to represent state hierarchies, but does not use method forwarding.

The Typestate-Oriented Programming ((Aldrich, Sunshine et al. 2009), (Sunshine, Naden et al. 2011)) supported by the Plaid language is a quite different approach. It is in line with Modes approach in the sense that state mechanisms are well integrated in the language. However, it only supports simple states. The reason is that the main objective is to define a corresponding type system that will make it possible to check that objects behave in accordance to the constraints specified by state types.

4. DISCUSSION

The implementation described above is based upon the existing language mechanisms of Java. Although it works, it is cumbersome and error-prone to have to make subclasses of the states that shall be extended (in order to cope with new events or to be changed from a simple state to a composite state), and in addition override referencing methods correspondingly.

Extension of states means extension of state classes. A solution would therefore be to define the class `State` as a virtual class (Madsen as a virtual class (Madsen and Møller-Pedersen 1989), see

Figure 10. Virtual classes are supported by a number of languages ((Madsen, Møller-Pedersen et al. 1993), (Ernst 1999), (Aracic, Gasiunas et al. 2006), (Bracha, Ahé et al. 2010)), however in the following the idea is simply just sketched graphically. Composite states are still handled by method forwarding.

A virtual class is just like a virtual method: it must be an inner class, and in a subclass of the enclosing class it may be given a new definition. While a virtual method may be overridden (that is completely redefined, except for its signature), a virtual class can only be extended, as if making a subclass of the virtual class. The reason that virtual classes can only be extended is obvious: it must be ensured that references typed by a virtual class can only denote objects with at least the properties of the virtual class.

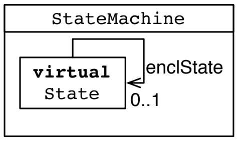

Figure 10 State as a virtual class in the framework

A specific state machine, e.g. the `Switch`, is then defined as a subclass of `StateMachine`, extending the virtual class `State` so that it implements the event methods for the switch (`on` and `off`), and define the states of `Switch` as subclasses of the extended `State` class, see Figure 11. The new subclasses of `State` are defined to be virtual classes as well, so that further specializations may extend them. Further specializations may therefore also redefine event methods for the states. The extended virtual class `State` in `Switch` is still virtual (although extended), so a further specialization of `Switch` may extend `State` in order to add new event methods.

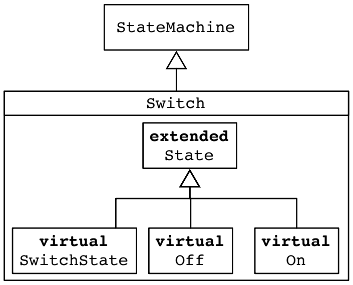

Figure 11 A specialized StateMachine with extended State and specific states

Figure 12 illustrates how the `MediaSwitch` is defined as a subclass of `Switch`. The class `State` is extended in order to implement the new event method `mode`, the states `CD` and `TV` are added as subclass of `State`, and `On` is extended in order to become a composite state. The fact that the state classes of a state machine are virtual classes implies that the construction of the state object hierarchy may be inherited and does not have to be made again for specialized state machines. As an example, the constructor for `Switch` in Figure 11 will have a statement that generates an `On` state object and sets the enclosing state to be an object of class `SwitchState`. The `MediaSwitch` state machine inherits this constructor, and as `On` has been extended, the inherited generation statement will now generate an object of the extended `On`.

In this respect a virtual class works the same way as a virtual method: like a call of virtual method implies a call of the overridden method in case the call is made in the context of a subclass, generation of an object of a virtual class will imply generation of an object of the extended class.

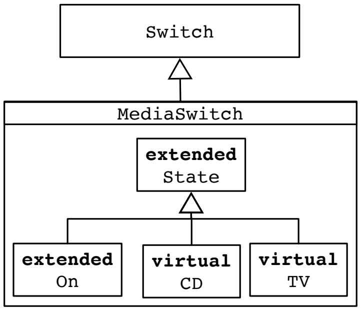

Figure 12 MediaSwitch as a specialization of Switch

As part of extending a virtual state class, it is possible to override inherited event methods. In principle an event method may be completely overridden, i.e. changing also the setting of the next state of the transition represented by the event method, and that is not desirable. A simple solution is to define the event methods as non-virtual (like final in Java) and then rather define for each event method a corresponding virtual action method that is called by the event method. This way a given event will lead to the same next state for all state machine specialization, while the action performed for this event may be tailored to the given specialization.

```
class SomeState extends State {
  public eAction(){/* action for event e */}
  public final e(){eAction();changeState(nextState)}
}
```

This way a given event will lead to the same next state for all state machine specialization, while the action performed for this event may be tailored to the given specialization.

REFERENCES

Adamczyk, P. (2003). *The Anthology of the Finite State Machine Design Patterns.* Pattern Languages of Program Design 2003 (PLoP'03).

Andresen, K., B. Møller-Pedersen and R. K. Runde (2015). *Combined Modelling and Programming Support for Composite States and Extensible State Machines.* MODELSWARD 2015 - 3rd International Conference on Model-Driven Engineering and Software Development, ESEO, Angers, Loire Valley, France, SciTePress.

Aracic, I., V. Gasiunas, M. Mezini and K. Ostermann (2006). *Overview of CaesarJ*. Transactions on AOSD I, LNCS, 3880: 135 – 173.

Bracha, G., P. Ahé, V. Bykov, Y. Kashai, W. Maddox and E. Miranda (2010). *Modules as Objects in Newspeak.* ECOOP 2010 – Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, Springer Berlin Heidelberg.

Chin, B. and T. Millstein (2008). *An Extensible State Machine Pattern for Interactive Applications.* ECOOP 2008.

Dyson, P. and B. Anderson (1998). State Patterns. Pattern Languages of Program Design 3. R. Martin , D. Riehle and F. Buschmann. Chichester, England:, John Wiley & Sons Ltd. Wiley.

Ernst, E. (1999). *gbeta - a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance.* PhD, University of Aarhus, Denmark.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Henney, K. (2003). *Methods for States*. First Nordic Conference of Pattern Languages of Programs (VikingPLoP 2002). Copenhagen, Denmark.

ITU (2011). *Z.100 series, Specification and Description Language SDL*.

Jacoub, S. M. and H. H. Ammar (1998). *A Pattern Language for Statecharts*. 5th Annual Conference on the Pattern Languages of Programs (PLoP '98). Illinois, US.

Madsen, O. L. and B. Møller-Pedersen (1989). *Virtual Classes—A Powerful Mechanism in Object-Oriented Programming.* OOPSLA'89 – Object-Oriented Programming, Systems Languages and Applications, New Orleans, Louisiana, ACM Press.

Madsen, O. L. and B. Møller-Pedersen (2010). *A Unified Approach to Modeling and Programming.* MoDELS 2010, Oslo, Springer.

Madsen, O. L., B. Møller-Pedersen and K. Nygaard (1993). *Object-Oriented Programming in the BETA Programming Language*, Addison Wesley.

Noble, J. (1998). *Classifying Relationships Between Object-Oriented Design Patterns*. Software Engineering Conference. Australia.

OMG (2015). *UML 2.5*.

Sane, A. and R. H. Cambell (1995). *Object-Oriented State Machines: Subclassing, Composition, Delegation and Genericity.* OOPSLA.

Taivalsaari, A. (1993). *Object-Oriented Programming with Modes*. Journal of Object-Oriented Programming **6**(3): 25-32.