Combined Modelling and Programming Support for Composite States and Extensible State Machines

Kjetil Andresen, Birger Møller-Pedersen and Ragnhild Kobro Runde
Department of Informatics, University of Oslo, Oslo, Norway

Keywords: State Machine, Design Pattern, Modelling, Programming.

Abstract: 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 specialization of extensible state machines. In this paper, we present 1) a state design pattern using delegation to support composite states and inheritance together with virtual classes to support extensible state machines, and 2) a framework, implemented in Java, which also supports history and entry/exit actions. Together, these form the basis for developing support for state machines in a combined modelling and programming language.

SCIENCE AND TECHNOLOGY PUBLICATIONS

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 central modelling concepts can be supported by programming language mechanisms, and vice versa. As one step in that direction, this paper reports from an investigation on programming language support for the modelling mechanisms of state machines, from simple state design patterns to the use of advanced programming language mechanisms. It has been an aim to rely as much as possible on existing mechanisms and not just introduce new mechanisms in order to support the combination.

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

The starting point for our approach is the *state design pattern* in (Gamma, Helm et al. 1995) which is commonly used when programming state machines. States are represented as objects of state classes, while events and their corresponding transitions are represented by *event methods*. The original state design pattern only supports simple state machines; there is e.g. no support for composite states, entry/exit actions and history.

Recent state design patterns support composite states. As already introduced in 1987 (Harel 1987) a composite state is a state with *substates* (contained states) such that all events and corresponding transitions that apply to the composite state by default apply to all of the substates, unless specified differently. State design patterns usually represent composite states by means of inheritance: classes for substates are defined as subclasses of the class for the composite state. The event methods of the composite state are therefore inherited, and event methods may be overridden for the substates where the default behaviour specified for the composite state shall not apply. This works for pure composite states, but cannot be used in combination with entry/exit actions on states, as entry/exit actions of the enclosing state should usually not be inherited.

Another development on state design patterns is the notion of *extensible state machine pattern* (Chin and Millstein 2008), which allows the extension (by inheritance) of state machines by adding states and events methods, and by overriding states. However, using inheritance for extension implies that this pattern does not support composite states.

In order to support both composite states and specialization of state machines we therefore pursue

Andresen K., Møller-Pedersen B. and Runde R..
Combined Modelling and Programming Support for Composite States and Extensible State Machines.
DOI: 10.5220/0005237302310238
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 231-238
ISBN: 978-989-758-083-3
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

the idea of representing composite states by state objects that are linked by *delegation*. Inheritance may then be used for specialization of state machines. With the delegation link from a substate object to the enclosing state object, an event method call on a substate that does not define this event method will be delegated to the composite state.

In order to support entry/exit actions and history, the design pattern above, with delegation and inheritance, has to be combined with a framework of predefined classes for e.g. states.

The remainder of this paper is organized as follows: Section 2 introduces our running example. In section 3 we introduce our framework. Section 4 describes how we support composite states. Section 5 describes how inheritance may be used to support specialization of state machines, by applying a pattern for extensible state machines to our composite state approach, and finally we show that a language with support for virtual classes would be the optimal for a combined modelling and programming language. Section 6 compares the resulting framework with related approaches, and section 7 concludes the paper.

2 MEDIA SWITCH EXAMPLE

For the purpose of illustrating our approach we use a simple state machine.

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 composite state `On` with its initial state `CD`. The mode is changed by the `mode` event. At any state in `On` the switch may be powered off (by the event `off`), entering the state `Off`.

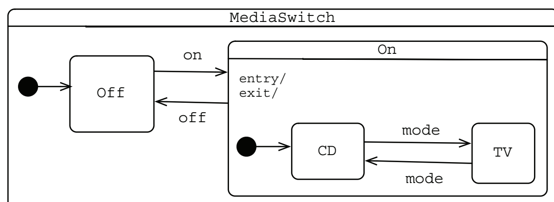


Figure 1: State Machine of a Media Switch.

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. Note that due to space limitations, we have not illustrated the use of history and entry/exit points, although they are supported. If e.g. the `on` transition from `Off` to `On` had targeted the history of `On` and not the `On` as such, then the first time it would enter the initial state `CD`, while subsequent `On` transitions from `Off` would enter the state that `On` were in (`CD` or `TV`) when it was turned off.

3 FRAMEWORK BASED UPON THE STATE DESIGN PATTERN

The original state design pattern (Gamma, Helm et al. 1995) represents each state as an object of a state class, with an *event method* for each event that applies to this state. The state machine has the same event methods, and calls of these are forwarded to the current state (maintained by a variable in the state machine), in order to be handled specifically for this state. An event method performs some action, followed by a *transition* to the next state by changing the current state variable.

The actual employment of the state design pattern depends on the mechanisms in the given language. We will assume that the language supports inner classes, so that the state classes can be defined as inner classes to the state machine class. In order to support our required mechanisms, we combine the state design pattern with a framework in terms of a pre-defined class `StateMachine` with an inner class `State`; a specific state machine is then defined by a subclass of `StateMachine` with subclasses of `State`.

The `MediaSwitch` state machine is defined as a subclass of `StateMachine` with a subclass (in our example `SwitchState`) of `State`, and finally specific states as subclasses of `SwitchState`, see Figure 2, here only illustrated with `Off` and `On`. Both `MediaSwitch` and `SwitchState` implements the event methods specific for this machine. The substates of `On` will be added to the example in the next section.

In this framework, the `StateMachine` class includes the method `cS` that returns the current state. All event method calls to a state machine shall be forwarded to the current state via the `cS` method. Without utilizing additional language mechanisms, `cS` would have to be typed by `State`:

```
abstract class StateMachine {
  private State cS(){...};
  ...
}
```

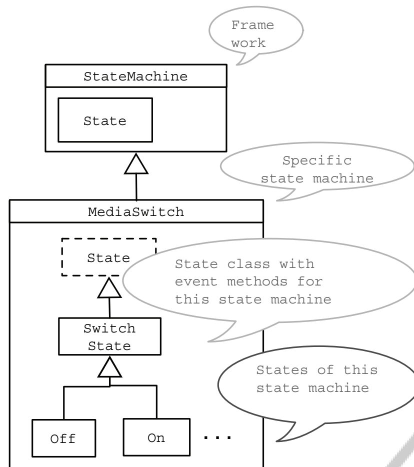


Figure 2: Use of the Framework.

The forwarding calls to the state maintained by `cS` would then have to be casted to the specific subclass of `State` in order to get access to the specific event methods defined in this subclass. As such type-casts are error-prone, we have in our framework instead made the class `StateMachine` generic with a type parameter `StateType` that represents the type of the states in the state machine, see Figure 3. `StateType` is a subtype of `IState` defining all the event methods. It is therefore based upon an interface `IState` that defines what is common to all states: entry/exit actions (in terms of methods) and an initial state (used in case of composite states).

```
interface IState {
  public void entry();
  public void exit();
  public Class<? extends IState> initialState();

}

abstract class StateMachine <StateType extends IState> {
  private StateType cS(){...}; //current state
  Class<? extends StateType> initialState();

  class State implements IState {
    public Class<? extends StateType> initialState() { ... }
  }
  ...
}
```

Figure 3: Generic State Machine Framework.

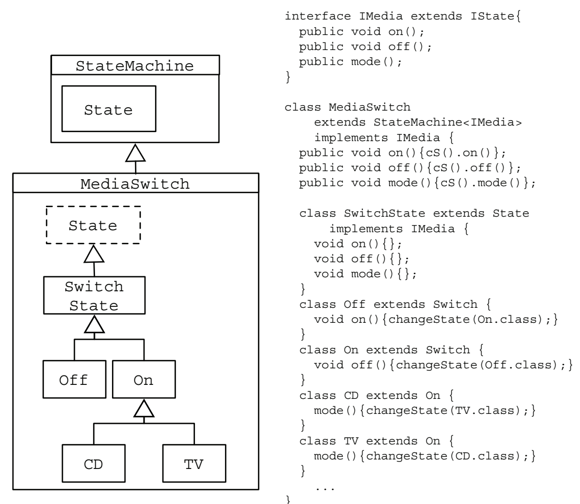
As the framework is implemented in Java with its support for generics, we represent states by means of `Class` objects. A `Class` object is the object that represents a class, so given the state class `On`, the expression `On.class` yields the `Class` object for the class `On`. The type of `initialState` is `Class<? extends StateType>`, as it shall be prepared for state classes that implement at least `IState`. For our media switch, the interface `IMedia` must therefore extend the `IState` interface and define the event methods that are specific for this state machine.

While a pure state design pattern approach would make the state machine class (including the inner state classes) specifically for the state machine at hand, the framework classes `StateMachine` and `State` provides the machinery for handling entry/exit actions, entry/exit points and history.

4 COMPOSITE STATES

4.1 Composite States by Inheritance

An obvious way of representing composite states in the state design pattern is to represent the substates by subclasses of the class for the composite state. Recall from above that for the composite state `On` the thing is that the event `off` shall apply to all substates (at any level) in `On` and perform the same transition (to `Off`). This is precisely what happens when the event method `off` is defined in class `On` and inherited (but not overridden) by the subclasses `CD` and `TV`, see Figure 4. Note that the graphical illustrations are just showing the class hierarchies, not the implemented interfaces and the generics – they are part of the corresponding code fragments.




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

class MediaSwitch
    extends StateMachine<IMedia>
    implements IMedia {
  public void on(){cS().on()};
  public void off(){cS().off()};
  public void mode(){cS().mode()};

  class SwitchState extends State
      implements IMedia {
    void on(){};
    void off(){};
    void mode(){};
  }
  class Off extends Switch {
    void on(){changeState(On.class);}
  }
  class On extends Switch {
    void off(){changeState(Off.class);}
  }
  class CD extends On {
    mode(){changeState(TV.class);}
  }
  class TV extends On {
    mode(){changeState(CD.class);}
  }
    ...
}
```

Figure 4: Implementation of the Media Switch state machine using inheritance.

Support for entry/exit points and shallow/deep history may be added to this approach to composite states. However, this approach does not work for entry and exit actions. The reason for this is that entry/exit actions of a composite state shall *not*

apply to the substates, while using inheritance for the substates will imply that they inherit also the entry/exit actions. 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. Existing approaches using inheritance for representing composite states do not provide a solution to the unwanted inheritance of entry/exit actions. In the following we will therefore pursue a different way of representing composite states.

4.2 Composite States by Delegation

In order to avoid the above-mentioned problem of entry/exit actions in combination with composite states by means of inheritance, we employ *delegation* instead of inheritance. With delegation, an event method call to a state is delegated to its enclosing (composite) state in case the event method is not defined specifically for the current state.

The benefit of using delegation is that it is a well-known mechanism; in addition we do not have to invent a mechanism just for the purpose of composite states. Delegation was first introduced in (Lieberman 1986) as a means to share behaviour specifications between objects in prototype-based languages, i.e. languages with only objects and not classes, and it is often used as an alternative to inheritance in prototype languages. Delegation is a mechanism that is often considered an alternative to inheritance, but taken literally there is no reason that a language may not support both inheritance and delegation. Inheritance is a mechanism for specifying specialization and therefore a relationship between classes, while delegation is a relationship between objects.

In the delegation approach to composite states, each substate object will have a delegation link to its composite state object, as illustrated for our media switch example in Figure 5. The SwitchState root state is now reached when event method calls are delegated to the root (i.e. not handled in any of the other states).

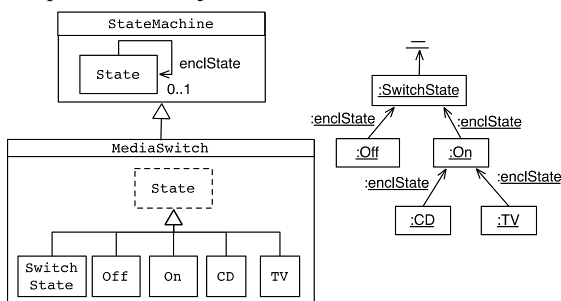


Figure 5: Class diagram and object diagram with delegation links for the MediaSwitch state machine.

While an inheritance approach creates the composite states by making the state class/subclass hierarchy, a delegation approach must specify the state structure. The delegation links in the right part of Figure 5 are links of a general delegation association from State to State. The composite state structure is set up as part of the constructors for the state classes. Each constructor gets a reference to the enclosing state object as a parameter. In Figure 6 it is demonstrated that the state CD will have On as it enclosing state (super(On.class) in the constructor for CD), while On will have the root state SwitchState as its encloser.

The delegation approach as illustrated in Figure 5 and Figure 6 is the simplest alternative, with all states being subclasses of State, and the state hierarchy being maintained by delegation links. State only provides the most basic framework methods, all state classes have to implement the appropriate interface, and users of the framework have to insert code that delegates a call of an event method (e.g. enclState().off() in the event method off in state CD in Figure 6) to a call on the enclosing state. In case a call of an event method shall not be delegated, the ordinary changeState method is used, e.g. changeState(On.class) in the event method on in state Off in Figure 6.

```
interface IMedia extends IState{
  public void on();
  // similar for off and mode
}

class MediaSwitch
    extends StateMachine<IMedia>
    implements IMedia {
  public void on(){cS().on()};
  // similar for off and mode

  class SwitchState extends State
      implements IMedia {
    void on(){};
    // similar for off and mode
  }
  class On extends State implements IMedia {
    On(){super(Switch.class)}
    void on(){ enclState().on(); }
    // similar for off and mode
  }
  class Off extends State implements IMedia {
    void on(){ changeState(On.class); }
    // similar for off and mode
  }
  class CD extends State implements IMedia {
    CD(){ super(On.class); }
    off(){ enclState().off(); }
    // similar for off and mode
  }
  ...
}
```

Figure 6: MediaSwitch by using delegation.

4.3 Framework Implementation

As Java does not support delegation, our framework implemented in Java[1] uses a delegation design pattern in order to represent composite states by delegation.

In the same way as the framework defines the class State as a superclass for all specific state classes, the framework also defines superclasses for entry/exit points and shallow/deep history.

Entry/exit actions are in the framework defined as methods in the class State; these may then be overridden in specific states, but the framework will ensure that they are called in the right order when states are entered/exited.

In order to support transition to shallow 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.

5 SPECIALIZATION OF STATE MACHINES

Modelling languages like SDL and UML have the notion of specialization of state machines. A special state machine may add states and events (with transitions), it may extend inherited states and it may override event methods from the general state machine. Recently, (Chin and Millstein 2008) has demonstrated how an extended notion of the state design pattern can support state machines that may be specialized, in (Chin and Millstein 2008) called *extensible state machines*. This is achieved by not using inheritance to specify composite states, but rather use inheritance to specify extension. The implication is that the state pattern only covers state machines with simple states and not composite states.

5.1 Extensible State Machine Design Pattern

In the following we apply this extensible state pattern to our delegation-supported composite states, and thereby we get a state design pattern that supports both composite states *and* specialization of state machines. We start out with a simple Switch state machine and then define the MediaSwitch state machine as a specialization of this, see Figure 7.

With the extensible state machine design pattern, a specialization is specified by defining a subclass of the enclosing state machine class. Extending a state so that it may handle additional events is done by adding a new inner state class that extends the corresponding class from the super state machine class. This is illustrated in Figure 7 with the On state class, as this has to handle the new event mode. In addition, the simple On state is extended to become a composite state.

The code for the simple Switch state machine is illustrated in Figure 8. State objects are created according to state classes. As an extended state machine makes subclasses of some of the inherited state classes, the extensible state pattern introduces the rule that state objects must be created by factory methods. These factory methods can then be overridden in order to have objects of the specialized state subclasses be generated instead of objects of the (original) state superclasses. Instead of referencing a state with name <state name> by means of '<state name>.class', it is therefore referenced by a factory method 'state<state name>, see Figure 8 for an illustration of this for state On (with factory method stateOn). The state class Off will have a similar factory method.

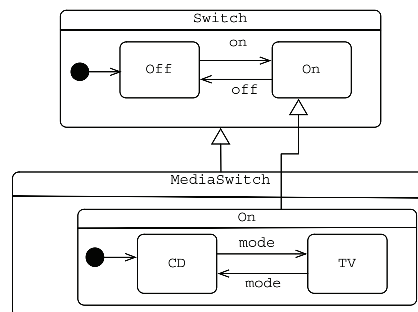


Figure 7: MediaSwitch as a specialization of Switch.

In the specialization of Switch in Figure 9, the states that are extended in order to accept new events or to become composite states are defined as subclasses of the corresponding state classes from the general state machine class. The corresponding factory methods have to be overridden to generate objects of these subclasses. As an example, the specialized state SpecialOn is a subclass of On (in order to introduce both new events and new

[1] The source code of the framework is available at http://folk.uio.no/kjetand/framework.zip

substates) and the factory method `stateOn` is overridden to reflect this extension.

Adding a new state to a specialized state machine implies the definition of a new state class as a subclass of `State`, the setting of its delegation link to its enclosing state, and a definition of the event methods that shall apply in this state; all other existing event method calls will be delegated to its enclosing state.

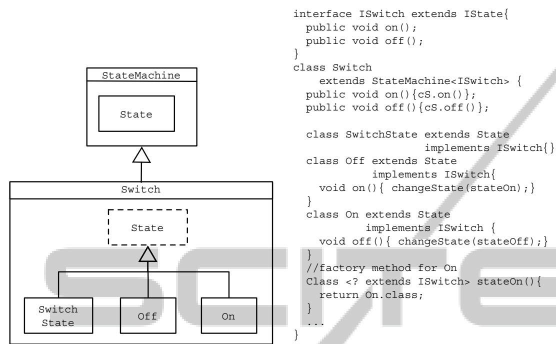


Figure 8: Switch as a subclass of StateMachine.

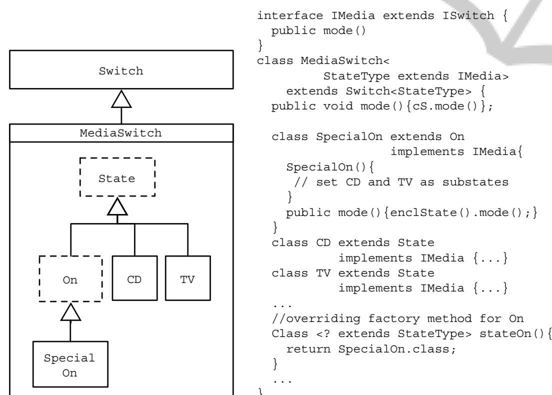


Figure 9: MediaSwitch as a specialization of Switch.

5.2 Language Support for Extensible State Machines

The extensible state machine design pattern is based upon the existing language mechanisms of Java (inheritance and generics), and we have shown above that this may be combined with our approach to composite states (by use of delegation). However, 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 factory methods correspondingly. In a combined modelling and programming language we would rather look for an existing programming language mechanism (as is the case with delegation and inheritance) that supports the extension of states.

Extension of states means extension of (inherited) states classes. The solution would therefore be to define the framework class `State` as a *virtual class* (Madsen and Møller-Pedersen 1989), see Figure 10. Composite states are still handled by delegation. 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.

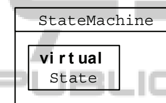


Figure 10: State as a virtual class in the framework.

A specific state machine, in our case 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 then define the states of switch as subclasses of the extended `State` class, see Figure 11.

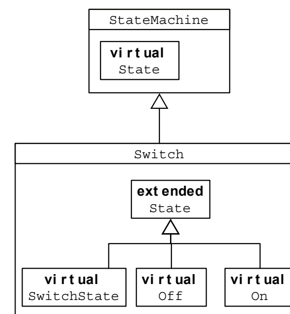


Figure 11: A specialized StateMachine with extended State and specific states.

The new subclasses of `State` are defined to be virtual classes as well, so that further specializations may extend them. 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. In order to be able to redefine event methods for given states and

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

events, the states of a state machine are also represented by virtual classes (see `On` and `Off` in Figure 11).

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 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 encloser 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 the extended class.

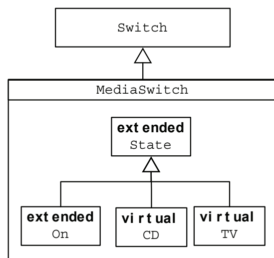


Figure 12: MediaSwith as a specialization of Switch.

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 next state of the transition, and that is not desirable. A simple solution is to define the event methods as non-virtual (final in Java) and then rather define for each event method a corresponding virtual action method that is called by the event method.

Java does not support virtual classes. While it is straightforward to obtain delegation by means of a design pattern, virtual classes are not that easy to do by a design pattern. The framework in Java therefore does not support the solution with virtual state classes. This is also the reason that this subsection just has illustrations of the solution, no code fragments. Except for the part of the code that expresses virtual classes and their extensions, the rest of the code, e.g. for the handling of events, will be the same as already described.

With virtual state classes there is no need for generics as described above. The `cS` would be typed by `State`, and along with extension of `State` in subclasses of `StateMachine` the type of `cS` is also extended.

6 RELATED WORK

As described in the introduction, the original state design pattern does not cover composite states. Most modelling languages have full support for state machines directly as language mechanisms. Existing state machine APIs in various programming languages also support full state machines, but without any attempts to integrate the state machine mechanisms with the mechanisms of language.

Among the approaches that are integrated with existing language mechanisms, the Actor model (Hewitt, Bishop et al. 1973) was the first approach. Actors can change state explicitly and thereby accepting a new set of messages. This idea has later been followed by proposals where an object may change its class and thereby the methods it will accept. 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. The solution in (Madsen 1999) takes the Modes approach a little further in that it supports composite states by means of state class inheritance.

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 delegation.

A quite different approach is taken by Typestate-Oriented Programming (Aldrich, Sunshine et al. 2009; Sunshine, Naden et al. 2011) supported by the Plaid language. It is in line with Modes and with our approach in 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.

None of the above approaches (except APIs, e.g. Sterkin 2008) have support for entry/exit actions or History.

7 CONCLUSIONS

While existing state design patterns either support composite states by inheritance of state classes (and then not specialization of state machines), or support specialization of state machines by inheritance (but then not support for composite states), our approach supports both composite states and specialization of state machines: delegation to handle composite states, and inheritance together with generics to support specialization of state machines.

Our approach is a combination of the state design pattern and a framework that handles history, entry/exit actions, and entry/exit points.

Our framework is implemented in Java. Therefore, delegation is handled using a delegation design pattern, and specialization is implemented, not using the most elegant solution with virtual classes, but instead by state subclasses and factory methods. A combined modeling and programming language with support for state machines according to our approach would call for a language including all of inheritance, delegation and virtual classes.

REFERENCES

Aldrich, J., J. Sunshine, S. Darpa and Z. Sparks (2009). Typestate-Oriented Programming. Onward ´09, OOPSLA´09, Orlando, Florida, USA.

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

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

Harel, D. (1987). "Statecharts: A visual formalism for complex systems." Science of Computer Programming 8(3).

Hewitt, C., P. Bishop and R. Steiger (1973). A Universal Modular Actor Formalism for Artificial Intelligence. International Joint Conference on Artificial Intelligence.

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

Lieberman, H. (1986). "Using prototypical objects to implement shared behavior in object-oriented systems." ACM SIGPLAN Notices 21(11): 214–223.

Madsen, O. L. (1999). "Towards integration of state machines and object-oriented languages." Technology of Object-Oriented Languages and Systems: 261–274.

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.

OMG (2011). UML - Unified Modelling Language.

Sterkin, A. (2008). State-Oriented Programming. Muliparadigm Programming with Object-Oriented Languages.

Sunshine, J., K. Naden, S. Stork, J. Aldrich and É. Tanter (2011). First-Class State Change in Plaid. OOPSLA’11, Portland, Oregon, USA, ACM.

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

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS