A Pattern Story for Combining Crosscutting Concern State Machines

Mark Mahoney
Carthage College
Kenosha, WI
mmahoney@carthage.edu

Tzilla Elrad
Illinois Institute of Technology
Chicago, IL
elrad@iit.edu

ABSTRACT

This paper describes a solution to a real world problem using a combination of well-known patterns. The problem deals with combining state machines that represent core concerns and crosscutting concerns in a loosely coupled manner. The state based behaviors are modeled with state machines and implemented with the State Pattern[3]. The coordination between the loosely coupled state machines is achieved with the Interceptor Pattern[9][11]. The Abstract Factory Pattern[3] is used to shield the original state machine developers from being aware that their state machines are being combined in new and different ways.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]: Modules and interfaces, Object-oriented design methods, State diagrams

General Terms
Design.

Keywords
Crosscutting Concerns, State Machines, Design Patterns.

1. INTRODUCTION

A pattern story describes the application of patterns to a specific design. This paper tells the story of the design of an application with core and crosscutting concerns. The concerns are state based and the patterns describe how to combine state machines in a manner that maximizes reusability and loose coupling.

Separating a software system into concerns is one way to deal with the increasing complexity of constructing large systems. However, not all concerns can easily be modularized. Some concerns crosscut others. A crosscutting concern is one that is scattered throughout a system and is tangled with other core application concerns. Fault tolerance, for example, is a crosscutting concern that is often tangled with many core application concerns. Aspect-Oriented Software Development (AOSD) [2] pro-vides a means to separate crosscutting concerns so that they can be reasoned about in isolation. It also provides the means to weave the crosscutting concerns into a set of core concerns to form a functioning system.

State machines are an excellent way to model reactive behavior. A state machine fully describes how an object or subsystem behaves in response to stimuli. State machines can easily be transformed into executable code using, for example, the State Pattern [3]. In addition, heavyweight tools such as Telelogic's Tau [13] can be used to build massively state based systems. State machine models typically do not cleanly separate the interaction between core and crosscutting concerns. There is a tendency to tangle concerns together in a single state machine. For example, in a banking application there may be state behavior in depositing into an account as well as separate state behavior for authentication and authorization. Traditional state based design techniques tend to mix these concerns together into the same state machine model even though the authentication and authorization behavior may be required in many other places in the system. A superior solution would allow the two independent reactive behaviors to be modeled separately and later be woven together. Each state machine would then be reusable in different contexts.

Once a set of state based core and crosscutting concerns have been separated into disparate state machines a mechanism is required to specify how they will interact. This weaving mechanism is currently not present in the most used state machine modeling languages. Our goal is to use a combination of patterns to create state based components that can easily interact in a loosely coupled manner. The state based behavior is implemented with the State Pattern [3] and the interaction between disparate implementations of the State Pattern is accomplished with the Interceptor Pattern [9][11]. The Abstract Factory Pattern [3] provides loose coupling in the cooperating state machines.

Using this approach will benefit developers who have recognized state based behaviors in the core and the crosscutting concerns. Traditionally, reactive systems are modeled with a set of state machines. Reactive systems tend to be embedded, distributed, or real-time in nature. However, as the size of non-reactive transformational systems get larger it is likely that some state based concerns will appear. Our pattern is targeted toward systems that are not entirely state based, but do have state based core and crosscutting concerns.

The rest of this paper is organized as follows: Section two describes the problem context, section three describes the forces, section four describes the solution, section five describes the forces resolved by our solution, and section six describes related work.

2. PROBLEM CONTEXT

Early in his career, the first author worked on wireless accessories for two-way radios that police officers and firefighters carry for communication. This section describes a simplified version of the devices. A two-way radio can be in an Idle state, a Transmit state (Tx), or a Receive State (Rx), see figure 1. Translation from a state machine model to an implementation of the State Pattern [3] is straightforward, see figure 2. An Abstract State class (TwoWayStates) is created from which all Concrete States (Rx, Idle, Tx) inherit. For each event in the state machine a method is placed in the Abstract State class. The derived Concrete State classes override the events that have meaning to them. A Context class (TwoWayRadio) creates and maintains a reference to each of the Concrete States and maintains a reference to the current state. The con-text object handles all events by sending them to the current state object.

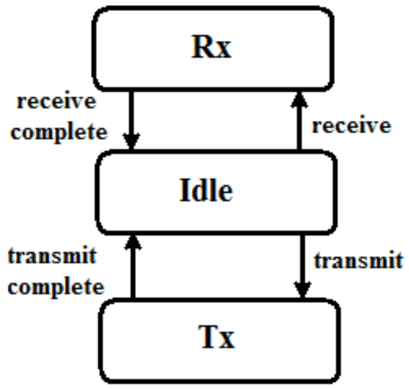


Figure 1. State Machine of a Two-Way Radio

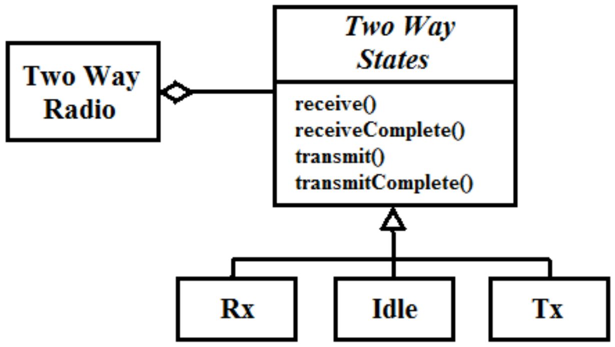


Figure 2. State Pattern Implementation for a Two-Way Radio

Imagine a state machine for a wireless accessory (perhaps using a technology like Bluetooth) that connects to the two-way radio wirelessly and includes both a speaker and a microphone to transmit and receive audio. The wireless accessory can be in an Idle state, a Connecting state, and an Audio state, see figure 3. The classes for the State Pattern implementation are shown in figure 4.

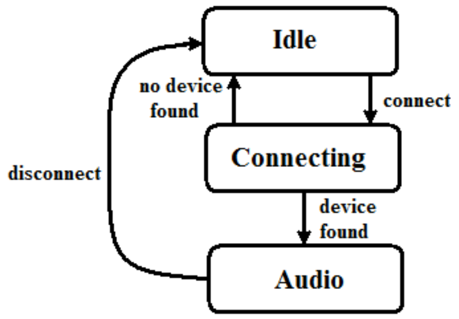


Figure 3. State Machine for Two-Way Radio Accessory

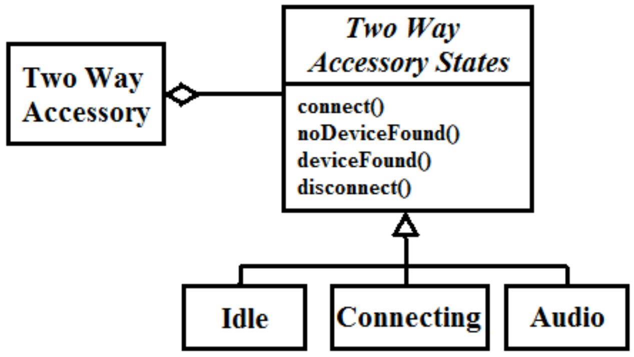


Figure 4. State Pattern Implementation for a Two-Way Radio Accessory

The control of the wireless accessory is a crosscutting concern because it must inter-act with the two-way radio in many different ways and in many different contexts. For example, when the two-way radio is in the 'Idle' state and a 'receive' event is received the wireless accessory must enter the 'Audio' State to transmit the two-way radio's audio to the speaker on the accessory. Similarly, when the two-way radio's battery falls below a certain threshold an audio alert is sent to the accessory and re-quires an audio connection. There are many times when the two-way radio's audio needs to be sent to the accessory.

3. FORCES

The forces influencing a solution to this problem have to do with the fact that the application is not entirely state based. Many of the requirements can be described as data-transformational in nature. Since the system is not entirely state based it doesn't make sense to use specialized tools [13][14] to create state machines. Using such tools might not even be possible in an embedded application environment. Rather, the solution should use standard Object-Oriented techniques to address the combination of state based concerns.

Ideally the solution will also allow the disparate state machines to be loosely coupled. Each state based concern should be reusable in different contexts. The two way radio, for example, should not be directly tied to the accessory because not every radio will have an accessory. Similarly, the audio accessory might be used with devices other than a two-way radio, like a cell phone. The combined state machines should not directly reference each other, rather, an intermediary should bind the state machines together.

Such an approach will be more complex but will allow for greater reuse.

4. SOLUTION

One can think of a state machine as the behavioral interface to a class, feature, or reactive subsystem. It is a metaphor for the subsystem. When the cooperating subsystems are also state based a method is required to compose them together. However, a desirable quality is to reduce coupling between the subsystems' state machines. In our previous work [7][8] we describe using state machines to implement reactive systems with crosscutting concerns. Each of the concerns is modeled with a state machine. The state machines can be used in isolation but they can also be brought together to share broadcast events. The events in the cooperating state machines are bound to each other to affect one another. Figure 5 shows two state machines with bound events.

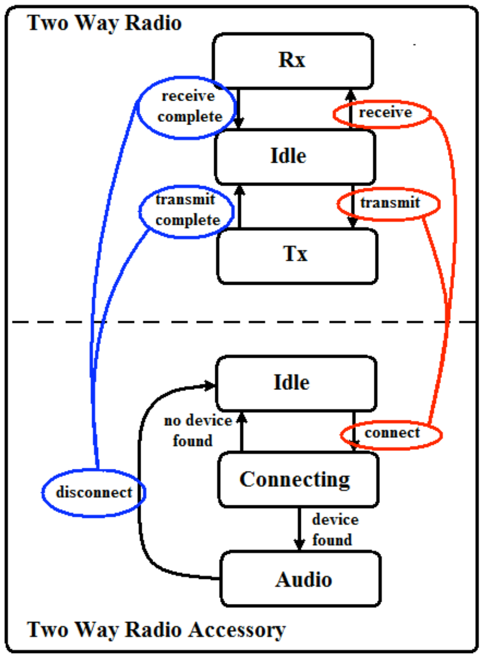


Figure 5. Combined State Machines with Event Bindings

For incoming audio, the 'receive' event in the two-way radio state machine is bound to the 'connect' event in the accessory state machine. The 'receive complete' event is bound to the 'disconnect' event. A similar approach is taken for outgoing audio. The primary benefit of this approach is that because neither state machine directly refers to the other, each one is reusable in different contexts. Only the weaving developer is aware of the interactions. One can imagine a developer creating complex systems by composing state machines from a library and simply specifying the bindings in a non-invasive manner.

In order to provide a means to combine independent state machine models and generate an executable system from them we propose using the State Pattern [3], the Interceptor Pattern [11] [9], and the Abstract Factory Pattern [3]. The State Pattern is used to create an executable implementation of a state machine while the Interceptor Pat-tern is used to coordinate the binding of events in different state machines. The Abstract Factory pattern is used to achieve obliviousness in the core state machine models.

The Interceptor Pattern [11] [9] allows one to monitor what an application is doing and add services transparently to a framework. We use Interceptor to monitor a core state machine and inject bound events into other state machines. The description from Bosak [9] varies slightly from pattern described in the POSA2 book [11], the structure of the pattern is shown in figure 6.

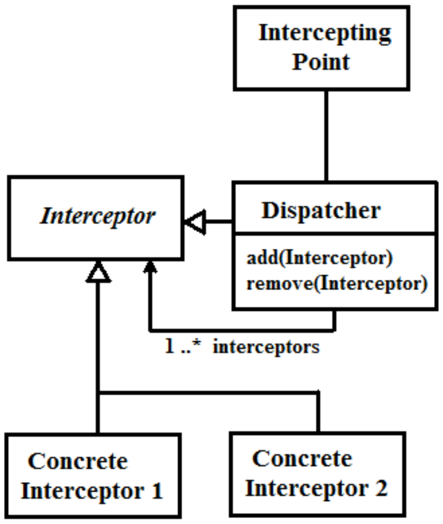


Figure 6. Interceptor Pattern Structure

In this variation of the pattern an Interceptor interface describes an object where behavior will be added before or after a call. The Interceptor Interface looks like an object that the Intercepting Point interacts within its domain. The core state machine's abstract state class will serve as the Interceptor interface in this example because it is necessary to know when to inject events in a crosscutting state machine. Concrete Interceptors implement the interface to provide additional services. The Concrete Interceptors will be responsible for notifying other state machines when certain events are handled.

The Dispatcher is responsible for coordinating the different calls to the Concrete Interceptors. It can accomplish this based on a priority for Interceptors or using some other intelligent scheme. Since some events are bound before, after, or in place of others the dispatcher provides the granularity needed to inject events at the right time. The Intercepting Point is associated with a Dispatcher and sends all requests to it. The State Pattern's context object will refer to Dispatchers rather than concrete State objects when binding occurs in those states. When a method from the Dispatcher is called the Dispatcher will call the associated methods of all the Concrete Interceptors associated with it.

State [3] and Interceptor [9][11] can be combined to allow independent state machines to interact, see figure 7. In this case the Abstract State from the State Pattern acts as the Interceptor interface. It has all the methods of a Concrete State and will act as a stand in when event binding is required. When combining state machines a weaving developer will inherit from the Abstract State class to create Concrete Interceptors. The State Pattern's Context object maintains a reference to the Dispatcher rather than the Concrete State object. When a bound event occurs the event is handled by the Dispatcher rather than the Concrete State object. The Dispatcher then coordinates the injecting of events in another state machine.

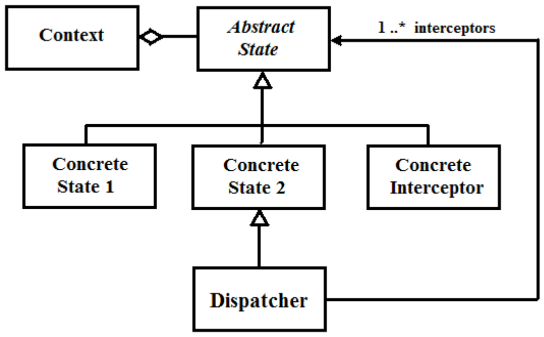
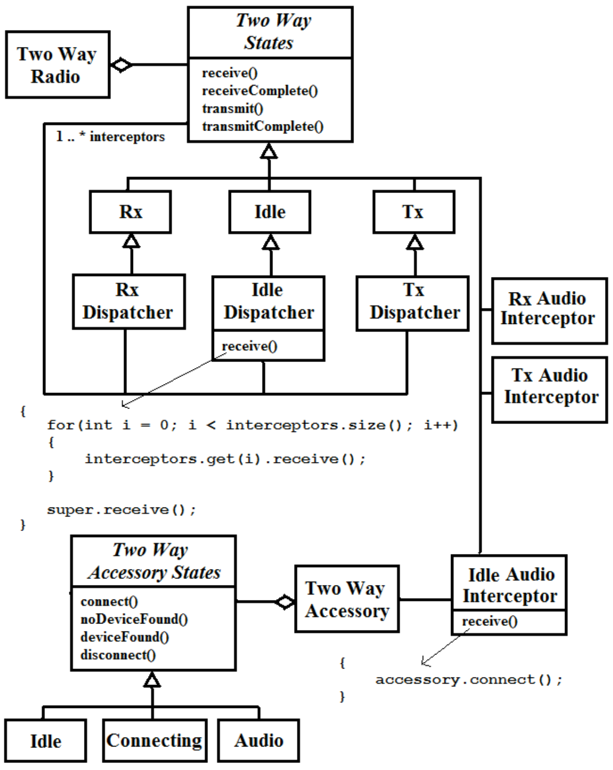


Figure 7. Combined State and Interceptor Patterns

To relate this approach to the example from above, the Two-Way Radio and Accessory state machines can be combined to bind the 'receive' event in the Two-Way Radio state machine to the 'connect' event in the Wireless Accessory state machine, see figure 8 and Listing 1.



```
{
  for(int i = 0; i < interceptors.size(); i++)
  {
     interceptors.get(i).receive();
  }

  super.receive();
}
```

```
{
   accessory.connect();
}
```

Figure 8. Combined Two-Way Radio and Wireless Accessory Implementation

```
public class IdleDispatcher extends Idle
{
  private List <TwoWayStates> interceptors;

  //...
  public void receive()
  {
    for(int i = 0;i < interceptors.size();i++)
    {
      interceptors.get(i).receive();
    }

    super.receive();
  }
  //similar for other events
  //...
}
```

```
public class IdleAudioInterceptor extends
TwoWayStates
{
  private TwoWayAccessory accessory;

  public IdleInterceptor(TwoWayAccessory a)
  {
    super(null);
    accessory = a;
  }
  public void receive()
  {
    accessory.connect();
  }
  public void transmit()
  {
    accessory.connect();
  }
}
```

Listing 1. Idle Dispatcher and Idle Interceptor

The key to making this an oblivious solution is using an Abstract Factory in the State Pattern's Context object to create State objects. The State Pattern's Context object uses a concrete factory to create the Dispatcher and Interceptors rather than a Concrete State class. The weaving developer is responsible for creating an implementation of a Concrete Binding Factory along with the Dispatcher and Concrete Interceptors, see figures 9 and 10 and Listings 2, 3, and 4.

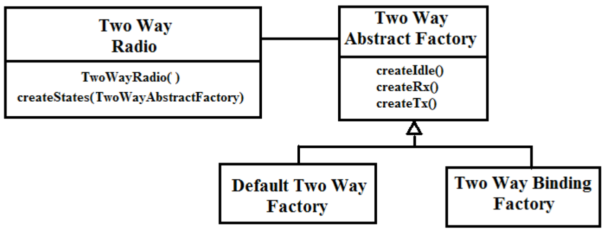


Figure 9. Abstract Factory Pattern

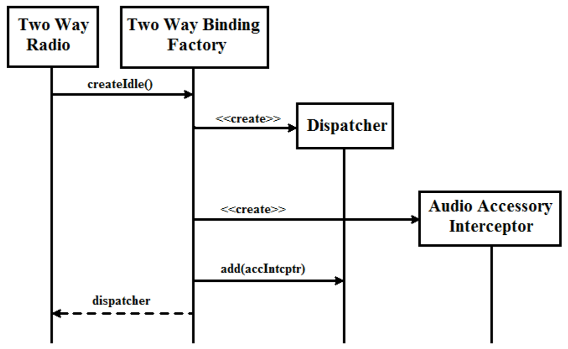


Figure 10. Sequence Diagrams for State Creation

```
public    class    TwoWayBindingFactory    extends
TwoWayAbstract-Factory
{
  //...
  //...
  public TwoWayStates createIdle()
  {
    //create an idle dispatcher
    IdleDispatcher idleDispatcher = new
                    IdleDispatcher(getContext());

    //add all interceptors
    idleDispatcher.addInterceptor(new
         IdleInterceptor(accessoryStatemachine));

    return idleDispatcher;
  }
  //...
  //...
}
```

Listing 2. Factory for Creating States

```
public class TwoWayRadio
{
  private TwoWayStates idle; //concrete idle state
  private TwoWayStates rx;   //concrete rx state
  private TwoWayStates tx;   //concrete tx state

  //current state in the state machine
  private TwoWayStates currentState;

  public  void  createStates(TwoWayAbstractFactory
factory)
  {
    //use the factory to create each of the states
    idle = factory.createIdle();
    rx = factory.createRx();
    tx = factory.createTx();
    //...
  }
}
```

Listing 3. Creating States in the Context Object

```
TwoWayRadio radio = new TwoWayRadio();

TwoWayAbstractFactory radioFactory = new
                  TwoWayBindingFactory(radio,
accessory);

radio.createStates(radioFactory);
```

Listing 4. Creating the Context Object

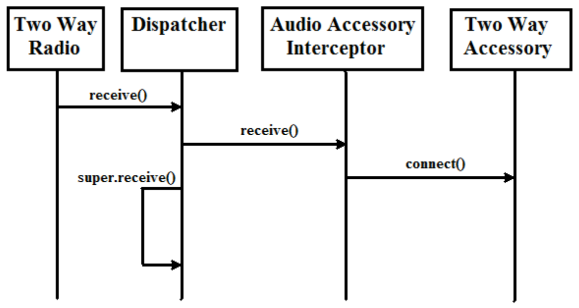


Figure 11. Sequence Diagram Showing the Two-Way Radio Interact with an Accessory Through an Interceptor

Here the two state machines are linked by the Dispatcher and a Concrete Interceptor used to bind ‘receive’ to ‘connect’. The Two-Way Radio Binding Factory object creates the Dispatcher and Audio Accessory Interceptor objects instead of the Concrete Idle State using a provided implementation of the Context’s Abstract Factory. The Two-Way Radio object treats the reference to the Dispatcher as if it were the Concrete Idle state. When the ‘receive’ event occurs in the Idle state the Two-Way Radio context object calls the Dispatcher’s receive() method, see figure 11. The Dispatcher then marches through all the Concrete Interceptors for this event in this state and calls receive() on those objects. The Idle Interceptor calls the Two-Way Accessory’s connect() method to inject the event into the state machine and then relies on the base class to do its normal processing by calling Idle’s receive() method (with a call to super.receive())

5. FORCES RESOLVED

The Interceptor Pattern [9][11] is ideal for allowing developers to add behavior into an existing system without requiring intimate knowledge of the system and without changing the base system. The Interceptor allows the State Pattern’s Concrete States to be extended in such a way that they can inject events into other state machines. When multiple state machines are combined the Dispatcher handles the coordination of injecting events. The Dispatcher may be built with a preference for handling certain interactions above others.

Other patterns related to this one that were considered were Decorator[3], Template Method[3], Chain of Responsibility[3], and Interceptor Filters[15]. Interceptor is very similar to the Decorator Pattern [3] but we were able to take advantage of inheritance of the concrete states to simplify the dispatcher. Interceptor provides more flexibility in the presence of multiple crosscutting state machines. Interceptor has just the right granularity to intelligently coordinate calls to multiple Concrete Interceptors.

Template Method [3] proved to be an inferior solution because it required all binding be done in one place. It is not easy to add and take away crosscutting behaviors with-out affecting the other state machines. Chain of Responsibility [3] proved to be an inferior solution because the crosscutting concern state machines would need to be aware of each other creating a tight coupling between them. Interceptor Filters [15] proved to be an inferior solution because it is slightly more complex than Interceptor. Interceptor proved to be the simplest solution that worked.

The Abstract Factory Pattern [3] allows the State Pattern's Context object to be oblivious to whether concrete states are being created or Interceptors to inject events in other state machines. The weaving developer is responsible for creating implementations of the Abstract Factory to create the correct objects.

This approach works best for systems that are not predominantly state based. In some telecommunication and avionic systems the predominant decomposition technique is to break the system down entirely into state machines. In a massively state based system the overhead and complexity of the design would warrant using a different approach. This method is designed for systems that encounter state based concerns but are not defined by them. Systems that are defined by massive state machines tend to have special tools and languages to help create them[13][14].

6. RELATED WORK

In our previous work [7][8] we implemented a framework for dealing with state based crosscutting concerns. This framework is called the Aspect-Oriented Statechart Framework (AOSF). There are classes in this framework for states, events, state ma-chine containers, etc. This framework does not make use of any patterns, is somewhat complex, and is language dependent. For these reasons we are proposing a more straightforward solution that still allows a developer to combine state machines in a loosely coupled, reusable fashion.

Volter [12] describes a general method of achieving AOSD using a patterns-based approach. In this work the use of interceptors, proxies, and factories is used to de-scribe how to achieve some of the same goals as AOP. We are extending that work a step further by apply those principles directly to state based AOSD.

In Aldawud et. al. [1] a similar approach for handling state based crosscutting concerns is addressed. In that work, different state machines model different concerns. The state machines are brought together in concurrent, orthogonal regions, however, the broadcast events are explicitly hard coded between disparate state machines making each model tightly coupled with each other and not reusable in different contexts.

In the work of Prehofer [10] feature composition is addressed for state based features. Each feature is modeled with a state machine and combined using one of two different approaches. In the first approach separate state machine models are combined in to a single model containing all the behavior by binding transitions. This leads to tangled state machine models that are hard to reason about. In the other proposed approach a similar method combining concurrent state machines is proposed with explicitly shared broadcasted events. This eliminates the reusability of the state machines in isolation or in different contexts. The author's implementation strategy did not use patterns and relied on a language specific pre-processing tool.

In France et. al. [5] State Machine Pattern Specifications model state machine interactions. The approach involves specifying binding statements that compose state machines together. The abstract state machines are not usable in isolation and the com-posed state machines are tangled and difficult to reason about

7. CONCLUSION

The State Pattern is ideal for creating implementations of state based behavior from a state machine. The problem with the it, however, is there is no easy way to combine state machines while keeping them loosely coupled and reusable. Our contribution is to provide a language/framework independent approach to loosely coupled state machines. Loosely coupled state machines can be reused in different contexts. Further, because no new languages or frameworks are required this approach can be used in legacy systems with no additional tool support.

We have described an approach to state based Aspect-Orientation that involves only the use of well-known patterns. The State Pattern is used for implementing state based behavior. The Interceptor Pattern coordinates event binding between state machines. The Abstract Factory permits core concern developers to be oblivious to additions made to their state machines. A concrete example was given describing use of the combination of patterns.

8. ACKNOWLEDGMENTS

We would like to thank Michael Weiss for helping us through the shepherding process with several iterations of useful and insightful comments. In addition, we would like to thank the 'Fu Dog' group at the writer's workshop at PLOP '07 for providing excellent feedback.

9. REFERENCES

[1] Aldawud, O., Elrad T., Bader A.. "Aspect-oriented Modeling- Bridging the Gap Between Design and Implementation". Proceedings of the First ACM SIGPLAN/SIGSOFT International Conference on Generative Programming and Component Engineering (GPCE). Pittsburgh, PA. October 6–8, 2002, pp. 189-202.

[2] AOSD web site: http://aosd.net.

[3] Gamma, Helm, Johnson, Vlissides; Design Patterns, Elements of Reusable Software Design, Addison-Wesley 1995.

[4] Filman R.E., Friedman, D.P. "Aspect-Oriented Programming is Quantification and Obliviousness", Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis.

[5] R. France, D. Kim, S. Ghosh, and E. Song, "A UML-Based Pattern Specification Technique," IEEE Transactions on Software Engineering, vol. 30, pp. 193-2006, 2004.

[6] Kiczales, G. et al., Aspect-Oriented Programming. Proc. European Conf. Object-Oriented Programming, Lecture Notes in Computer Science, no. 1241, Springer-Verlag, Berlin, June 1997, pp. 220–242

[7] Mahoney, M., Elrad, T. A Pattern Based Approach to Aspect-Orientation for State Based Systems, Workshop on Best Practices in Applying Aspect-Oriented Software Development (BPAOSD ' 07) at the Sixth International Conference on Aspect-Oriented Software Development (AOSD 2007). March 2007. Vancouver, BC.

[8] Mark Mahoney, Atef Bader, Tzilla Elrad, Omar Aldawud, Using Aspects to Abstract and Modularize Statecharts, The 5th Aspect-Oriented Modeling Workshop in Conjunction with UML 2004 Lisbon, Portugal, October 2004

[9] Bosak, R., Daily Development Blog. http://dailydevelopment.blogspot.com/2007/04/interceptor-design-pattern.html. April 2007

[10] C. Prehofer, Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams, International Workshop on Feature Interaction in Telecommunications and Software Systems, Ottawa, Canada, June, 2003, IOS Press

[11] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects Volume 2. Wiley and Sons Ltd., 2000

[12] Volter, M. Patterns for Handling Cross-Cutting Concerns in Model-Driven Software Development, In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee, Germany, July 2005

[13] Telelogic Tau, http://telelogic.com

[14] ILogix Rhapsody, http://telelogic.com

[15] Core J2EE Patterns - Interceptor Filters, Core J2EE Pattern Catalog, http://java.sun.com/blueprints/corej2eepatterns/Patterns/InterceptingFilter.html