Using Aspects to Abstract and Modularize Statecharts

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

Atef Bader
Lucent Technologies
Naperville, IL
abader@lucent.com

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

Omar Aldawud
Lucent Technologies
Naperville, IL
oaldawud@lucent.com

ABSTRACT

The statechart modeling mechanism is an essential element in the UML standard. Software engineers use statecharts to capture dynamic behaviors of objects and their interactions. In this paper we demonstrate how abstracting statecharts can help in building models that are easy to maintain and extend. Through the use of aspect-oriented techniques, we can build orthogonal abstract statecharts that can be reused in different contexts. Adding statechart submodels into the core statechart submodel involves a weaving process that a modeler can utilize during the design phase. This paper attempts to demonstrate the benefits of employing aspect-oriented techniques in the pursuit of abstracting statecharts.

Keywords

Aspect-oriented software development, statecharts, UML.

1. INTRODUCTION

Separation of concerns is a fundamental software engineering principle that has been addressed since the early days of the programming discipline [1]. Separation of concerns is perceived as an essential principle to decompose software systems into smaller modules in order to improve comprehensibility, maintainability, and adaptability of software systems. A crosscutting concern is one that is not easily modularized into a modular language construct. Tangling is when two concerns are implemented in such a way that their codes are intermixed, making it hard to distinguish the implementation of each concern separately. Aspect-oriented technology [2,3,19] focuses on the identification and modular representation of crosscutting concerns to generate 'single-concerned' software components that can be developed in isolation from each other and at a later stage be woven together to produce a fully operational software system.

The main rationale behind modeling software systems is to raise the level of abstraction and focus on capturing and understanding software requirements early in the analysis and design phase of the software lifecycle. UML [12] and object-oriented modeling techniques provide the user with a rich set of constructs and design principles to model object-oriented software systems. However, designers often find it difficult to map certain requirements (logging, security, etc.) to a single design concern. These requirements end up being scattered across many different core concerns. It is necessary to recognize and identify the interactions between crosscutting concerns and core concerns before we attempt to express them explicitly in submodels.

Statecharts [5] provide a mechanism to model and capture the dynamic behavior of objects using extended finite state machine concepts. A statechart is attached to a class that specifies all behavioral aspects of the objects in that class. When a crosscutting concern is tangled with a core concern the statechart that represents the core contains overlapping concerns which are hard to maintain and comprehend. In other words, standard statecharts do not resolve the issue of the misalignment of requirements, design, and implementation since requirements that belong to different concerns may crosscut different design model elements. The tangling problem originally discovered at the programming level cannot be avoided at the design level with current statechart modeling techniques and practices. Therefore, a process to identify and separate requirements into their prospective model elements will greatly simplify system design and comprehensibility.

The essence of aspect-oriented modeling [11] is to build on object-oriented modeling by enhancing it with the ability to model crosscutting concerns separately. The constructed submodels simplify the verification and validation of the constructed software system and ease maintenance and extension of the submodels. The system's core functionality is represented by a submodel, and each of the crosscutting concerns can be represented by a separate submodel. Aspect-oriented modeling can help in bridging the gap between software design and implementation through the use of advanced features of UML/Statecharts. The main deficiency in current aspect-oriented modeling techniques [11] however, is a lack of abstraction that would allow for the reuse of crosscutting concern design elements.

This paper describes a framework and design methodology that extends [11] to permit an object's behavior to be captured in a statechart according to its individual concerns, and a mechanism for weaving crosscutting concerns (modeled in one or more separate statecharts) together into the rest of the system. The framework takes one or more statechart designs along with the declarations of how they should be woven and generates executable code from them. Abstraction capabilities are provided to allow crosscutting concern design elements to be reused with other core concerns in other applications. Our approach uses a declarative method of event reinterpretation between orthogonal statecharts that are instructed how to interpret each others events in order to treat foreign events as if they were native events.

The rest of this paper is organized as follows. Section 2 describes classical behavioral modeling with statecharts. Section 3 describes current aspect-oriented modeling techniques with statecharts and their strengths and weaknesses. Section 4 describes a proof of concept framework that addresses the weaknesses from Section 3. Section 5 goes through an example using the framework. Section 6 discusses related work. Section 7 states the conclusions we found in our work.

2. BEHAVIORAL MODELING WITH STATECHARTS

Statecharts are a tool used to model the dynamic behavior of a class. A statechart is an effective design tool when the decision on what action to take in response to a given input is dependent not only on the input, but also on the current state of the system. An object's behavior can be decomposed into states. A state captures a system's history while abstracting away unhandled outside stimuli. An event handled in one state may cause a completely different reaction than the same event in another state. A statechart specifies three things: the stable states in which an object may live, the events that trigger a transition from state to state, and the actions that occur on each state change [13].

The basic principle in a reactive system is [5,20,21]:

"When event *a* in occurs in state *A*, if condition *C* is true at the time, the system transfers to state *B*"

States are drawn as rounded rectangles in statecharts.

Events are stimuli in the system that may cause transitions between states, they are drawn as directed arrows with a label specifying the event name. An optional condition may be included that is required to evaluate to true before a transition can take place. Conditions are placed in a set of '[' ']' after the event name. Actions are executable sequences that can be associated with transitions between states, they are specified with a '/':

event[conditional guard]/action

Upon entering or exiting a state, an action may be invoked. The actions associated with the entrance to, and exit from states are not event dependent, they happen regardless of how a state is entered or exited. A statechart must have a starting state indicated with a transition emanating from a large dot.

A simple statechart is said to have an 'exclusive-or' relationship between the states. This 'Or-composition' means that a statechart may be in one, and only one, state at a time. A statechart may also exhibit 'And-composition' by employing orthogonal regions. A statechart with orthogonal regions is composed of two or more independent statecharts. Being in a statechart with orthogonal regions means that in every region, one and only one, of the states from each composed statechart will be active. Orthogonal regions allow you to avoid intermixing the independent behaviors as a cross product and, instead, to keep them separate [15].

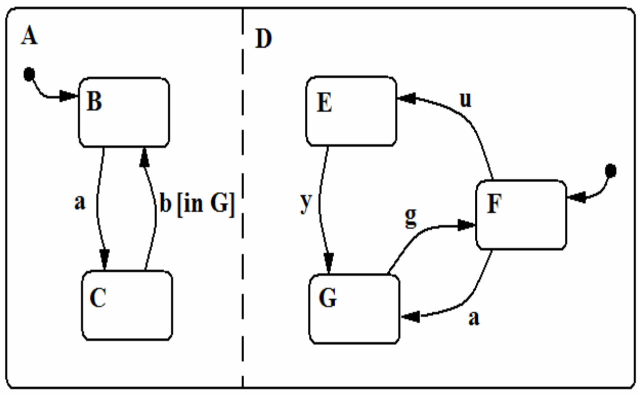

Figure 1. Statechart with orthogonal regions from Harel's 'A Visual Formalism for Complex Systems' [14]

For example, when the statechart in Figure 1 is entered the two composed statecharts (separated by a dashed line) will both be active and one state from each will always be selected, that is, there is still 'Or-composition' among each of the individual statecharts. A key feature of orthogonal regions is that events from every composed statechart are broadcast to all others. Therefore, an event can cause transitions in two or more orthogonal statecharts simultaneously. If event 'a' in Figure 1 is introduced while the active states are B and F, a transition will occur from B to C, as well as from F to G.

The composed statecharts are allowed to communicate through these broadcast events. Orthogonality is a way to avoid an explosion in the number of possible states. If this technique wasn't employed in Figure 1 then the system would require the states BE, BG, BF, CE, CG, CF or the cross product of all the states in all the statecharts. Instead, with the use of orthogonal regions, the number of states is simply the sum of all the states in all the statecharts. Each subsystem can be in its own orthogonal region as a way to decompose a complex system.

Statecharts are an elegant way of expressing the behavior of objects by means of states, events, and actions based on extended finite state machine concepts. These statecharts, used for designing behavior in objects, can even be the basis for executable code as described in [15]. However, like most design methodologies, it is possible to have concerns that do not easily fit into a single logical unit. Statecharts are no exception.

Consider the bounded buffer example. There are different concerns for a bounded buffer, such as synchronization and scheduling, which crosscut the core requirements of adding and removing elements from the buffer. Without aspect-orientation these concerns would be tangled with the core functionality. The tangled crosscutting concerns make it difficult to reason about all three concerns in isolation.

3. ASPECT-ORIENTED DESIGN

Aspect-oriented software development (AOSD) strives to keep concerns separate at all levels of the development process. Separation of concerns promises to reduce the complexity of system development by decomposing the system into cohesive units. Quantified programming statements are ones that have an effect on many loci in the underlying code. Aspect-oriented programming can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers [18]. With AOSD, all concerns are separated and developed in isolation, then the crosscutting concerns are woven together with the core concerns using quantified statements about how to interact with the core.

Currently, the Unified Modeling Language (UML) [5] is the language of choice when modeling systems using statecharts. In our recent work [11], we have proposed an aspect-oriented design technique that uses UML/Statecharts to simplify the design of object-oriented software systems. The approach provides developers with a methodology to create semi-independent statecharts that broadcast events to orthogonal regions that represent core and aspect behaviors.

In the design methodology laid out in [11] both core concerns and crosscutting concerns are modeled as static classes. Each class is then assigned a statechart that describes the dynamic behavior of the class. Associations between core and aspect classes form the relationships necessary for the classes, and thus the statecharts, to communicate. Each core and crosscutting concern becomes one orthogonal region in a statechart. This is how implicit weaving is accomplished. Each orthogonal region will broadcast events and receive events from other orthogonal regions. Because class diagrams and statecharts are used to fully describe objects it is possible to take these descriptions and generate code from them.

The process prescribed by the methodology from [11] consists of nine steps. In the first step of the process the developer identifies core objects in the domain. These often are the easiest to discover because they are exactly the same objects that would be candidates in a non-aspect-oriented approach. These objects represent core concerns and will be referred to as ‘the core’ of the system.

The next step is to identify the crosscutting concerns in the system and classify them into aspect objects. This is an important step because this is where the developer attempts to find and modularize concerns that typically get scattered across many core concerns. This step may need to be repeated many times during the development of the system. Often, crosscutting concerns are not as easy to identify when developers are first specifying a system. As developers knowledge of the system grows, so too does their awareness and ability to identify crosscutting concerns. These objects will be referred to as ‘aspects’.

The third step is to create associations between the core and aspect classes that dictate how communication will be achieved between the core and aspect classes. In this methodology this is equivalent to specifying the temporal order that events will be propagated from statecharts in orthogonal regions. That is, the structure of the relationships between the classes specifies the order that events will move through the statecharts. This allows aspect statecharts to access an event either before or after the core statecharts. Creating the associations specifies that order. The primary reason the associations between classes determines the order that statecharts handle events is because the authors wanted to make use of existing CASE tools. Using this approach amounted to a lightweight extension of UML.

The fourth step is to model each core and aspect object as a formal class. Next in step five a class diagram is built using the output from the previous steps. Step six involves identifying the states, transitions, and dependencies between the composed statecharts for each of the classes. States and transitions are identified for each class and dependencies between class’s states are identified. Step seven models the output from step six into formal statecharts associated with each of the classes. Each class’s statechart is composed into orthogonal regions.

The dependencies between individual classes’ statecharts are hard coded events that need to be propagated to achieve communication between the disparate statecharts. This is accomplished in step eight. Finally, step nine is to repeat the process steps one through eight throughout the rest of the development process as other crosscutting concerns are discovered.

The benefit of this process is to avoid creating incredibly complex statecharts with core and crosscutting concerns tangled together. In a tangled implementation, crosscutting concerns are usually handled by using complex sets of guard conditions that block transitions to states. For example, a bounded buffer statechart that handles synchronization of the buffer may have additional guards at each state specifying whether a transition can be made if the buffer is full, empty, or partially full. This way of specifying behavior is very difficult to reason about and is incredibly difficult to maintain. The tangled buffer statechart diagram loses its simplicity and elegance once synchronization is introduced.

In Figure 2, the process is applied to the bounded buffer problem. One can create a ‘Synchronization’ statechart and a ‘Scheduling’ statechart and put them in orthogonal regions with the core ‘Buffer’ statechart and the result is a protected ‘Bounded Buffer’ statechart.

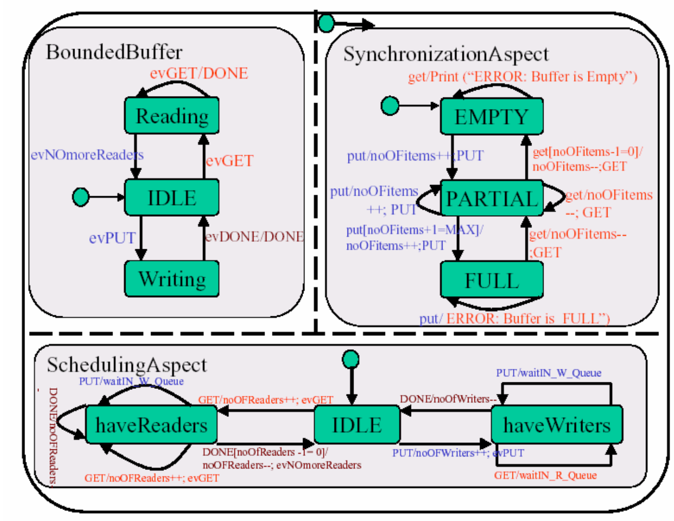

Figure 2. Solution to the bounded buffer problem with scheduling and synchronization aspects and hard coded event translation from [11]

Using this approach in Figure 2, when a 'GET' event is received in the scheduling aspect an 'evGet' event is propagated to the buffer. The novelty of this approach is the development of additional statecharts to handle crosscutting concerns. These additional statecharts introduce events that are broadcast to other statecharts in the other orthogonal regions creating a sequence of event transitions that protect the buffer. The main drawback of this approach, however, is the explicit event propagation performed by the developer.

This approach requires that the developer hard code the propagation of events to specify the order needed to protect the buffer. There is tight coupling between the core and aspect statechart and a lack of obliviousness resulting in the core developer and the aspect developer being aware of each other's state transitions and events. If quantification and obliviousness could be introduced into such a system the level of abstraction would be raised significantly and reuse of core and aspect classes would be possible.

Other than the coupling problem the solution is quite elegant. Each concern is encapsulated and abstracted into its own first class unit and the concerns are implicitly woven together using the broadcast events. Our approach extends the ideas in [11] by replacing hard coded event propagations with high level declarations about how an event in one statechart can be treated like a completely different event in another statechart. An aspect-oriented statechart framework was created to implement these ideas in a proof of concept tool. With the framework, a developer can take two or more independently developed executable statechart objects and weave them together with some high level declarations about how the statecharts crosscut each other and which events shall be reinterpreted in the orthogonal regions. This can be done without hard coding events in the submodels. Our approach increases the level of abstraction and makes it easier to develop complex systems that have a tyranny of crosscutting concerns.

4. ASPECT-ORIENTED STATECHART FRAMEWORK

We have taken a step forward from the work described in [11] by designing and implementing a framework that permits a statechart design to be translated into skeleton code for a class. In addition, two or more of these statechart objects may be joined to create orthogonal regions. Next, we permit crosscutting statecharts in orthogonal regions to be woven together by specifying which events in one statechart shall be reinterpreted to have meaning in another. This eliminates the need for hard coding of event propagations. Event reinterpretation is done obliviously to both the core and aspect statechart designer.

Libraries of core and aspect statecharts can be developed concurrently and independently, and combined only when needed for a particular application. For example, a 'Synchronization' aspect statechart class may be developed independent of any core functionality statechart. A weaving designer can reuse the 'Synchronization' aspect for any system that requires synchronized access to a resource. The weaving designer only needs to specify which statechart to crosscut and which events in the core to interpret as requests to access a resource.

The proof of concept framework has been implemented in Java and consists of classes that represent events, states, statecharts, event handlers, orthogonal regions, and orthogonal region containers. To create an executable statechart that represents a core or crosscutting concern one inherits from the framework's Statechart class. State objects are added to the Statechart object and transitions between states are specified with Statechart methods. Event handler objects have methods that are called in response to transitions between states, they represent actions. A table of event handler objects is created and the framework manages the calling of the appropriate methods based on receiving an event in a given state. The region container maintains an event queue and dispatches events to all the orthogonal regions in such a way that all the statecharts see the broadcast event before the next event is handled.

The handling of crosscutting concerns is achieved by specifying that one statechart crosscuts another. This is achieved by a Statechart method call that takes another Statechart as a parameter. The effect is that the crosscutting statechart is placed in an orthogonal region with the core statechart. Statecharts that crosscut one another interact by examining broadcast events and treating them as native events. The mapping between events in disparate statecharts is performed by the weaving developer.

Events are intercepted and 'reinterpreted' from one event type in the core to another event type in the aspect statechart. This causes transitions to take place in both the core and the aspect statecharts. Further, the order of aspect state transitions and statechart event handlers can be specified. One can choose to have an aspect statechart's state transition occur before or after the core state transition. This solves the problem of event ordering that is introduced by eliminating hard coded event propagation from [11]. The benefit of this is that the aspect event handler can alter, consume, replace, or introduce new events before or after the core statechart handles an event.

The declaration of event reinterpretation can be described by stating that (see Figure 3):

If the core statechart is in State 'X' and event 'y' is introduced, and if the aspect statechart is in State 'A' treat 'y' exactly as if it were event 'b', and allow the aspect statechart to make the transition either before or after the core.

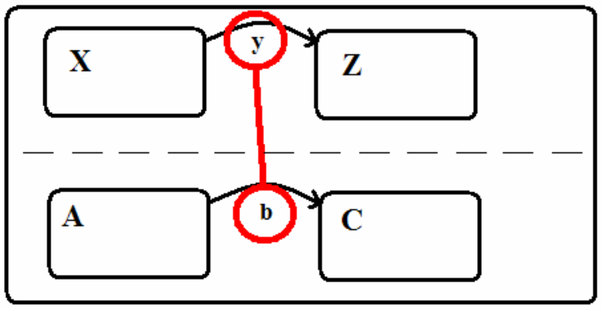

Figure 3. Event reinterpretation

5. EXAMPLES USING THE FRAMEWORK

Perhaps the best way to describe the framework is to look at a solution to a problem using it. Imagine a communication protocol represented as a statechart. Figure 4 presents the statechart diagram.

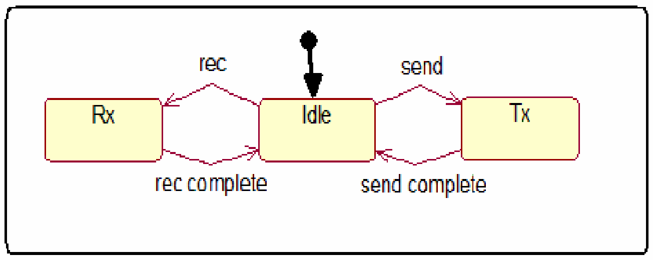

Figure 4. Core Communication Statechart

Whenever a request to send is received for this object a 'send' event will be introduced into the statechart. Whenever there is data to be received a 'rec' event will be introduced. The event data (that travels with the event object) may be an IP like packet with source and destination addresses and the communication data to transmit or receive. Once the send or receive operation completes an event will be generated that will cause a transition back to the idle state.

This statechart can be modeled in the framework as a set of classes representing states, transitions, and actions by inheriting from the 'Statechart' class. Classes that inherit from 'Statechart' often include an interface that allows non-Statechart objects to interact with them. These methods then determine which events shall be injected into the statechart and introduce the events into the system.

Now imagine that a requirement was added to handle encryption and decryption of any message that is sent or received outside of a certain network. In a traditional system this concern might be tangled with the communication concern. It is better in terms of comprehensibility and reusability, however, to keep them separate. A statechart could be created to handle this situation, see Figure 5.

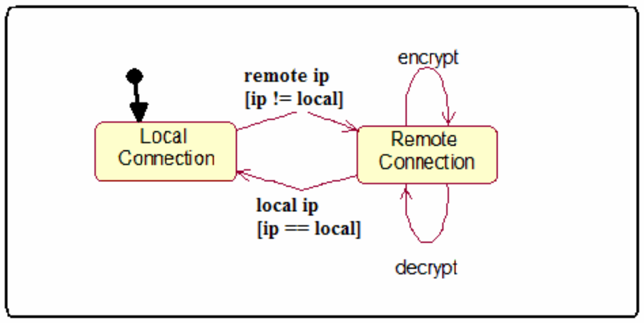

Figure 5. Encryption Aspect Statechart

This statechart starts in the 'Local Connection' state and will transition to 'Remote Connection' whenever a 'remote ip' event is handled that has, as part of the event data, a packet with a non-local address in it. When in the 'Remote Connection' state an 'encrypt' event will cause the statechart to encrypt the relevant part of the event data. Similarly, when a 'decrypt' event is handled in this state it will decrypt the relevant part of the event data. Lastly, when in the 'Remote Connection' state a 'local ip' event will cause a transition to the 'Local Connection' state if the packet contains a local address.

Weaving these two statecharts together involves declaring that certain events in the core communication statechart should be intercepted and reinterpreted as encryption related events in the encryption aspect statechart. In particular, to handle the transitions to and from the 'Local Connection' and 'Remote Connection' states a weaving developer can take these two statecharts and declare:

1. If the core is in the 'Idle' state and a 'send' event is introduced, and if the aspect is in the 'Local Connection' state treat 'send' exactly as if it were a 'remote ip' event in the aspect statechart. Allow the aspect statechart to handle the event before the core.
2. If the core is in the 'Idle' state and a 'send' event is introduced, and if the aspect is in the 'Remote Connection' state treat 'send' exactly as if it were a 'local ip' event in the aspect statechart. Allow the aspect statechart to handle the event before the core.

The aspect statechart will only make the transitions if the event data contains either a non-local or local address, respectively. The guards in the aspect statechart guarantee this by examining the IP-like packet in the event data. The weaving developer would state that the aspect statechart should handle the reinterpreted 'send' event before the core statechart handles it. He would do this so that one ‘send’ event could cause a transition to ‘Remote Connection’ and be treated as an ‘encrypt’ event that would cause the data to be encrypted (see rule 3 below).

Next the weaving developer would specify that ‘send’ events should also be treated as ‘encrypt’ events:

3. If the core is in the ‘Idle’ state and a ‘send’ event is introduced, and if the aspect is in the ‘Remote Connection’ state treat ‘send’ exactly as if were an ‘encrypt’ event in the aspect statechart. Allow the aspect statechart to handle the event before the core.

The aspect statechart must again handle the event before the core. This will cause the packet in the ‘send’ event’s data to be encrypted before the core communication statechart sends the packet. The core would be completely oblivious that the encryption took place.

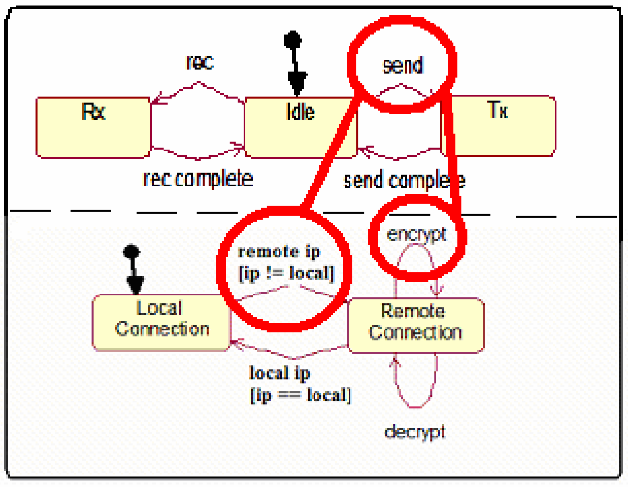

Figure 6. Reinterpreted events

The framework provides a simple, clean interface for specifying weaving and event reinterpretation. In the example above, weaving can be specified using a reference to the core ‘Statechart’ object and a reference to the aspect ‘Statechart’ object:

```
core.crosscutBy(encryption);
```

This will take the two statechart submodels and weave them into orthogonal regions. In the course of this weaving, methods will be called to map events in the core and aspect statecharts. These methods hold the declarations of which events need to be reinterpreted. A weaving developer would fill in these details in a subclass of the encryption aspect ‘Statechart’ and are equivalent to the three rules stated above:

```
reinterpretEvent(core,"IDLE","send",
"LOCAL_CONNECTION","remoteip",
Statechart.PREHANDLE);
```

```
reinterpretEvent(core,"IDLE","send",
"REMOTE_CONNECTION","localip",
Statechart.PREHANDLE);
```

```
reinterpretEvent(core,"IDLE","send",
"REMOTE_CONNECTION","encrypt",
Statechart.PREHANDLE);
```

To understand exactly what happens in the framework imagine that these two statecharts were woven together into orthogonal regions where the communication statechart is in the ‘Idle’ state while the encryption aspect is in the ‘Local Connection’. If a ‘send’ event is introduced, then before the transition to ‘Tx’ in the core, ‘send’ will be treated as ‘remote ip’ and a transition to ‘Remote Connection’ will occur (if the guard evaluates to true). Then, since ‘send’ should be interpreted as ‘encrypt’ before the core handles it, a transition will occur from ‘Remote Connection’ to itself and the data inside the packet will be encrypted. Lastly, the transition to ‘Tx’ will occur and the communication statechart will send an encrypted packet.

Similar declarations of reinterpretation would be stated that handle the receiving and decryption of network data. After weaving these two statecharts together and specifying how events should be interpreted, encryption on the network is accomplished. This is done without either statechart explicitly knowing about the other. The aspect statechart can be used in any situation that requires encryption of IP like packets.

The novelty of this approach is to allow events to be reinterpreted in different orthogonal regions and to alter the semantics of orthogonal statecharts. In traditional orthogonal statecharts when an event is broadcast the order that the composed statecharts handle it is non-deterministic. Our approach is to create a deterministic relationship between certain event transitions. We guarantee that, if such a relationship exists, an aspect statechart will have the ability to alter the event or event data and make a transition either before or after the core. This is similar to the AspectJ notion of before and after advice. The ordering of events was achieved in [11] by explicitly propagating hard coded events to different orthogonal regions. The cost of that solution was a more complex set of statecharts that were not reusable and that lacked obliviousness.

In this new approach, quantification is achieved by allowing statecharts to be composed orthogonally so that events occurring in one statechart may be mapped, or reinterpreted, to events in the composed statecharts. There is no restriction on which statecharts can crosscut one another, that is, an aspect statechart may crosscut a core statechart or it may crosscut another aspect statechart. All that is required for weaving is that the statecharts inherit from the framework’s ‘Statechart’ class. The scope of the quantification is the set of events in a composed set of statecharts. The action interaction (how disparate actions communicate) between the core and the aspect is in the event and the event data. Since events in the aspect statecharts are aliases (equivalent to reference parameters in procedural programming languages) for core events, the aspects can communicate through the event data. In particular, the event and event data may be altered or consumed.

6. RELATED WORK

Different approaches [7,8,9,10,11] have been proposed to address the issue of aspect-oriented modeling. Few of these approaches have focused on the modeling process while others have focused on UML profiles. The profile based approaches [8,10] have attempted to present the modeler with a UML profile for aspect-oriented modeling. The work presented in [9] has focused on modeling the crosscutting concerns by means of stereotyped packages. The approach presented in [10] has argued that aspects can be best represented by means of stereotyped classifiers. However, previous approaches haven't addressed the issue of abstracting statecharts as a mechanism by which quantification and obliviousness can be supported at the model level.

7. CONCLUSION

Our approach presented in this paper combines the expressive power of Harel's statecharts with aspect orientation to handle crosscutting concerns in the design and development of software systems. Our aspect-oriented statechart framework provides the capability to specify the dynamic behaviors of a crosscutting concern without committing to where, when, and on what core behaviors the aspect will be applied to. The decision about where, when and what core behaviors will be affected by an aspect statechart can be specified independently of the development of the core and independently of the development of the aspect. The two development teams do not have to be cognizant of each other. Only the *weaving* developers need to be cognizant of the core and aspect statecharts. This simplifies the design of core and crosscutting concerns and provides the obliviousness that is desirable in aspect-oriented approaches.

We have shown that aspect orientated techniques can be used to abstract and modularize statecharts. We have extended previous work in this area to separate and provide a level of reusability for crosscutting concerns. In addition, we created a framework to prove that the ideas are sound. What was missing from the previous work [11] was a method to have events act as aliases for other events and a way to specify the order that events should be handled. Our approach eliminates the need for hard coded event propagation. We hope to extend this work to examine how real time constraints might be modeled using composed statecharts in orthogonal regions.

8. ACKNOWLEDGEMENTS

We would like to thank Shangping Ren for her constructive criticism and insights into the development of the framework and this paper.

9. REFERENCES

[1] D. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," Comm. ACM, vol. 15, no.12, 1972, pp. 1053–1058.

[2] Elrad, T., Filman, R. E., and Bader, A. 2001. Aspect-oriented programming. *Comm. ACM 44*, 10 (Oct.), 29–32.

[3] G. Kiczales 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.

[4] Rashid, A., P. Sawyer, A. Moreira and J. Araujo. "Early Aspects: A Model for Aspect-Oriented Requirements Engineering," IEEE Joint International Conference on Requirements Engineering, IEEE Computer Society Press, 2002, pp. 199-202.

[5] Harel, D. and Gery, E., "Executable Object Modeling with Statecharts" *IEEE Computer* 30 (7). 1997, pp. 31-42.

[6] K. Koskimies, T. Systä, J. Tuomi, and T. Männistö "Automated Support for Modeling OO Software," IEEE Software, 15(1), 1998, pp. 87-94.

[7] Aldawud, O., Elrad, T., and Bader, A. 2001. A UML profile for aspect-oriented modeling. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA)*, (Tampa, Florida). http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/submissions/26-aldawud.pdf.

[8] Aldawud, O., Elrad, T., and Bader, A. 2001. A UML profile for aspect-oriented modeling. AOM: Aspect-oriented Modeling with UML Workshop 2003. 3rd *International Conference on Aspect-Oriented Software Development (AOSD)* http://lglwww.epfl.ch/workshops/uml2002/index.html

[9] Clarke, S. and Walker, R. J. 2001. Composition patterns: An approach to designing reusable aspects. In *23rd Int'l Conf. Software Engineering (ICSE)*, (Toronto). IEEE, 5–14.

[10] Stein, D., Hanenberg, St., Unland, R., *A UML-based Aspect-Oriented Design Notation For AspectJ*, in: Proc. of AOSD '02 (Enschede, The Netherlands, Apr. 2002), ACM, pp. 106-112.

[11] Elrad T., Aldawud O., 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.

[12] Object Management Group, Unified Modeling Language Specifications. Version 1.5 Mar 2003.

[13] Booch, Rumbaugh, Jacobson. 1999. The Unified Modeling Language User Guide. Addison Wesley.

[14] Harel, David. 1987. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8 231-274.

[15] Samek, Miro. 2002. Practical Statecharts in C/C++. CMP Books.

[16] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. 2001. An overview of AspectJ. In *ECOOP 2001—Object-Oriented Programming, 15th European Conference*, (Budapest), J. L. Knudsen, Ed. LNCS, vol 2072. Springer-Verlag, Berlin, 327–353.

[17] Aldawud, O., A Bader and T. Elltad, Weaving with Statecharts, Aspect-Oriented Modeling with UML workshop at the 1st International Conference on Aspect-Oriented

Software Development, April 2002.", http://citeseer.ist.psu.edu/aldawud02weaving.html

[18] R.E. Filman and D.P. Friedman, Aspect-Oriented Programming is Quantification and Obliviousness, Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis. http://ic-www.arc.nasa.gov/ic/ darwin/oif/leo/filman/text/oif/aop-is.pdf",

[19] P. Tarr, H. Ossher, W. Harrison, St. Sutton Jr.: *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, in: Proc. of ICSE '99, (Los Angeles, CA, May 1999) ACM, pp. 107-119

[20] Douglass, B. P. 1997. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, Reading, Massachusetts.

[21] Douglass, B. P. 1999. UML statecharts: A white paper. *Embedded Systems Programming 12*, 1 (Jan.).

[22] AspectJ Development Team. 2003. AspectJ FAQ. http://dev.eclipse.org/viewcvs/indextech.cgi/aspectj-home/doc/faq.html?rev=1.7#q:crosscutting

[23] Harel, David, Politi, Michal. 1998. Modeling Reactive Systems with Statecharts, The Statemate Approach. McGraw-Hill

[24] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring and M. Trakhtenbrot, 1990. STATEMATE: A Working Environment for the Development of Complex Reactive Systems, IEEE Trans. on Software Engineering 16:4, 403-414.