Distributing Statecharts to Handle Pervasive Crosscutting Concerns

Mark Mahoney
Carthage College
mmahoney@carthage.edu

Tzilla Elrad
Illinois Institute of Technology
elrad@iit.edu

ABSTRACT
Statecharts are a tool to model the reactive behavior of an object or system. Statecharts can be extended to execute on separate machines and coordinate with each other to achieve complex reactive behavior. We propose using a framework of classes to take statechart diagrams and make them executable on separate machines to handle pervasive computing concerns.

Keywords
Statecharts, distributed statecharts, aspect-orientation, aspect-oriented statechart framework.

1. INTRODUCTION
The world is seeing an increase in the amount of seamless coordinated computing among individual devices. Whether this is in the form of gamers participating in group gaming activities or deeply embedded devices communicating to provide value added services to their users, there is no doubt that the freedom and mobility provided by pervasive computing is growing in popularity. Along with this growth is the need for new design and implementation methods to handle pervasive computing. We propose using statecharts to model the behavior of distributed objects and a mechanism to coordinate the activities of distributed statecharts in a pervasive environment.

The rest of this paper is organized as follows: section 2 introduces traditional statecharts and a framework created to take statechart diagrams and transform them into executable code, section 3 proposes alterations to the framework to allow statecharts to exist on separate machines, section 4 discusses related and future work, Section 5 provides the conclusions of this paper.

2. MODELING REACTIVITY WITH STATECHARTS

2.1 Simple Statecharts
Statecharts [4] are an excellent way to model the reactivity of an object. They are visually very expressive and can be formally verified and used to generate code. Statecharts are an extension of extended finite state machines. An *extended* finite state machine is one in which variables are assigned to the model and conditions can be used to guard transitions to different states. Actions can be associated with transitions between states. An action is a nearly instantaneous occurrence that takes ideally zero time. Statecharts use a Mealy-automata approach to actions [8]. Actions are associated with events and transitions.

event/action

If this combination is associated with a transition it means that ‘event’ in the current state will cause a transition and the ‘action’ will take place.

A guard condition is specified with a set of square brackets. A guard evaluates a condition of one of the extended finite state machine variables:

event [condition]/action

Only if the event is introduced and the guard evaluates to true will the transition take place.

Actions alone are not well suited to represent behavior in a complex environment. Some behavior requires a great deal of computation time. For these, activities are appropriate. An activity is a non-zero length occurrence. Activities are associated with entries and exits from states. Upon entering a state an activity may be started and continue indefinitely until the state is exited.

2.2 Advanced Features of Statecharts
[4] introduced the idea of hierarchy into state machine diagrams. Rather than having large, unruly, flat descriptions of state behavior he allowed the nesting of states inside one another. This reduced the complexity of statechart diagrams and allowed more expressive diagrams to be built.

A simple statechart is said to have an ‘exclusive-or’ relationship between the states. This ‘Or-composition’ means that a statechart may have one, and only one, active 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 somewhat independent statecharts. Being in a statechart like this means that in every orthogonal region one of the states of each composed statechart may be active. Orthogonal regions allow you to avoid mixing the independent behaviors as a cross product and, instead, keep them separate.

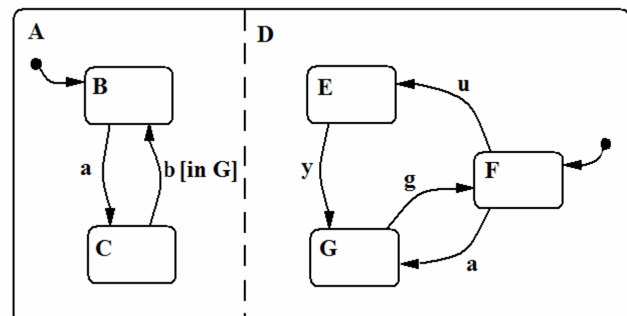

Figure 1. Statechart with Orthogonal Regions

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 in one diagram can cause a transition in that diagram and in other composed statecharts simultaneously (for example, event 'a' in figure 1). The composed statecharts are orthogonal, or independent, of each other but are allowed to communicate through broadcast events. Orthogonality is a way to avoid an explosion in the number of possible states. If this technique weren't used above then the system would require the states BE, BG, BF, CE, CG, CF or the multiplicative 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.

2.3 Aspect-Oriented Statechart Framework

Aspect-Oriented Software Development (AOSD) [1] aims to reduce the effect of crosscutting concerns. A crosscutting concern is one that cannot easily be modularized into a single unit. Rather, it is spread out, or scattered, with the implementation of other concerns. The implementations of crosscutting concerns are tangled with core concerns and are hard to reason about and limit their reuse.

For example, synchronization is a concern that developers deal with whenever guarded access to a resource is required. Access to devices, files, and memory are all things that may require synchronization. However, the implementation of synchronization among those concerns is usually repeated in several places. A change to the synchronization policy would require manual changes to many components. This is time consuming and error prone.

AOSD provides a means to separate crosscutting concerns into their own first class units, called Aspects, that specify *where* the concern should be applied and *what* actions should take place at those points. There are several Aspect-Oriented Programming languages [5][3] that are gaining support in academia and industry. We have previously developed a framework of classes in Java for handling crosscutting concerns in reactive systems using statecharts [2][6][7] that we call the Aspect-Oriented Statechart Framework (AOSF).

Using our approach one can model core and crosscutting concerns with statecharts. One can manually transform a statechart diagram into a set of classes within the framework to get executable code (currently this is a manual transformation but in the future we plan on developing tools to automate the process). Each executable statechart that represents a core concern may then be composed into orthogonal regions with statecharts that represent crosscutting concerns. Further, because the statecharts may have been developed in isolation, a developer can declare that broadcast events from one statechart to have new meaning in one or more of the orthogonal regions.

As an example of how to use the AOSF, imagine one was asked to implement a communication protocol in software. Because the sending and receiving of data over the network is a complex process the developer decides to use a state-based implementation. In particular, the developer has come up with the following statechart to model the protocol software:

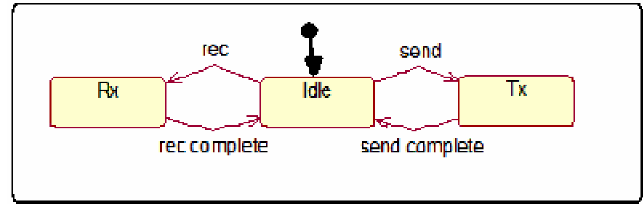

Figure 2. Core Communication Concern

This statechart says that while in the 'Idle' state if a 'send' event is received, transition to the 'Tx' state and begin transmitting the data associated with the 'send' event. The rest of the states and transitions are self-explanatory.

Using the AOSF, this functionality can be implemented with the following class:

```
import statechartframework.*;
public class CoreCommunication extends Statechart
{
    /**
    * these are the core states of the statechart
    */
    private State rx;
    private State tx;
    private State idle;
    …
    …
```

This is only a partial specification, see [2] for the full example and code.

Now let's imagine that there was a requirement to encrypt all data sent to machines on different subnets. We could go to the core communication statechart and make changes there, however this would render the core communication class unusable for users who don't need to handle encryption. It would be better to come up with a solution for this crosscutting concern that is non-invasive.

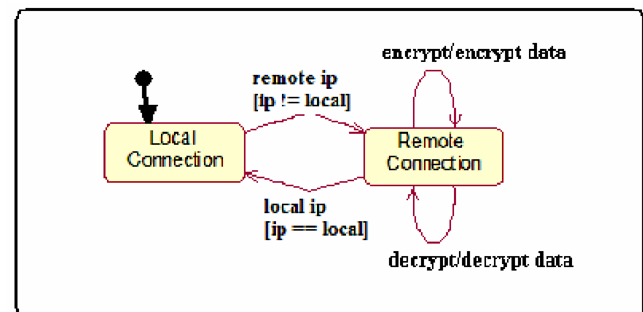

Figure 3. Crosscutting Encryption Concern

The statechart in figure 3 handles encryption based on the state of the connection, either local or remote. The code for this statechart can be created from the framework in a similar manner to the core communication statechart.

Now that the weaving developer has identified two independent statecharts that he believes can give him the functionality he

desires, he has to perform the weaving and specify how events should be translated.

The basic structure of the AOSF can be described in the diagram below:

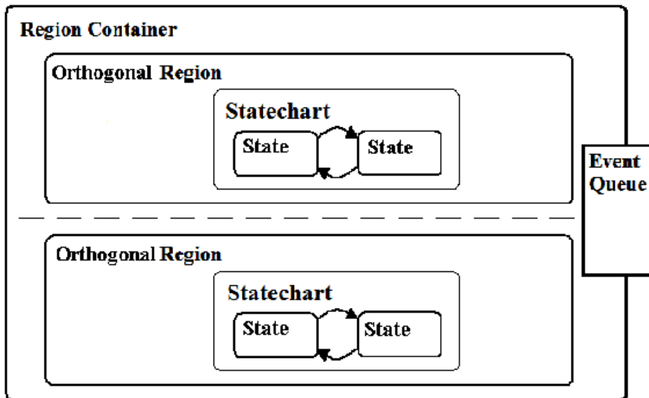

Figure 4. Basic Containment Hierarchy of the AOSF

A region container holds one or more orthogonal regions. An orthogonal region is made up of a single statechart. A statechart is made up of states, transitions, activities, and actions. The event queue is used to broadcast events to all the orthogonal regions.

The novel contribution of our work is to allow independent statecharts that are capable of executing in isolation in their own region containers to be woven together into a single region container and for events to be reinterpreted from one type to another.

Event reinterpretation can be summarized by the following diagram and statements:

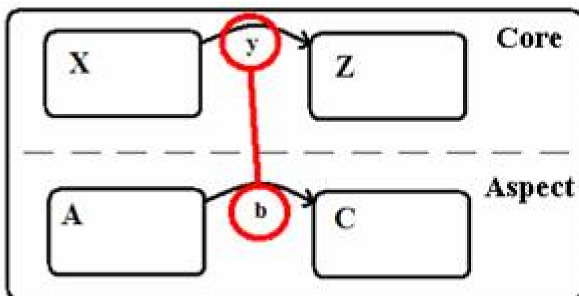

Figure 5. Reinterpreting Events

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'.

The aspect statechart can handle the event before or after the core statechart.

If one takes the two models and weaves them together into orthogonal regions of the same statechart and reinterprets some events one can non-invasively add behavior to achieve encryption. The communication and encryption statecharts will be unaware that they are working together making both simpler and more reusable. Weaving these two models would result in a model that looks like this.

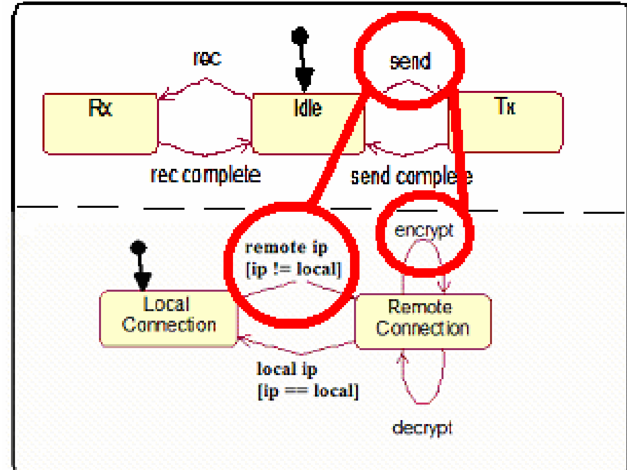

Figure 6. Core and Crosscutting Statecharts Woven Together with Event Reinterpretation

Using the framework code, the weaving is accomplished by the following code.

```
core.crosscutBy(enc);
```

This will take the encryption statechart object (enc) and place it in an orthogonal region within the core communication statechart's region container. All events from the statecharts will be broadcast to each other. However, since the two statecharts were developed in isolation the events have no meaning to each other. This is where event reinterpretation comes in. The two event reinterpretations in figure 6 look like this:

```
reinterpretEvent(core,"IDLE","send","LOCAL_CONNECT
ION","remoteip",Statechart.PREHANDLE, false);
```

and

```
reinterpretEvent(core,"IDLE","send","REMOTE_CONNEC
TION","encrypt", Statechart.PREHANDLE, false);
```

The first declaration of event reinterpretation is saying that if the core's current state is 'IDLE' and a 'send' event is received, and if the aspect statechart's current state is 'LOCAL_CONNECTION' then treat 'send' as if it were a 'remoteip' event. In addition, the encryption statechart should handle these events before the core.

Imagine the statecharts have been woven together and the current state for each statechart is 'Idle' and 'Local Connection'. When a 'send' event is introduced in the system the encryption statechart will treat 'send' as 'remoteip'. Because the weaving developer specified that the encryption statechart should handle the event before the core communication statechart the guard will be evaluated to see if the destination address is not local. If it is not local then a transition to 'Remote Connection' will be made. Then, because 'send' is to be treated as an 'encrypt' event in the 'Remote Connection' state, the self-transition will be made and the action of encrypting the event's data will occur. Only after that will the transition from 'Idle' to 'Tx' be made and the newly encrypted event data will be transmitted securely through the network.

3. DISTRIBUTED STATECHARTS AND PERVASIVENESS

The current implementation of the AOSF was created without pervasiveness in mind. However, we believe the AOSF is applicable in ubiquitous environments. The key to using the AOSF in pervasive environments is to allow the orthogonal regions (and their contained statecharts) to execute on remote machines. The broadcasting mechanism in the AOSF works by having a global event queue (associated with a region container) that sends events to all orthogonal regions so that they may be handled by the statecharts. We believe we can alter the AOSF in such a way that orthogonal regions can be distributed on different machines yet still participate and coordinate with other statecharts in orthogonal regions. Event reinterpretation will be used to supply a context to the disparate statecharts in participating orthogonal regions.

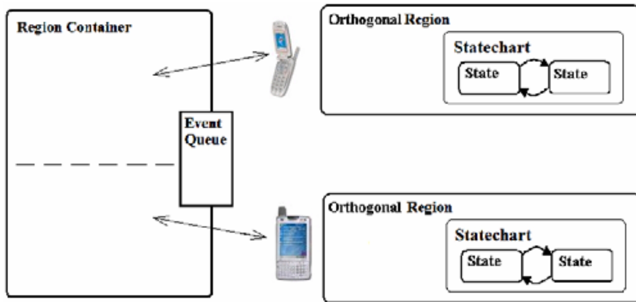

Figure 7. Distributed Statecharts Coordinating Using the AOSF

3.1 Assembly Line Example

One of the goals of this approach is to be able to coordinate disparate devices' behavior. For example, there may be a factory with several independent machines (chip placement machines, screening machines, etc.) that make up an assembly line. Each piece of equipment may be controlled by a statechart that describes its behavior. The machines on this assembly line may be rearranged for every new product that the factory is asked to produce. Using the AOSF, one can assign each machine a statechart to control its individual behavior. In addition, one could create a library of statecharts that is used to control the assembly line for any given product. The individual machines' statecharts will be unaware that they are participating in an assembly line and are reusable from assembly line to assembly line.

Imagine the product that was being built on the assembly line was a light fixture and that the assembly line was made up of two machines- one to press the assembly together and one to rivet the pieces. Each machine is controlled by a separate statechart that shares no events in common with the others. In fact, the machines are not even aware that they are constructing a light fixture, that is, the next time these machines are used they may be building a completely different product.

One could add a statechart to control the assembly of the light fixture.

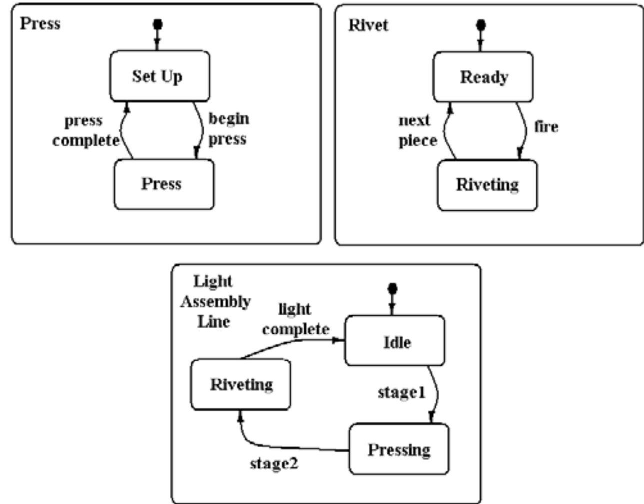

Figure 8. Simple Assembly Line Statecharts

These (simplified) statecharts can be brought together into orthogonal regions and the assembly line can be controlled using all three. The coordinating statechart would reinterpret events from each of the three machines to control flow down the line. When the press machine issued a 'press complete' event, the coordinating statechart can reinterpret this event to 'stage2' and transition to the 'Riveting' state. This transition might be reinterpreted in the rivet machine to a 'fire' event, and so on.

The benefit of having a controlling statechart is that the machines do not need any special modifications for each assembly line. They are simply brought together, have their statecharts join each other in orthogonal regions, and allow the coordinating statechart to control the line using event reinterpretation.

The coordinating statechart may be able to handle assembly line issues. For example, if a machine breaks down or runs out of parts it can halt the other machines. The coordinating statechart can gather statistics about completion times, defects, and faulty assemblies all without involving the individual machines.

3.2 Air Traffic Control Example

Another example of the use of distributed statecharts comes in an air traffic control system. Imagine that every plane that flies in an air space has a statechart that describes its current state. A simplified statechart may look like this:

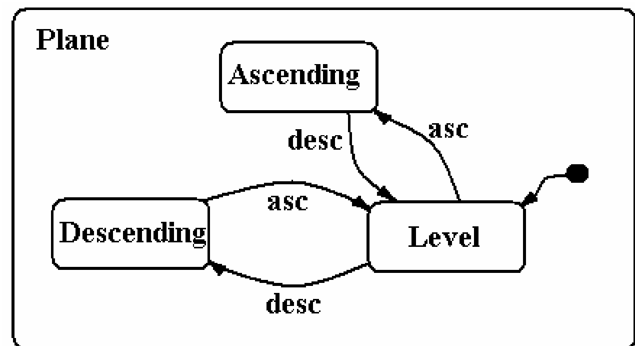

Figure 9. Simple Statechart Describing an Airplane's State

As a plane flies across the country it would send its statechart to the air traffic control tower whose airspace the plane is flying in. Each air traffic control tower has its own statechart to monitor the planes in its airspace. An air traffic control tower in Washington D. C. might have a statechart that looks like this:

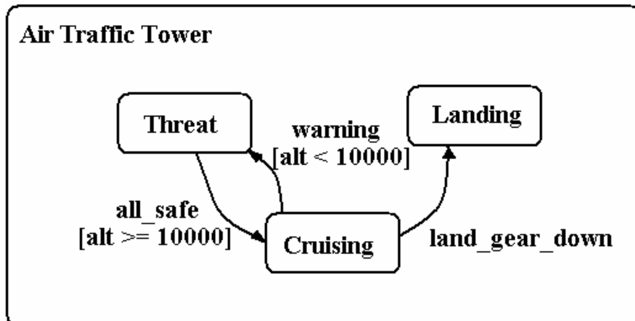

Figure 10. Air Traffic Control Tower Statechart

Each airplane's statechart would occupy an orthogonal region in the air traffic control tower's statechart. The air traffic control tower would then specify which events to reinterpret. In this case, when an airplane descended it would trigger a 'desc' event that would be reinterpreted as a 'warning' event in the air traffic control tower's statechart. If the guard condition evaluated to true, then the air traffic control tower may go into a state of 'threat' and an action or activity may take place.

As a plane travels across the country it would continually add and remove its statechart to air traffic control towers. Not all air traffic control towers will have the same statecharts, and not all planes will have the same statecharts. Each tower will interpret events differently. Air traffic control towers would then have the ability to monitor all the planes in its airspace. Because there is loose coupling in the associated statecharts an air traffic control tower would remain relatively immune to new types of airplane statecharts. When a new model airplane is introduced all that is required is that a weaving developer specify which events to reinterpret. This specification may be stored as a policy and will allow seamless integration of planes in an airspace.

4. RELATED AND FUTURE WORK

The idea of statecharts executing on separate machines is not new. [9] introduced a similar idea for distributed workflow executions. In this work, however, there is no idea of event reinterpretation or determinism in the order that broadcast events are handled. Their work relates to different workflows existing on remote servers and the state of completed workflows in orthogonal regions.

We are still early in our research in applying distributed statecharts to pervasive computing. We have yet to implement the distribution mechanism in the AOSF. We believe we can extend the AOSF with a Java RMI implementation to allow orthogonal regions to exist on separate machines and communicate with a region container. Because pervasive concerns were not the primary focus of the framework's implementation there are still many things we are examining. We believe, however, that the use of distributed statecharts can greatly aid in the handling of concerns dealing with ubiquitous computing.

5. CONCLUSIONS

This paper has introduced statecharts and a framework of classes for creating executable code from statecharts. We have proposed an extension to the framework for dealing with pervasive concerns. We have shown that distributed statecharts can be useful in the coordination of a large number of remote devices with minimal impact on the software in each of those devices. A coordinating statechart is used to reinterpret broadcast events to take on meaning in orthogonal regions.

6. ACKNOWLEDGMENTS

This work is partially supported by CISE NSF grant No. 0137743.

7. REFERENCES

[1] AOSD web page. http://aosd.net

[2] Aspect-Oriented Statechart Framework website. http://ulysses.carthage.edu/faculty/mmahoney/AOP/AOSF/default.htm

[3] Aspectwerkz website. http://aspectwerkz.codehaus.org/

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

[5] 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.

[6] Mahoney, M., Bader, A., Aldawud, O., Elrad, T., *Using Aspects to Abstract and Modularize Statecharts*. The 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004. http://www.cs.iit.edu/~oaldawud/AOM/mahoney.pdf

[7] Mahoney, M., Elrad, T. *Modeling Platform Specific Attributes of a System as Crosscutting Concerns using Aspect-Oriented Statecharts and Virtual Finite State Machines*, the 6th International Workshop on Aspect-Oriented Modeling as part of AOSD'05 (Chicago, USA, March 2005)

[8] Samek, M. *Practical Statecharts in C/C++*. 2002. CMP Books.

[9] Wodtke, D., Weikum, G. *A Formal Foundation for Distributed Workflow Execution Based on State Charts.* In 6th Intl. Conf. ICDT, 1997.