

# Modeling Platform Specific Attributes of a System as Crosscutting Concerns using Aspect-Oriented Statecharts and Virtual Finite State Machines

Mark Mahoney  
Carthage College  
Kenosha, WI

mmahoney@carthage.edu

Tzilla Elrad  
Illinois Institute of Technology  
Chicago, IL

elrad@iit.edu

## ABSTRACT

Model Driven Development (MDD) emphasizes the importance of modeling business requirements and separating platform independent details of a system from platform specific details. Virtual Finite State Machines (VFMS) are an MDD construct that model control behavior in a finite state machine to prolong the life of a design. The platform specific behaviors in a system can be considered crosscutting concerns rather than core concerns. These crosscutting concerns can be modeled as statecharts using the Aspect-Oriented Statechart Framework (AOSF) and woven into a VFMS to reduce coupling and maximize reuse of system models.

## Keywords

Aspect-Oriented Software Development, Model Driven Development, Virtual Finite State Machines, Aspect-Oriented Statechart Framework.

## 1. INTRODUCTION

It has long been known that separating concerns in the development of software is a beneficial process [9]. When each concern is isolated it makes it easier to reason about the concern, maintain that concern, and comprehend the entire system as a collection of related concerns. Separation of concerns has taken many forms over the years from functions and procedures to objects and classes. In each case, the concerns are abstracted so that one can comprehend a complex system.

Aspect-Oriented Software Development (AOSD)[6][3] attempts to further decompose a system into core application concerns and a special type of concern called a 'crosscutting concern'. A crosscutting concern is one that is not easily modularized because the concern tends to be spread throughout a system. Crosscutting concerns tend to be tangled with core application concerns. As an example, in a banking system the core concerns might take the form of withdrawing from an account, depositing in to an account, and querying the balance of an account. Each one of these core concerns is easily modularized into functions or methods of a class. However, there are other concerns in a banking application that are not as easy to modularized. Security, for example, may be necessary for all the above core concerns. The security concern is tangled the core banking concerns. If the security concern needed to be updated or removed it would involve modifying all of the core concerns.

AOSD allows crosscutting concerns to be separated from the core concerns so that they can be reasoned about in isolation and later woven back to form a functioning system. The key tenets of AOSD are quantification and obliviousness [3]. Quantification is the ability to specify one or more points in a program where a single crosscutting concern can be woven in with the core. Obliviousness has to do with the core developer being oblivious to the fact that crosscutting concerns are woven into the core concern.

Model Driven Architecture (MDA)[7] falls into the category of Model Driven Development. MDA is a new technology that attempts to separate the platform independent details from the platform specific details of a system in a series of models. The models are then used to generate code. The idea is that platforms change relatively frequently (programming language, programming paradigm, hardware, operating system, etc.) but the core business models change infrequently. The developer devotes his efforts to developing Platform Independent Models (PIM) that persist for long periods of time. When a platform change is required all that needs to be done is to create a new Platform Specific Model (PSM) to handle the differences in platform, the core business models remain unchanged.

One particular type of MDD model is a Virtual Finite State Machine (VFMS) [11]. VFMSs, like MDA implementations, separate platform independent behavior from platform specific behavior. The platform independent behavior is modeled in an extension to Extended Finite State Machines (EFSM) and the platform specific behavior is modeled in the form of input processor and output processor classes that interact with the VFMS. There is a high degree of coupling between the state machine and the input/output processors. This paper argues that platform specific details in a VFMS are crosscutting concerns and can be modeled using Aspect-Oriented Modeling techniques [8] and VFMSs.

The Aspect-Oriented Statechart Framework [8] was created to model core and crosscutting concerns separately as statecharts and to allow them to be woven into a functioning system. This paper describes some research that is being conducted to determine if platform specific statechart models can be woven with platform independent statechart models to minimize the coupling between the two.

The organization of the rest of this paper is as follows, in section 2 VFMSs will be described in detail. In section 3 the Aspect-Oriented Statechart Framework will be described. In section 4

these two ideas will be combined in order minimize coupling and maximize the reuse of models and code. In section 5 related work will be discussed. In section 6 future work will be discussed. Finally, in section 7 the conclusions of this work will be discussed.

## 2. VIRTUAL FINITE STATE MACHINES

The VFMS methodology consists of a design paradigm, in which the control behavior of a software module is specified as a finite state machine, and an implementation paradigm that consists of a design structure that defines the interface between the control specification and the rest of the implementation [4]. The benefits of modeling the control behavior of the system in a platform independent state machine are that the behavior is easily understood by the developers involved, the behavior rarely changes so the shelf life of the model is extended, the model can be used to generate code, automated validation of the system is possible, and the maintainability of the system increases. The goal is to get many years of use out of the description of the behavior of the system even if platforms change. The description of the behavior abstracts platform specific details out of the models.

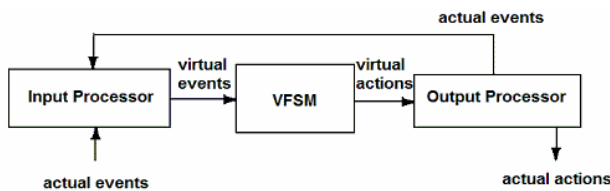


Figure 1. VFMS Components

The VFMS methodology uses three modeling components (figure 1). The VFMS is a finite state machine that contains the core application behavior. The VFMS responds to virtual events and reacts with virtual actions. A virtual event is a simple string that is an abstraction of an actual event in a platform specific context. There are no data or parameters associated with the event. A virtual action is also an abstraction for an action that will take place in a platform specific context. The input processor translates actual events into virtual events and feeds them into the VFMS. The VFMS will then react to the virtual events and perhaps produce virtual actions. The output processor maps virtual actions into actual actions. An example is given below.

Imagine an air conditioning system whose behavior is described with a finite state machine (see figure 2).

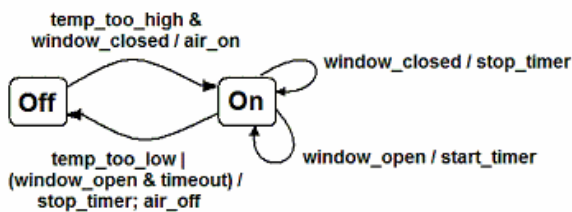


Figure 2. Air Conditioning Control System

There are two states for the air conditioning control system, ‘On’ and ‘Off’. When in the ‘Off’ state there will be a transition to ‘On’ when the temperature is too high and the windows are

closed. The virtual action associated with this transition is to turn the air conditioning system on (air\_on). When in the ‘On’ state if a window open virtual event occurs a self-transition will occur and the virtual action associated with this transition will be to start a timer. If the window is closed in the ‘On’ state there will be another self-transition and the timer will be stopped. Finally, if in the ‘On’ state and the temperature becomes too low or if the window is open and the timer expires then a transition will be made to the ‘Off’ state. The actions associated with this transition will be to stop the timer and turn the air conditioning system off.

Notice that there is not any mention of platform specifics or data in this state machine. Rather, there are simple events called ‘temp\_too\_high’, ‘temp\_too\_low’, ‘window\_closed’, ‘window\_open’, and ‘timeout’. This is attractive because these platform specific characteristics and data are the most likely to change over time. It is up to the input processor to interpret the current temperature and decide if it is too high or too low. The exact temperature and the method of reading the temperature are platform specific details that may change over time. The behavior of the system, however, is not likely to change. Figure 3 shows a combined representation of the input processor, output processor and VFMS.

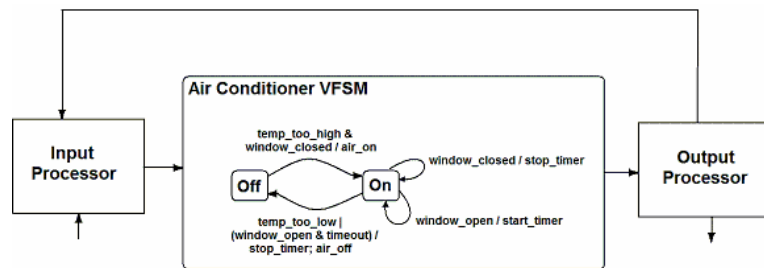


Figure 3. Air Conditioning VFMS

A typical implementation would require the platform specific input and output processors to be modeled as classes. These classes would require direct associations to the Air Conditioner VFMS. In particular, when the input processor determined that the temperature was too high it would have to ask the VFMS to introduce the virtual event ‘temp\_too\_high’ into the system. This might be implemented as a method temp\_too\_high() of the VFMS. Therefore, there is a high degree of coupling between the input processor and the VFMS implementation. For output processing, the VFMS would generate an interface that any implementation of an output processor would have to adhere to. The VFMS would have to maintain a reference to the output processor. In the example above the output processor would have to implement the methods air\_on(), stop\_timer(), start\_timer(), air\_off() or one method for each virtual action. Again, there is a high degree of coupling between the output processor and the VFMS. It would be very hard to reuse the input processor, output processor, or VFMS components in other contexts.

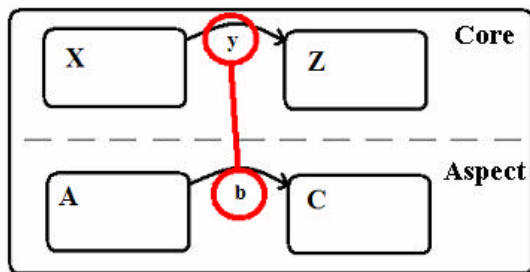
## 3. ASPECT-ORIENTED STATECHART FRAMEWORK

The Aspect-Oriented Statechart Framework (AOSF) [8] is a framework of classes in Java that one can use to transform a statechart model into working code. The AOSF addresses

crosscutting concerns by allowing a core concern to occupy an orthogonal region of a statechart and each crosscutting concern to occupy a separate orthogonal region of the same statechart. Further, the orthogonal regions interact by broadcasting events. The statechart semantics are based on David Harel's statecharts [5]. Previous attempts at modeling crosscutting concerns as statecharts [1][2] resulted in tightly coupled models. The goal of the AOSF is to reduce coupling in the model by making declarative statements about how crosscutting statecharts interact.

Events in orthogonal regions are broadcast to all other orthogonal regions of the same statechart. The AOSF allows broadcast events to be reinterpreted to have alternate meanings in other orthogonal regions. In other words, two independent statecharts can be combined (or woven together) and a declaration about how they crosscut each other can be made. For example, using the two statechart models that have been woven into orthogonal regions in figure 4 one can state:

*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'.*

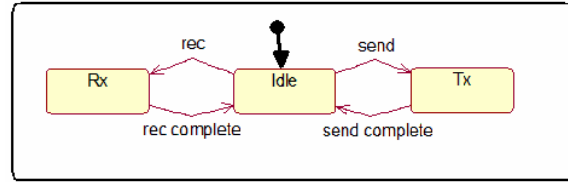


**Figure 4. AOSF Reinterpretation**

Broadcast events provide a means to quantify how the crosscutting concerns interact with the core. Obliviousness is achieved because the declarations are non-invasive.

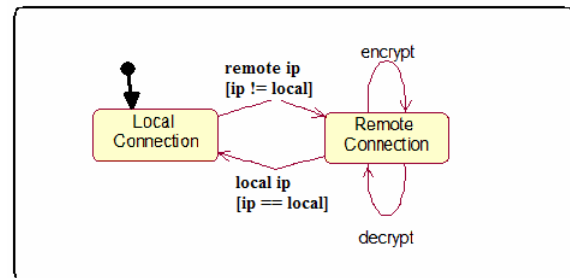
The semantics of the AOSF are such that aspect statecharts can handle events before or after the core statechart based on the declaration of crosscutting. That is, determinism is introduced in the order that broadcast events are handled among the orthogonal regions. This is different than traditional statecharts with orthogonal regions, but is beneficial because an aspect statechart may wish to introduce new events into the core statechart, alter event data before the core handles the event, or consume the event so that the core does not get an opportunity to handle the event. The advantages of this approach compared to [1][2] are that eliminating hard coded events in the models reduces coupling, reuse is possible among a collection of statechart classes, and there is a clean mapping of events in orthogonal regions.

For example, one can model a communication protocol using a statechart.



**Figure 5. Core communication statechart**

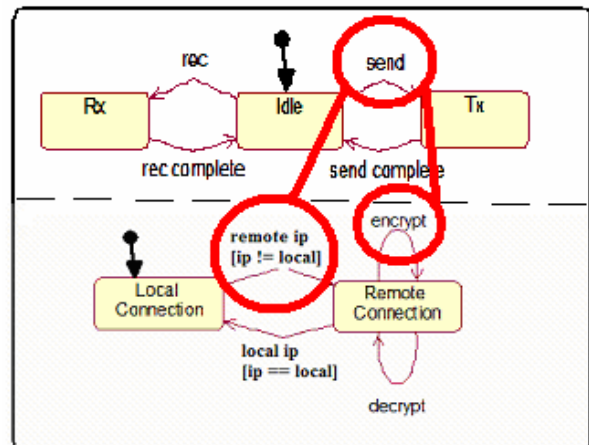
The statechart in figure 5 will transition to the transmission state 'Tx' when a 'send' event is received in the 'Idle' state. When the transmission is complete a 'send complete' event will cause transition back to 'Idle'. This statechart models a simple communication protocol. But what if the developers of this statechart were asked to introduce an encryption protocol when sending data across a non-local network? This is clearly a crosscutting concern. Using the AOSF one could create a statechart to handle this situation (figure 6).



**Figure 6. Aspect encryption statechart**

The IP like address associated with the event data can determine whether the packet is destined for a local connection or a remote connection. Once in the 'Remote Connection' state an encrypt event will cause a self-transition and the action of encrypting the event data will occur. Similarly, when a decrypt event occurs the event data will be decrypted.

Now these two statecharts can be woven and events from one can be reinterpreted (figure 7).



**Figure 7. Reinterpreted Events**

This process is relatively simple using the framework, below is the statement needed to combine the core communication statechart object and the aspect encryption statechart object.

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

Next we would have to declare how we want to reinterpret events from one statechart to another:

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

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

Here, the specification of crosscutting behavior states that while in the core's 'Idle' state if a 'send' event is received and if the aspect statechart is in the 'Local Connection' state treat 'send' as if it were a 'remote ip' event. In addition, the aspect statechart should make the transition before the core. The next statement says, if in the core's 'Idle' state if a 'send' event is received and if the aspect statechart is in the 'Remote Connection' state treat 'send' as if it were an 'encrypt' event. Again, we want the aspect statechart to handle the event before the core so that the event data can be encrypted before the data is sent in the 'Tx' state.

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 as the action associated with the transition. Lastly, the transition to 'Tx' will occur and the communication statechart will send an encrypted packet.

#### 4. VFSM'S AND ASPECT-ORIENTED STATECHARTS

It is our belief that platform specific models (input and output processors) are not the core concerns in a VFSM implementation. The input and output processors are crosscutting concerns that don't need to be tangled with core VFSM state machine. In the spirit of MDA (if not the exact process of automated transformations- see the future work section) we seek to separate platform independent characteristics from platform specific characteristics of a system and weave them together as needed. We propose using the AOSF to model the input and output processors as well as the VFSM as statecharts that crosscut each other. Then the statecharts can be woven together and the events from the input statechart can be reinterpreted to have meaning in the core VFSM. Further, events in the core VFSM that are related to virtual actions can be reinterpreted to have meaning in the output processor statechart.

There are several benefits to this approach. First, statecharts are an excellent way to model platform specific behavior. Although it

is not prohibited to use statecharts for input and output processors in traditional VFSMs a developer can now use the exact same framework code to create the platform independent behavior and the platform specific behavior. More importantly, there is little direct coupling between the input/output processors and the VFSM. In fact, any statechart created using the framework can be used as an input/output processor. All that needs to be done is to weave the statecharts together and map events from the input processor to the VFSM and from the VFSM to the output processor. This increases the likelihood of achieving reuse from a statechart library.

For example, one could create a statechart for the input processor and the output processor in the air conditioning system as in figure 8.

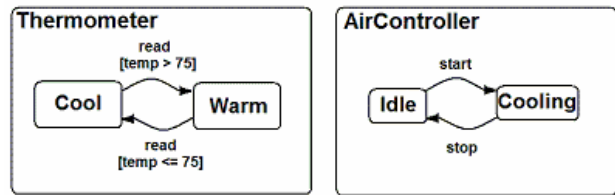


Figure 8. Input Processor Statechart and Output Processor Statechart for the Air Conditioning System

Next, a weaving developer would combine these statecharts together with the core VFSM and reinterpret the appropriate events. In this case, the input processor's event 'read' when the temperature was greater than 75 degrees (in the top orthogonal region in figure 9) would be reinterpreted as a 'temp\_too\_high' event in the core VFSM (middle orthogonal region in figure 9). The virtual action 'air\_on' in the VFSM (which is modeled as an event, which is appropriate because virtual actions are just abstractions of actual actions) would be reinterpreted as the 'start' event in the output processor statechart (bottom orthogonal region in figure 9). Similar declarations can be made for the other virtual events and actions.

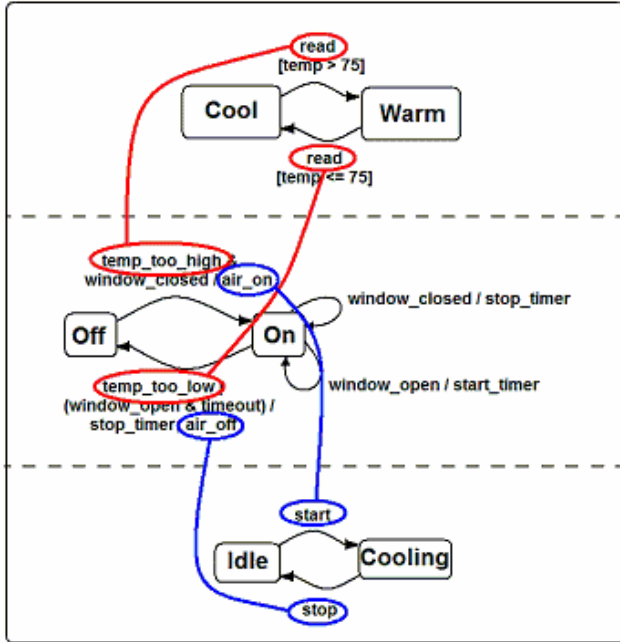


Figure 9. Woven Statecharts

To show how flexible the system is we will demonstrate how easy it is to replace the input processor. In the example above it was assumed that a thermometer and a specific temperature value controlled the transition from ‘Cool’ to ‘Warm’. Let’s imagine that we are moving this air conditioning system to an enclosed room with many people occupying the space and we use feedback from the occupants to determine an acceptable temperature. This is an example of a change of platform. All that is required is to build a statechart to model that behavior (see figure 10).

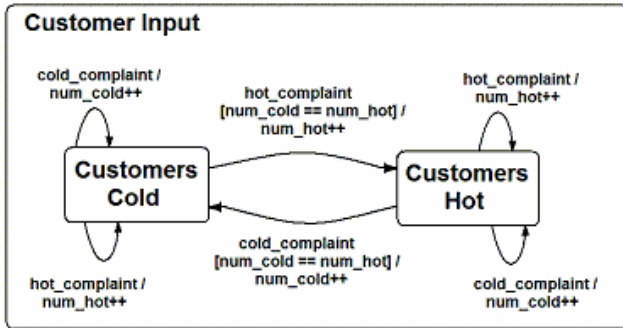


Figure 10. Customer input statechart

In this statechart a majority of complaints determines whether our customers are too hot or too cold. Now we can weave this new input processor statechart in with the old components and reinterpret some events (figure 11).

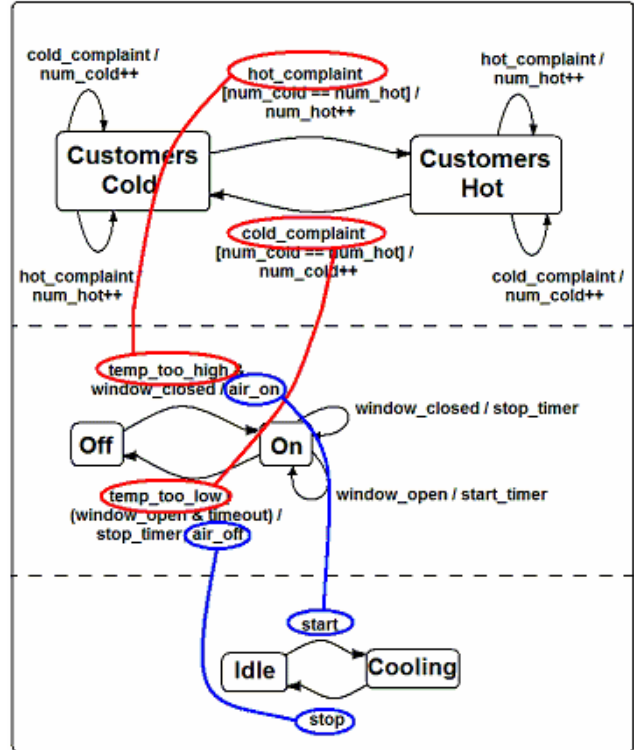


Figure 11. Customer complaints driving the air conditioning system

None of the components need to be altered to achieve this change in platform. All that is required is a new declaration of how the statecharts crosscut each other.

## 5. RELATED WORK

Hierarchical statecharts [5][10] are means to describe the behavior of a system as a series of nested statecharts. Originally proposed by Harel, a hierarchical statechart a state can contain nested statecharts. One state from each nested statechart is considered active at one time. We are aware of an (as yet unpublished) use of modeling concerns with hierarchical statecharts. With this methodology crosscutting concerns are represented as statecharts and core concerns are modeled as nested statecharts within a crosscutting statechart. The methodology uses statecharts to guard access to web pages and is UML compliant. We have chosen to use orthogonal regions as opposed to hierarchical statecharts because a statechart is meant to describe the history of a particular behavior that may persist for long periods of time. With hierarchical statecharts a transition out of a super state (crosscutting concern) implies transitioning out of all the nested states (core concerns) as well. From our perspective we find this somewhat limiting, the use of orthogonal regions permits the core and crosscutting behavior to exist and persist independently.

## 6. FUTURE WORK

We are planning on building a graphical tool and code generator to aid in generating framework code (currently the transformations from statechart model to framework code are

done manually). The graphical tool will create a specification of core and crosscutting behavior, and the specification will then be fed into the code generator to create AOSF code. Until these tools are designed and built it is hard to say exactly what relationship VFMS's and the AOSF will have to MDA. MDA requires automatic transformation of PIM and PSM models. We can easily imagine a tool to transform VFMS models into AOSF code to handle core (PIM) and crosscutting concerns (PSM). Whether this technique will be superior to current MDA tools will be studied in the future.

In addition to tool creation, we are examining other visual methods of specifying event reinterpretation than those in the figures in this paper. The models in this paper tend to be cluttered, we hope to come up with a cleaner method of describing event reinterpretation. In addition, with the current models it isn't clear which reinterpretations occur before or after the core.

## 7. CONCLUSION

The use of VFMSs can extend the life and maintainability of a system far longer than a system susceptible to platform changes. We believe that platform specific behavior is a crosscutting concern and can be modeled as such using the AOSF. By combining the AOSF and VFMS we can achieve the benefits of long-lived design and implementations that have less coupling and more reuse possibilities than traditional VFMS implementations.

## 8. REFERENCES

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

[2] 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.

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

[4] Flora-holmquist, A., Staskauskas, M. "Formal Validation of Virtual Finite State Machines" Proceedings of the workshop on Industrial-Strength Formal Specification Techniques 1995, pp 122-129

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

[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] Kleppe, A., Warmer, J., Bast, W. "MDA Explained The Model Driven Architecture: Practice and Promise" Addison-Wesley, 2003

[8] 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>

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

[10] Samek, M. 2002. "Practical Statecharts in C/C++". CMP Books.

[11] Wagner, F. "VFMS Executable Specification" on CompEuro 1992, Hague, Hollan