Stateful Aspects: The Case for Aspect-Oriented Modeling

Thomas Cottenier
Motorola Labs
Illinois Institute of Technology
thomas.cottenier@mot.com

Aswin van den Berg
Motorola Labs
aswin.vandenberg@mot.com

Tzilla Elrad
Illinois Institute of Technology
elrad@iit.edu

ABSTRACT

In this position paper, the authors argue that Aspect-Oriented Modeling (AOM) technologies have the potential to simplify the deployment and the ability to reason about a category of crosscutting concerns that have been categorized in the literature as stateful aspects. Stateful aspects trigger on a sequence of join points instead of on a single join point. Their trigger condition is history sensitive. We identify three properties of Aspect-Oriented Modeling languages that enable them to provide more natural solutions to the stateful aspect problem. The first factor is the ability of modeling languages to support different system decompositions paradigms. This ability allows a system to be decomposed according to the paradigm that fits the nature of the problem the best. Second, these decompositions allow AOM languages to capture and reason about pointcut descriptors at a higher level of abstraction. Finally, the system decomposition mechanisms can be exploited to advance the compositional expressiveness of aspects defined in AOM languages. The paper concludes with an invitation to our colleagues in the "general purpose" programming community to embrace modeling technologies in the context of automatic code generation.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces, Object-oriented design methods, State diagrams.* D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *Object-oriented programming, Structured programming.*

Keywords

Aspect-Oriented Modeling, Model-Driven Software Development, State machine diagrams

1. INTRODUCTION

The "jumping aspects" problem formulated by Brichau et al. [1] presents examples of crosscutting concerns for which it is desirable to change the behavior of aspects based on the history of the program execution. Brichau et al. further maintain that history-sensitive triggering conditions should be captured by pointcut descriptors, rather than being embedded in aspect advices.

The concept of history-sensitive pointcut descriptors enables aspects to capture the completion of a *protocol* of run-time events. Protocols define the permitted or unacceptable sequences of actions of a module and define the reactions of this module to external events, in particular contexts that depend on its state.

Protocols are pervasive in a wide range of computing applications, including communication systems, application servers, service-oriented architectures, user-interfaces and business processes.

Stateful aspect languages partially address the "jumping aspect" problem, by enabling aspects to recover the state of the system through pointcuts that recognize the significant events that bring the system in a particular state. Stateful aspect where introduced by Douence et al., following on their work on Event-Based AOP and trace-based aspects [2, 3, 4]. Trace-based aspects trigger on a sequence of join points instead of on a single join point. The triggering condition of an advice is history-sensitive. The formal model developed by Douence et al. was also conceived to control and analyze interactions among aspects and provide expressive aspect composition mechanisms.

Aspect-Oriented Programming languages that support stateful aspect, such as Declarative Event Patterns (DEP) [5] or JAsCo [6] provide language mechanisms to capture entire protocols, or particular sections of a protocol in pointcut descriptors. These pointcut descriptors implicitly, in the case of DEP, or explicitly in the case of JAsCo, maintain a finite state machine to determine the triggering condition of aspect advices.

Stateful aspect languages *recover* the state-machine based behavior of the system. We believe that the need for such mechanisms is the symptom of a problem that has deeper causes: the base system is not decomposed according to decomposition abstractions that fit the problem domain. Systems that implement protocols can be categorized as being *reactive*, discrete systems as opposed to *transformational* systems [7].

The overwhelming majority of Aspect-Oriented Programming languages are built on top of procedural, functional, object-oriented or component-based decomposition languages and frameworks. These languages provide excellent support for *transformational* systems, such as data-processing systems.

Yet, it is mostly recognized that these languages do not provide sufficient abstractions for the specification and implementation of *reactive* systems [7]. Reactive systems include modules that are repeatedly prompted by the outside world or other modules, and continuously respond to external inputs. The behavior of a reactive system is highly dependent on its history and needs to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Workshop AOM '07, March 12-13, 2007, Vancouver, British Columbia, Canada
Copyright 2007 ACM 1-59593-658-5/07/03... $5.00

maintain a continuous relationship with its environment. Examples include phones, automobiles, communication networks, avionics systems, and the user interface of many kinds of ordinary software [7].

The natural decomposition for transformational systems is the generalized procedure. A more natural decomposition for reactive behavior is the Harel Statechart [7], which extends state machine diagrams for scalability purposes. State machine implementation languages such as the Specification and Description Language (SDL) [8] have not been widely adopted in the "general purpose" software development community. Yet, they are widely used in the industry for the development of critical, real-time distributed systems, especially in the telecom and the avionics domain where reactivity is commonplace.

Increasingly, general-purpose applications need to accommodate both transformational and reactive modules. In fact, a large proportion of systems developed using procedural or object-oriented languages such as Java implement functionalities that are essentially of reactive nature. In Java, many widely used libraries such as AWT make heavy use of asynchronous events, handled by threads that are specifically spawned to continuously respond to multiple events in a responsive manner. Other examples are applications that handle client-service interactions, such as web or application servers, and business process engines. Reactive systems are therefore not only found in real-time, embedded and distributed system domains, but are in fact pervasive, across the general purpose computing landscape.

When a reactive system is implemented in an object-oriented style, the behavior that is implemented by a method tends to highly depend on the history of the object. During execution, the method needs to take important *decisions* about the behavior to execute in the particular state of the object. When crosscutting concerns apply to such behavior, the signature of a method simply does not provide sufficient information.

In a reactive world, the procedural decomposition does not provide the right hooks where aspects can apply. Stateful aspect languages therefore need to *recover* the natural state-machine based decomposition of the behavior of the system. We believe that stateful aspects can be expressed in a more elegant and natural way if the base system itself is defined according to its natural decomposition.

2. THE CASE FOR ASPECT-ORIENTED MODELING

The purpose of modeling languages is to provide different decomposition paradigms that can be used concurrently to specify and implement a system according to different perspectives. Each perspective focuses on a particular property or functionality of the system, while omitting other properties.

These different paradigms provide complementary decomposition mechanisms. UML Class Diagrams and Sequence Diagrams support the traditional procedural and Object-Oriented decomposition. Composite-Part diagrams capture the run-time structure of a system. Activity diagrams support a workflow-based decomposition and provide abstractions for specifying concurrency and synchronization. State-Machine diagrams support the specification and the implementation of reactive behavior.

The strength of modeling languages lies in the ability to decompose the modules of a system according to the decomposition paradigm that fits the problem domain best. Transformational behavior can be implemented in an object-oriented style using an action language or sequence diagrams, while reactive sections of the behavior can be implemented using state machines that execute actions along transitions or upon state entry and exit.

The use of adequate decomposition mechanisms simplifies the development of hybrid systems, but is absolutely essential to cope with the complexity of large reactive distributed systems. Furthermore, the decomposition that is used to structure the base system has an important impact on the abstractions that can be used for the integration of crosscutting concerns.

Modeling and Aspect-Oriented Modeling techniques enable pointcut descriptors to be specified at a higher level of abstraction and augment the composition capabilities of Aspects.

2.1 Stateful Aspects in JAsCo

Figure 1 illustrates a simple stateful aspect in JAsCo, adapted from [6]. We omit the JAsCo connector for simplicity reasons and assume the methods captured by the hook are bound to methods with the same signature. The hook captures a sequence of events, methodA - methodB - methodC and attaches an advice to methodC, whenever it is called in the context of this protocol. The advice replaces the execution of methodC and throws an exception indicating that the sequence is not allowed in the context in which this Aspect is deployed.

```
class ProtocolSecurityChecker {

   hook StatefulProtocolCheck {

      StatefulProtocolCheck(methodA(..arg),
                            methodB(..arg),methodC(..arg)){

         ATrans: call(methodA) > BTrans;
         BTrans: call(methodB) > CTrans;
         CTrans: call(methodC) > ATrans;
      }

      replace CTrans () {
         throw new SecurityException("This protocol on
              module"+thisJoinPoint.getClassName()
              +" is not allowed!");
      }
   }
}
```

Figure 1. A protocol security checker stateful Aspect in JAsCo

The stateful aspects in JAsCo paper [6] does not give an example of a module on which this aspect could be deployed on. Following the discussion in Section 1, we can tell that such a module exhibits reactive behavior: the protocol methodA - methodB – methodC is one possible succession of events that are executed in a particular state of the module, as a response to an external trigger.

Figure 2 gives an example of such a module. It is a representative pseudo-code sample of behavior typically encountered in Java Applets. Such reactive behavior is widespread in the user interface modules of many kinds of ordinary software.

```
1. void actionPerformed(ActionEvent evt) {
2.    switch(this.STATE){
3.      case Init:
4.         serviceA.methodA();
5.         if(evt.getSource() == Enter){
6.            serviceB.methodB();
7.            if(evt.getActionCommand() == "Previous"){
8.               serviceC.methodC();
9.               this.STATE = Previous;
10.              return;
11.           }else
12.           if(evt.getActionCommand() == "Next"){
13.              serviceD.methodB();
14.              this.STATE = Next;
15.              return;
16.           }
17.        }else
18.        if (evt.getSource() == Cancel){
19.         }
20.         break;
21.     case Previous:
22.        serviceA.methodE();
23.        break;
24.     case Next:
25.        if(evt.getActionCommand() == "Proceed"){
26.           serviceA.methodC();
27.        }
28.        break;
29.   }
30.   serviceB.methodB();
31.   this.STATE = Done;
32.   return;
33. }
```

Figure 2. Pseudo-code for the reactive behavior of a typical Java Applet

The applet acts as listener to user interfaces components such as Buttons and Checkboxes, and reacts to events by accessing some resources, performing some calculations or refreshing a display. The state of the Applet is explicitly captured by the *STATE* variable of the Applet class.

When applied to the Applet of Figure 2, the aspect of Figure 1 prohibits transitions from state *Init* to state *Previous*. Yet, it does so implicitly by monitoring the succession of events that are characteristic of this transition, methodA - methodB – methodC. The JAsCo "transitions" used in the pointcut descriptor of Figure 1 correspond to sections of the transition from *Init* to *Previous*. These sections are delimitated by *Decision Actions*. Decision Actions are conditional statements that have a significant impact on the future behavior of the module. They impact the *Next State* of the module.

Examples of Decision Actions are the highlighted conditional statements of lines 5, 18 and 7, 12. Note that the conditional statement of line 25 does not qualify as a Decision Action because it does not affect the future reactive behavior of the applet.

Without support for stateful sspects, the aspect of Figure 1 would need to explicitly check for the current state of the module it is applied to, which is symptomatic of the "jumping aspect" problem. This can either be done by directly accessing the variable that characterize its state such as the *STATE* variable and the *EventSource* argument using joinpoint context passing and the *if(…)* pointcut descriptor, or indirectly, by monitoring the execution of the events that characterize the state transition. The aspect advice then needs to mimic the decisions made by the base module in order to capture the intended behavior. A detailed knowledge about the internals of the module is required, which leads to undesirable coupling between base modules and aspects.

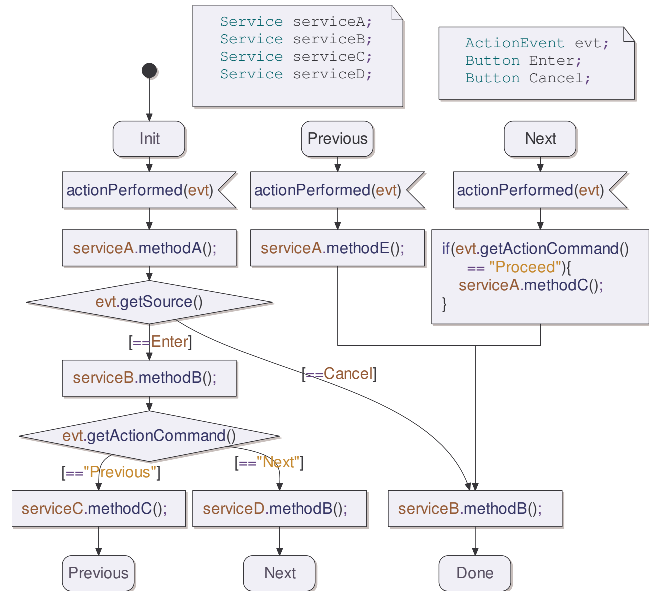

Figure 3. Implementation of the reactive behavior of the Applet as a transition-centric state machine

2.2 Model-Driven Implementation

A Model-Driven implementation of the same Applet module is illustrated in Figure 3. Figure 3 fully implements the behavior of the module of Figure 2 as a *transition-centric* state machine [9]. Such state machines are fully executable. They can be compiled into executable artifacts.

Compared with the code of Figure 2, the implementation model of Figure 3 makes the control flow of the reactive behavior more explicit. The actions executed along a transition and the control flow of a transition, for example from *Init* to *Done,* are clear from the first inspection, by following the flow lines that lead from state *Init* to state *Done*. In the case of the Java code, a developer needs to follow the code lines 3, 4, 5 and then jump to lines 17, 18, then 30, 31 to understand the flow of the transition.

The graphical representation of a modeling language helps in the understanding and the specification of complex software systems. Yet, the graphical representation is not the only advantage of modeling languages over traditional programming languages

The key difference between the implementation of Figure 2 and the implementation of Figure 3 is in the decomposition abstractions that are provided by the language. The Java implementation is decomposed according to a procedural decomposition - the generalized procedure is the main driver in the system decomposition. The implementation of Figure 3 is decomposed according to the state chart paradigm, where State, Transitions and Decision Actions are the dominant abstractions.

The modeling language provides explicit language constructs for distinguishing States of the system from ordinary variables and

make Decision Actions explicit, which are represented as diamonds. The language also checks that the ranges of Decision Actions such as *[==Enter]* and *[==Cancel]* are mutually exclusive and that their conjunction is always true, through static analysis.

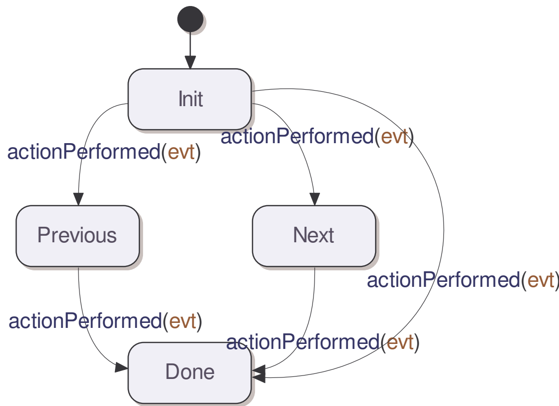

Figure 4. Specification of the reactive behavior of the Applet module as a state-centric state machine

2.3 Behavioral Module Specifications

Model-Driven Software Development methodologies encourage the definition of specifications of module behavior.

Figure 4 illustrates a specification that corresponds to the implementation of Figure 3 as a *state-centri*c state machine [9]. It defines the possible transitions of the system, but does not describe how those transitions are executed. In particular, it does not define how the decision is made whether to access states *Previous*, *Next* or state *Done,* starting from state *Init*. Such specifications are developed early in the life cycle of the system and semi-automatically validated though simulation. During the simulation of the system, the user is prompted to enter the information necessary for the execution and to test the conformance to the system requirements.

The conformance between the implementation of the system and its specifications is maintained during the development life-cycle through testing and basic model checking techniques.

The behavioral specifications of a module act as interfaces. They capture the assumptions that can be held for the module, without exposing its implementation details.

3. ABSTRACTIONS FOR STATEFUL ASPECTS

The authors' position can be summarized as follow:

1. The need for stateful aspects indicates that the base system exhibits reactive behavior
2. Being reactive, the base system is better decomposed according to a state machine decomposition language
3. Aspect-Oriented Modeling languages can capture stateful aspects in terms of module specifications as opposed to implementation-dependent sequences of events
4. Aspect-Oriented Modeling techniques can achieve better decoupling between aspects and base modules

The state machine implementation of Figure 3 provides for more natural ways to capture history-sensitive pointcut descriptors. Pointcut descriptors can be defined at a higher level of abstraction, in terms of the states of the system rather than the characteristic events executed along a transition. The use of state machine decompositions also makes it possible to augment the compositional expressiveness of aspect, as discussed in Section 4.

The techniques illustrated in this Section are implemented in a tool developed at Motorola, the Motorola ***WEAVR*** [10-13]. The ***WEAVR*** tool supports a profile to define aspects in Telelogic TAU [14] and performs weaving of state machines before code generation.

3.1 *WEAVR* Aspects

The WEAVR support two types of joinpoints: action joinpoints, which correspond to an action performed along a transition such as a method call or an object instantiation, and transition joinpoints. A transition joinpoint is either a state transition or a section of a state transition.

Figure 5 illustrates a ***WEAVR*** aspect that corresponds to the JAsCo stateful aspect of Figure 1. The aspect expresses that whenever a resource access brings the system in an unauthorized state, an exception should be thrown and the system should be forced into a failure state.

The aspect defines a new interface and contains a pointcut descriptor and a *connector*. A connector is the equivalent of an advice in AspectJ. The *throwSecurityException* connector is bound to the *securityViolation* pointcut descriptor through a <<binds>> dependency. The binding dependency specifies how context arguments and parameters are passed from joinpoints to *connector instances*. A connector instance is the connector behavior instantiated at a joinpoint.

The pointcut descriptor of Figure 5.b captures transitions from some state, triggered by a resource access event that brings the system in an *unauthorized* state.

The connector of Figure 5.c implements the equivalent of the *replace* advice of Figure 1. The connector invokes the reflective API of the ***WEAVR*** through the thisJoinPoint, thisTransition and thisStateMachine keywords. These keywords are used to retrieve information about the context of the joinpoint, but also specify the scope of connector introductions.

Connectors can introduce owned members such as variables and operations in classes, but they can also introduce new states and labels into state machines. The connector of Figure 5.c. introduces a new state, whose scope is the state machine of the joinpoint.

The connector is instantiated *before* the sections of transitions from some state to an unauthorized state, as indicated by the *Start* symbol in the connector state machine. These transitions are aborted by the connector, as it does not invoke the *proceed* keyword.

The aspect of Figure 5 is defined independently of the base modules. The specification of the aspect is defined by its pointcut descriptors and the specifications of its connectors. The aspect replaces transitions to unauthorized states by transitions to a new failure state, and throws an exception.

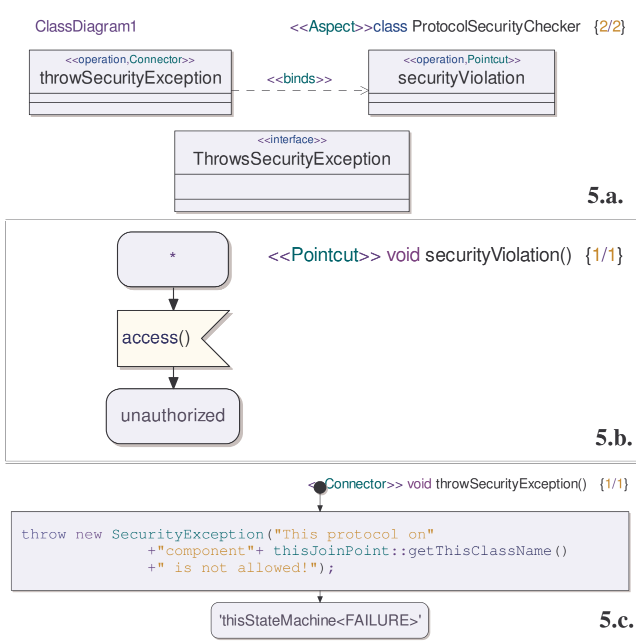

Figure 5. The protocol security checker Aspect implemented in *WEAVR*

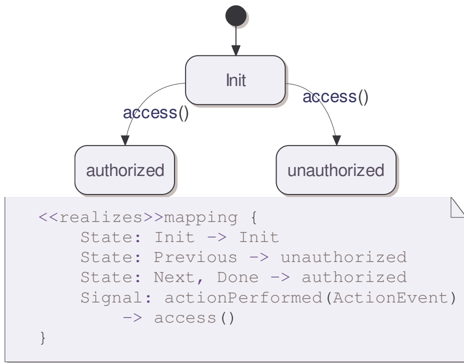

Figure 6. A perspective of the state machine specification of Figure 4 as defined by a realization mapping

3.2 *WEAVR* Specification Mappings

In order to deploy the aspect of Figure 5 on the Applet module, we need to map the specification of the Applet to a *perspective* that defines resource accesses and unauthorized states. This mapping is performed using realization relationships between state machine specifications. The mapping of Figure 6 defines the unauthorized state as being the *Previous* state, and a resource access as being the execution of the *actionPeformed* method.

Realization mappings provide perspectives of the specification of a module, which respect to a particular concern. They implicitly capture the awareness of the module with respect to the concern.

3.3 Model Weaving

In the ***WEAVR***, aspect pointcut descriptors are defined in terms of module specification elements but match points that might be located deep in the implementation of a module. The tool performs weaving of implementation state machines in terms of module specifications.

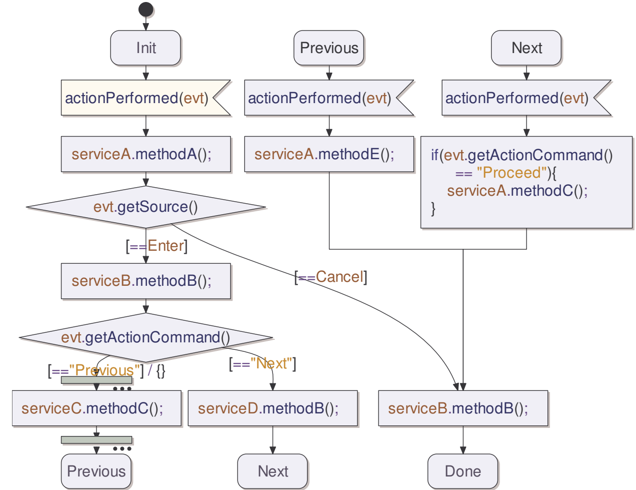

Figure 7. Section of a transition selected by the pointcut descriptor of Figure 5.b, through the mapping of Figure 6

Figure 7 shows the transition section selected in the implementation by the pointcut descriptor of Figure 5.b through the realization mapping of Figure 6, as visualized in the ***WEAVR*** tool.

The transition sections are delimitated by small shadowed delimitations boxes. The pointcut descriptor select the portions of the transition for which it can be statically determined that the module will transition into an *unauthorized* state, in this case, the *Previous* State.

The semantics of the selection mechanism are as follow. A transition pointcut designator from state S to state T triggered by i matches the complement of the execution paths from S to NOT T, triggered by i, from all the execution paths from S to T, triggered by i.

$$sel(S \xrightarrow{i} T) = \{ \bigcup path(S \xrightarrow{i} T) \} \setminus \{ \bigcup path(S \xrightarrow{i} NOT(T)) \}$$

This matching method is very powerful because it can localize the important decision points in the execution of a state machine.

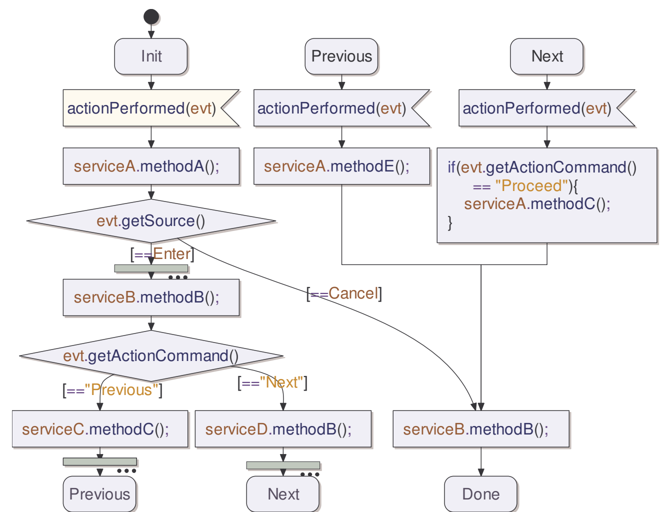

Figure 8. Section of the transition selected by the pointcut descriptor of Figure 5.b after modification of the mapping

The *before* location on a transition section always corresponds to a branch of a decision action. In the case of Figure 7, the decision selected is the *getActionCommand() == "Previous"* expression. The decision itself is a point in the implementation, but it can be characterized in terms of the module specification. The transition selection mechanism enables the ***WEAVR*** to capture locations in the implementation of modules, in terms of their specification.

Now, suppose that the application requires the user to be authenticated before entering any of the *Previous* and *Next* states. This modification can be implemented by modifying the state machine specification realization mapping as follow:

```
State: Previous, Next -> unauthorized
State: Done -> authorized
```

Figure 8 shows the corresponding transition sections in the state machine implementation. The decision action matched is the *getSource() == "Enter"* expression.

The same modification, when applied to the Aspect of Figure 1 would require the following modification in the transition sections of the pointcut descriptor:

```
ATrans: call(methodA) > BTrans;
BTrans: call(methodB) > CTrans;
CTrans: call(methodC) || call(methodB) > ATrans;
```

This modification necessitates a detailed inspection of the implementation of the module. It requires precise knowledge about the particular sequences of events that characterize a state transition.

Furthermore, the ***WEAVR*** aspect is able to detect unauthorized transitions earlier in the execution flow of the state machine than the JAsCo aspect. The ***WEAVR*** aspect matches a particular outcome of the *getSource()* decision, whereas the JAsCo aspect matches call to *methodC()* or *methodB(),* occurring after calls to *methodA()* and *methodB().*

In order to capture the *getSource()* decision action, Aspect-Oriented Programming languages would need to match the decision explicitly, which would couple the aspect tightly to the implementation of the base module. State machine implementation languages allow stateful aspects to be expressed at a higher level of abstraction, in terms of the specification of a module rather than its implementation. The awareness of the module with respect to the crosscutting concern is implicitly captured by a perspective of its specification, defined by a realization mapping.

4. COMPOSABILITY OF ASPECTS

The control flow structures provided by state machines (states, decision actions and labels) make it possible to introduce complex control flow structures in the system. These augment the conciseness and composition expressiveness of state machine aspects.

Aspects can introduce states and labels into state machines. A state or label introduction occurs when a connector refers to a state or a label that is not declared in its pointcut descriptors. When introducing states and labels, the scope of the introduction needs to be specified. States and labels can be introduced per state machine, per Transition or per Joinpoint. A state that is introduced per Joinpoint, *thisJoinPoint<state_name>*, is unique to the connector instance bound to a specific joinpoint. A state that is introduced per transition, *thisTransition<state_name>* is shared by all connector instances bound to joinpoints of the same transition. State and label introductions make it possible to define behavior that spans across different instantiations of a connector.

The action joinpoints within a transition are partially ordered. A connector can therefore reference states and labels introduced by other connector instances using the *previousJoinPoint <state_name, default_state>* notation. *previousJoinPoint* resolves to a default state in the case the previous joinpoint is not defined.

Figure 9 shows a transaction atomicity aspect that makes use of label and state introductions. The pointcut descriptor of Figure 9.b. matches invocations of recoverable resources that return an error code of type *ERR_t*. Failure of the resource access is detected through the return value of the invocation.

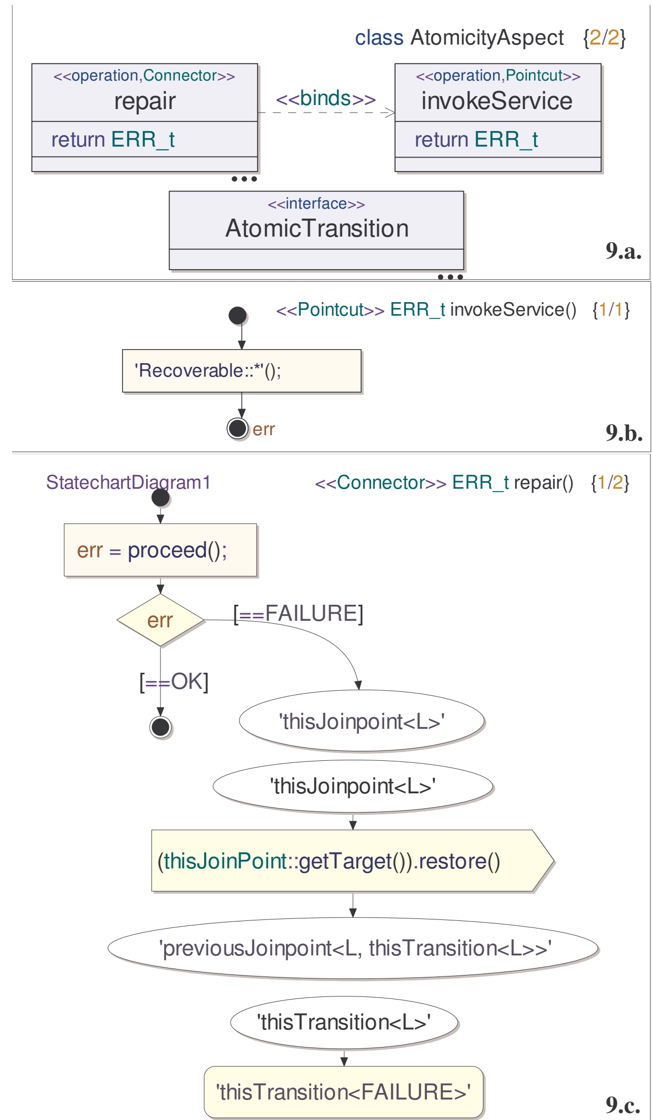

Figure 9. An Aspect that implements the atomicity property along state transitions

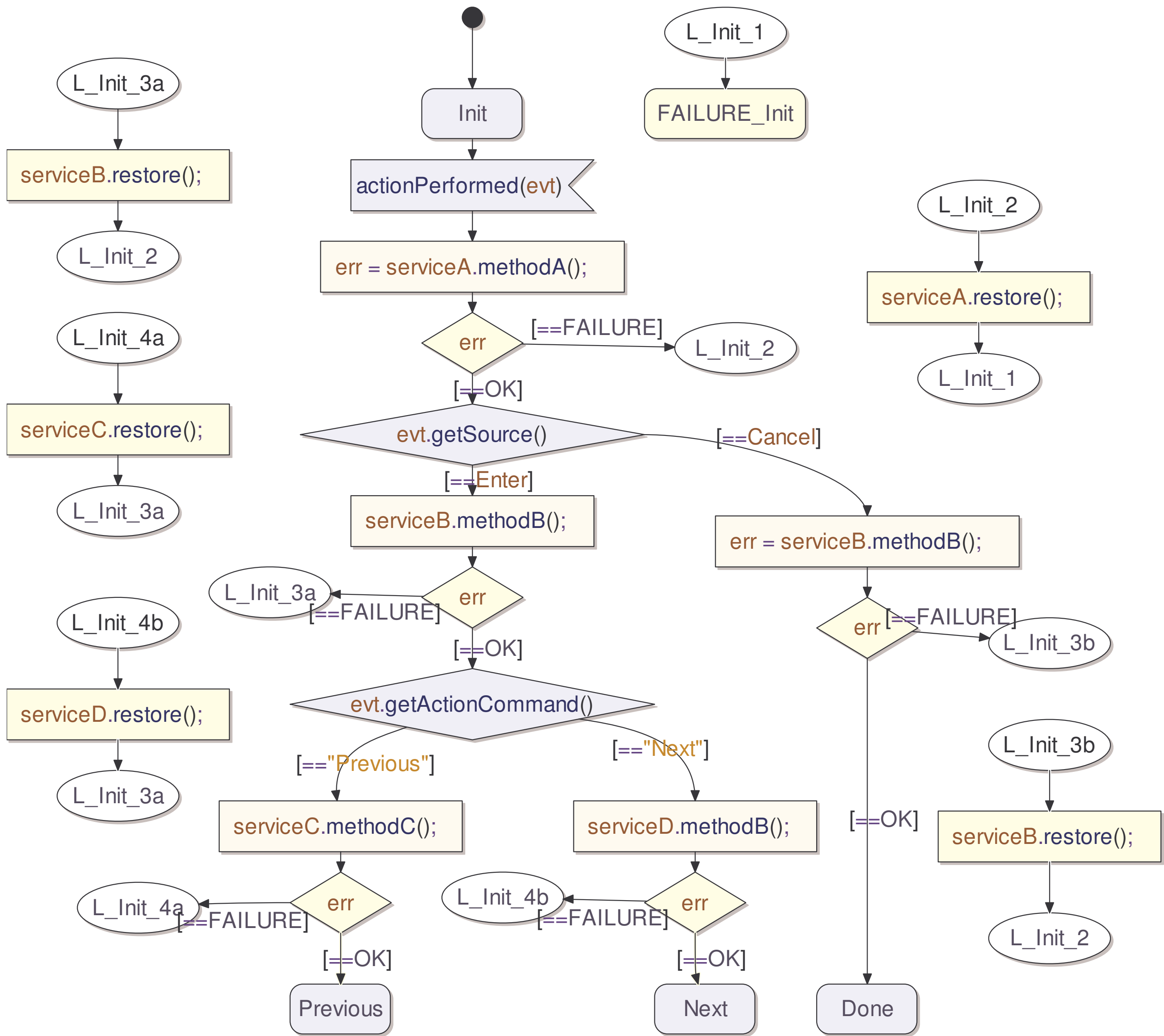

Figure 10. Partial representation of the result of weaving the Transition Atomicity aspect in the Applet module of Figure 3

Whenever a resource access fails, the aspect connector of Figure 9.c. ensures that all the recoverable resources accessed along this transition are restored to the state in which they were before the transition was fired. Resource recovery is performed by invoking the restore *method* on recoverable resources.

The connector builds a control structure that corresponds to the reversed order in which resources are accessed, by referring to the label introduced by the previous connector instantiation at a joinpoint along this transition. The connector also introduces a per transition state which characterize the failure of the transition, *thisTransition<FAILURE>* and a default label for this transition, *thisTransition<L>*. This state can be referred to by other connectors, in order to provide a recovery mechanism.

These constructs allow the ***WEAVR*** to construct complex control structures, as shown in Figure 10. The first joinpoint encountered along the transition is the call to serviceA.methodA(). This joinpoint does not have a previous joinpoint. The *previousJoinpoint* label therefore resolves to the default label for this transition, *thisTransition<L>*.

At the code level, the implementation of the corresponding aspect would be problematic. The use of state machine based abstractions augments the compositional expressiveness of aspects.

In the case of Figure 10, the ordering relationships between joinpoints can be resolved statically. In the more general case, the runtime environment needs to maintain a jump table to resolve the precedence relationships between joinpoints at runtime.

Figure 10 is a partial *representation* of the result of weaving the aspect of Figure 9 in the state machine of Figure 3. Developers are not supposed to inspect woven models. The weaving is only performed at the level of the model and not at the level of the model presentation.

5. CONCLUSIONS

We believe that the need for stateful aspects is a symptom that the base behavior of the system is not decomposed according to abstractions that fit its problem domain. Modules that exhibit reactive behavior are more naturally decomposed using state machine implementation language such as SDL rather than a procedural or OO programming language.

The use of state machine implementation languages allow stateful aspects to be defined at a higher level of abstraction, in terms of the specification of the system, as opposed to implementation-dependent sequences of events.

This is beneficial both in terms of comprehensibility and modularity. Aspects do not need to be aware of the particular sequence of events that characterize a state transition. Furthermore, the use of state machine abstractions, such as states and transitions allow aspects to build control structures that spawn multiple advice instantiations. This ability augments to compositional expressiveness of aspects.

The uses of state machine implementation languages is widespread in the industry, especially in the telecom and avionics domains where reactivity is commonplace, but have not been widely adopted in the "general purpose" programming community. Increasingly, applications need to accommodate both transformational and reactive behavior. There is therefore a strong incentive to adopt languages that can accommodate both paradigms simultaneously, such as modeling languages, in a Model-Driven Software Development environment.

6. REFERENCES

[1] Brichau, J, De Meuter, W, de Volder, K. Jumping Aspects. (position paper), In Workshop on Aspects and Dimensions of Concern at the European Conference on Object-Oriented Programming, Cannes, France, 2000.

[2] Douence, R., Fradet, P., Sudholt, M. A framework for the detection and resolution of Aspect interactions. In Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Module Engineering, Pittsburgh, USA, LNCS 2487, pp.173-188, Springer-Verlag, October 2002.

[3] Douence, R., Fradet, P., Sudholt, M. Composition, Reuse and Interaction Analysis of Stateful Aspects. In Proceedings of the 3th International Conference on Aspect-Oriented Software Development, Lancaster, UK, pp. 141-150, ACM Press, March 2004.

[4] R. Douence, P. Fradet, and M. Sudholt. Trace-based Aspects. In Aspect-Oriented Software Development, pp 201-218, Addison Wesley, September 2004.

[5] Walker, R.J., Viggers, K. Implementing Protocols via Declarative Event Patterns. In Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Newport Beach, USA, pp. 159-169, ACM Press, November 2004.

[6] Vanderperren, W., Suvee, D., Cibrán, M. A., De Fraine, B. Stateful Aspects in JAsCo, Software Composition Workshop at the European Joint Conferences on Theory and Practice of Software, Edinburgh, Scotland, LNCS 3628, pp. 167-181, Springer-Verlag, April 2005.

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

[8] ITU, Z. 100: Specification and Description Language (SDL), International Telecommunication Union, 2000.

[9] Bjorkander, M.: Graphical Programming Using UML and SDL, IEEE Computer, 33(12), pp. 30-35, IEEE Press, December 2000.

[10] Cottenier, T., van den Berg, A., Elrad, T. The Motorola WEAVR: Model Weaving in a Large Industrial Context. in Proceedings of the International Conference on Aspect-Oriented Software Development, Industry Track, Vancouver, Canada, 2006

[11] Cottenier, T., van den Berg, A., Elrad, T. Motorola WEAVR: An Add-In for Aspect-Oriented Modeling in TAU. Telelogic User Group Conference, Denver, Colorado, USA, 2006

[12] Zhang, J., Cottenier, T., van den Berg, A., Gray, J., Aspect Interference and Composition in the Motorola Aspect-Oriented Modeling Weaver. Workshop on Aspect-Oriented Modeling at the 9th International Conference on Model Driven Engineering Languages and Systems, Milan, Italy, 2006

[13] Cottenier, T., van den Berg, A., Elrad, T. Modeling Aspect-Oriented Compositions. Proceedings of the Satellite Events at the 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, LNCS 3844, pp. 100-109, Springer-Verlag, 2005

[14] Telelogic. TAU homepage, http://www.telelogic.com/products/tau/index.cfm, 2005.