

Towards Aspect-Oriented State Machines

Gefei Zhang

Ludwig-Maximilians-Universität München
gefei.zhang@pst.ifi.lmu.de

Abstract

UML state machines provide an operational view of the behavior of software systems. However, properties of the execution history of state machines cannot be expressed modularly. This often leads to model elements addressing the same concern scattered all over the machine. We present an initial approach to aspect-oriented state machines, which show considerably better modularity in designs of history dependent behavior than normal UML state machines.

1 Introduction

Aspect-oriented Software Development (AOSD [7]) aims at achieving a clear separation of concerns in the entire process of software development. In particular, in the design phase, aspect-oriented modeling could help to increase the modularity of software models and thus make them less error-prone and better maintainable (see e.g. [5, 19]). Separating concerns of GUI and business logic, content and access control etc. in software design would be no less advantageous than in program code. However, aspects are not yet sufficiently supported in models. There still does not exist a general purpose aspect-oriented modeling language. This makes it difficult, if not impossible, for software designers to understand, specify, document their systems and communicate in aspects.

The Unified Modeling Language (UML) is the *lingua franca* in object-oriented software analysis and design. It provides, among others, state machines as an operational view of the behavior of software systems. However, state machines are not aspect-oriented. Their execution semantics as described in the UML specification [15] considers state machines as memoryless automata: state transitions fired by an event are determined only by the state of the system at the time point the event is dispatched. Properties of the execution history thus cannot be expressed modularly. Instead, the modeler has to keep track of the execution by programming with additional variables. The resulting “program”, even though addressing one concern, is often scat-

tered all over the state machine.

We propose an initial approach to aspect-oriented state machines. We extend the execution semantics of UML state machines to make these history-aware. Aspect-oriented state machines thus allow the modeler to specify alternative behavior in dependence of the execution history modularly.

The rest of this paper is organized as follows: We give a brief overview of UML state machines in Sect. 2 and show their deficiency in the support of modeling history-dependent behavior in Sect. 3. An initial definition of history-based aspect-oriented state machines is given in Sect. 4. Related work is discussed in Sect. 5. Finally, we conclude by outlining some future work.

2 UML State Machines

The following overview of UML state machines and the automated teller machine (ATM) example used are both adapted from [3].

Figure 1 shows the state machine of an ATM. It verifies a bank card and the PIN. Money is dispensed if both of them are valid, otherwise, our ATM simply gets idle again.

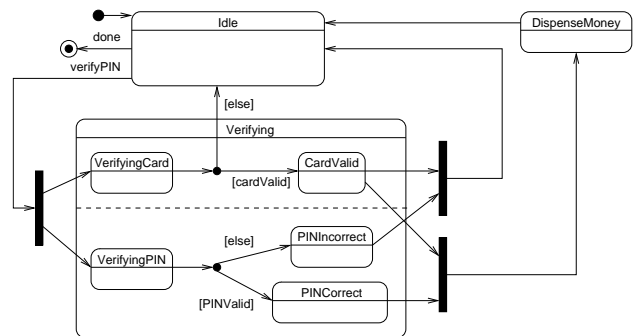


Figure 1. State machine for an ATM

A state machine consists of a finite set of *states* and a finite set of *transitions* between states. States may be *simple* (such as Idle and DispenseMoney) or *composite* (such as Verifying); a *concurrent* composite state contains several other composite states, called *orthogonal regions* of the

```

1 executeStateMachine( $SM$ )  $\equiv$ 
2    $activeStates \leftarrow initActiveStates(SM)$ 
3   while  $SM.Evt \neq \phi$  do
4      $e \leftarrow dequeue(SM.Evt)$ 
5      $ET \leftarrow \{t \in SM.T \mid enabled(t)\}$ 
6     foreach  $t \in ET$  do
7        $activeStates \leftarrow activeStates \setminus \{t.src\}$ 
8        $runAction(t.action)$ 
9        $activeStates \leftarrow activeStates \cup \{t.target\}$ 
10    od
11  od

```

Figure 2. Simplified execution schema of UML state machines

concurrent state and separated by dashed lines. Moreover, *fork* and *join* (pseudo)-states, shown as bars, synchronize several transitions to and from orthogonal regions; *junction* (pseudo)-states, represented as filled circles, chain together multiple transitions. States and pseudostates are also called statevertices. When a statevertex is contained in a composite state, we say it is a subvertex of its container. The relationship subvertex-container is a 1:n-association, thus the statevertex hierarchy of a state machine forms a tree structure, with a composite state top as root, representing the entire machine. Transitions between states are triggered by *events*. Transitions may also be guarded by conditions and specify actions to be executed or events to be emitted when the transition is fired. For example, the transition from the state *Idle* to the fork pseudostate requires signal *verifyPIN* to be present; the transition branch from *VerifyingCard* to *CardValid* requires the guard *cardValid* to be true.

The actual state of a state machine is given by its *active state configuration* and its *event queue*. The active state configuration of a state machine is the set of currently active states. In particular, when a (simple or composite) state is active, so is its container, too. An active state configuration thus is also a tree. When a concurrent composite state is active, so is also each of its regions. If a non-concurrent composite state is active, then exactly one of its substates is also active.

The execution of a state machine consists in changing its active state configuration in dependence of the active state configuration and the first element in the event queue. We call the change from one active state configuration to another an execution step. Given a state machine SM , let $SM.S$ be its set of states, $SM.T$ the set of its transitions, and $SM.Evt$ its event queue. For each $t \in SM.T$ we write $t.src \in SM.S$ and $t.target \in SM.S$ for the source and target state of t , resp. Let Act be some set of actions of the system, t may be associated with some $t.action \in Act$, which

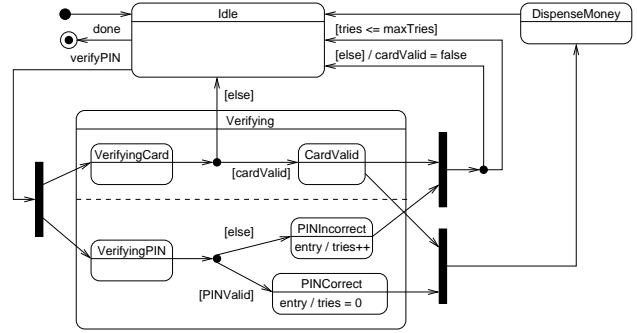


Figure 3. Limiting authentication failures in UML state machines: scattered design

is executed when t takes place. Besides, if t is guarded by any condition we denote it as $t.guard$. The execution of SM as specified in the UML specification is sketched in Fig. 2 (This is a very abstract view, for a detailed formal semantics see [11]). The function *initActiveState* calculates the first active state configuration; *dequeue* returns the first element of a queue and removes it; *runAction* executes an action. The value of *enabled(t)* is true iff $t.src \in activeStates$ and $t.guard$, if any, is satisfied. Note that this algorithm only keeps track of the current one active state configuration. The execution history is simply forgotten.

3 Insufficient Modularity of UML State Machines

Due to their historylessness UML state machines are often insufficiently modular. In particular, the necessity of keeping track of the execution history by means of additional variables may lead to model elements addressing the same concern scattered all over the state machine.

A useful feature of an ATM may be limiting how many times in a row an invalid PIN may be inputted. When the upper limit *maxTries* is reached, the ATM should make the card invalid. Once the valid PIN is inputted, the previous failed tries are forgotten.

Figure 3 extends the UML state machine shown in Fig. 1 to model this feature: besides an additional variable *tries*, which is needed as a counter and has to be initialized with 0 when the card is created, two actions must be introduced, one for incrementing *tries* when the input is wrong, the other for resetting *tries* when the input is right. Moreover, the transition from the join point after *CardValid* must be split into two for the cases of whether the limit is reached or not. It is obviously unsatisfactory that adding such a simple feature to the original state machine causes modifications almost everywhere in it.

dependent behavior, aspects do not have to be dynamic. In fact, static, history independent aspects are simply special cases of dynamic, history-based aspects, where the pointcut is not dependent of the entire execution history, but only the last state of it.

Figure 6 gives an example. This aspect extends the ATM with the feature of waiting for the user to input the desired amount of money after successful verification, checking if the account is covered and, if not, stopping dispensing money. The *before* pointcut is satisfied when the state `DispenseMoney` is about to become active and it will then be the last state in the execution history (visualized by the circle \circ in the state). Note that when a final pseudostate named `n` is reached, the execution of the advice is terminated, and the state with the name `n` in the base model gets active. The transition of the *else* branch thus makes our ATM idle again and prevents `DispenseMoney` from getting active. Static aspects like this may be used to specify model transformations in the context of *Model Driven Architecture* (MDA).

5 Related Work

Dynamic aspect-oriented programming is an active research field. Advantages of JAsCo [18], Object Teams [9] etc. over languages with no or only limited support for dynamic aspects like AspectJ [10] are getting visible (cf. [8]). Our history-based aspects are reminiscent of the trace-based aspects in Arachne [6].

On the level of software design, however, aspect-oriented modeling is still in its infancy. Most existing work (see e.g. [4, 1, 16]) focuses rather on representing aspect-oriented programs in models than making modeling languages aspect-oriented. Altisen et. al [2] both model reactive programs and define aspects with Mealy automata. Due to the acceptance of the UML as well as the more “user-friendly” pointcut syntax our approach seems to be better applicable as a modeling language. Aspect-Oriented Statechart Framework (AOSF [13]) weaves several independent state machines into one concurrent composite state. In comparison to our approach, AOSF supports only static aspects. Besides, our advice in the form of a state machine is also more powerful.

Existing join point selection approaches such as [17] may be used in our approach, if extended by quantification over execution history of state machines.

6 Conclusions and Future Work

We have given an initial design of aspect-oriented state machines, which help to design history dependent behavior modularly.

Currently, our aspect-oriented state machines consider only the changes of the active state configuration. An ex-

ension to including also variables is straight-forward and expected to be useful. Moreover, it may be also interesting to include quantification over previous transitions. We intend to investigate the usefulness of such extensions in case studies.

The advice can be made more powerful too. In particular, it would be desirable to refer to previous active states. This would e.g. facilitate to model modularly roll backs in transactional applications.

An important issue of future work is model verification. We plan to extend the UML state machine model checker Hugo/RT¹ [12] to check aspect-oriented state machines. However, model checking requires a formal semantics. We are now working on an operational semantics of our aspect-oriented state machines.

Finally, code generation in an aspect-oriented programming language and introducing aspect-orientation into other UML diagrams are also planned.

Acknowledgments

This research has been partially sponsored by the Deutsche Forschungsgemeinschaft (DFG) within the project MAEWA (WI 841/7-1). The author also wishes to thank Matthias Hölzl and the anonymous reviewers for valuable comments and suggestions.

References

- [1] O. Aldawud, T. Elrad, and A. Bader. UML Profile for Aspect-Oriented Software Development. In *3rd Int. Wsh. Aspect-Oriented Modeling (AOM)*, Boston, March 2003.
- [2] K. Altisen, F. Maraninchi, and D. Stauch. Aspect-Oriented Programming for Reactive Systems: Larissa, a Proposal in the Synchronous Framework. *Sci. Comp. Prog., Special Issue Foundations of Aspect-Oriented Programming*, 2006. To appear.
- [3] M. Balsler, S. Bäumler, A. Knapp, W. Reif, and A. Thums. Interactive Verification of UML State Machines. In J. Davies, W. Schulte, and M. Barnett, editors, *Proc. 6th Int. Conf. Formal Methods and Software Engineering (ICFEM'04)*, volume 3308 of *Lect. Note Comp. Sci.*, pages 434–448. Springer-Verlag, 2004.
- [4] E. L. A. Baniassad and S. Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. In *Proc. 26th Int. Conf. Software Engineering (ICSE'04)*, pages 158–167. IEEE, 2004.
- [5] H. Baumeister, A. Knapp, N. Koch, and G. Zhang. Modelling Adaptivity with Aspects. In D. Lowe and M. Gaedke, editors, *5th Int. Conf. Web Engineering (ICWE'05)*, volume 3579 of *Lect. Notes Comp. Sci.*, pages 406–416. Springer-Verlag, 2005.

¹<http://www.pst.ifi.lmu.de/projekte/hugo>

- [6] R. Douence, T. Fritz, N. Lorient, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An Expressive Aspect Language for System Applications with Arachne. In Mezini and Tarr [14], pages 27–38.
- [7] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.
- [8] R. E. Filman, M. Haupt, and R. Hirschfeld, editors. *Proc. 2nd Dynamic Aspects Wsh. (DAW'05)*. Technical report 05.01. Research Institute for Advanced Computer Science, 2005.
- [9] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In M. Aksit, M. Mezini, and R. Unland, editors, *Rev. Papers Int. Conf. NetObjectDays (NODE'02)*, volume 2591 of *Lect. Notes Comp. Sci.*, pages 248–264. Springer-Verlang, 2003.
- [10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *Proc. 15th Eur. Conf. Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lect. Notes in Comp. Sci.*, pages 327–353. Springer-Verlag, 2001.
- [11] A. Knapp. Semantics of UML State Machines. Technical Report 0408, Institut für Informatik, Ludwig-Maximilians-Universität München, 2004.
- [12] A. Knapp, S. Merz, and C. Rauh. Model Checking Timed UML State Machines and Collaborations. In W. Damm and E. R. Olderog, editors, *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 2469 of *Lect. Notes Comp. Sci.*, pages 395–416. Springer-Verlag, 2002.
- [13] M. Mahoney, A. Bader, T. Elrad, and O. Aldawud. Using Aspects to Abstract and Modularize Statecharts. In *Proc. 5th Wsh. Aspect-Oriented Modeling*, Lisboa, 2004.
- [14] M. Mezini and P. L. Tarr, editors. *Proc. 4th Int. Conf. Aspect-Oriented Software Development (AOSD'05)*. ACM, 2005.
- [15] Object Management Group. Unified Modeling Language Specification, Version 1.5. Specification, OMG, 2003. <http://www.omg.org/cgi-bin/doc?formal/03-03-01>.
- [16] D. Stein, S. Hanenberg, and R. Unland. A UML-based Aspect-oriented Design Notation for AspectJ. In *Proc. 1st Int. Conf. Aspect-Oriented Software Development (AOSD'02)*, pages 106–112, 2002.
- [17] D. Stein, S. Hanenberg, and R. Unland. Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design. In R. E. Filman, editor, *Proc. 5th Int. Conf. Aspect-Oriented Software Development (AOSD'06)*, pages 15–26. ACM, 2006.
- [18] W. Vanderperren, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive programming in JAsCo. In Mezini and Tarr [14], pages 75–86.
- [19] G. Zhang, H. Baumeister, N. Koch, and A. Knapp. Aspect-Oriented Modeling of Access Control in Web Applications. In *6th Int. Wsh. Aspect Oriented Modeling (AOM'05)*, Chicago, 2005.