Enhancing UML State Machines with Aspects

Gefei Zhang, Matthias Hölzl, and Alexander Knapp[⋆]

Ludwig-Maximilians-Universität München
{gefei.zhang,matthias.hoelzl,alexander.knapp}@pst.ifi.lmu.de

Abstract. Separation of Concerns (SoC) is an important issue to reduce the complexity of software. Recent advances in programming language research show that Aspect-Oriented Programming (AOP) may be helpful for enhancing the SoC in software systems: AOP provides a means for describing concerns which are normally spread throughout the whole program at one location. The arguments for introducing aspects into programming languages also hold for modeling languages. In particular, modeling state-crosscutting behavior is insufficiently supported by UML state machines. This often leads to model elements addressing the same concern scattered all over the state machine. We present an approach to aspect-oriented state machines, which show considerably better modularity in modeling state-crosscutting behavior than standard UML state machines.

1 Introduction

Separation of Concerns (SoC) is an important issue in software engineering. A clear SoC could improve the modularity of software artefacts, reduce the complexity of software systems, and thus make them less error-prone and better maintainable.

Recent advances in programming language research propose the use of Aspect-Oriented Programming (AOP [11]) to achieve a better SoC in programs. AOP helps in particular to solve the problem of "scattered code": code, that would be otherwise scattered all over the program, may be collected together in a new language construct called *aspect*. This way, AOP is particularly helpful for separating cross-cutting concerns, i.e. concerns that are involved in other concerns, like logging or transaction management.

However, not only code may be scattered all over a program, but also model elements all over a software design model. In particular, the Unified Modeling Language (UML [16]), the *lingua franca* in object-oriented software analysis and design, lacks aspect-like language constructs to address the problem of "scattered model elements" by centralizing model elements involved in one concern at a dedicated location.

We propose to extend the UML with aspect-oriented language concepts. In particular, we present a design of aspect-oriented state machines. Aspect-oriented state machines show considerably better modularity in, among others, the design of state-crosscutting behavior, such as state synchronization and trace-based behavior.

The remainder of this work is organized as follows: in Sect. 2 we summarize the syntax and semantics of UML state machines and in Sect. 3 we demonstrate some of

[⋆] This work has been partially sponsored by the EU project SENSORIA (IST-2005-016004) and the DFG project MAEWA (WI 841/7-1).

their weaknesses w.r.t. SoC. In Sect. 4 we present our proposal to aspect-oriented state machines and show how they may be used to achieve a better SoC by modularizing state interaction and trace-based behavior. An algorithm for for translating aspects into UML state machines is presented in Sect. 5. Related work is discussed in Sect. 6. Finally, we conclude and outline some future work.

2 UML State Machines

A UML state machine provides a behavioral view of its context. Figure 1 shows a state machine of a process which contains two parallel threads. After creation, the process is first initialized in the state Init, then the two threads run parallel in the state Running, until they receive the events astop and bstop while they are in the states A3 and B3, respectively. In this case each thread waits for the other to receive his stop signal in waiting states A4 and B4, respectively, before the threads terminate conjointly.

We briefly review the syntax and semantics of UML state machines according to the UML specification [16] by means of Fig. 1. A UML state machine consists of *regions* which contain *vertices* and *transitions* between vertices. A vertex is either a *state*, where the state machine may dwell in and which may show hierarchically contained regions; or a *pseudo state* regulating how transitions are compound in execution. Transitions are triggered by *events* and describe, by leaving and entering states, the possible state changes of the state machine. The events are drawn from an *event pool* associated with the state machine, which receives events from its own or from different state machines. The *context* of a state machine, a UML classifier, describes the features, in particular, the attributes, which may be used and manipulated in execution.

A state of a state machine is *simple*, if it contains no regions (such as Init and all states contained in Running in Fig. 1); a state is *composite*, if it contains at least one region; a composite state is said to be *orthogonal* if it contains more than one region, visually separated by dashed lines (such as Running). Each state may show an *entry* behavior (like actB2 in B2), an *exit* behavior (like actA2 in A2), which are executed on activating and deactivating the state, respectively; a state may also show a *do activity* (like in Init) which is executed while the state machine sojourns in this state. Transitions are triggered by events (a12, a23), show guards (condB), and specify actions to be executed when a transition is fired (actA23). Completion transitions (transition leaving

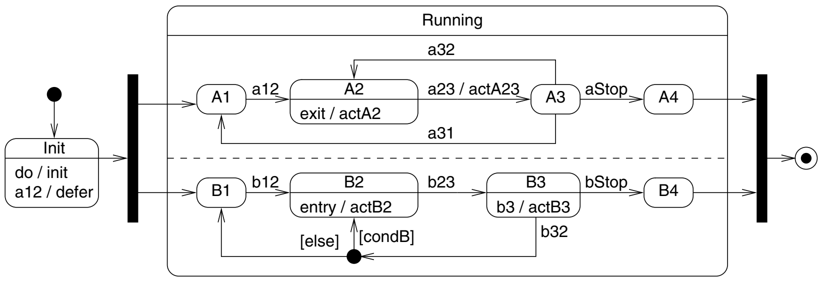


Fig. 1. State machine of a process containing two parallel threads

Init) are triggered by an implicit *completion event* emitted when a state completes all its internal activities. Events may be *deferred* (as a12 in Init), that is, put back into the event pool, if they are not to be handled currently but only later on. By executing a transition its source state is left and its target state entered; transitions, however, may also be declared to be *internal* to a state (b3 / actB3), thus skipping the activation-deactivation scheme. An *initial* pseudo state, depicted as a filled circle, represents the starting point for the execution of a region. A *final* state, depicted as a circle with a filled circle inside, represents the completion of its containing region; if all regions of a state machine are completed the state machine terminates. *Junction* pseudo states, also depicted as filled circles (see lower region of Running), allow for case distinctions. Transitions to and from different regions of an orthogonal composite state can be synchronized by *fork* and *join* pseudo states, presented as bars. For simplicity, we omit the other pseudo state kinds (entry and exit points, shallow and deep history, and choice).

During runtime, a state gets active when entered and inactive when exited as a result of a transition. The set of currently active states is called the active *state configuration*. When a state is active, so is its containing state. The active state configuration is thus a set of trees starting from the states in the top-level regions down to the innermost active substates. The execution of a state machine consists in changing its active state configuration in dependence of the current active states and a *current event* dispatched from the event pool. We call the change from one state configuration to another an *execution step*. First, a maximally consistent set of prioritized, enabled compound transitions is chosen. Transitions are combined into *compound transitions* by eliminating their linking pseudo states; for junctions this means to combine the guards on a transition path conjunctively, for forks and joins to form a fan-out and fan-in of transitions. A compound transition is *enabled* if all of its source states are contained in the active state configuration, its trigger is matched by the current event, and its guard is true. Two enabled compound transitions are consistent if they do not share a source state; an enabled compound transition takes priority over another enabled compound transition if its source states are below the source states of the other transition in the active state configuration. For each compound transition in the set, its least common ancestor (LCA) is determined, i.e. the lowest composite state containing all the compound transition's source and target states. The compound transition's main source state, i.e., the direct substate of the LCA containing the source states, is deactivated, the transition's actions are executed, and its target states are activated.

Given a state machine M, we denote by the sets $states(M)$, $transitions(M)$, $compounds(M)$ and $events(M)$ the states, transitions, compound transitions and events of M. Given a state $s \in states(M)$, we write $compounds(M, s)$ for the compound transitions leaving s. Given a transition $t \in states(M)$ we write $trigger(t)$ for the event triggering t and $guard(t)$ for the guard of t.

3 Pervasive Modification of State Machines

UML state machines work fine as long as the only form of communication among states is the activation of the subsequent state via a transition. More often than not, however, an active state has to know how often some other state has already been active and/or if

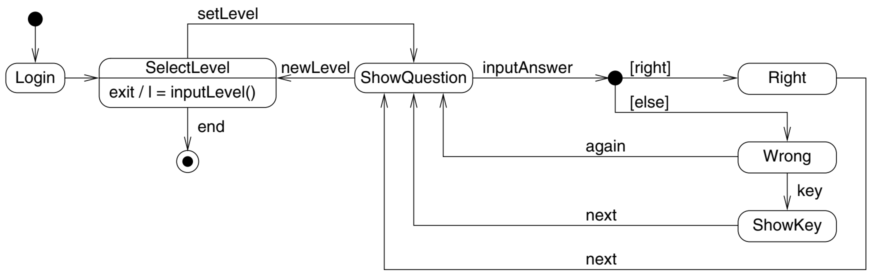


Fig. 2. State machine of an e-learning system

other states (in other regions) are also active. Unfortunately, behavior that depends on such information cannot be modeled modularly in UML state machines.

For an example, consider an e-learning system as modeled in Fig. 2: the user first logs in, selects a level of difficulty, and then proceeds to answering questions of this level. The system tells him whether his answer is right or not, if it is right, the user may proceed to the next question; if not, he may choose to try again, to see the key to the question, or to proceed to the next question. Instead of answering the current question, the user may also choose to go to another level.

Suppose the selection of a level $l > 0$ should be allowed only if the user has already answered minRight questions of level $l-1$ in a row, otherwise the system should give an error message. Figure 3 shows how this restriction might be modeled in a standard UML state machine: a new attribute variable crir is introduced for counting the length of the current stroke of correct answers (current right in a row), it is incremented in state Right and reset to 0 once Wrong is active. An array r is introduced to store the maximal length of crir at each level. Once the user gives a wrong answer, the system has to check if this record should be updated. In order to know whether the user has selected in SelectLevel a different level than the current one or just continues with the same level, another new variable cl stores the current level each time SelectLevel is entered. The transition from SelectLevel to ShowQuestion is split into two to handle the cases whether the level selected by the user is selectable or not. Finally, the variable crir has to be reset to 0 when the user has successfully changed to another level. Obviously, it is unsatisfactory

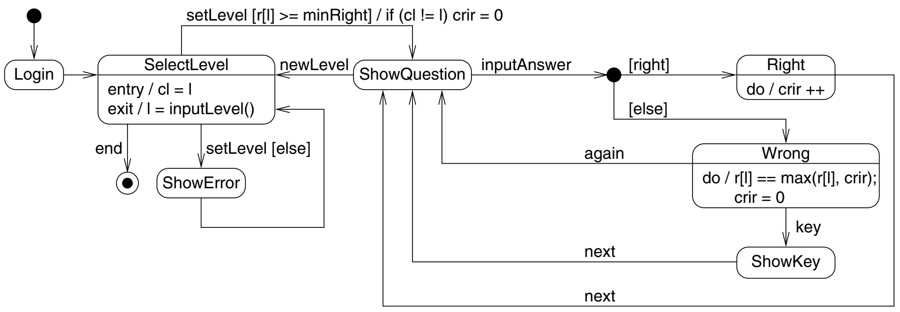


Fig. 3. Modeling the level selection restriction using standard UML

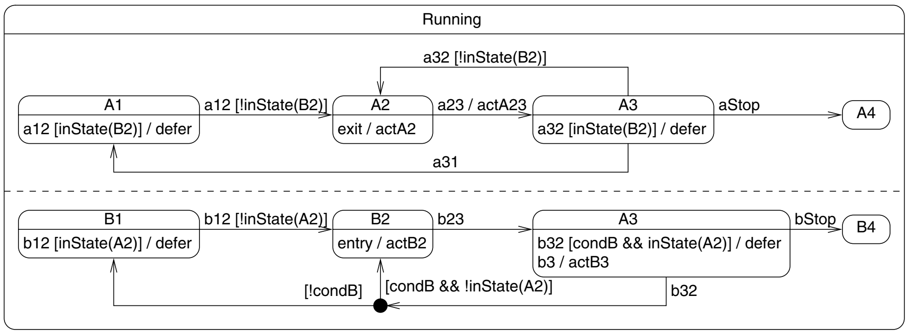


Fig. 4. Mutual exclusion using standard UML

that the model elements involved in this one feature are scattered all over the state machine, switching the feature on and off thus is difficult and requires modifications all over the state machine. The state machine gets rather hard to understand and maintain.

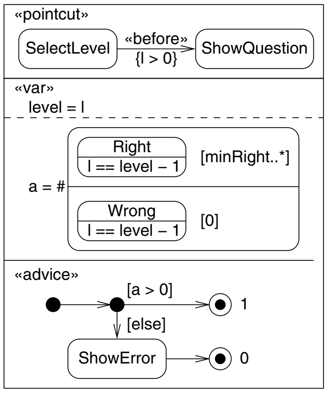
Besides history-based behavior, state synchronization is also difficult to model in UML state machines. Suppose for our state machine shown in Fig. 1 an additional mutual exclusion rule that the states A2 and B2 must not be active at the same time: before any thread entering any of these two states, the process must check if the other critical state is active; the thread may only proceed if this is not the case, otherwise it must wait for the other thread to leave the "blocking" state.

Figure 4 extends the state Running of the state machine shown in Fig. 1 with this rule: The OCL [15] predicate inState is used for checking whether a particular state is active; every transition leading to A2 and B2 must be guarded by such a predicate in order to ensure that the A2 and B2 do not become active at the same time. Simultaneously, the source states of these offending transitions defer the triggering event such that it will be reconsidered if the activation condition of either A2 or B2 changes. (Note that we have taken the liberty of adding a guard to the deferring of an event in order to make a peek to the current event, which is not covered by the UML metamodel. However, this construct can be replaced by adding an internal transition involving the negated deferring guard.) Again, it is a tedious and error-prone task to model a simple thread synchronization rule including no more than two states like this, since invasive modifications to the original state machine are necessary almost everywhere in it.

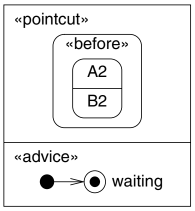
In both examples, having to model interaction between several states by the pervasive usage of sometimes intricate state machine features makes the resulting state machine rather complex and no longer intuitively understandable, in contrast to what models in a graphical modeling language are supposed to be.

4 Aspect-Oriented State Machines

Our answer to the problem of scattered model elements in UML state machines is, motivated by the success of aspect-oriented programming in solving the problem of scattered code, to enhance UML state machines with aspects. We introduce a new language construct, *aspect*, to the UML, which contains a *pointcut* and a piece of *advice*. The



(a) Restricting level selection



(b) Mutual exclusion

Fig. 5. Aspects for a better separation of concerns

pointcut specifies some special point of time in the execution of a UML state machine, which is called the *base state machine*. The advice defines some additional or alternative behavior. An aspect is thus a (graphical) statement expressing that at the special point of time specified by the pointcut, the behavior defined by the advice should be performed. The complete behavior of the context of the base state machine is therefore given by the composition of the base state machine and the aspect. The process of composition is called *weaving*.

4.1 Concrete Syntax and Informal Semantics

We first define a new language construct, *superstate*, to be used in aspects. A superstate contains a subset of the states of the base state machine and may be guarded by constraints.

A pointcut is either a *configuration pointcut* or a *transition pointcut*. A configuration pointcut consists of a superstate, stereotyped by ≪before≫ or ≪after≫. It specifies, depending on the stereotype, the point of time just before or just after any state configuration that contains all states in this superstate is active. A transition pointcut consists of two superstates, connected by a transition with a stereotype ≪before≫ or ≪after≫. It specifies, depending on the stereotype, the point of time just before or just after this transition is fired. In the aspect shown in Fig. 5(a), the transition from SelectLevel to ShowQuestion is stereotyped ≪before≫ and guarded by a constraint $l > 0$. The pointcut thus specifies the point of time just before the base state machine is about to change from an active state configuration, which contains the state SelectLevel, to a new active state configuration which would contain the state ShowQuestion, where the value of the variable l is greater than zero.

The behavior to execute at the point of time specified by the pointcut is defined in the advice. A piece of advice is a UML state machine enhanced with some pre-defined final states. The variables of the base state machine may be used, the aspect may also define local variables, in particular *trace variable*s for counting the occurrences of a certain sequence of active state configurations in the execution history. The local variables may also be used in the advice.

In Fig. 5(a), two variables are introduced: a normal variable level which stores the current value of the variable l each time the advice is executed, and a trace variable a. The value of a is the number of occurrences (indicated by #) of the sequence specified by the pattern on the right hand side: it is a sequence that contains at least minRight ([minRight..*]) active state configurations in which Right is active while the variable l has the value of level $-$ 1, and that does not contain any active state configuration ([0]) where Wrong is active while l has the value of level $-$ 1. The terms l == level–1 are added as constraints to the respective states. The advice in Fig. 5(a) checks whether such a sequence can be found in the trace ([a $>$ 0]) or not ([else]), and resumes the execution of the base state machine (final state named 1) or prevents the base state machine from firing the specified transition by making it stutter at the current state configuration (final state named 0). As a whole, this aspect implements additively to the base state machine that changing to level $l > 0$ is only allowed when the user has already answered $l - 1$ questions correctly in a row. Note that no more invasive modification to the base state machine is necessary and that the model elements used are now gathered at a dedicated location instead of scattered all over the state machine.

Figure 5(b) shows another aspect, which is applied to the state machine given in Fig. 1 and implements the desired mutual exclusion. Its configuration pointcut contains only one superstate and specifies the point of time just before («before») any state configuration containing A2 and B2 gets active (this configuration should be avoided according to the mutual exclusion requirement). The advice does not need any variable, but simply makes the base state machine stutter (final named waiting). The difference between 0 and waiting is that returning to the base state machine after 0 the event that kicked off the aspect is consumed and the base state machine therefore needs explicity another instance of this event if it should try again, while after waiting the base state machine does not need such an event but will be waiting to complete the interrupted transitions as soon as possible, e.g., as soon as the next active state configuration would not be caught by the pointcut again. Figure 5(b) thus models the logic of mutual exclusion highly modularly and non-invasively, switching on and off this feature is now a very easy task.

4.2 Resuming from Advice

Note that waiting and 0 are not the only "stuttering" final states that may be used in the advice. When the base state machine is told to stutter, it needs not only information on whether it should try to resume the interrupted transition "automatically" or only upon an explicit event, but also whether it should react to other events or not. For example, suppose the base state machine shown in Fig. 1 tries to enter the state A2 from A3, but has to stay in A3 since B2 is active, then what should it do when it now receives an a31 event? Should it proceed to A1 or not?

Therefore, we could distinguish four stuttering strategies:

1. The base state machine tries automatically to resume the interrupted transition without any explicit event and reacts to other events; this is our case waiting.

2. The base state machine needs an explicit event to make another try of the interrupted transition and reacts to other events; this is our case 0.
3. The base state machine tries automatically to resume the interrupted transition without any explicit event and does not react to other events. We say the base state machine is in this case "pinned" to the interrupted transition and model this case with a final state with the name pinned.
4. The base state machine needs an explicit event to make another try of the interrupted transition but does not react to other events.

We currently have no examples that require case 4 and therefore do not include it in our aspect language. However, both assigning this case a name and extending our implementation (see Sect. 5) are straightforward so that a simple extension would make our language to cover this case as well.

We call final states with label 0 or 1 *progressing* and final states with labels waiting or pinned *inhibiting*.

5 Translation of Aspects

The weaving process transforms a state machine and an aspect into a new state machine. As combining several aspects presents a number of additional challenges we concentrate on weaving a single aspect into a state machine. Weaving proceeds in the following stages which we describe in more detail in the rest of the section.

1. Normalize the state machine to eliminate syntactic variation.
2. Identify the states and transitions which have to be modified.
3. Construct the finite automata that track the relevant history.
4. Insert variables and actions.
5. Insert advice.

Normalization. UML state machines allow a number of notational variations. To simplify the translation process we transform the state machine into an equivalent canonical state machine which is well-suited for the next stages of the translation process. We require that the normalization process transform the state machine into a form where all transitions that can lead to a state configuration in which the pointcut applies are explicit. Hence the normalization process ensures that all states and transitions which may potentially be modified by the introduction of the aspect can be determined from the advice or a variable declaration.

Identification of Relevant States and Transitions. There are two different kinds of relevant state machine elements: Some elements are necessary for inserting advice. We call these elements *advice-relevant* and write $\mathsf{arel}(M, \mathcal{A})$ for the set of advice-relevant elements of state machine M and aspect $\mathcal{A}$. Other elements are relevant for keeping track of the history of active state configurations; these are called *history-relevant*. The set of all history-relevant elements is written as $\mathsf{hrel}(M, \mathcal{A})$. The set of elements which are history-relevant for a single trace variable a is written $\mathsf{hrel}(M, a)$. It is, of course, possible for an element to be both advice-relevant and history-relevant.

The advice-relevant elements can be found from the pointcut specification:

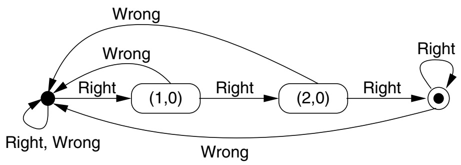


Fig. 6. Finite automaton for trace variable a

- If the stereotyped element of the pointcut is a state configuration and the stereotype is ≪before≫, then all simple states in this state configuration and all transitions with such a state as target are advice-relevant. If the stereotype is ≪after≫, then all simple states in the state configuration and all transitions with one of these states as source are advice-relevant.
- If the stereotyped element of the pointcut is a transition, then all the transitions in the state machine which go directly from the source state of the stereotyped transition to its target state are advice-relevant; furthermore all simple states contained in the source state of the stereotyped transition are advice-relevant.

A state is history relevant if and only if it appears on the right hand side of a variable declaration. Transitions are never history-relevant.

Construction of History Automata. Aspect-oriented state machines can take decisions based on their history of active state configurations. It would obviously be prohibitively expensive to store and analyze the complete history of the state machine in order to decide which transition to take. Therefore our aspect language is designed such that the whole relevant history for unbounded executions can be stored in a finite amount of memory unless the context variables referenced by trace variables in some aspect take an infinite number of values.

This is possible because a trace pattern for a trace variable a defines a regular language over the alphabet $\mathsf{hrel}(M, a)$. For example, the definition of the trace variable a with $\mathsf{minRight} = 3$ in Fig. 5(a) is translated into the non-deterministic finite automaton shown in Fig. 6. The initial state of this automaton accepts the whole alphabet $\mathsf{hrel}(M, a)$; this represents the fact that we are looking for occurrences of the pattern anywhere in the history. The automaton for recognizing a sequence of trace patterns can be obtained by connecting the automata for the individual trace patterns with ϵ-transitions.

The rest of this section is rather technical as it describes the construction of the non-deterministic finite automaton (NFA) for integer trace variables of the form $a = \#(s)$ for some superstate s.

We assume that the states in $\mathsf{hrel}(M, a)$ are numbered from 0 to n and write s_i for the state with number i. If the constraint on a state s is of the form $n_1..n_2$ we call $\min(s) := n_1$ the *minimal* and $\max(s) := n_2$ the *maximal number of occurrences* of that state. If the constraint on s is of the form $n..*$ we define $\min(s) := \max(s) := n$. in this case we define $\mathsf{infinite?}(s) = \mathsf{true}$, otherwise $\mathsf{infinite?}(s) = \mathsf{false}$.

To distinguish states of the finite automaton from states of the state machine we refer to the former as *hstates*. We define a hstate for each tuple $(a_0, \ldots, a_n)$ with $0 \leq a_i \leq \max(s_i)$ and write *hstate*$(a_o, \ldots, a_n)$ for this hstate. Furthermore we write $tuple(H) = (a_0, \dots, a_n)$ iff $H = hstate(a_o, \dots, a_n)$. For all other tuples we define $hstate(a_0, \dots, a_n) = \perp$. We write $\mathcal{H}(\mathcal{A})$ for the set of all hstates, i.e., $\mathcal{H}(\mathcal{A}) := hstate[\mathbb{N}^{n+1}] \setminus \{\perp\}$.

Algorithm 1. Computation of the transitions $\mathcal{T}$

```
1: $H \leftarrow \mathcal{H}(\mathcal{A})$
2: $\mathcal{T} \leftarrow \emptyset$
3: $h_0 \leftarrow hstate(0, \dots, 0)$
4: for all $i$,   $0 \leq i \leq n$ do
5:     $\mathcal{T} \leftarrow \mathcal{T} \cup \{h_0 \xrightarrow{s_i} h_0\}$
6: end for
7: for all $h \in H$ do
8:     $(a_0, \dots, a_n) \leftarrow tuple(h)$
9:     for all $k \leftarrow (a_0, \dots, a_i + 1, \dots, a_n)$,   $0 \leq i \leq n$ do
10:        if $hstate(k) \neq \perp$ then
11:            $\mathcal{T} \leftarrow \mathcal{T} \cup \{h \xrightarrow{s_i} hstate(k)\}$
12:        else if $\mathsf{infinite?}(s_i)$ then
13:            $\mathcal{T} \leftarrow \mathcal{T} \cup \{h \xrightarrow{s_i} h\}$
14:        else
15:            $\mathcal{T} \leftarrow \mathcal{T} \cup \{h \xrightarrow{s_i} h_o\}$
16:        end if
17:    end for
18: end for
```

The set of transitions $\mathcal{T}$ is generated as described in Algorithm 1. This algorithm inserts a transition with label s_i between hstates of the form $hstate(a_0, \dots, a_i, \dots, a_n)$ and $hstate(a_0, \dots, a_i + 1, \dots, a_n)$; it inserts a self-transition with label s_i for hstates of the form $hstate(a_0, \dots, a_n)$ if $a_i = \max(s_i)$ and s_i allows infinitely many repetitions, otherwise it inserts a transition into the initial hstate h_0. By inserting (non-deterministic) transitions $h_0 \xrightarrow{s} h_0$ for all states $s \in \mathsf{hrel}(M, a)$ we ensure that the automaton keeps track of all sequences of states in its history.

A hstate $hstate(a_0, \dots, a_n)$ is accepting if $\min(s_i) \leq a_i \leq \max(s_i)$ for all i with $0 \leq i \leq n$. To achieve the desired semantics for the aspect-oriented state machine we execute a transition of the automaton for each run to completion step of the state machine. By construction the finite automaton enters an accepting state from a non-accepting state for the n-th time precisely when the state machine is in the corresponding superstate for the n-th time. The construction of the automaton described above takes into account overlapping patterns, a slight modification of the construction of the NFA and the functions described in the next section would disallow them.

Depending on the space/time trade-off for the state machine we can either convert the NFA into a deterministic finite automaton (DFA) or simulate the execution of the DFA with the NFA. For the sake of concreteness we assume that convert the NFA into a DFA which we call the *history automaton for* a, $\mathfrak{H}(a)$, with hstates $states(\mathfrak{H}(a))$, transitions $transitions(\mathfrak{H}(a))$, initial hstate h_0^a (corresponding to the translation of $hstate(0, \dots, 0)$), and accepting hstates $accepting(\mathfrak{H}(a))$. If h is a hstate of $\mathfrak{H}(a)$ we write $\mathfrak{H}(a)(h, s)$ to denote the result of executing the transition labeled s starting from hstate h of the automaton $\mathfrak{H}(a)$.

Insertion of Variables and Actions. Having defined the history automata for all trace variables we are now in a position to introduce the necessary modifications for deciding the applicability of aspects into the context of the base state machine. Let a be a trace variable. We write $\mathsf{vars}(a)$ for the set of context variables in the trace patterns of a. For example, in Fig. 5(a), $\mathsf{vars}(a) = \{l\}$. We need to keep track of a separate history for each value of each variable in $\mathsf{vars}(a)$. This is also the case when manually modifying the state machine, as can be seen in Fig. 3 where an array $r[l]$ is used to store the number of correct answers for each level.

For each trace variable we therefore introduce the following elements into the context of the base state machine:

- $history$: This is a finite function with domain $\mathsf{vars}(a) \times E(a)$, where $E(a)$ is the set of environments for $\mathsf{vars}(a)$, i.e., the set of all assignments of a value to each variable in $\mathsf{vars}(a)$. The value for each key is a pair $\langle \mathfrak{H}(a), h \rangle$, where h is the current state of the state machine $\mathfrak{H}(a)$ (for the values in e). The initial value for $history$ is the function mapping each value $\langle a, e \rangle$ in its domain to $\langle \mathfrak{H}(a), h_0^a \rangle$.
- val: This is a finite function with domain $\mathsf{vars}(a) \times E(a)$ as described above and range $\mathbb{N}$, evaluating the current value of a in environment e. The default value is a function mapping each value in its to domain to 0
- $updateHistory$: This is a function with arguments $\langle a, s, e \rangle$, where a and e are as described above and s is a state of the base state machine. This function updates the result $\langle \mathfrak{H}(a), h \rangle$ of $history(a, e)$ by executing transition s of $\mathfrak{H}(a)$ in hstate h, i.e., $history(a, e)$ is updated to $\langle \mathfrak{H}(a), \mathfrak{H}(a)(h, s) \rangle$. If h is not accepting and the result of this transition is an accepting hstate of $\mathfrak{H}(a)$, i.e., if $h \notin accepting(\mathfrak{H}(a))$ and $\mathfrak{H}(a)(h, s) \in accepting(\mathfrak{H}(a))$, then $val(a, e)$ is incremented by one.

To keep track of the state machine's history, it is now sufficient to add an entry action $updateHistory(a, s, e)$ (where e is the local environment for $\mathsf{vars}(a)$) to every state s in $\mathsf{hrel}(M, a)$ for each trace variable a. To evaluate an expression X in which a occurs we replace a by $val(a, e)$ (with the current environment e for $\mathsf{vars}(a)$) in X, i.e., we evaluate $X[a/\ val(a, e)]$.

Inserting Advice. The final step in the weaving process is the insertion of the advice itself. We write $\mathsf{init}(\mathcal{A})$ for the initial transition of the advice, $\mathsf{final}(\mathcal{A}, l)$ for the set of final transitions of the advice with label l, $\mathsf{source}(t)$ for the source state of a transition t, $\mathsf{target}(t)$ for the target state of a transition, and $guard(t)$ for the guard of a transition. We write $\mathsf{source}(t) \leftarrow s$ to denote the operation of replacing the source state of transition t with state s, similar for the target node. The operation $\mathsf{copy}(\mathcal{A})$ copies the advice of an aspect; all states and transitions of the copy are disjoint from the original. We write $\mathsf{inhibited}(s)$ for the set of all compound transitions leading from s to an inhibited final state.

To simplify the translation algorithm we place the following restrictions on advice: the advice itself may only contain a single region; for each compound transitions all final states must either be progressing or inhibiting. The first restriction ensures that

Algorithm 2. Inserting Advice

```
1: $\mathcal{T}_{old} \leftarrow \mathit{transitions}(\mathsf{arel}(M, \mathcal{A}))$
2: $\mathit{transitions}(M) \leftarrow \mathit{transitions}(M) \setminus \mathcal{T}_{old}$
3: **for all** $t \in \mathcal{T}_{old}$ **do**
4:     $A \leftarrow \mathit{copyAdvice}(\mathcal{A})$
5:     $\mathsf{source}(\mathsf{init}(A)) \leftarrow \mathsf{source}(t); \mathit{guard}(\mathsf{init}(A)) \leftarrow \mathit{guard}(t)$
6:     ADD BEHAVIOR FOR INHIBITED FINAL STATES($A$)
7:     UPDATE TRANSITION TARGETS($A, t$)
8: **end for**
9: $\mathit{transitions}(M) \leftarrow \mathit{transitions}(M) \cup \mathit{transitions}(A)$
10: $\mathit{states}(M) \leftarrow \mathit{states}(M) \cup \mathit{states}(A)$
```

Algorithm 3. Adding behavior for inhibited final states

```
1: **function** ADD BEHAVIOR FOR INHIBITED FINAL STATES(A)
2:     **for all** $s \in \mathit{states}(A)$ **do**
3:         $ew \leftarrow \emptyset; ep \leftarrow \emptyset$
4:         $\forall e \in \mathit{events}(A).\ (g_e^w \leftarrow \mathsf{false}; g_e^p \leftarrow \mathsf{false})$
5:         **for all** $t' \in \mathsf{inhibited}(s)$ **do**
6:             $g \leftarrow \bigwedge\{\mathsf{inState}(s') \wedge \mathsf{constraints}(s') \mid s' \in \mathit{states}(\mathsf{arel}(M, A)) \setminus \mathsf{target}(t')\}$
7:             **if** $t' \in \mathsf{final}(A, \mathsf{waiting})$ **then**
8:                 $g_e^w \leftarrow g_e^w \vee (\mathit{guard}(t') \wedge g); \quad ew \leftarrow ew \cup \{\mathsf{event}(t')\}$
9:             **else**
10:                $g_e^p \leftarrow g_e^p \vee (\mathit{guard}(t') \wedge g); \quad ep \leftarrow ep \cup \{\mathsf{event}(t')\}$
11:            **end if**
12:            $\mathit{guard}(t') \leftarrow \mathit{guard}(t') \wedge \neg g$
13:        **end for**
14:        **for all** $e \in ew$ **do**
15:            Add behavior e[$g_e^w$]/defer to $s$
16:        **end for**
17:        **for all** $e \in ep$ **do**
18:            Add entry behavior $v_e^s$ = false to $s$
19:            Add the following behavior to $s$:
20:                $e$[$g_e^p$]/$v_e^s$ = true;    $e$[$g_e^p \wedge v_e^s$]/defer
21:                $\forall e' \in \mathit{events}(A) \setminus \{e\}.\ e'$[$g_e^p \wedge v_e^s$]/
22:        **end for**
23:    **end for**
24: **end function**
```

Algorithm 4. Updating transition targets

```
1: **function** UPDATE TRANSITION TARGETS($A, t$)
2:     **for all** $t' \in \mathsf{final}(A, 1) \cup \mathsf{final}(A, \mathsf{waiting}) \cup \mathsf{final}(A, \mathsf{pinned})$ **do**
3:         $\mathsf{target}(t') \leftarrow \mathsf{target}(t)$
4:     **end for**
5:     **for all** $t' \in \mathsf{final}(A, 0)$ **do**
6:         $\mathsf{target}(t') \leftarrow \mathsf{source}(t)$
7:     **end for**
8: **end function**
```

compound transitions cannot have multiple final states, the second restriction simplifies the introduction of guards in the weaving process.

Algorithms 2, 3 and 4 describe the process of inserting advice into the state machine for configuration pointcuts: For each advice-relevant transition t we remove t from the base state machine, attach the advice to the source state of t and connect the final states of the advice with appropriate states of the state machine. We also modify the advice relevant states of the resulting state machine to defer events for advice ending in a final state with label waiting or pinned and disable the outgoing transitions once the advice arrives in a pinned state. The algorithm for transition pointcuts is similar.

As an example of the weaving process consider Fig. 4 which is the result of weaving the mutual exclusion advice in Fig. 5(b) into the base state machine in Fig. 1. The result of manually extending the e-learning example in Fig. 3, however, differs from the result of the automatic weaving process, as this process uses a history automaton for the trace variables instead of explicitly manipulating counters.

6 Related Work

Our idea of dynamic aspects of state machines has been inspired by dynamic aspect-oriented programming languages such as JAsCo [20] and Object Teams [9]. Such languages are recognized as useful for separation of concerns (cf. [7]); a recent example of using control flow based aspects to build powerful debuggers is given in [4]. In particular, using trace variables to quantify over the trace is reminiscent to the trace aspects of Arachne [6]. Aspect-oriented modeling, aiming at a better separation of concerns on the level of software design, is still in its infancy. Most existing work (e.g. [1,17,19]) focuses on modeling aspect-oriented programs rather than making modeling languages aspect-oriented.

In the realm of modeling languages and state machines in particular, Altisen et al. [2] propose aspects for Mealy automata. Pointcuts are also defined as automata. In comparison, the weaving algorithm of our approach is due to the richer language constructs of the UML much more elaborate. Thanks to the wider acceptance of the UML, our approach is more tightly connected to common practice. Mahoney et al. [13] propose to combine several state machines into one orthogonal composite state and to relate by textual notations triggering events in different regions so that related transitions can be fired jointly. This approach can be used to modularize the synchronization of state machines, although having to declare all events of the wrapping state machine to be executed before triggering transitions in the base state machine may lead to quite complicated annotations. JPDD [8] is a pointcut language that facilitates the definition of trace-based pointcuts. In comparison, our approach also allows the modeler to define state machine synchronization modularly. Moreover, we have also defined a translation semantics for our aspects including both pointcuts and advice.

Theme/UML [5] models different features in different models (called themes) and uses UML templates to define common behavior of several themes. It does not contain a pointcut language, model elements have to be bound to formal parameters explictly by textual notations (A first step towards using JPDD as the pointcut language is presented in [10], although there are still compatibility problems between these two approaches).

The definition of history-based and modular modeling of state machine synchronization does not seem possible.

7 Conclusions and Future Work

We have defined a syntax and a translation semantics of aspect-oriented state machines. Using aspects may improve the modularity of UML state machines considerably by separating state interaction and trace-based behavior from other concerns. Our weaving algorithm works with a bounded amount of memory in realistic cases, i.e., as long as the variables used in the constraints of superstates do not take infinite many values.

Both the pointcut language and the advice language may be extended. In particular, it is expected to be straightforward yet useful to allow the pointcut to contain more than two superstates and/or to quantify over variable traces (and thus allow data-oriented aspects as proposed in [18]). It would also be interesting to allow the advice to make the base state machine not only stutter, but jump into the past. This might be useful for modeling compensations in long-running transactions in service-oriented systems. Moreover, in order to allow the application of several aspects to a state machine, a notation should be designed for defining the order of weaving in case of conflicts; the implementation described in Sect. 5 should be extended as well.

Another important issue of future work is model validation. We plan first to extend an existing UML model checker, such as Hugo/RT [12], to validate the weaving product. In a second step, it would be interesting to investigate techniques of compositional validation, in order to allow validation of larger models. Finally, extending aspect-orientation to other UML diagrams and generating aspect-oriented programs from aspect-oriented models are also part of our future research.

References

1. Aldawud, O., Elrad, T., Bader, A.: UML Profile for Aspect-Oriented Software Development. In: AOM. Proc. 3rd Int. Wsh. Aspect-Oriented Modeling, Boston (2003)
2. Altisen, K., Maraninchi, F., Stauch, D.: Aspect-Oriented Programming for Reactive Systems: Larissa, a Proposal in the Synchronous Framework. Sci. Comp. Prog. 63(3), 297–320 (2006)
3. Barry, B., de Moor, O. (eds.): AOSD'07. Proc. 6th Int. Conf. Aspect-Oriented Software Development, ACM Press, New York (2007)
4. Chern, R., De Volder, K.: Debugging with Control-Flow Breakpoints. In: Barry, de Moor [3], pp. 96–106 (2007)
5. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. Addison-Wesley, Reading (2005)
6. Douence, R., Fritz, T., Loriant, N., Menaud, J.-M., Ségura-Devillechaise, M., Südholt, M.: An Expressive Aspect Language for System Applications with Arachne. In: Mezini, Tarr [14], pp. 27–38
7. Filman, R.E., Haupt, M., Hirschfeld, R. (eds.): Proc. 2nd Dynamic Aspects Wsh. (DAW'05). Technical Report 05.01. Research Institute for Advanced Computer Science (2005)
8. Hanenberg, S., Stein, D., Unland, R.: From Aspect-Oriented Design to Aspect-Oriented Programs: Tool-Supported Translation of JPDDs into Code. In: Barry, de Moor [3], pp. 49–62

9. Herrmann, S.: Object Teams: Improving Modularity for Crosscutting Collaborations. In: Aksit, M., Mezini, M., Unland, R. (eds.) NODe 2002. LNCS, vol. 2591, pp. 248–264. Springer, Heidelberg (2003)
10. Jackson, A., Clarke, S.: Towards the Integration of Theme/UML and JPDDs. In: Proc. 8th Wsh. Aspect-Oriented Modeling, Bonn (2006)
11. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
12. Knapp, A., Merz, S., Rauh, C.: Model Checking Timed UML State Machines and Collaborations. In: Damm, W., Olderog, E.R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 395–416. Springer, Heidelberg (2002)
13. Mahoney, M., Bader, A., Elrad, T., Aldawud, O.: Using Aspects to Abstract and Modularize Statecharts. In: Proc. 5th Wsh. Aspect-Oriented Modeling, Lisboa (2004)
14. Mezini, M., Tarr, P.L. (eds.): AOSD'05. Proc. 4th Int. Conf. Aspect-Oriented Software Development, ACM Press, New York (2005)
15. Object Management Group. Object Constraint Language, version 2.0. Specification, OMG (2006), `http://www.omg.org/cgi-bin/doc?formal/06-05-01.pdf`
16. Object Management Group. Unified Modeling Language: Superstructure, version 2.1.1. Specification, OMG (2007), `http://www.omg.org/cgi-bin/doc?formal/07-02-05.pdf`
17. Pawlak, R., Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L., Martelli, L.: A UML Notation for Aspect-Oriented Software Design. In: AOM'02. 1st Int. Wsh Aspect-Oriented Modeling, Enschede (2002)
18. Rashid, A., Chitchyan, R.: Data-Oriented Aspects. In: Hannemann, J., Baniassad, E., Chen, K., Chiba, S., Masuhara, H., Ren, S., Zhao, J. (eds.) AOASIA'06. Proc. 2nd Asian Wsh. Aspect-Oriented Software Development, pp. 24–29. National Institute of Informatics, Tokyo (2006)
19. Stein, D., Hanenberg, S., Unland, R.: A UML-based Aspect-Oriented Design Notation for AspectJ. In: AOSD'02. Proc. 1st Int. Conf. Aspect-Oriented Software Development, pp. 106–112. ACM Press, New York (2002)
20. Vanderperren, W., Suvée, D., Verheecke, B., Cibrán, M.A., Jonckers, V.: Adaptive Programming in JAsCo. In: Mezini, Tarr [14], pp. 75–86