Weaving Semantic Aspects in HiLA

Gefei Zhang
arvato Systems Technologies GmbH
gefei.zhang@pst.ifi.lmu.de

Matthias Hölzl
Ludwig-Maximilians-Universität München
matthias.hoelzl@pst.ifi.lmu.de

Abstract

UML state machines are widely used for modeling software behavior. Due to the low-level character of the language, UML state machines are often poorly modularized and hard to use. High-Level Aspects (HILA) is an aspect-oriented extension of UML state machines which provides high-level language constructs for behavior modeling. HILA considerably improves the modularity of UML state machines by extending them by semantic aspects. This paper presents the weaving process for HILA that we have shown to be sound with respect to the transition-system semantics of HILA. In particular, we show how our weaving process deals with implicit state activation (and deactivation), maps semantic pointcuts to syntactic elements, and resolves potential conflicts between different aspects. The process has been implemented in an extension of the Hugo/RT UML translator and model checker, the correctness of our weaving is validated by model checking.

Categories and Subject Descriptors D2.2 [*Software Engineering*]: Design Tools and Techniques—State diagrams

Keywords Aspect-Oriented Modeling, UML, Algorithms

1. Introduction

UML state machines [15] are widely used for modeling software behavior. They are considered as simple and intuitive, and are even deemed to be "the most popular modeling language for reactive components" [7]. However, UML state machines exhibit modularity problems, and even some simple behaviors may be hard to model, see [17] for examples.

Proposals have been made [3, 5, 12, 14] to use aspect-oriented modeling to tackle the modularity problems of UML state machines. In the prevalent proposals, aspects define model transformations, most commonly graph transformations: an aspect specifies in its pointcut a fragment of the base model, and defines in its advice a graph to replace the parts of the base model matched by the pointcut. The weaving is then the process of actually performing the transformation.

Aspects in these proposals are therefore syntactic constructs: they define how to modify (the syntax tree of) the base model. While these approaches provide a means for modeling parts of the system separately, they hardly help to reduce the complexity of the model, because, even though the factorization provided by the match-and-replace semantics might eliminate some redundancy, the modeler still has to define the system behavior in every little detail. Moreover, the aspects (which are in fact model transformations) are defined in syntactic terms, and do not have a behavioral semantics. When they are applied to behavioral models like state machines, the semantics of the overall system can therefore be only obtained by carefully studying the weaving result.

Our approach, High-Level Aspects (HILA), in contrast, extends UML state machines by semantic aspects. In HILA, a pointcut defines specific points of time in the execution of the base machine, and the advice defines some additional or alternative behavior to execute at these points of time. The behavior of HILA aspects can therefore be understood independently of any weaving process, and indeed the semantics of HILA is defined by a transition-system model rather than by a graph transformation that weaves aspects into the base state machine, see [17]. State machines using HILA aspects are therefore easier to read and also easier to construct than their graph-transformation-based counterparts, since the modeler only has to define what to do instead of how to do it.

To implement HILA aspects it would therefore be possible to develop an interpreter that directly implements the semantics of HILA. In order to leverage the sophisticated possibilities for formal analysis and code generation offered by the Hugo/RT system[1], and to avoid the overhead of evaluating the dynamic activation and deactivation of aspects specified by the semantics of HILA at run time, we have opted to implement HILA by weaving aspects and base state machine together. Given the nature of HILA aspects, this weaving process is more elaborate than the one required by

AOSD'12, March 25–30, 2012, Potsdam, Germany.

[1] `http://www.pst.ifi.lmu.de/projekte/hugo`

graph-transformation-based aspect systems: The modeler no longer directly specifies the elements that have to be modified by the weaver; instead the weaver has to determine which model elements are modified by HILA aspects. This is rather challenging, as even a simple aspect may touch many model elements that are scattered throughout the state machine.

In this paper, we present the weaving algorithms of HILA aspects. In particular, we show how we handle syntax variations of UML state machines, how the semantic pointcuts are mapped to syntactic elements, and how we resolve conflicts between aspects. The presented algorithms have been implemented as an extension of the model checker Hugo/RT, model checking HILA aspects is thus an easy task. Many of the issues encountered by weaving aspects for HILA also apply to the implementation of semantic aspect systems for concurrent, state-based languages in general. Therefore, the techniques presented in this paper are not only a description of the implementation strategy we have used for our current HILA system, they can also serve as guideline for implementors of other aspect-oriented languages. We will address one example of this in Sect. 6.

The rest of this paper is structured as follows: In the next two sections we give short introductions to UML state machines and HILA. The fourth section, the core of this paper, details the weaving algorithm; it is followed by a discussion of the implementation in Hugo/RT and some remarks on formal validation of the weaving results. Finally, we present related work and conclude.

2. UML State Machines

A UML state machine provides a model for the behavior of an object or component. Figure 1(a) shows a state machine modeling (in a highly simplified manner) the behavior of a player during a part of a game. The player—a magician—starts in a state where she has to chose a NewLevel. Upon completion of the preparations she is transferred into the Play state which contains two concurrent regions, corresponding to two parallel threads of execution. The upper region describes the possible movements of the player, the lower region specifies her behaviors. After completing the current level, the player can go to the next level which has the same topology and general gameplay.

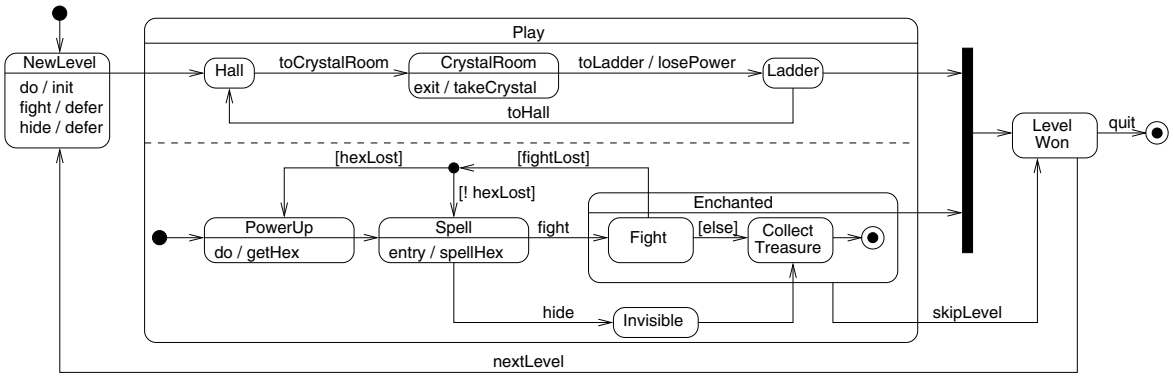
In each level the player initially starts in an entrance hall (Hall), from there she can move to a room in which magic crystals are stored (CrystalRoom) and on to a room containing a Ladder. From this room the player can either move back to the hall or, after excavating a treasure consisting of gold coins, exit the level. After starting the level, the player first has to acquire enough magical power (PowerUp), then cast a Spell that, make here invisible, or transforms her into an Enchanted state. Once she is in this enchanted state, the player may either try to fight the guard of the treasure (Fight), collect the treasure (CollectTreasure) and then exit the level via the ladder, or she may use her magical power to exit the current level without fighting the guard and collecting the gold (in which case she may encounter the guard again later on in the game). The player may lose the fight against the guard in which case she is transferred back to one of the previous states, depending on the severity of the defeat.

2.1 Syntax and Informal Semantics

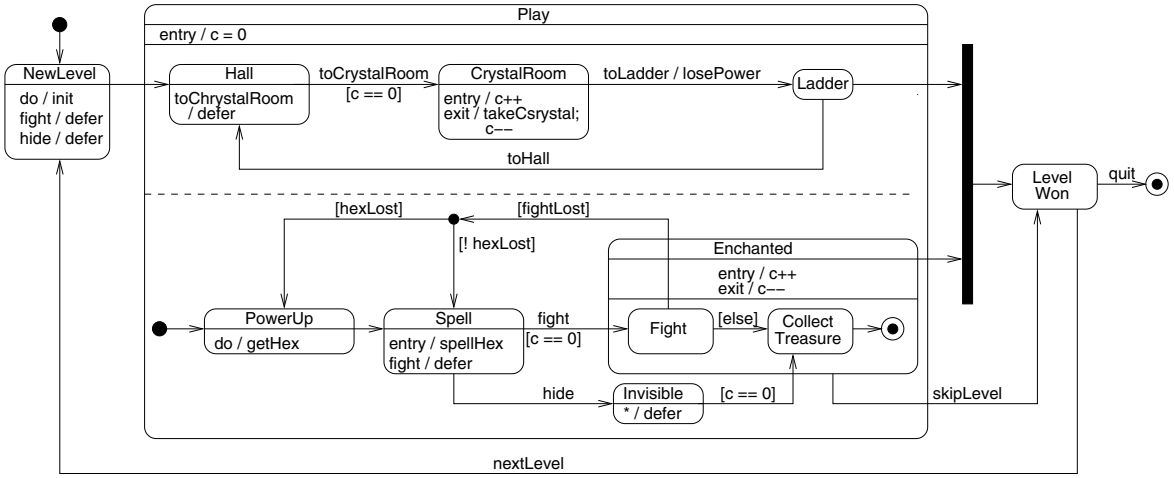
We briefly review the syntax and semantics of UML state machines according to the UML specification [15] by means of Fig. 1(a). A UML state machine consists of *regions* which contain *vertices* and *transitions* between vertices. We require every state machine to have a top-level region called top.[2] 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.

A state of a state machine is *simple*, if it contains no regions (such as NewLevel in Fig. 1(a)); 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 Play). Each state may show an *entry* behavior (like spellHex in Spell), an *exit* behavior (like takeCrystal in CrystalRoom), which are executed on activating and deactivating the state, respectively; a state may also show a *do activity* (like in NewLevel) which is executed while the state machine sojourns in this state. Transitions are triggered by events (toCrystalRoom, fight), show guards (fightLost), and specify effects to be executed when a transition is fired (losePower). Completion transitions (e.g., the transition leaving NewLevel) are triggered by an implicit *completion event* emitted when a state completes all its internal activities. Events may be *deferred* (e.g., fight and hide in NewLevel), that is, put back into the event pool if they are not to be handled currently. By executing a transition its source state is left and its target state entered; transitions may also be declared to be *internal* (not shown in this example), 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 the region top of a state machine is completed the state machine terminates. *Junction* pseudo states, also depicted as filled circles (see lower region of Play), allow for case distinctions. Transitions to and from different regions of an orthogonal composite state can be synchronized by *fork* (not shown here) and *join* pseudo

[2] This was required by UML 1.x. In UML 2.x, this restriction was dropped. However, we can always put a top-level region around a state machine without changing its semantics.



(a) Movement and basic actions



(b) Mutual exclusion: player cannot be in crystal room while enchanted

Figure 1. UML state machine for a magician in a computer game

states, presented as bars. For simplicity, we omit the other pseudo state kinds (entry and exit points, shallow and deep history, choice, and terminate).

At run time, states get activated and deactivated as a consequence of transitions being fired. The active states at a stable step in the execution of the state machine form the active *state configuration*. Active state configurations are hierarchical: when a composite state is active, then exactly one state in each of its regions is also active; when a substate of a composite state is active, so is the containing state too. The execution of the state machine can be viewed as different active state configurations getting active or inactive upon the state machine receiving events.

For example, an execution trace, given in terms of active state configurations, of the state machine in Fig. 1(a) might be (NewLevel), (Play, Hall, PowerUp), (Play, Hall, Spell), (Play, Hall, Enchanted, Fight), (LevelWon), followed by the final state, which terminates the execution.

2.2 Modularity Problems

Models of plain UML state machines may exhibit modularity problems, in particular when modeling synchronization of parallel regions or history-based behaviors, see [17, 18]. For simplicity, we show only an example of modeling mutual-exclusion.

Assume in the example above an additional rule that the player cannot enter the crystal room while she is enchanted (because, for instance, she might damage the room). That is, in Fig. 1(a) the states CrystalRoom and Enchanted must not be simultaneously active. In plain UML, this rule has to be modeled *imperatively*. An example is given in Fig. 1(b), where a variable c is introduced and used to control the access to the two critical states: it is initialized as 0 in the entry action of Play, increased whenever CrystalRoom or Enchanted is activated, and decreased whenever one of the two states is deactivated. The three transitions that activate the two states (from Hall to CrystalRoom, from Spell to Fight, and from Spell to CollectTreasure) are extended by a guard, such that they are only fired when c equals 0, which means that the other critical state is currently inactive and the mutual exclusion rule is satisfied. A subtle point is that we have to declare the events toCrystalRoom and fight to be deferrable in the states Hall and Spell, respectively, and we also have to declare the completion event of state Invisible to be de-

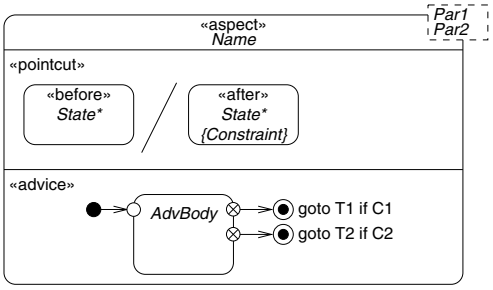


Figure 2. HiLA: concrete syntax

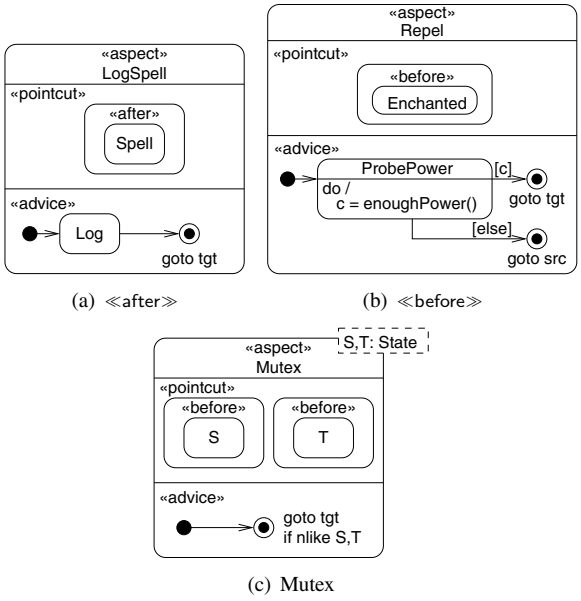


Figure 3. HiLA aspects

ferrable (we use the notation */defer for deferring the completion event; UML provides no standard syntax for this). In this way the transitions are only postponed if the other critical state is active, and will be automatically resumed without requiring the events to be sent again. Otherwise the events would be lost in case exactly one of the critical states were active, since the event would then be taken from the event pool without firing a transition.

It is obviously unsatisfactory that modeling even such a simple mutual exclusion rule requires modification of many model elements, which are scattered over the state machine. Furthermore, it is easy to introduce errors which are hard to find, as evidenced by the need to defer events. Such modeling makes maintenance difficult, the models are complex and prone to errors.

3. HiLA in a Nutshell

As a possible solution of UML state machines' modularity problems, the language High-Level Aspects (HiLA, [17]) was defined as an aspect-oriented extension for UML state machines. HiLA provides high-level constructs for declarative behavior modeling. The concrete syntax of a HiLA aspect is shown in Fig. 2 and explained in the following.

Syntactically, a HiLA aspect is a UML template containing at least a name, a pointcut and an advice. The template parameters allow easy customization, so that aspects for functionalities such as logging, transactions or mutual exclusions can easily be reused in many places.

An aspect is applied to a UML state machine, which is called the *base machine*. An aspect defines some additional or alternative behavior of the base machine at some points in time during the base machine's execution. The behavior is defined in the *advice* of the aspect; the points in time to execute the advice are defined in the *pointcut*. The advice (stereotype ≪advice≫ in Fig. 2) also has the form of a state machine, except that the final states may carry a label. The "body" of the advice, i.e. the part without the initial vertex, the final states, and the connecting transitions, models the behavior to carry out. A label on a final state names the state that should be activated when the advice is finished and the execution of the base machine should be resumed. We refer to this state as the *resumption state* of the final state. The label may optionally be guarded by a *resumption constraint*, which is indicated by the keyword if and has the form (like|nlike) StateName* where StateName is the set of qualified names of the base machine's states. like S is true iff after the resumption all states contained in S will be active, otherwise nlike S is true.

The pointcut (≪pointcut≫) specifies the points in time when the advice is executed. These points in time may be 1) when a certain state of the base machine is just about to become active or 2) a set of states has just been left.[3] The semantics of a pointcut can be regarded as a selection function of the base machine's transitions: a pointcut ≪before≫ s selects all transitions in the base machine whose firing makes state s active, and a pointcut ≪after≫ S selects those transitions whose firing deactivates S. Note this does not mean "a pointcut ≪before≫ s selects all transitions whose target is s" or "a pointcut ≪after≫ S selects all transitions whose source is a state contained in S". The semantics of UML state machines is actually more involved, which makes the weaving more complex, see Sect. 4.

Overall, an aspect is a graphical model element stating that at the points in time specified by the pointcut the advice should be executed, and after the execution of the advice the base machine should resume execution by activating the state given by the label of the advice's final state, when the conditions given there are satisfied. For ≪before≫ and ≪after≫ pointcuts, this "point in time" is always the firing of a transition; we say that this transition is *advised* by the advice.

[3] Actually there is still another kind of pointcut, ≪whilst≫, which defines the time spans during which certain states are active, see [17]. For simplicity, ≪whilst≫ pointcut is not discussed in this paper.

For example, aspect LogSpell in Fig. 3(a) states that a log message is written whenever the player has just cast a spell, i.e., when state Spell has just turned inactive (≪after≫). At such points of time, i.e., whenever the transition from Spell to Enchanted should be fired, the advice of the aspect LogSpell is executed: the state Log is activated and writes a log message, then the final state of the advice is activated, and the base machine continues the advised transition and goes to the original target state. Since no constraint is specified for the label of the final state, the original transition will be resumed as soon as the advice is finished.

To show a more involved example, we now consider an additional behavior of the magician. Suppose the magician is not always allowed to be enchanted by casting a spell, he also need a certain amount of power (we abstract from details of the magician gaining or losing power). This feature is realized by the aspect Repel in Fig. 3(b). Aspect Repel activates state ProbePower whenever state Enchanted is just about to turn active. This state sets the boolean variable c to true when the player is more powerful than the guard and to false otherwise. The aspect then either proceeds to Enchanted (goto tgt) if c is true, or it returns to the source of the advised transition (label goto src).

An even more complex feature involves mutual exclusion, as discussed in Sect. 2.2. That is, the magician should not be able to enter CrystalRoom while being Enchanted. Since mutual exclusion is a very common requirement in parallel systems, we define a HILA template to model it, see Fig. 3(c). The template takes two State parameters, S and T. The pointcut is a shortcut of "≪before≫ S or ≪before≫ T", and specifies all the points in time when either S or T is just about to get active. In such moments the advice is executed, which contains an empty body and simply conducts the base machine to resume the advised transition (by going to its target), when after the resumption the states S and T would not be both active (condition nlike S, T). Compared with the UML solution, the imperative details of mutual exclusion are now transparent for the modeler, the modeling is non-intrusive, the semantics of the aspect (template) is much easier to understand hence less error-prone. Instantiating the template by binding S to Enchanted and T to CrystalRoom elegantly prevents our magician from entering the crystal room while being enchanted and also from becoming enchanted while in the CrystalRoom.

4. Weaving

Weaving is the process of transforming the base machine to incorporate the behaviors defined in aspects. It includes determining (according to the pointcut) which transitions are advised, and to advise these transitions correctly (so that the behavior of the advice is executed and then the base machine is resumed correctly).

Due to the semantic nature of HILA aspects, the final check whether a transition is actually advised by some aspect, and therefore if the advice should be executed, can only be performed at run time. However, given an aspect A, we can statically determine the set $C(A)$ of *candidate transitions* that may be advised by A. Our weaving process actually applies a transformation to each candidate transition to implement the run-time logic that checks dynamically whether the transition is advised and, if so, executes the advice and returns to the base machine according to the label of the final state.

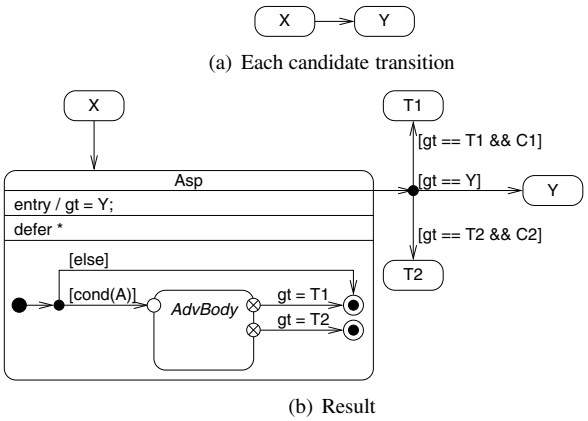
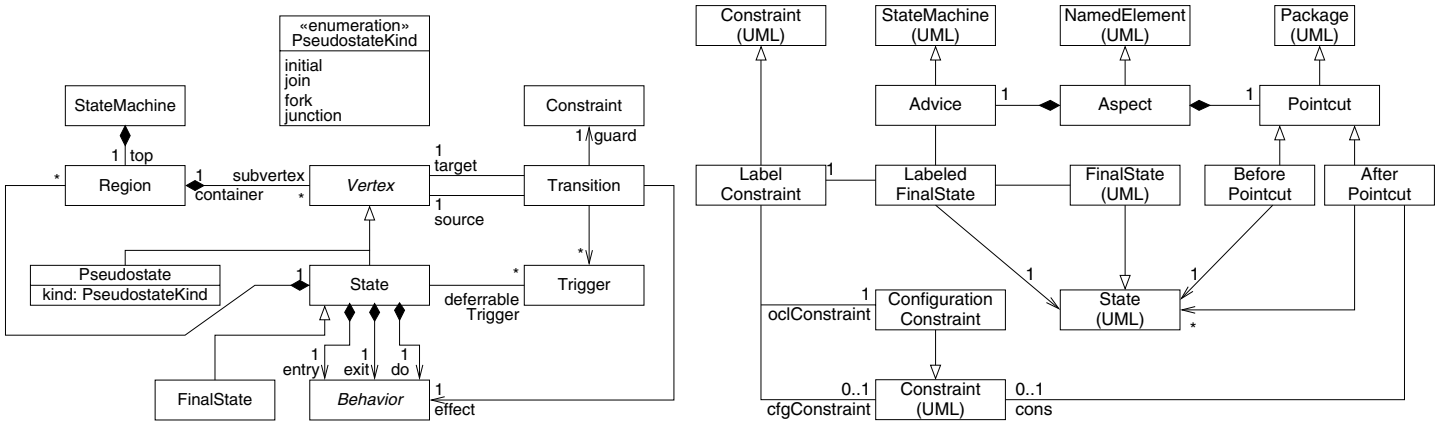


Figure 4. Weaving a single aspect A

In the following, we first define in Sect. 4.1 some notation that we need for the discussion. Then, in Sect. 4.2, we consider the case that there is only one aspect to weave. Even on this simplistic stage, some rather elaborate techniques are required to calculate $C(A)$, implement the check whether a transition is actually advised, and resume the base machine by activating the states demanded by the label. These techniques will be extended in Sect. 4.3 to handle the more realistic situation of a multitude of aspects being applied simultaneously. In particular, our weaving is designed in such a way that minimizes possible conflicts between aspects, and remaining conflicts at least can be detected at run time. To keep the size of the examples manageable we use a simplified excerpt from the state machine in Fig. 1(a) to demonstrate the weaving process. In this excerpt, shown in Fig. 7(a), we have removed some transitions which are not essential to show features of the normalization and weaving process.

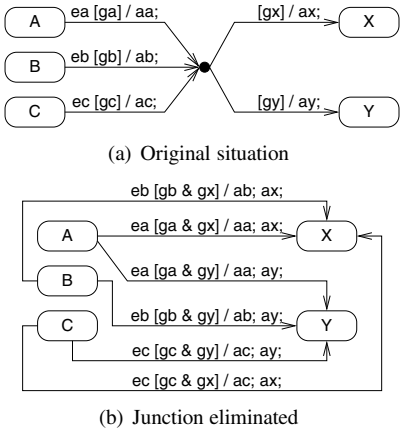
4.1 Notation

The abstract syntax of UML state machines and HILA is defined in Fig. 5. When discussing the weaving algorithms, we use the usual UML convention and write prop(el) to refer to the property prop of el. In order to better focus on the weaving process, we use a simplified metamodel of UML and do not consider entry and exit points, history, choice and terminate vertices. History and choice vertices can be simulated using other model elements, and terminate vertices can be included by a simple extension of our weaving, see [17].



(a) UML (b) HiLA

Figure 5. Metamodel (simplified)



(a) Original situation

(b) Junction eliminated

Figure 6. Elimination of junction vertices

Moreover, we require the base machine to be free of junction vertices,[4] which can be achieved performing the transformation shown in Fig. 6 to eliminate undesired junctions.

Given states s and z, and transition t, we write $\mathsf{src}(t)$ for $\mathsf{source}(t)$, $\mathsf{tgt}(t)$ for $\mathsf{target}(t)$, $\mathsf{LCR}(s,z)$ for the least region containing both s and z. Given region r, we write $\mathsf{substate}^+(r)$ to represent all states (directly or recursively) contained in r. For a state s, we write $\mathsf{substate}^+(s)$ to represent $\bigcup_{r\in\mathsf{region}(s)} \mathsf{substate}^+(r)$ and write $\mathsf{substate}^*(s)$ for $\mathsf{substate}^+(s) \cup \{s\}$.

4.2 Weaving A Single HiLA Aspect

The transformation applied to the candidate transitions in the case of weaving one single aspect is described in Fig. 4. Given an aspect A, where we assume it has the very general advice as given in Fig. 2, we introduce for each transition in the candidate set $C(\mathsf{A})$ (we call its source state X and its target state Y) a composite state, which we call Asp here and in actual weaving is assigned a unique name. The state Asp contains a slightly modified copy of the advice where: 1) labels of final states are removed (in plain UML state machines there are no labels), 2) gotos are implemented by storing the state that should be activated in the *resumption variable* gt, and 3) an additional case distinction is added, which is carried out when Asp is activated and ensures that the advice body is only executed when the transition is really advised. The construction of the condition to check here, cond(A), will be explained in Sect. 4.2.2. After the execution of Asp, the transition to fire is selected depending on the value of gt. The transition from the junction to the final state in Fig.4(b) is called the *bypass transition* of aspect A. It ensures that the advice body is skipped if cond is not satisfied.

In the following, we show in detail how we implement 1) the selection defined by the pointcut and 2) the resumption conditions. For both tasks, we first need to introduce *trace variables* into the base machine to trace its execution.

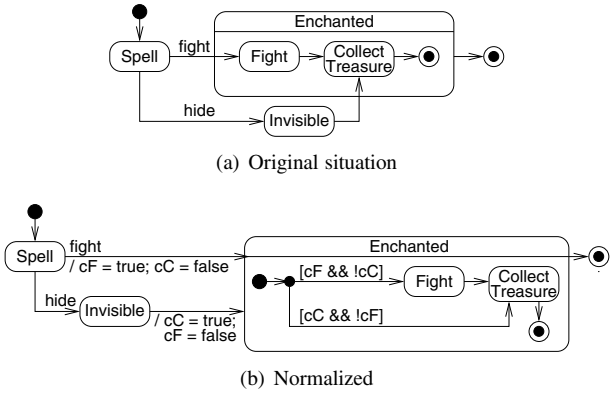
4.2.1 Trace Variables

In plain UML state machines, no information is provided on which states (in different regions) are currently active or on which states have just turned inactive. Since HiLA aspects may define behaviors to be executed just after a multitude of states have been active, we must make such information explicit by introducing additional variables.

For each state s of the base machine, we introduce a variable a_s, and set it to true in the entry action of s, and to false in the exit action. We initialize all variables a_s with false. Obviously, a_s is true iff state s is active.

The basic idea of tracking states that have just turned inactive is also to introduce for each state s a variable l_s. l_s is set to true by the exit action of s and set to false by the exit action of the state activated after s. Since there may be several transitions leaving s, and we cannot determine statically which one will actually be fired (and therefore

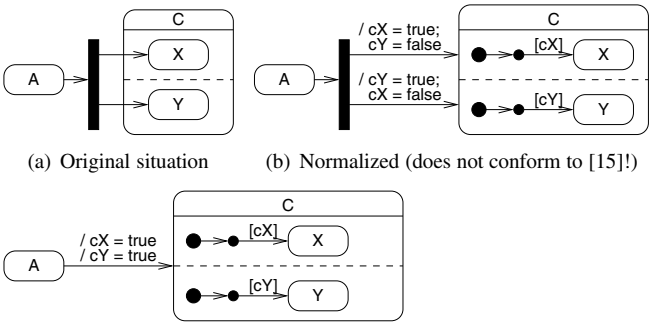
[4] Except the targets of the transitions leaving initial vertices, if these are junctions. Note that every region may contain an initial vertex.



(a) Original situation

(b) Normalized

Figure 7. Transformation: before



(a) Original situation

(b) Normalized (does not conform to [15]!)

(c) Fork and assignments of false removed

Figure 8. Normalization of fork vertices

which state will be the next active one), we set l_s to false in the exit action of each state that may turn active after s. Seen from the target's point of view: for all states z and x, we set the variable l_x to false in the exit action of z if x may have been active just before z turns active.

Note, however, that generally states may be composite and contain several orthogonal regions. State x, which "may have been active just before z", is not necessarily the source of some transition leading to z. Instead, for each state z, we set in its exit action variable l_x to false, for each state x such that there is a transition entering z from x, from an ancestor state of x, or from any state contained in any ancestor state of x.

For example, we introduce for the state LevelWon in Fig. 1 the entry action $a_{\mathsf{LevelWon}} = \mathsf{true}$, setting variable a_{LevelWon} to true to indicate that state LevelWon is currently active.

The exit action of LevelWon is more complex:

$$
\begin{aligned}
l_{\mathsf{LevelWon}} &= \mathsf{true} \\
l_{\mathsf{Ladder}} &= \mathsf{false}; & l_{\mathsf{Enchanted}} &= \mathsf{false}; \\
l_{\mathsf{Fight}} &= \mathsf{false}; & l_{\mathsf{CollectTreasure}} &= \mathsf{false}; \\
l_{\mathsf{Play}} &= \mathsf{false}; & l_{\mathsf{Hall}} &= \mathsf{false}; \\
l_{\mathsf{CrystalRoom}} &= \mathsf{false}; & l_{\mathsf{Ladder}} &= \mathsf{false};
\end{aligned}
$$

The first line indicates that after the execution of the exit action LevelWon is the state (more generally, one of the states) that has just turned inactive; the second line indicates that Ladder and Enchanted are no longer such states (these are the states from which—via a join vertex—LevelWon is directly reached); the other three lines indicate that the substates of Enchanted, as well as the states contained in the other region of Play are no longer such states either. It is not necessary to set l_{PowerUp} or l_{Spell} to be false, because they cannot have been the last active states before LevelWon turned active.

4.2.2 Pointcut

Equipped with trace variables, we are now in the position to implement pointcuts.

≪before≫ A pointcut of the form ≪before≫ s advises all transitions which activate the state s. In UML, a transition t may active a state s iff one of the two following cases is true:

1. $\mathsf{tgt}(t) = \mathsf{s}$, (for example, in Fig. 1, the transition T from NewLevel to Hall activates state Hall),
2. (recall s may be a composite state) $\mathsf{tgt}(t) \in \mathsf{substate}^*(\mathsf{s})$, (transition T also activates Play).

To solve the problem of capturing both transitions (in particular the second one), we first transform the base machine to eliminate the second form of state activation. The transformation is straightforward. For each transition t "goes into" (see below) some state, we "redirect" t to the container state of $\mathsf{tgt}(t)$ by setting $\mathsf{tgt}(t) \leftarrow \mathsf{state}(\mathsf{container}(\mathsf{tgt}(t)))$, and setting a fresh variable to indicate what to do when the container state of $\mathsf{tgt}(t)$ turns active. This variable is called an *initial variable* and will be checked by the initial transition to determine the (first) state to activate after the container's activation. We say that a transition t "goes into" a state s iff $s = \mathsf{state}(\mathsf{container}(\mathsf{tgt}(t)))$ and s is contained in $\mathsf{LCR}(\mathsf{src}(t), \mathsf{tgt}(t))$. When a transition t of this kind is fired, it activates not only $\mathsf{tgt}(t)$, but also the container state s. Our transformation above removes such transitions.

After this transformation, every state is activated iff a transition leading to the state is fired. Since a pointcut ≪before≫ s does not contain any other constraints, we set cond(s) simply to be true. That is, the aspect advises exactly all the transitions leading to s. Therefore we do not need a bypass transition for ≪before≫ aspects.

For example, if an aspect with pointcut ≪before≫ Enchanted is applied to Fig. 7(a), the base machine is first transformed to Fig. 7(b), and the aspect advises the transitions from Spell and Invisible to Enchanted. The variables cF and cC are used to indicate whether to activate Fight or CollectTreasure when Enchanted gets active.

Forks Note that the normalization step described above has to be extended if the source of the transition going into state s is a fork. For example, when Fig. 8(a) is normalized, the result (Fig. 8(b)) will be violating the constraint that "all

transitions outgoing a fork vertex must target states in different regions of an orthogonal state" [15, p.556]. Therefore, to all such transitions, we apply an additional transformation to remove the fork as well as those assignments where initial variables are set to false. The final result is given in Fig. 8(c).

≪after≫ Differently than in the case of ≪before≫, a pointcut of the form ≪after≫ S selects all transitions that are fired just after the *configuration* S gets inactive.[5] Obviously, we have to check if any state contained in S is deactivated. In UML, a state s can be deactivated by any one of the following transitions

1. all transitions t, such that $\mathsf{src}(t) = \mathsf{s}$ (for example, in Fig. 1, when the Transition $\mathcal{T}$ from Enchanted to LevelWon is fired, Enchanted is deactivated),
2. all transitions t, such that $\mathsf{src}(t) \in \mathsf{substate}^+(\mathsf{s})$ and $\mathsf{tgt}(t) \notin \mathsf{substate}^+(\mathsf{s})$ ($\mathcal{T}$ also deactivates Play),
3. all transitions t, such that $\exists S \in \mathsf{subvertex}(L), L = \mathsf{LCR}(\mathsf{src}(t), \mathsf{tgt}(t)), \mathsf{tgt}(t) \notin \mathsf{substate}^*(S) \cdot s, \mathsf{src}(t) \in \mathsf{substate}^+(S)$ ($\mathcal{T}$ also deactivates all states in the upper region of Play).

Given an ≪after≫ S aspect, its candidate set is therefore $T = \bigcup_{s \in \mathsf{S}}(T_1(s) \cup T_2(s) \cup T_3(s))$, where $T_1(s) = \{t \mid \mathsf{src}(t) = s\}$, $T_2(s) = \{t \mid \mathsf{src}(t) \in \mathsf{substate}^+(s) \wedge \mathsf{tgt}(t) \notin \mathsf{substate}^+(\mathsf{s})\}$, $T_3(s) = \{t \mid s, \mathsf{src}(t) \in S$ for some $S \in \mathsf{subvertex}(\mathsf{LCR}(\mathsf{src}(t), \mathsf{tgt}(t)))\}$ such that $t \notin \mathsf{substate}^*(S)$.

On each transition t contained in the candidate set, we still need to implement the run-time check whether t is actually advised by a pointcut ≪after≫ S. When t is fired, we know that *one of* the states contained in S has just turned inactive, we still have to check whether *all* states in S were active before the transition was fired. This is the case iff (at run time) $z = \bigwedge_{s \in \mathsf{S}} a_s \vee l_s$ is true, i.e. for each state s contained in S, either s is active, or s has just turned inactive. Recall that when this check is performed, i.e. when any transition $t \in T$ is fired, it is not possible that all states in S are active.

Finally, the constraint of the pointcut is also integrated (by conjunction) into cond(A) to ensure the aspect is only executed when the condition was satisfied. We therefore set condition c in Fig. 4(b) to be $\mathsf{cond}(\mathsf{A}) = z \wedge \mathsf{cons}(\mathsf{A})$. Recall cons(A) is the OCL constraint of the ≪after≫ pointcut, see Fig. 5(b).

Joins The above definition of T works fine if the base machine does not contain any join vertex. In order to include joins as well, we extend the definition of the candidate to $\{t \in T \mid \mathsf{tgt}(t)$ is not join$\} \cup T'$, where T is defined above, and T' contains all transitions such that $\mathsf{src}(t)$ is a join and for which t' and X exist, such that $\mathsf{tgt}(t') = \mathsf{src}(t)$, $X \in \mathsf{subvertex}(\mathsf{LCR}(\mathsf{src}(t), \mathsf{tgt}(t)))$, $\mathsf{src}(t') \in \mathsf{substate}^+(X)$ and $\exists s \in \mathsf{S} \cdot s \in \mathsf{substate}^+(X)$. The basic idea here is that if a transition's target is not a join, then the definition above works fine, and if the target is a join, then we select the section leaving the join.

For example, let the base machine be given by Fig. 1(a), an aspect ≪after≫ Enchanted would advise the transitions from Enchanted and the join to LevelWon, as well as from Fight to Spell and PowerUp. Recall that since no constraint of the pointcut is explicitly given, the default is true.

4.2.3 Resumption

When the execution of the aspect is finished (at a final state of the advice), control is given back to the base machine by activating the state specified in the label of the final state, provided that the condition of the goto label is satisfied.

This is implemented as follows: we store in a *resumption variable* the state to activate, introduce transitions from the aspect state to all possible states (i.e. all states indicated by all the final states in the advice), and decide where to go by checking the value of the resumption variable at run time.

On every transition leading to a final state f of the aspect a (except the bypass $b(a)$, see the introduction part of Sect.4.2), we assign the state of $\mathsf{label}(f)$ to the resumption variable. If a final state does not contain a label, the resumption variable is set to be the target of the advised transition. This variable will be used to determine the control flow of the base machine when the advice is finished. The bypass transition $b(a)$ does not carry an effect, so that the default value of the resumption variable is the target of the advised transition.

We also have to make sure the condition of the final state is satisfied before control is given back to the base machine. To this end, we guard the transitions leaving the aspect state by the condition $\mathsf{cond}(f)$. The oclConstraint part of the condition can be copied into the guard. The other part, cfgConstraint, which ensures that after the resumption a certain state configuration is (or is not) active, is implemented by checking if all states contained in the given configuration are active except for the target of the transition. In the implementation this amounts to checking the trace variables that were introduced in Sect. 4.2.1.

Note that we also declare the completion event of the aspect state to be deferrable[6], to make sure that this event is not lost even if the condition is not satisfied when the execution of the aspect state is finished.

For example, Fig. 9 shows the result of weaving Fig. 3(b) to the base machine Fig. 7(b). Note that both transitions from Spell and Invisible to Enchanted are advised, and that the resumption variables in the different aspect states are given unique names (by the weaver).

4.3 Multiple Aspects

Obviously, in any non-trivial system multiple aspects are needed to model the different features. For this reason, it is

[5] Actually, S getting inactive is caused by the transitions being fired.

[6] Using the syntax */defer, as before.

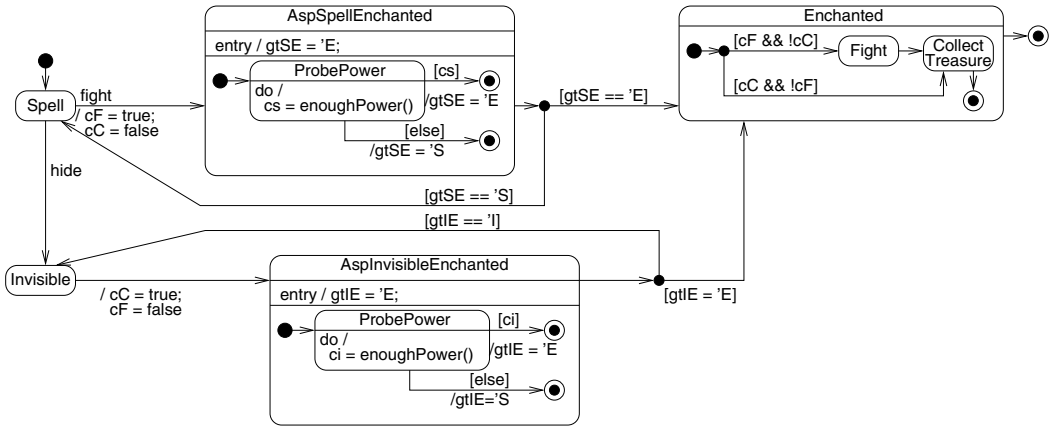


Figure 9. Weaving result of Fig. 3(b) to Fig. 7(b)

essential to design the weaving in a *confluent* way, i.e. the result is independent of the order of the individual aspects being woven. Particular care has to be taken for two kinds of conflicts: created joinpoints and shared joinpoints.

4.3.1 Created Joinpoints

After the execution of an aspect, if state x should be activated, the HiLA semantics considers this also as a situation to execute the ≪before≫ x aspects, if any. Our weaving algorithm should ensure that in such cases, after the execution of the aspect and before state x is actually activated, all ≪before≫ x aspects are really executed.

For the implementation, we introduce another normalization step: before any weaving, we first "unify" the transitions leading to the same states. That is, we introduce for each state s a junction, which we call $\mathsf{b4}(s)$, introduce a transition $\mathsf{tb}(s)$ from $\mathsf{b4}(s)$ to s with no trigger, no guard and no effect, and make all transitions entering s lead to $\mathsf{b4}(s)$. In other words, we replace every transition from state x to s by two transitions: one from x to $\mathsf{b4}(s)$, as well as $\mathsf{tb}(s)$. Given an aspect with ≪before≫ s, the set of candidate transitions $T(s)$ contains then only $\mathsf{tb}(s)$. We also implement goto s by a transition from the aspect state to $\mathsf{b4}(s)$.[7] This way, when an aspect is finished with goto s, the transition $\mathsf{tb}(s)$, and therefore ≪before≫ s aspects do get executed.

Note that a prerequisite for this normalization is, in the effect of the transition from x to $\mathsf{b4}(s)$, to store the source state of the original transition in a variable, otherwise the label goto src of the advice could not be implemented correctly.

For example, applying this idea to Fig. 7(a), and then weaving Fig. 3(b) to it, the result is shown in Fig. 10. Variable s is used to indicate the (source of the) transition actually leading to the junction and then to AspEnchanted. The value is then read in AspEnchanted for the implementation of goto src.

[7] Except s is the target of the advised transition. In this case goto s is implemented by a transition leading to s.

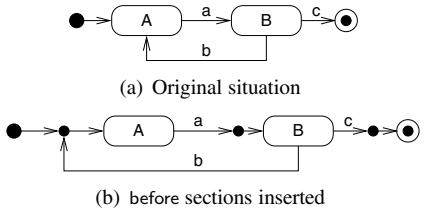


Figure 11. Weaving: unifying transitions leading to the same state

4.3.2 Shared Joinpoints

In non-trivial systems, joinpoints selected by (the pointcuts of) two or more aspects are generally not disjiont. In the context of HiLA, this happens when at a certain point in time in the execution of the base machine, two or more aspects are all supposed to run. In aspect-oriented approaches, it is important yet challenging to make sure that the weaving result does not (semantically) depend on the order in which the aspects with shared joinpoints are woven, and that possible conflicts between the aspects are minimized, see [1].

In HiLA, we benefit from the concurrent nature of UML state machines, and weave aspects with shared joinpoints into orthogonal regions of the aspect state. At run time, the aspects (that is, their advices) will be executed in parallel, and no goto is executed until all of the aspects are finished. This weaving is beneficial in that

- the weaving result is semantically independent of the order in which the individual aspects are woven, since different weaving orders only result in permutations of the regions of the aspect state. At run time, all the regions are, independently of their relative position, executed in parallel,
- and possible conflicting gotos of different aspects can be detected at latest at run time.

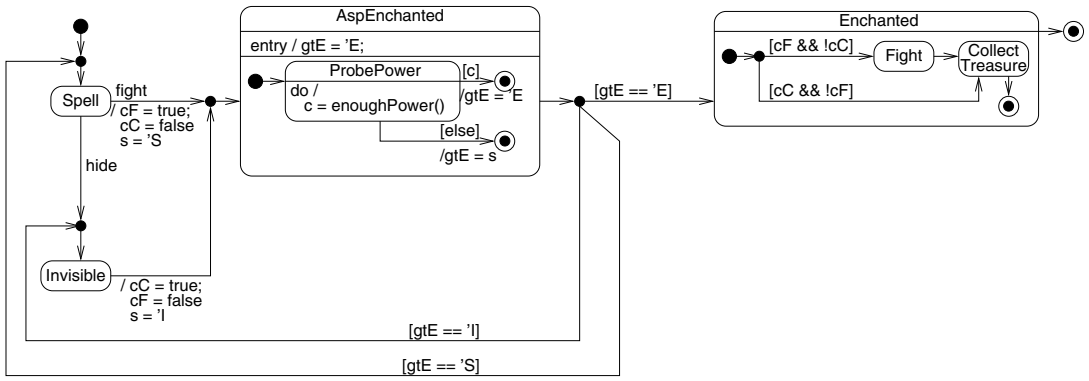


Figure 10. Normalization: before-section

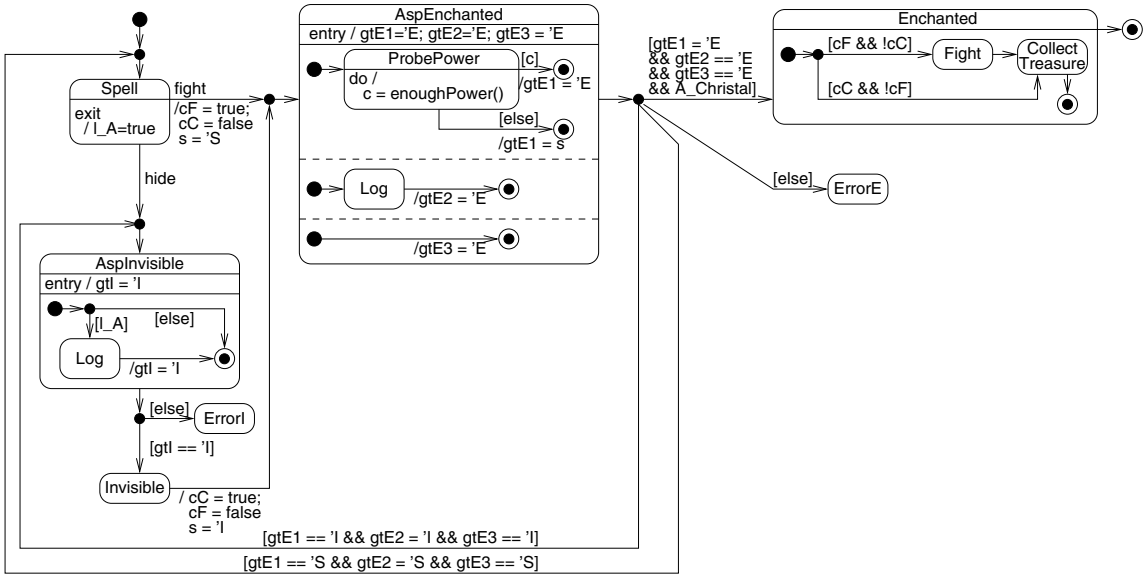


Figure 12. Weaving: shared joinpoints

For example, the result of weaving all three aspects defined in Fig. 3, with (S, T) bound to (Enchanted, CrystalRoom), to Fig. 7(a) is given in Fig. 12.

The idea of parallel execution of aspects is also valuable in other transition systems, beyond UML state machines. An example if given in Sect. 6.

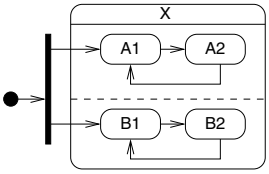


Figure 13. Base machine

5. Implementation and Validation

As a proof of concept, the above weaving process has been implemented in Hugo/HiLA, an extension of the UML translator and model checker Hugo/RT. The case studies described in [10, 19], as well as several smaller examples have been translated and model checked using this implementation. Model checking the weaving result has helped validate our weaving algorithms: we discovered some subtle errors in earlier definitions of weaving and our implementation of the weaver when model checking did not produce the expected results. Having used our weaver in these small to medium scale projects demonstrated not only the advantages of applying aspect-oriented techniques to state-machine models, but also that the implementation approach described in this paper can be applied to non-trivial examples.

Currently Hugo/HiLA does not support the graphical notation of HiLA. Instead, we have extended ute, the textual input format of Hugo/RT, to accept aspects as well. As an example, the ute definition of the simple base machine in Fig. 13 is shown in Fig. 14, and the ute presentation of the (instantiated) mutual-exclusion aspect in Fig. 3(c) is given

```
statemachine M1 {
  states {                transitions {
    initial INIT;           INIT -> FORK{}
    fork FORK;              FORK -> X.a.A1{}
    state X {               FORK -> X.b.B1{}
      region a {            X.a.A1 -> X.a.A2 {}
        state A1{}          X.a.A2 -> X.a.A1 {}
        state A2{}          X.b.B1 -> X.b.B2 {}
      }                     X.b.B2 -> X.b.B1 {}
      region b {          }
        state B1{}      }
        state B2{}
      }
    }
  }
```

Figure 14. UTE: base machine

```
aspect BeforeAspect {
  before config {         advice {
    state X.a.A2;           states {
    state X.b.B2;             initial AI;
  }                           labeledfinal AaF {
                                goto tgt
                                  f nlike {A2, B2}
                              }
                            }
                            transitions {
                              AI -> AaF{}
                            }
                          }
                        }
```

Figure 15. UTE: instantiated aspect

in Fig. 15, with S and T being bound to A2 and B2. That is, we implement a mutual-exclusion rule to prevent A2 and B2 from being active simultaneously.

After weaving, we ask Hugo/HILA if in the weaving result it is still possible for the states A2 and B2 to be active simultaneously, i.e. whether or not the states are really mutually excluded. This is expressed by the following formula in linear temporal logic (LTL):

```
F (inState(X.a.A2) and inState(X.b.B2))
```

The operator F (finally) states that the formula given as its argument will eventually hold, in this case that the states A2 and B2 will be active simultaneously at some point in time during the execution of the state machine. Hugo/HILA answers that this is no longer possible, which confirms that the mutual-exclusion aspect performs its intended function.

On the other hand, we also want to be sure that the aspect does not break other properties of the base machine. For example, we can also ask Hugo/HILA if it is still possible for the states A1 and B2 to be simultaneously active in the weaving result. This is expressed by the formula

```
F (inState(X.a.A1) and inState(X.b.B2))
```

and Hugo/HILA answers that this result is indeed still possible.

6. Related Work

Prevalent approaches of incorporating aspect-orientation into UML state machines, such as [3, 5, 12, 14], are mainly syntactic: their semantics are typically defined by graph transformation systems, such as Attributed Graph Grammar (AGG, [16]). In contrast, HILA has a transition-system semantics that is independent of any particular implementation strategy. We can therefore define the semantics of aspects in a purely behavioral manner and show that the weaving process described in this paper is sound with respect to the semantics; see [17] for details. Similar considerations apply to weaving processes for systems like Mealy-automata [4] or UML activity diagrams.

In static approaches, consistency checks are supported by a confluence check of the underlying graph transformation, see e.g. [12]. Due to the syntactic character of the aspects, this check is also syntactic: there may be false alarms if different weaving orders lead to syntactically different but semantically equivalent results. In contrast, in our approach described in Sect. 4.3.2, the error state is only entered when the resumption variables are really conflicting.

The pointcut language JPDD [9] also allows the modeler to define "stateful" pointcuts. Compared with HILA, a weaving process is not defined. State-based aspects in reactive systems are also supported by the Motorola WEAVR tool [6, 20]. Their aspects can be applied to the modeling approach Rational TAU[8], which supports flat, "transition-centric" state machines. In comparison, HILA is also applicable to UML state machines, that in general include concurrency. Moreover, HILA also considers hierarchical states and comes up with a more elaborate weaving process.

Ge et al. [8] give an overview of an aspect system for UML state machines. They do not give enough details for a thorough comparison, but it appears that the HILA language is significantly stronger the theirs, and that the issues presented in this paper are not addressed by their solution.

The position of HILA in a model-driven software development process is described in [2]. HILA has been successfully applied to model a crisis management system [10] and in the area of Web Engineering [19].

7. Conclusions and Future Work

We have presented HILA, a high-level aspect language for UML state machines, and in particular the weaving process underlying the HILA implementation in Hugo/RT. By eschewing a purely graph-transformation based approach in favor of a semantic one, HILA provides powerful, reusable abstractions that can greatly increase the modularity of state-machine models. Therefore HILA serves as an example of how a carefully designed aspect system can improve the expressive power and modularity of the base language. By basing our implementation on Hugo/RT it becomes possible to

[8] http://ibm.com/software/awdtools/tau/

validate the desired properties of model and aspects as well as the correctness of the weaving process.

The normalization step presented in this paper is heavily dependent on the semantics of UML state machines. It converts them into a form that is well-suited to static analysis since it clarifies the relationship between transitions and the states they activate and deactivate. Therefore this work may also be useful for static analysis of UML state machines.

Meanwhile, the issues addressed by the weaving process described in this paper are generally applicable to languages that support concurrency and hierarchical states. For example, we are involved in the development of the POEM language [11], an aspect-oriented modeling language for self-aware, autonomic ensembles. Aspects in POEM pose similar semantic challenges as aspects in HiLA, and we expect that the implementation of POEM will utilize the mechanisms described in this paper.

Currently, a graphical editor for HiLA aspects is under development [13] which will automate the translation step from graphical models to the textual ute notation, and therefore make HiLA easier to use in software development.

Acknowledgments

This work has been partially sponsored by the EU project ASCENS, 257414.

References

[1] M. Aksit, A. Rensink, and T. Staijen. A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points. In K. J. Sullivan, A. Moreira, C. Schwanninger, and J. Gray, editors, *Proc. 8th Int. Conf. Aspect-Oriented Software Development (AOSD'09)*, pages 39–50. ACM, 2009.

[2] M. Alférez, N. Amálio, S. Ciraci, F. Fleurey, J. Kienzle, J. Klein, M. E. Kramer, S. Mosser, G. Mussbacher, E. E. Roubtsova, and G. Zhang. Aspect-Oriented Model Development at Different Levels of Abstraction. In R. B. France, J. M. Küster, B. Bordbar, and R. F. Paige, editors, *Proc. 7th Eur. Conf. Modelling Foundations and Applications (ECMFA'11)*, volume 6698 of *Lect. Notes Comp. Sci.*, pages 361–376. Springer, 2011.

[3] S. Ali, L. Briand, and H. Hemmati. Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems. *Software and Systems Modeling (SoSyM)*, 2011. To appear.

[4] K. Altisen, F. Maraninchi, and D. Stauch. Aspect-Oriented Programming for Reactive Systems: Larissa, a Proposal in the Synchronous Framework. *Sci. Comp. Prog.*, 63(3):297–320, 2006.

[5] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: the Theme Approach*. Addison-Wesley, 2005.

[6] T. Cottenier, A. van den Berg, and T. Elrad. Joinpoint Inference from Behavioral Specification to Implementation. In E. Ernst, editor, *Proc. 21st Eur. Conf. Oriented Programming (ECOOP'07)*, volume 4609 of *Lect. Notes Comp. Sci.*, pages 476–500. Springer, 2007.

[7] D. Drusinsky. *Modeling and Verification Using UML Statecharts*. Elsevier, 2006.

[8] J.-w. Ge, J. Xiao, Y.-q. Fang, and G.-d. Wang. Incorporating Aspects into UML State Machine. In *Proc. Advanced Computer Theory and Engineering (ICACTE'10)*. IEEE, 2010.

[9] S. Hanenberg, D. Stein, and R. Unland. From Aspect-Oriented Design to Aspect-Oriented Programs: Tool-Supported Translation of JPDDs into Code. In B. M. Barry and O. de Moor, editors, *Proc 6th Int. Conf. Aspect-Oriented Software Development, (AOSD'07)*, pages 49–62. ACM, 2007.

[10] M. M. Hölzl, A. Knapp, and G. Zhang. Modeling the Car Crash Crisis Management System with HiLA. *Trans. Aspect-Oriented Software Development (TAOSD)*, 7:234–271, 2010.

[11] M. M. Hölzl, M. Wirsing, A. Klarl, N. Koch, S. Reiter, M. Tribastone, R. D. Nicola, D. Latella, M. Massink, U. Montanari, R. Bruni, F. Plasil, J. Kofron, J. Sifakis, S. Bensalem, J. Combaz, E. Vassev, and F. Zambonelli. ASCENS White Paper, 2011. http://www.ascens-ist.eu/.

[12] P. K. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Proc. 10th Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *Lect. Notes Comp. Sci.*, pages 151–165. Springer, 2007.

[13] A. M. Kourtessi. Entwicklung eines Modellierungswerkzeugs für aspekt-orientierte Zustandsmaschinen. Diplomarbeit, Ludwig-Maximilians-Universität München, 2011. In German.

[14] M. Mahoney, A. Bader, T. Elrad, and O. Aldawud. Using Aspects to Abstract and Modularize Statecharts. In *Proc. 5th Int. Wsh. Aspect-Oriented Modeling*, Lisboa, 2004.

[15] OMG. UML 2.3 Superstructure. Specification, Object Management Group, 2010. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/.

[16] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Rev. Sel. Papers 2nd Int. Wsh. Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *Lect. Notes Comp. Sci.*, pages 446–453. Springer, 2003.

[17] G. Zhang. *Aspect-Oriented State Machines*. PhD thesis, Ludwig-Maximilians-Universität München, 2010.

[18] G. Zhang and M. M. Hölzl. HiLA: High-Level Aspects for UML State Machines. In S. Ghosh, editor, *Rep. & Rev. Sel. Papers Wshs at MODELS'09*, volume 6002 of *Lect. Notes Comp. Sci.*, pages 104–118. Springer, 2009.

[19] G. Zhang and M. M. Hölzl. Aspect-Oriented Modeling of Web Applications with HiLA. In A. Harth and N. Koch, editors, *Rev. Sel. Papers Wshs. at ICWE'11*, Lect. Notes Comp. Sci. Springer, 2011. To appear.

[20] J. Zhang, T. Cottenier, A. van den Berg, and J. Gray. Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. *Journal of Object Technology*, 6(7):89–108, 2007.