Aspect-Oriented State Machines

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität München

zur Erlangung des Grades
Doctor rerum naturalium (Dr. rer. nat.)

vorgelegt von
Gefei Zhang

München, November 2010

Erstgutachter	Prof. Dr. Martin Wirsing
Zweitgutachter	Prof. dr. Antonio Vallecillo
Tag des Rigorosums	30. November 2010

Abstract

UML state machines are a widely used language for modeling software behavior. They are considered to be simple and intuitively comprehensible, and are hence one of the most popular languages for modeling reactive components.

However, this seeming ease to use vanishes rapidly as soon as the complexity of the system to model increases. In fact, even state machines modeling "almost trivial" behavior may get rather hard to understand and error-prone. In particular, synchronization of parallel regions and history-based features are often difficult to model in UML state machines.

We therefore propose High-Level Aspect (HILA), a new, aspect-oriented extension of UML state machines, which can improve the modularity, thus the comprehensibility and reusability of UML state machines considerably. Aspects are used to define additional or alternative system behaviors at certain "interesting" points of time in the execution of the state machine, and achieve a high degree of separation of concerns. The distinguishing feature of HILA w.r.t. other approaches of aspect-oriented state machines is that HILA aspects are defined on a high, i.e. semantic level as opposed to a low, i.e. syntactic level. This semantic approach makes HILA aspects often simpler and better comprehensible than aspects of syntactic approaches.

The contributions of this thesis include 1) the abstract and the concrete syntax of HILA, 2) the weaving algorithms showing how the (additional or alternative) behaviors, separately modeled in aspects, are composed with the base state machine, giving the complete behavior of the system, 3) a formal semantics for HILA aspects to define how the aspects are activated and (after the execution) left. We also discuss what conflicts between HILA aspects are possible and how to detect them. The practical applicability of HILA is shown in a case study of a crisis management system.

Zusammenfassung

UML-Zustandsmaschinen werden sehr oft verwendet, um das Verhalten von Software-Systemen zu modellieren. Sie gelten als intuitiv verständlich, und sind eine der populärsten Sprachen für die Modellierung reaktiver Komponenten.

Allerdings nimmt diese empfundene Einfachheit rapide ab, sobald die Komplexität des zu modellierenden Systems zunimmt. Selbst Zustandsmaschinen, die „fast-triviale“ Verhalten modellieren, können schwer verständlich und fehleranfällig werden. Insbesondere Synchronisierung paralleler Regionen und history-basierte Features sind oft nur schwer mit UML-Zustandsmaschinen zu modellieren.

Als eine mögliche Lösung solcher Probleme präsentieren wir die Sprache High-Level Aspect (HILA), eine neue, aspekt-orientierte Erweiterung von UML-Zustandsmaschinen, welche die Modularität, die Lesbarkeit und die Wiederverwendbarkeit von UML-Zustandsmaschinen erheblich erhöht. Aspekte werden verwendet, um zusätzliche oder alternative Verhalten des Systems zu bestimmten „interessanten“ Zeitpunkten in der Ausführung der Zustandsmaschinen zu definieren. HILA hebt sich von anderen Ansätzen aspekt-orientierter Zustandsmaschinen dadurch ab, dass HILA-Aspekte auf einer hochen, d. h. semantischen Abstraktionsebene und die Aspekte anderer Ansätze auf einer niedrigen, d. h. syntaktischen Abstraktionsebene definiert sind. Durch unseren semantischen Ansatz sind HILA-Aspekte oft einfacher und besser verständlich als die von syntaktischen Ansätzen.

Zu den Beiträgen dieser Dissertation gehören 1) die abstrakte und die konkrete Systex von HILA, 2) die Weavings-Algorithmen, wodurch die in Aspekten separat modellierten (zustätzlichen oder alternativen) Verhalten mit der Basis-Maschine komponiert werden und so das komplette Systemverhalten ergeben wird, 3) eine formale Semantik für HILA-Aspekte, welche definiert, wie die Aspekte aktiviert und (nach deren Ausführung) terminiert werden. Wir diskutieren auch, welche Konflikte zwischen HILA-Aspekten möglich sind, und wie sie festgestellt werden können. Die praktische Anwendbarkeit unseres Ansatzes wird durch eine Fallstudie eines Crisis-Management-Systems validiert.

Mom and Dad

Acknowledgements. I am indebted to Prof. Dr. Martin Wirsing for giving me the chance to conduct a PhD research within his uniquely excellent research group (PST). I am very grateful for his patience and for always believing in me, even more than I ever did. I learned very much from Prof. Wirsing, a great scientist, and a wonderful, generous man.

I thank Prof. dr. Antonio Vallecillo for being my external referee and providing valuable discussions related to this thesis.

I thank Matthias "Genius" Hölzl for the *infinitely* many hours of discussion and many great ideas in designing and implementing HILA. I thank Alexander Knapp for sharing his deep insight into state machines and his expertise of LaTeX. I thank all my PST colleagues for the very friendly atmosphere, which makes PST a great place to work at. I thank the *Deutsche Forschungsgemeinschaft* (DFG) for partially supporting this research within the project MAEWA.

I thank Bingbing for taking care of Emily on so many weekends. Last but not least, I thank you, Emily, for brightening up my life with your sunshine.

Contents

Part 1. HiLA Ante Portas 1

Chapter 1. Introduction 3
1.1. UML State Machines: Nice but Not Nice Enough 4
1.2. Modularization 5
1.3. Aspect-Oriented Software Development 5
1.4. Dynamic vs. Static, Declarative vs. Imperative, High-level vs. Low-level 6
1.5. Conflicts between Aspects 7
1.6. Goals 7
1.7. Organization of This Thesis 8

Chapter 2. UML State Machines 9
2.1. Syntax and Informal Semantics 9
2.2. Hard-to-Model Features 14
2.3. Feature Interference 16
2.4. Wanted: Better Separation of Concerns 17

Chapter 3. Static Aspects 19
3.1. Syntax and Informal Semantics 20
3.2. Metamodel 21
3.3. Weaving 21
3.4. Consistency Checking 22
3.5. Discussion 23

Part 2. HiLA des Ingénieurs 25

Chapter 4. HiLA 27
4.1. HiLA in a Nutshell 28
4.2. Examples 30
4.3. Abstract Syntax and Informal Semantics 35
4.4. Big Picture 43
4.5. Discussion 43

Chapter 5. Weaving 45
5.1. Main Idea 46
5.2. Prerequisites 47
5.3. Preprocessing 51
5.4. Weaving History Properties 60
5.5. Weaving Aspects 63
5.6. Postprocessing 69
5.7. Correctness 71

5.8. Implementation 72
5.9. Discussion 77

Part 3. HiLA d'Ivoire 79

Chapter 6. Formal Semantics 81
6.1. Abstract Transition Systems 81
6.2. UML State Machines 82
6.3. History Properties 84
6.4. Structural Extension by ≪whilst≫Aspects 85
6.5. Behavior Extension 86
6.6. Weaving and Semantics 90
6.7. Discussion 92

Chapter 7. Interaction of aspects 95
7.1. Change of State Reachability 97
7.2. Conflict Detection 99
7.3. Discussion 101

Part 4. HiLA du Monde 103

Chapter 8. Case Study 105
8.1. Overview and Static Structure 106
8.2. Modeling the Behavior of the CCCMS 109
8.3. Validation of the Model 117
8.4. Discussion 119

Chapter 9. Related Work 123
9.1. Event Condition Action Systems and Programming Languages 123
9.2. Modeling Languages Supporting Static Aspects 123
9.3. Modeling Languages Supporting Dynamic Aspects 124
9.4. Aspect Interference 125

Chapter 10. Conclusions and Future Work 127
10.1. Summary 127
10.2. Future Work 128

Appendix A. Remaining Use Cases of the CCCMS 129
A.1. Use Case 2: Capture Witness Report 129
A.2. Use Case 3: Assign Internal Resource 130
A.3. Use Case 4: Request External Resource 133
A.4. Use Case 5: Execute Mission 135
A.5. Use Case 6: Execute SuperObserver Mission 135
A.6. Use Case 7: Execute Rescue Mission 137
A.7. Use Case 8: Execute Helicopter Transport Mission 139
A.8. Use Case 9: Execute Remove Obstacle Mission 139

Bibliography 141

Index 147

Part 1

HiLA Ante Portas

CHAPTER 1

Introduction

Contents

1.1. UML State Machines: Nice but Not Nice Enough 4
1.2. Modularization 5
1.3. Aspect-Oriented Software Development 5
1.4. Dynamic vs. Static, Declarative vs. Imperative, High-level vs. Low-level 6
1.5. Conflicts between Aspects 7
1.6. Goals 7
1.7. Organization of This Thesis 8

The Unified Modeling Language (UML, [57]) is the *lingua franca* in object-oriented analysis and design. UML state machines are widely used to model dynamic behaviors of software systems. State machine models, however, often show insufficient *separation of concerns* [19] due to missing language constructs for structuring and modularization.

It has been proposed to apply techniques of *Aspect-oriented Modeling* (AOM) to solve this problem. In AOM, different parts (for instance, different features) of the system behavior are modeled in separate constructs called *aspects* and composed together according to some predefined *weaving* algorithm. Usually, a *base model* is used to define a common basis, which is usually the core functionality of the system, for the aspects to be woven to. The aspects then define modifications of the base model.

The prevalent proposals of aspect-oriented state machines view aspects as syntactic, as opposed to semantic, modifications of the base model. That is, aspects are actually model transformations to the base model. These aspects are called static or transformation aspects.

Since model transformations (and hence transformation aspects) define in the first instance syntactic modifications to the base model, the change of the semantics to the base model is often hard to grasp: it often necessitates studying the weaving result carefully to understand what the new, composed semantics is. Moreover, conflict detection, i.e. if weaving two or more aspects (model transformations) in different orders would lead to syntactically different result models, is available only on the syntactic level. Detection of semantic conflicts, for instance, one aspect being made effectless by another, is not directly supported.

To overcome these problems, we present High-Level Aspects (HILA), an aspect-oriented UML extension for state machines. HILA aspects are semantic aspects, that is, they define in the first instance modifications of the base model semantics

rather than of its syntax; the semantics of an aspect is defined locally in the aspect rather than in the weaving result. Therefore modeling with HILA is more declarative than it is with plain UML state machines or with static aspects.

The semantic nature of HILA aspects also give rise to several other benefits: HILA facilitates a higher level separation of concerns: modeling and reasoning of separate aspects is possible in a greater extend than in the case of using plain UML state machines or transformation aspects; conflict detection is also more accurate. Moreover, HILA is supported by Hugo/HILA, an extension of the UML translator and model checker Hugo/RT [45]. The seamless integration with Hugo/RT facilitates easy model checking of HILA aspects.

In the following, we explain the motivation and goals of this thesis in more detail.

1.1. UML State Machines: Nice but Not Nice Enough

UML state machines originated from Harel [32] and are nowadays a widely used language for modeling software behavior. Their application areas include embedded systems [51, 59], web applications [20, 46, 76], software product lines [73], etc. In fact, they are even considered "the most popular language for modeling reactive components" [23]. A large number of books, research papers on and tools for state machines testify this popularity.

The reasons for this popularity seem to be clear: state machines are considered easy to use, their syntax simple, and their semantics intuitively comprehensible. In particular, using UML state machines to obtain a better understanding of the system under modeling seems to be common wisdom: "To understand complex classes better, particularly those that act in different manners depending on their state, you should develop one or more UML 2 state machine diagrams" [5].

However, this seeming ease to use vanishes rapidly as soon as the complexity of the system to model increases. In fact, even state machines modeling "almost trivial" behavior may get rather hard to understand and error-prone. Maybe it is not only coincidence that "whenever people write about state machines, the examples are inevitably cruise controls or vending machines" [29], even for modestly sized systems intelligible state machines are hard to find.

How can this situation arise? We believe the reason is that state machines are too low level a language. They provide only case distinction and *goto* for control flow. More elaborate behavior, like history-based features or synchronization of parallel regions, thus easily torpedos the brevity of state machines: elements modeling one feature get scattered all over the machine, while single elements are involved in multiple features.

Moreover, it is difficult to modularize state machines, the only structural element being submachines. Different features of the system can hardly be modeled separately and then combined together precisely. As a consequence, it is rarely possible to "develop more UML state machine diagrams" to model complex behavior as proposed in [5]. Instead, a single machine has to define the complete behavior of the object under modeling. Thus, although drawing "a state machine diagram for a single class to show the lifetime behavior of a single object" [29] is not a desirable development methodology, it is more often than not the only choice to construct a UML state machine.

1.2. Modularization

Modularization, however, is the quintessence of efficient software development. Ideally, artefacts of a software system are organized in small, independent units (modules), each specifying, designing, or implementing a part of the whole system. There should be no interference between the modules for the software developer to take care of, the interaction of the modules should be defined precisely and made transparent to the developer, the composition of the modules giving the complete behavior of the system.

The effort to achieve better modularization has been a main line of Software Engineering research. The best known example is supposedly the evolution of programming paradigms from unstructured over structured and procedural to object-oriented programming, with procedures, methods and classes modularizing algorithms and data, hiding information and reducing interference as much as possible.

The advantages of a clean modularization are obvious: the achieved information hiding allows us to realize and reason about different concerns of the system locally, without having to take care about other modules, and thus reduces the complexity of the system. Therefore, it makes a great contribution to enhance the comprehensibility and maintainability of software systems, and reduction of their error proneness.

Object-orientation is the most established modern software development paradigm. As powerful as it is, object-orientation is considered insufficient with respect to modularization in many systems, see [39]. In object-oriented systems, there are still many so called "cross-cutting concerns", which cannot be modeled or implemented in one separate module, but only by several modules jointly (*scattering*). Meanwhile, there may also be modules that are responsible for several concerns (*entangling*).[1] Both scattering and entangling make software maintenance and evolution, even modeling or implementing in the first place, a hard job, since some module, although involved in some concern, can be easily forgotten, or the one part of a module, addressing one concern, may have unintended repercussions on other parts addressing other concerns. The aforementioned difficulty of modeling non-trivial behavior using UML state machines is actually a result of their insufficient modularity. Concrete examples will be given in Chap. 2.

1.3. Aspect-Oriented Software Development

One answer to the problem of insufficient separation of concerns in object-oriented systems is Aspect-Oriented Software Development (AOSD), firstly introduced in the form of Aspect-Oriented Programming (AOP, [40]).

A new programming language construct, *aspect*, was introduced by AOP. Aspects are used to define additional or alternative behavior of the program at certain "interesting" points of time (called *join point*s) in the execution. Possible join points are before or after method invocation, before or after constructor call, etc. An aspect contains a *pointcut* and an *advice*. The advice defines the additional or alternative behavior, the pointcut selects a subset of join points of a certain program (called *base program*) to apply the advice. This way, an aspect defines a part of the system behavior separately and non-intrusively, i.e. without modification of the

[1] For a more formal definition of entangling and scattering see [68].

base program. The aspects, or, more precisely, their advices, are then woven together with the base program, the result of the weaving process finally realizes the desired, complete behavior of the system. Since aspects provide a means to source out features from the base program, they can help improve modularity, readability, maintainability and the chance of correctness considerably. Success stories of aspect-oriented programming can be found in [1].

In other software development phases, the idea of enhancing object-orientation with aspects to separate concerns also enjoys great popularity. Aspect-orientation has been applied in requirements analysis, design and implementation of software development, for an overview see [27]. In particular, modeling languages may benefit from aspect-orientation in the same way as programming languages do. After all, models are nothing else than abstract programs.

1.4. Dynamic vs. Static, Declarative vs. Imperative, High-level vs. Low-level

In Aspect-Oriented Modeling, aspects can be classified as dynamic or static, see [28]. Static aspects are defined as syntactic transformation of the base model. Their semantics is structural and can (and can only) be specified as a function on a set of model elements (the base model). Static aspects are therefore the right choice to define structural modifications of the base model, or, if the base model contains only structural information of the system under modeling, as do, for example, class diagrams. For behavior modeling base models, such as state machines, using static aspect to define behavior modification is often not a good idea, since the aspect *per se* does not define any behavior, only the composition model (the weaving result), in the form of a new set of model elements, defines a behavior. Therefore, the semantics of the aspects can be only grasped after careful study of the weaving result.

Dynamic aspects, on the contrary, are defined as modifications of the behavior of the base model. Their semantics is defined on top of the semantics, as opposed to syntax, of the base model. Therefore, a dynamic aspect has its own semantics, which is a modification of the behavior defined in the base models. Modeling and reasoning about behavior is therefore facilitated on the behavior level and thus easier than it is in the case of static aspects.

In this sense, dynamic aspects are declarative, whilst static aspects are imperative. With dynamic aspects, the modeler simply defines the intention of the modification to the base model's behavior, the implementation details being hidden behind the weaving process. With static aspects, it must be modeled how to realize the modification by adding model elements to or removing model elements from the base model. Obviously the latter is in most cases much more cumbersome than the former.

Note that static approaches are generally more powerful than dynamic aspects. In fact, since dynamic aspects are also implemented by introduction of model elements to the base model or removal of model elements from the base model, every dynamic aspect can also be "implemented" by static aspects. Oppositely, it is rather easy to construct examples of static aspects that are difficult to be simulated by dynamic aspects, if it is possible at all. The real advantage of dynamic aspects is not being more powerful, but rather being more "user-friendly", since the modeler no longer has to worry about the implementation details. In this sense, dynamic aspects are high-level, static aspects are low-level aspects.

1.5. Conflicts between Aspects

Obviously, only in simplest cases one aspect suffices for designing the system behavior. Realistic systems often necessitate a large number of aspects. Their interference, or rather conflicts, has always been a hot topic of the AOSD research.

In fact, aspects are deemed to be a two-edged weapon. The arguments of why Aspect-Oriented Programming is considered harmful [16] or why its success paradoxical [64] are also valid in the context of Aspect-Oriented Modeling. In particular, compared to classes and methods in object-oriented systems, the interaction between aspects is much more subtle and often only implicitly defined. While individual aspects may correctly realize single concerns in separation, their composition may not show the desired properties, since some aspect interferes with others, see [2]. Therefore, it is very important in the design of aspect-oriented languages to minimize the interference between aspects, and to deliver mechanisms for interference detection.

The optimal interference detection should mark all semantically interfering aspects, but none syntactically conflicting ones. Roughly spoken, aspects are semantically interfering if weaving them in different orders yields different composition models defining different behavioral semantics; aspects are syntactically conflicting if weaving them in different orders yields syntactically different composition models. Clearly, semantically interfering is stronger than syntactically conflicting; if aspects are semantically interfering, they must be syntactically conflicting but not vice versa.

However, "marking all semantically-conflicting aspects" is an undecidable problem. This task can only be achieved by simulating the complete system. More practical are conservative approximations, which means automatic warnings are given at each potential conflict, and it is a human expert that decides whether a potential conflict is also an actual conflict. The most conservative approximation is to give a warning at each syntactic conflict. Undoubtedly, improving the accuracy of the conflict detection mechanism, establishing detection rules less yet sufficiently conservative, is more than desirable.

1.6. Goals

The goal of this research is therefore to define an aspect-oriented extension of UML state machines. Our language is called High-level Aspects (HILA). The principle requirement for HILA is that it be a high level language, that is, HILA aspects should be used to simply specify what is to be done by the system, without the modeler having to define the details of how it should be done.

The other goals of this research are as follows

- In order to enable seamless integration of HILA and UML, HILA should to be embedded into the UML by an abstract syntax defined as an extension of the UML metamodel.
- Detection of conflicts between aspects has always been an important topic in the research of aspect-oriented software development. The weaving algorithm of HILA should be designed in such a way that the potential of conflicts between aspects is minimized. Moreover, mechanisms for automatic detection of remaining conflicts should be given.

- The semantics of HILA should be defined precisely as a structural operational semantics. Formal validation of its models should be enabled, so that modeling mistakes, if any, could be uncovered as early as possible.
- Finally, weaving of HILA aspects should be automated by some tool support.

The following chapters will show in detail how these goals are reached.

1.7. Organization of This Thesis

The rest of this thesis is organized as follows:
In the rest of Part 1, we

- briefly review the syntax and semantics of UML state machines, and show why it is hard to model certain problems with them modularly (Chap. 2),
- give an introduction of the static approaches of aspect-oriented state machines, and show why they do not provide a satisfactory answer of the modularity problems of the UML state machines (Chap. 3).

In Part 2, we

- define the abstract and concrete syntax and an informal semantics of HILA, and show how it can be used to enhance the modularity of UML state machines (Chap. 4),
- show the weaving algorithms of HILA aspects (Chap. 5).

In Part 3, we

- define a formal semantics of HILA aspects (Chap. 6),
- discuss potential conflicts between HILA aspects, and their detection (Chap. 7).

Finally, in Part 4, we

- show how HILA is applied to a larger scale application to demonstrate its practical applicability (Chap. 8),
- discuss related work (Chap. 9),
- and give some concluding remarks and an outline of some future work (Chap.10).

CHAPTER 2

UML State Machines

Contents

2.1. Syntax and Informal Semantics **9**
2.1.1. Syntax 10
2.1.2. Metamodel 11
2.1.3. Informal semantics 12
2.1.4. Execution model 13
2.2. Hard-to-Model Features **14**
2.2.1. Synchronization 14
2.2.2. History-based Behavior 15
2.3. Feature Interference **16**
2.4. Wanted: Better Separation of Concerns **17**

UML state machines are a very popular language for modeling system behaviors, especially for reactive systems. They are deemed as simple and intuitive [29], and are applied to software development in quite a few areas. However, the complexity of UML state machine models may increase rapidly as soon as the system under modeling gets complex. This is due to two inherent weaknesses of UML state machines: 1) the language is very low-level, the only control flow constructs being case distinctions and *goto*s; 2) modularization is only insufficiently supported, the only modularization construct being submachines. As a result, state machines modeling non-trivial systems can get rather hard to read, hard to construct, and prone to errors.

In the following, we study by means of a few examples the syntax and informal semantics of UML state machines, as well as the weaknesses of the language. To mitigate these weaknesses is the main motivation of this thesis and therefore poses the requirements of the UML state machine extension that we envisage. The requirements will be discussed at the end of this chapter.

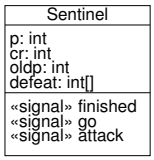
Publication Notice. Part of this chapter was published in [82, 83].

2.1. Syntax and Informal Semantics

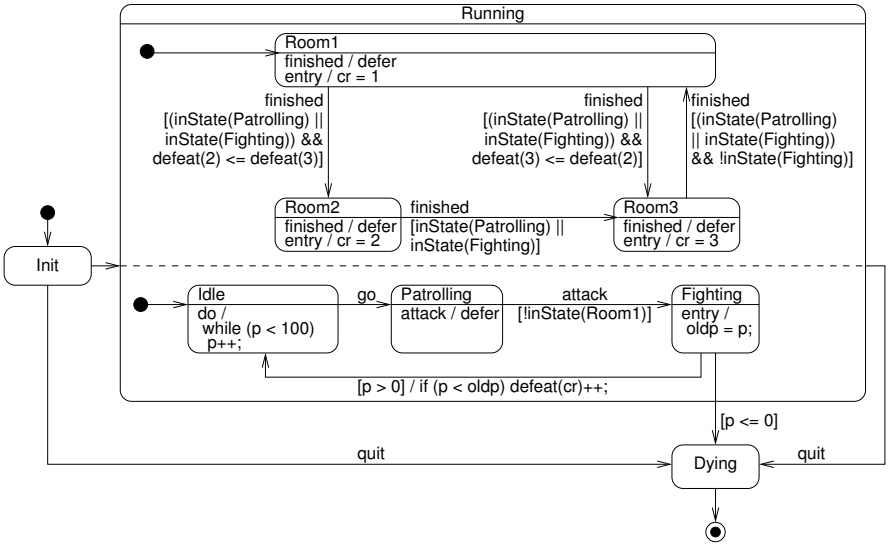
We use the sample state machine as given in Fig. 2.1b to review the syntax and semantics of UML state machines according to the UML specification [57].

The state machine models a sentinel in a computer game. The sentinel has an integer attribute p, which indicates his power between 0 and 100, 0 meaning that he is dying. The main behavior of the sentinel is modeled in the state Running, where, very briefly speaking, he may be Idle to tank his power, Patrolling through three rooms, or Fighting. His power changes during fighting (not modeled explicitly in this machine), an increase indicating a win, a decrease a loss. If the sentinel lost

too much power and p is no longer positive, he dies. The class diagram is given in Fig. 2.1a.



(a) Class diagram



(b) State machine

Figure 2.1: Example: Sentinel

The sentinel is supposed to exhibit some intelligence in the following features:

(1) The sentinel should not be Fighting while he is in Room1.
(2) Changing room is only allowed when the sentinel is Patrolling or Fighting.
(3) When leaving Room1, the sentinel is required to enter the room where it has lost the fewest fights so far.

Moreover, it is designed so that the user can quit the game any time.

The details of how these features are modeled in Fig. 2.1b will be discussed later on, but a brief glance makes clear that even a UML state machine modeling a small number of rather simple features like these may get rather complex, hard to understand, and thus invite mistakes.

2.1.1. Syntax. A UML state machine consists of *regions* which contain *vertices* and *transitions* between vertices. The sentinel machine in Fig. 2.1b contains one region. A vertex is either a *state*, in which the state machine may dwell, and which may hierarchically contain its own regions; or a *pseudo state* used for building compound transitions. 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.

A state is *simple*, if it contains no regions (such as Init, Idle, etc. in Fig. 2.1b); it is *composite*, if it contains at least one region; a composite state is said to be *orthogonal* if it contains several regions, visually separated by dashed lines (Running). A state may show an *entry* behavior (like in Fighting) and an *exit* behavior (not shown in Fig. 2.1b), which are executed on activating and deactivating the state, respectively; a state may also show a *do activity* (Idle) which is executed while the state machine sojourns in this state. Transitions are triggered by events (attack, go, etc.), may show guards (!inState[Room1] etc.), and specify *actions* to be executed when a transition is fired (not shown). Completion transitions (such as the ones leaving Fighting) are triggered by an implicit *completion event* emitted when a state completes all its internal activities. Events may be *deferred* (like in Patrolling and the Room states), that is, put back into the event pool, if they are not to be handled currently but only later on. Note that we write the deferring of completion events as * / defer. By executing a transition, its source state is left and its target state entered. 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.[1]

Some other model elements are not shown in this example: *Junction* pseudo states, also depicted as filled circles, may be used 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. *Entry* and *exit* points make explicit the entry and the exit of composite states.

In this research, we do not consider shallow and deep history pseudostates and choice pseudostates. They can be simulated by other model elements; and are omitted in this thesis for simplicity.

2.1.2. Metamodel. The metamodel of UML state machines that we study in this thesis is given in Fig. 2.2. There are some differences between this metamodel and the one given in the UML Specification [57].

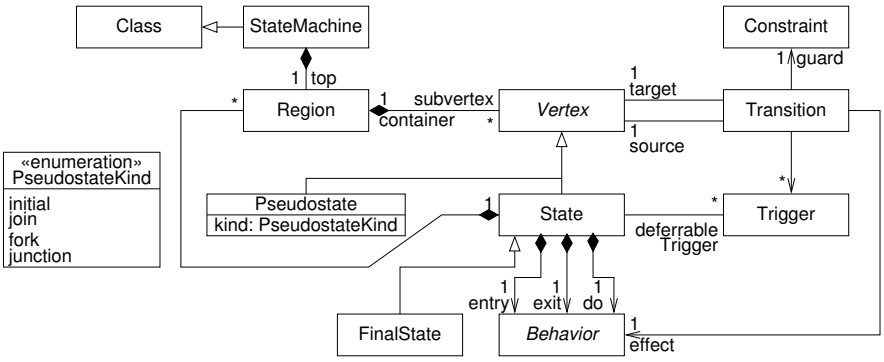


Figure 2.2: Simplified metamodel of UML state machines

[1]The two paragraphs above are mainly taken from [44], with some minor modifications of the text.

- While according to [57, p. 527] there may be several regions in a state machine, we assume that a state machine always has one region, which we call top. Parallelism can be modeled in a orthogonal state, concurrent regions on the top level are not necessary.
- We do not distinguish the different kinds of transitions. A distinction would not have any effect on the discussion in the later chapters.
- We do not consider entry and exit vertices. They are only syntactic constructs, and can be easily eliminated from the state machine in a semantics preserving way.
- We do not consider deep and shallow history vertices. These can be simulated by variables to store the last active state.
- We do not consider terminate or choice vertices. Including these vertices would make our discussion in Chaps. 5 and 6 unnecessarily complex, without producing any substantial scientific contribution. Choices can be simulated by simple states.
- We removed the container association between Transition and Region. The semantics of this association is not clearly defined in [57].
- We require that a state always have an entry, an exit and a do behavior, which may be skip, having no effect on the environment.
- We require that a transition always have an effect, which may be skip, and that a transition always have a guard, which may be true.
- We require that if there are transitions leading to a composite state S, then each region of S must contain an initial vertex. Though this is actually not required by [57], it is necessary since otherwise the semantics of S's activation would be undefined.
- As opposite to [57, p. 532], we allow final states to have entry and exit behaviors. Since FinalState is a subclass of State, we think it more natural to allow final states to have the properties that states have. However, we stick to the requirements of [57, p. 532] that final states must not have regions or outgoing transitions, since their semantics would be highly dubious.

Apart from these minor differences, Fig. 2.2 defines the same metamodel as defined by the UML Specification [57, p. 527].

2.1.3. Informal semantics. Aa state gets active when entered and inactive when exited as a result of a (compound) 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 tree starting from the states in the top-level region 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.[2]

EXAMPLE 2.1 (Run). A run of the sentinel state machine (Fig. 2.1b) may begin as follows:

(1) When the state machine is started, state Init gets active.
(2) The transition leading to Running is fired, resulting in the deactivation of Init and the activation of Running, which means that its two parallel regions are executed in parallel. The first stop they make is at Room1 and Idle. The active state configuration thus consists of three states: Running, Room1 and Idle. Upon entering Room1, the variable cr is assigned the value 1.
(3) In this configuration, the next event that can be handled is go. An event finished would be deferred, since the OCL constraint inState(Patrolling) || inState(Fighting) is not satisfied, and therefore no transition leaving Room1 is enabled. Any other event would be simply discarded since no transition is enabled. Upon go, the sentinel starts patrolling, the active state configuration now consists of Running, Room1, and Patrolling.
(4) In this configuration, the sentinel may go to another room. Since all three elements of the array defeat have the value 0, and inState(Patrolling) is true, both transitions leaving Room1 are enabled. Therefore, he may nondeterministically choose to enter Room2 or Room3.
(5) Upon a quit event, state Running amongst its substates will be deactivated, and Dying activated. The active state configuration will contain only Dying.

2.1.4. Execution model. The semantics above is very abstract. The UML Specification [57] intentionally leaves much freedom for the actual implementation. In the following, we highlight some important assumptions of the execution model of UML state machines our research is based on. This model is in accordance to the semantics given in [44].

- It cannot be emphasized enough that state machines generally model parallel behaviors. If a stable state configuration contains a state in one region, then it also contains one from each of other regions of its container state.
- In a stable state configuration, there is at most one active state in each region. If a region contains an active state, it is referred to as an active region.
- In general, a multitude of transitions are fired at each execution step: one from each active region, unless the active state in a region does not react to the current event.

[2]The paragraph above is mainly taken from [44], with some minor modifications of the text.

- The enabled transitions are fired in pseudo parallel mode. The order of the enabled transitions being fired is not predictable.
- Firing a transition is considered atomic, which means that the following actions cannot be interrupted: 1) deactivating the source, 2) executing the exit action of the source, 3) executing the effect of the transition, 4) executing the entry action of the target, and 5) activating the target.
- Synchronization of parallel regions is based on messages: when two states in two different regions are active, and there are transitions leaving from each of them that are enabled by the current event, then these two regions should make a step (pseudo) simultaneously.
- If the current event does not enable any transition, it is discarded, unless it is declared as "deferrable". In this case it is put back to the event pool, and will be handled when it does enable some transition.

2.2. Hard-to-Model Features

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 other states (in other regions) are also active. Unfortunately, behavior that depends on such information can hardly be modeled modularly in UML state machines. In the following, we illustrate by examples some of the features that may make UML state machines all messed up.

2.2.1. Synchronization. Clearly, synchronization of parallel regions is a cross region feature, which means that it breaks the most natural way of separation of concerns by regions. Both simultaneous action execution in parallel regions and mutual exclusion of states being simultaneously active necessitate pervasive change of states and transitions modeling other features of the system. Moreover, in case transitions cannot get active because of some synchronization rule, it is often necessary to declare the trigger events to be deferrable since otherwise they would be simply lost.

For example, consider the mutual exclusion requirement for our sentinel modeled in Fig. 2.1b that he not get Fighting while he is in Room1. How is this simple feature modeled there? Consider only the lower region for simplicity. It might be obvious that the transition entering Fighting must be guarded by the OCL-constraint !inState(Room1), but this is only the half of the truth: Patrolling still has to declare the event attack to be deferrable, otherwise attack would be lost if Room1 and Patrolling are both active, since the transition to Fighting is not enabled. The sentinel would then stay in the state Patrolling until the next rest event is handled.

Another example is the feature of the sentinel that when leaving Room1 he should always choose a room to enter where he has so far lost at the least. Modeling this feature requires quite much of cross region thinking:

- In the upper region, every room must claim to be the current room by assigning the variable cr the corresponding value.
- In the lower region, Fighting must contain an entry action to save the value of p when the fight starts. Moreover, the transition from Fighting to Idle first has to check if the last fight was lost (if (p $<$ oldp)) and then to update the array defeat which stores how often he has lost in which of the rooms.

- Back in the upper region, this information is used to expand the guard of every transition to ensure that it is only enabled when the sentinel has its target the least frequently.

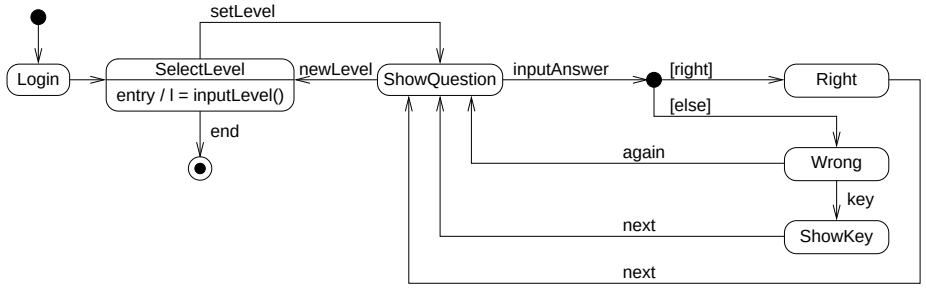
Obviously, it is not satisfactory that the model elements modeling such a simple feature are scattered all over the state machine, and that the comprehension of the model requires carefully studying the semantics of the assignments and the guards.

2.2.2. History-based Behavior. Since the UML state machine does not keep track of the execution history, it is also difficult to model history-based behaviors, no matter if cross region or not.

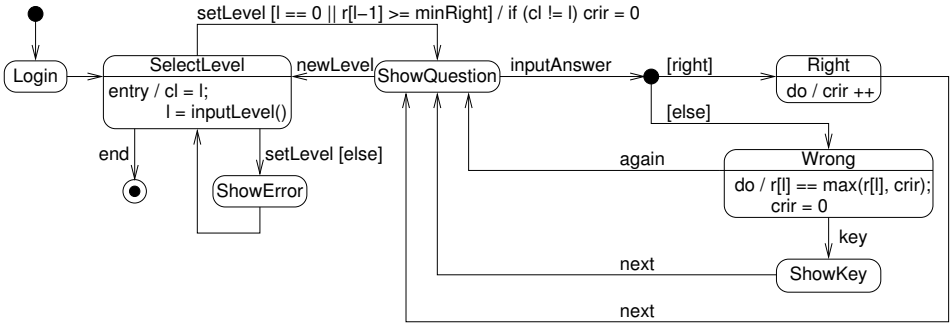
For example, consider Fig. 2.3a, which models an e-learning system, where the user first logs in (Login), selects a level of difficulty (SelectLevel), and then proceeds to answering questions of this level. The system tells him whether his answer is right (Right) or not (Wrong). If it is right, the user may proceed to the next question; if not, he may choose to try again, to see the key to the question (ShowKey), 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. We call this feature *minRt*. Figure 2.3b, which is taken from a previous publication [83], shows how *minRt* 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 using the predefined function max to assign r[l] the greater value of the current value of r[l] and crir. 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. It is rather unsatisfactory that the model elements such as crir involved in this one simple 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.

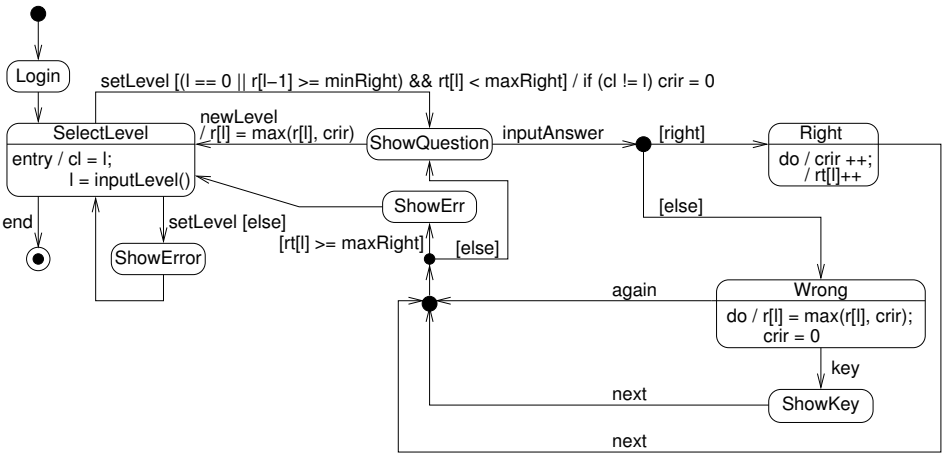
If this is not bad enough, this model is even *wrong*! Consider the following scenario: the user answers minRight times correctly at level 0, and, without answering any question wrong, proceeds to selecting level 1. Since in this case the state Wrong does not get active, the state machine does not have a chance to update the array r, and consequently, when a new level is selected in SelectLevel, the transition to ShowQuestion compares the old value of r[l-1] that does not reflect the latest lucky streak of the user to minRight, and may therefore falsely decide not to allow him to proceed. This mistake was not discovered until we model checked Fig. 2.3b using the tool Hugo/RT [45], six months after the publication of [83]. This mistake is corrected in Fig. 2.3c, where another feature is also modeled, see below.



(a) Basic feature



(b) Modeling level selection restriction using standard UML (erroneously)



(c) Modeling two additional features using standard UML

Figure 2.3: Example: an e-learning system

2.3. Feature Interference

Different features of a software system are seldom orthogonal, instead, they may interact with each other, impairing or even canceling the effects of each other. Therefore, for software systems with a multitude of features, quite a lot modeling complexity is caused by conflicts between different features. This also applies to state machines, where coordination of individual features may also get annoying when building UML state machines. Conflict detection requires carefully studying

the model, the reconciliation accurate manipulation of the transition guards. Moreover, additional model elements become necessary to prevent events being lost if one feature is overruled by another.

Back to our e-learning system (Fig. 2.3a), consider an additional feature, which we call *maxRt*, with the intention of keeping it challenging: as soon as the user has mastered a level l, which is assumed when the user has answered maxRight question on this level correctly, he is no longer allowed to answer a question on the level any more (and is forced to choose a more difficult level).

This feature is modeled in Fig. 2.3c. Yet another array rt (right) is used to store the number of right answers given in each level. The transitions leading to ShowQuestion must be extended by a guard to guarantee that the maximal number of right answers has not been reached yet, otherwise the transition is stopped, and an error message is shown. Note we reused the state ShowError for this error message too, a distinction of too many or tow few right answers would just make the model more complex.

Like most features in realistic projects, including *maxRt* in the system poses a number of questions. While the feature *minRt* already defines a restriction of level selection, *maxRt* defines another. How do these two restrictions interact? Is it possible that one of them allows the selection of a level and the other forbids it? If yes, which one should overrule the other? Moreover, it might be obvious that when only one of the restrictions is applied, every level is, independently of the actual value of minRight or maxRight, somehow reachable, but is it still the case when both of them are applied?

Why are such interactions hard to handle in UML state machines? First, given an even rather simple state machine like Fig. 2.3c, the detection of such interactions requires carefully examination of the machine. Second, even when conflicts have been detected, their reconciliation it is not an easy job, since the "who-rules-whom" decision has to be implemented by precisely arranging the "and"s and "or"s in the guards of the transitions. In Fig. 2.3c, we decided to require that both restrictions be satisfied for the user to select a new level. A biased prioritization of the two rules would lead to a model that has slight differences in syntax and major differences in semantics than Fig. 2.3c.

A third, perhaps the most annoying problem is the need of feature coordination, which means a separation of concerns by features is, albeit very natural, hardly possible, since the modeler has to pay much attention to the coordination of the feature with each other or with the base machine. For example, modeling feature 1 (p. 10) in Fig. 2.1b hinders the sentinel from always being capable to react to an event attack while he is Patrolling, and requires therefore to declare attack as deferrable in Patrolling, since otherwise the event would be lost when the sentinel is Patrolling but cannot get Fighting since he is in Room1. Having to change model elements in the base machine, or those modeling other features (not shown in this example), is obviously poisonous for separate modeling of different features.

2.4. Wanted: Better Separation of Concerns

The discussion above made clear that even supposedly simple features may be hard to model in UML state machines, the models may therefore get hard to read, and be, more often than might be expected, erroneous. Highly desirable would

be therefore new language constructs which enable a better separation concerns, which means

(1) different features are no more entangled, but modeled separately from each other,
(2) model elements concerning one feature are not scattered all over the model, but gathered in one place,
(3) coordination of features no longer needs to be modeled explicitly, but is taken care of by a precisely predefined algorithm which composes the features together,
(4) possible interferences between features are revealed automatically, priority of certain rules over others is defined explicitly as opposed to implicit modeling, e.g. by combining transition guards with “and” or “or”.

As stated in Chap. 1, Aspect-Oriented Modeling is deemed to be a promising paradigm for separation of concerns in software design. In the following, we will study why the so called *static aspects* do not fully satisfy these requirements before presenting our dynamic approach to solving the aforementioned problems of UML state machines.

CHAPTER 3

Static Aspects

Contents

3.1. Syntax and Informal Semantics **20**
3.2. Metamodel **21**
3.3. Weaving **21**
3.3.1. Graph Grammar 21
3.3.2. Rules implementing aspects 22
3.4. Consistency Checking **22**
3.5. Discussion **23**

The success of Aspect-Oriented Programming in enhancing the separation of concerns in programs gave rise to aspect-oriented modeling approaches to achieving a better modularization of cross-cutting concerns in software design models. In most prevalent approaches, a base model is used to specify the core functionality of the system, and in aspects, separated from the base model, other parts of the system behavior are modeled. The composition of the aspects and the base model is achieved in a weaving process.

The aspects fall into two categories:

- Static (low-level) aspects define modifications of the syntax of the base model. A static aspect defines a transformation of model elements of the base model. Static aspects are therefore also called transformation aspects.
- Dynamic (high-level) aspects define modifications of the semantics of the base model. A dynamic aspect defines a transformation of the behavior defined by the base model.

The state of the art approaches of aspect-oriented state machines are static in the sense that in these approaches aspects are defined as syntactic modifications of the base model. While these approaches provide valuable support for separate modeling of parts of the system behavior, modeling non-trivial systems with these approaches may still get cumbersome and error prone, since they do not increase the degree of abstraction of the modeling language, and the modeler still has to implement the system behavior in every detail.

While our work [79] was one of the first approaches of static aspects, the focus of this thesis is on high-level, dynamic aspects. In this chapter, we first present the definition of our static aspects, then we show how this language can be used in Aspect-Oriented Modeling, and discuss why static aspects alone are not sufficient for effective software development with state machines. The reasons why it is not very satisfactory to model non-trivial systems with our static aspects also apply to other static aspect approaches.

3.1. Syntax and Informal Semantics

In HILA, the syntax of static aspects is a slight adaption of the one presented in [79]. The syntax defined by other approaches that view aspects as model transformations, such as [75], could also be used.

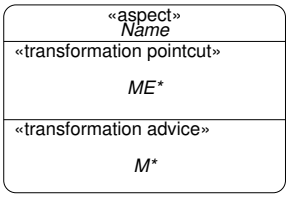


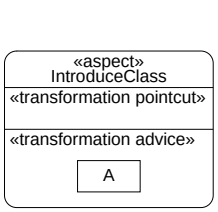
Figure 3.1: Transformation aspects

HILA's static aspects are not only applicable to state machines, but rather to any UML diagram (called the *base model*). A static aspect contains a ≪transformation pointcut≫ and a ≪transformation advice≫, see Fig. 3.1. The transformation pointcut contains a pattern. The pattern contains an arbitrary set of model elements *ME**, e.g. classes, vertices, or transitions, and can match a fragment of the base model. We require that at most one fragment of the base machine can be matched. That is, our static aspects, as opposed to other prevalent approaches, do not support quantification *per se*. This is less a restriction that it seems to be, since aspects can be defined as UML templates, and quantification can be realized by multiple instantiations.

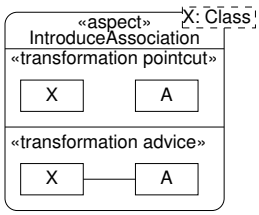
The meaning of the aspect is that the fragment of the base machine that matches *ME** should be replaced by the model elements contained in the advice, *M**. Note that weaving is not recursive, but executed only once.

There are some special cases defined:

- If the (transformation) pointcut is empty, then the (transformation) advice is added to the base model.
- Elements having the same name in the pointcut as in the advice refer to the same elements.
- Internal variables have names that begin with a question mark.



(a) Introducing class A to the system



(b) Introducing an association between X and A

Figure 3.2: Example: static aspects

EXAMPLE 3.1. Figure 3.2 shows two static aspects. In Fig. 3.2a the pointcut is empty. The aspect therefore means that the advice, consisting of class A only, should be added to the base model. In Fig. 3.2b an aspect template is defined: it has a formal parameter X of the type UML Class, the pointcut matches the pair of X and another class A, the advice introduces an association between them.

3.2. Metamodel

The metamodel of static aspects is very simple. A TransformationAspect consists of a TransformationPointcut and a TransformationAdvice. All three metaclasses are defined as subclasses of UML Package (see Fig. 3.3), thus being capable of containing other UML elements.

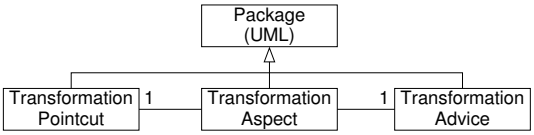


Figure 3.3: Metamodel: static aspects

3.3. Weaving

The graphical character and the "match-and-replace" semantics of static aspects suggest the use of graph transformation for implementation. We implement our static aspects with the Attributed Graph Grammar System (AGG, [67]). Weaving is realized by the following procedure:

(1) translate the base model and the aspects to a graph grammar system,
(2) use AGG to realize the transformations,
(3) translate the resulting graph back to UML.

The weaving of our approach [79] was prototypically implemented according to this procedure in [72].

3.3.1. Graph Grammar. Graph grammars are used to define modifications of graphs. A graph contains nodes and edges. A graph grammar consists in a set of rules $\{r_i \mid i \in \mathbb{N}\}$. A rule r has a left hand side $\mathsf{LHS}(r)$, a right hand side $\mathsf{RHS}(r)$, and an optional negative application conditions $\mathsf{NAC}(r)$, each of the three components being a set of nodes and edges.

The semantics of a graph grammar is defined in terms of Category Theory [26] and is beyond the scope of this thesis, the interested reader is referred to [24]. Informally, a rule r is applicable to a graph G iff. $\mathsf{LHS}(r)$ matches some fragment F of G while the $\mathsf{NAC}(r)$, if any, does not match F. When r is applied, the matched fragment is replaced by $\mathsf{RHS}(r)$. Applying a graph grammar to G means applying all its rules to G: first the set R of applicable rules are determined, then one element $r \in R$ is selected non-deterministically and applied to G, yielding another graph g', which is used in the next iteration. These two steps are repeated until there is no applicable rule any more.

3.3.2. Rules implementing aspects. The implementation is straight forward. We consider the base model as a graph, and construct for each aspect a rule r, with the pointcut as LHS(r) and the advice as RHS(r). This way, the graph grammar defines the model transformation semantics of the aspects.

Note that a static aspect in our approach is woven only once, while in graph grammar systems in general and AGG in particular, a rule can be applied recursively. Especially in aspects for introduction of model elements into the base model (e.g. Fig. 3.2b), where the pointcut is a subset of the advice, we need to implement some mechanism to stop avoid recursive, thus non-terminating, weaving of such aspects.

To this end, for each aspect, if its pointcut is a subset of its advice, we weave into the corresponding grammar rule a NAC, consisting of the same elements of the RHS. The rule is only applicable when the graph does not match the NAC, i.e. the RHS. At run time, after the (first) application of this rule, the resulting model will contain the RHS, thus the NAC will prevent the rule from being executed again.

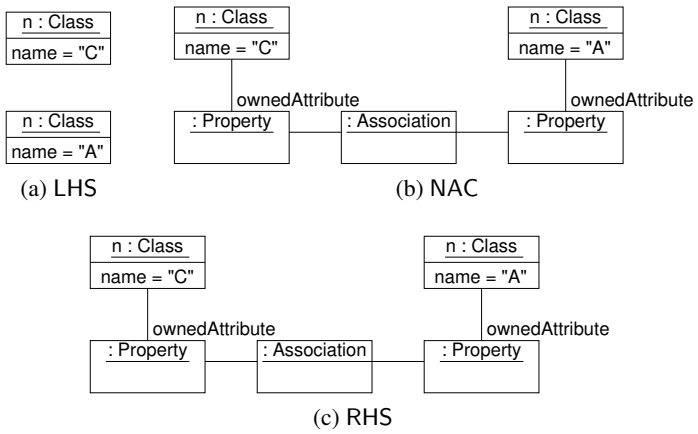


Figure 3.4: Example: weaving of Fig. 3.2b, X bound to C

If the pointcut is not a subset of the advice, after one execution of the rule, the fragment matching the pointcut will be replaced by one that does not, and the rule will not be applicable to the resulting graph any more.

EXAMPLE 3.2 (Weaving of static aspects). Figure 3.4 shows how an instance of the aspect IntroduceAssociation (Fig. 3.2b) is translated to a graph grammar rule, where the formal parameter X was bound to a class C. The LHS (Fig. 3.4a) contains the pointcut, the RHS (Fig. 3.4c) contains the advice. Since LHS $\subseteq$ RHS, a NAC $=$ RHS is also generated to avoid recursion application of this rule.

3.4. Consistency Checking

Two or more aspects are conflicting if weaving them in different orders leads to different result models. For example, the two aspects in Fig. 3.2 are conflicting if the basis model does not contain a class A: first weaving Fig. 3.2a and then Fig. 3.2b yields a model which contains class A, connected with associations from selected classes (the actual parameters X in Fig. 3.2b is instantiated with). First weaving

Fig. 3.2b and then Fig. 3.2a, on the contrary, would yield a model in which class A is contained in isolation, i.e. not connected to other classes. The reason is that when weaving Fig. 3.2b first, there is still no class A in the base model, which means the pointcut does not match any part of the model, hence there is nothing to weave. Then, in the next step, weaving of Fig. 3.2a, class A is woven into the model, without associations.

In graph grammar systems, consistency checking is also an important topic. Given a graph G, two or more grammar rules are conflicting if applying them to G in different orders leads to different result graphs. If in a graph grammar system the rules are not conflicting, the system is called confluent.

Since we implement our static aspects by graph grammars, the question of consistency between aspects amounts to the confluence of the graph grammar , which can be checked automatically by the tool of AGG. This technique is widely used by static aspect approaches, see [38].

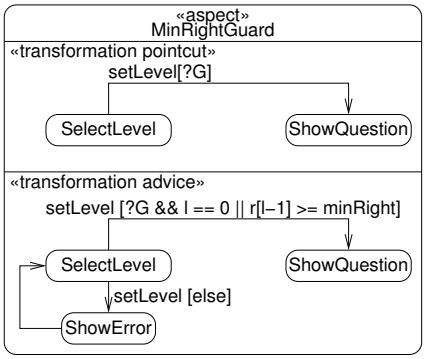
3.5. Discussion

Static aspects are a rather popular way for separation of concerns on the level of software design, proposals presented so far include [2, 66, 75]. Compared with these approaches, the pointcut language of our approach does not allow quantification by pattern matching. Instead, we use UML templates to define aspect templates and source out quantification to the binding of the formal parameters. The outsourcing makes the semantics of our static aspects easier to understand, without reducing the expressive power of the language. Selection languages like JPDD [31] can be easily used to select the model elements a formal parameter should be instantiated with.

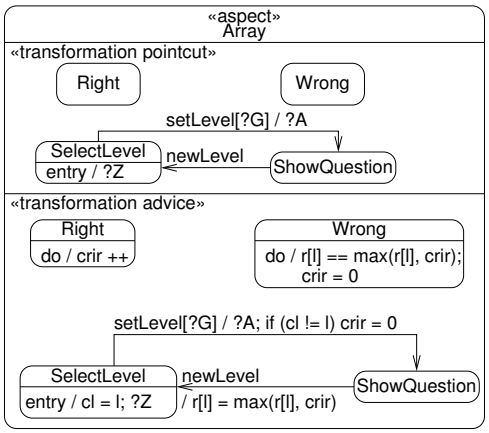
Static aspects are actually model transformations. Consistency checking via automatic confluence checking of the underlying graph grammar system helps the modeler design conflict free aspects [33, 74].

On the other hand, static aspects are low level constructs: they define only syntactic modifications. The understanding of the weaving result's semantics requires carefully studying the result model; modeling using static aspects is as error-prone as directly modifying the base model. This can be seen in the following example.

EXAMPLE 3.3 (Modeling the *minRt* feature of the e-learning example with static aspects). Figure 3.5 shows a static aspect based implementation of the level selection restriction *minRt* of our e-learning system, using Fig. 2.3a (p. 16) as the base model. For separation of concerns, the required modifications of the base model are defined in two aspects: MinRightGuard models mainly how to split the transition from SelectLevel to ShowQuestion into two to handle the cases whether the user may or may not change level. In particular, the guard of the transition in the base model is stored in ?G in the pointcut and reused in the advice. An additional state ShowError is introduced to give an error message. More complex is the aspect Array, modeling how to introduce an array r and a variable crir to do the counting. The value of crir is increased in Right, set to zero in Wrong, where the array r is also updated to store the current maximal length of sequential right answers. Note that the transition from SelectLevel to ShowQuestion, already modified once in aspect MinRightGuard, needs modification again, i.e. its effect must be extended by an action to set crir to zero if the user has chosen a new level (cl != l, cl is set by an extension of the entry action of SelectLevel). Finally, the transition from



(a) Splitting the transition from SelectLevel to ShowQuestion into two



(b) Implementing the array to count the current number of right answers in a row

Figure 3.5: Example: static aspects modeling the *minRt* feature of the e-learning system

ShowQuestion must also extend its action to set r[l], otherwise this modeling would exhibit the same mistake as the one in Fig. 2.3b.

This example reveals some of the reasons why static aspects are sometimes suboptimal

- Static aspects are less declarative than imperative. It is seldom possible for the modeler to define *what* the desired semantics of the system is. He often has to define in every little detail *how* to do it. In the above example, instead of just saying "allow change to level $l + 1$ only when minRight right answers in a row on level l", we have to define in Fig. 3.5 how the elements of the base model should be modified.
- In the first place, it is the syntax that is modified by static aspects. The modification of the semantics is not directly defined. For example, the behavior of the result model of weaving Fig. 3.5 to Fig. 2.3a is everything but intuitive.
- Since the modeler has to implement everything imperatively, modeling with static aspects is also error-prone. In Fig. 3.5, it is the modeler who has to pay attention not to forget to extend the effects of the transition from ShowQuestion to SelectLevel. Otherwise this aspect would be just as erroneous as Fig. 2.3b.

Therefore, static aspects do not provide a totally satisfactory answer to the problems of the UML state machines discussed in Sect. 2.2. What we need is an approach of high level, i.e. declarative, behavioral aspects. HILA is such an approach, as will be shown in the following chapters.

Part 2

HiLA des Ingénieurs

CHAPTER 4

HILA

Contents

4.1. HILA in a Nutshell **28**
4.1.1. Pointcut 29
4.1.2. Advice 30
4.1.3. History properties 30
4.2. Examples **30**
4.2.1. Mutual Exclusion 31
4.2.2. Synchronization 32
4.2.3. Static aspects 32
4.2.4. History 33
4.2.5. Additional Events 34
4.3. Abstract Syntax and Informal Semantics **35**
4.3.1. Aspect 35
4.3.2. Configuration Selector 35
4.3.3. Transition Selector 36
4.3.4. Transition pointcut 37
4.3.5. Configuration Pointcut 38
4.3.6. Advice 38
4.3.7. History properties 40
4.3.8. Priorities 42
4.4. Big Picture **43**
4.5. Discussion **43**

The discussion in Part 1 revealed two weaknesses of UML state machines: the lack of modularization constructs and being low level. Static aspects (model transformations) have been proposed to solve the first problem; indeed they do help to source out parts of the overall model and to increase the degree of separation of concerns.

On the other hand, model transformations are themselves low level instruments: they transform only model elements, as opposed to the behavior of the collection of model elements. Consequently, as shown in Sect. 3.5, the modeler has to design every detail of the desired behavior, the semantics of the result model can only be comprehended after carefully studying the weaving result, modeling with static aspects is therefore as error-prone as modeling by modifying the base model directly. Moreover, due to the syntactic character of static aspects, consistency checking is only possible on the syntactic level. There may be false alarms in situations where weaving aspects in different order would result in syntactically different yet semantically equivalent models.

We therefore present our approach, High-Level Aspect (HILA). HILA is defined as an aspect-oriented extension of the UML. The most distinguishing feature

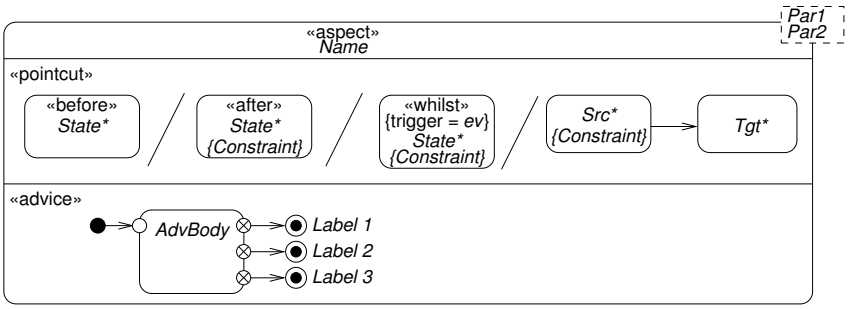


Figure 4.1: Concrete syntax of aspects

of HILA w.r.t. other proposals of aspect-oriented state machines is that HILA aspects have a run time semantics rather than being defined as pure syntactic transformations. That is, rather than a syntactic modification, a HILA aspect defines an alternative or additional behavior of the base machine, to be executed at some "interesting" points of time in the execution of the base machine. HILA aspects are therefore more declarative and more intuitively comprehensible than static aspects. The weaving algorithm of HILA exploits the concurrent nature of UML state machines, and thus minimizes interactions between aspects.

In the following, we first give a general introduction to HILA, show modeling examples, define the metamodel of HILA in the form of a conservative extension of the UML metamodel, and then give an informal description of its semantics.

Publication Notice. Sections 4.1 and 4.2 are an extension of [82, 83].

4.1. HILA in a Nutshell

In HILA, an aspect contains a *pointcut* and an *advice*. The advice defines an alternative or additional behavior, to be executed by the base machine at some "interesting" points of time, the pointcut specifies the interesting points of time. The set of all possible interesting points of time are called *join point*s, a pointcut defines a subset of the join points. We define the *constellation* of a state machine to be the pair of its active state configuration and its environment, i.e. the valuation of its variables. Given a set of states S and a proposition p over the environment of a state machine SM, the current constellation is *matched* by the pair (S, p) iff. S is a subset of SM's current active state configuration, and its environment satisfies p.

Our join points are the points in time just before, just after, or whilst the state machine being in a certain constellation. As will be explained in more detail later on, these three types of points in time can be actually subsumed under the firing of some transitions: the moment in which an advice should be executed can always be comprehended as a moment of some transition being fired. In this sense, an aspect is a graphical statement to interrupt the execution of certain transitions, "advise" them by executing some additional behavior, and then resume the execution of the base machine. The transitions advised by an aspect are specified, or selected, by the pointcut, the additional behavior to execute is defined in the advice.

Summarizing, a HILA aspect can be considered as a graphical representation of an event-condition-action rule: the pointcut defines the *precondition* to fire an

action, the advice defines the action to fire. The informal semantics of the aspect is that whenever the precondition is satisfied, the action should be executed.

The concrete syntax of HILA aspects is shown in Fig. 4.1, the *oblique* identifiers being place holders. As shown in this figure, an aspect consists of a ≪pointcut≫ and an ≪advice≫ compartment. There are several variants of pointcut definition, see Sect. 4.1.1. The advice is itself a state machine, its final states may exhibit a label and a constraint. For example, the advice in Fig. 4.1 shows three labels. Aspects are usually defined as UML templates with parameters to increase reusability. The parameters are then bound by the designer of concrete aspects. In Fig. 4.1 two parameters Par1 and Par2 are defined.

4.1.1. Pointcut. The pointcut of a HILA aspect may be either a "single configuration pointcut" or a "between pointcut".

Single configuration pointcuts. A "single configuration pointcut" maybe a ≪before≫, an ≪after≫, or a ≪whilst≫ pointcut. In the concrete systax, the single configuration pointcut is identified by one of these stereotypes, see Fig. 4.1.

The simplest form of single configuration pointcuts is that of ≪before≫ pointcuts. In a ≪before≫ pointcut, only a set of states *State** is given, the pointcut specifies the join points just before any of the states in *State** becomes active.

An ≪after≫ pointcut selects all transitions to be fired when the current constellation of the base machine is matched by the pair (*State**, *Constraint*). That is, it selects all transitions leaving some state in *State** provided that 1) the state machine is in a state configuration containing all states in *State** and 2) the environment of the state machine satisfies *Constraint*. ≪before≫ and ≪after≫ pointcuts select transitions that are already present in the base machine.

Note that in the case of ≪before≫ pointcuts we require only one of the states becoming active, and do not specify a constraint. The reason is that the join points for a ≪before≫ pointcut is used to select are *in the future*, and it is in general an undecidable question which states (possibly in other regions of the state machine) will be active and what values the state machine's variables will have after a transition being fired (and in particular after the execution of its effect and the entry behavior of the target state), and if their valuation will satisfy a certain constraint.

A ≪whilst≫ pointcut contains a set of states *State**, a constraint *Constraint* and a trigger *ev*. It specifies the join points that the current constellation of the base machine is matched by the pair (*State**, *Constraint*) and the current event is *ev*. We require that *ev* be a new event, that is, in the base machine there be no transition triggered by *ev* leaving any state contained in *State**. Therefore, a ≪whilst≫ pointcut actually introduces new transitions to the base machine: the sources are the states contained in *State**, the trigger is *ev*. The target of each transition is the same state as the source.

Between pointcuts. The pointcut may also contain two sets of states, *Src**, enhanced by a constraint *Constraint*, and *Tgt**, and a transition. In the concrete syntax, a transition pointcut does not carry a stereotype, see Fig. 4.1. We also refer to between pointcuts as "≪between≫ pointcuts".

Such a pointcut matches all transitions which are fired when the current constellation is matched by the pair (*Src**, *Constraint*) and the base machine is about to enter a state contained in *Tgt**. Intuitively, all transitions fired "between" two state configurations matched by *Src** and *Tgt** are selected. For the same reason as why ≪before≫ pointcuts do not have a constraint, *Tgt** does not, either.

In the terms of event-condition-action rules, the aspect is a rule stating "whenever P holds, do A. The pointcut defines the precondition P. More concretely,

- a ≪before≫ pointcut is satisfied whenever a state (of the base machine) contained in the pointcut is about to become active;
- an ≪after≫ pointcut is satisfied whenever a constellation of the base machine has just become inactive, in which all the states contained in the configuration were active and the constraint was satisfied;
- a ≪whilst≫ pointcut is satisfied whenever a constellation of the base machine is active, in which all states contained in the configuration are active and the constraint is satisfied, and the trigger is the current event.
- a ≪between≫ pointcut is satisfied whenever a constellation of the base machine has just become inactive, in which all the states contained in *Src** were active and the constraint was satisfied, and a state contained in *Tgt** is about to become active.

4.1.2. Advice. The advice of an aspect defines the behavior to execute when the precondition defined by the pointcut is satisfied. The advice is also a UML state machine, except that the final states may have a label and a constraint. The aspect is executed in the same way as a UML state machine is executed. When a final state is reached, i.e., when the execution of the aspect is finished, the execution of the base machine is resumed according to its label and constraint.

4.1.3. History properties. In order to address properties of the execution history, an aspect may contain *history properties*, defined in a ≪history≫ compartment, see Fig. 4.2. A history property contains a name and a pattern consisting of sets of states and constraints. The pattern is used to match contiguous subsequences of the execution history. The value of the history property is the number of matches of the pattern in the base machine's execution history where the constraints are satisfied. Figure 4.2 shows two history values, hs1 and hs2. The values of a history property can be used in pointcuts and advices to define history-sensitive behavior.

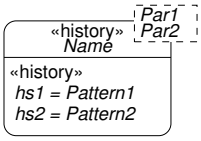
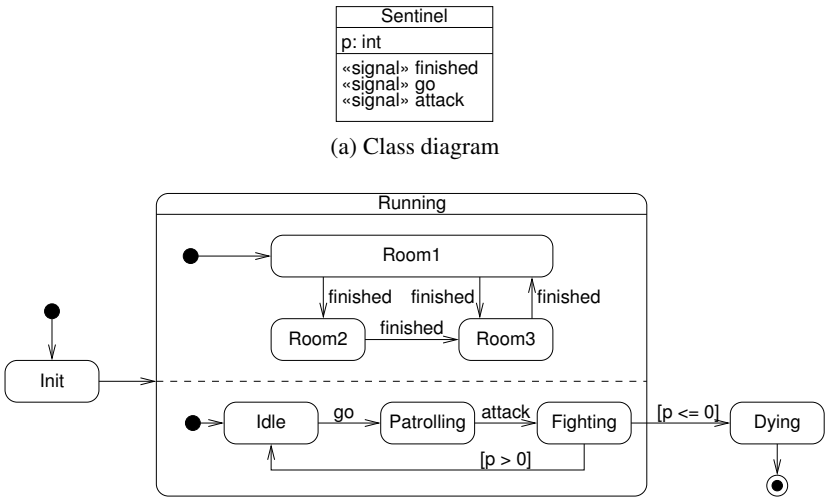


Figure 4.2: History properties

4.2. Examples

We give some examples of modeling with HILA, and describe the concrete syntax and informal semantics of HILA in more detail.

Modeling a system behavior with HILA starts with a base model which is as simple as possible. Figure 4.3b shows a very simple base machine for the sentinel introduced in Sect. 2.2.2. It consists of two orthogonal regions, one describing the location of the sentinel, the other his current action. Note this state machine is "unfinished" in the sense that it only models the very basic behavior of the sentinel.



(a) Class diagram

(b) Base machine

Figure 4.3: Example: modeling the sentinel with aspects

By leaving out a lot of details of the sentinel's behavior, it describes these fundamentals much more concisely than the state machine in Fig. 2.1b where the basic behavior is intertwined with several extensions. Equipped with HiLA, we are no longer forced to pack the complete behavior into one single state machine. Instead, we can (and we should) start with the simplest possible base machine. The class diagram is given in Fig. 4.3a. Note since the base machine is very simple, the class diagram can also get rid of the "unnecessary" properties cr, defeat and oldp that were used in Fig. 2.1b. In other words, using HiLA also helps us separate domain information and properties that are needed for the implementation.

Ideally, the only form of communication among states in the base machine should be, as is in this example, "the activation of the subsequent state via a transition" (p. 14). If modeling a behavior involves a multitude of model elements (vertices, transitions, or even regions), then this behavior ought to be modeled as individual aspects. This way, their modeling is kept at a dedicated place, without tangling with base machine elements. It is this out sourcing that keeps the base machine simple and easy to define.

In the following subsections we show how HiLA can be used to model more complex behaviors modularly. As stated in Chap. 2, our sentinel is supposed to show the following intelligent features (see p. 10):

(1) The sentinel should not be Fighting while he is in Room1.
(2) Changing room is only allowed when the sentinel is Patrolling or Fighting.
(3) When leaving Room1, the sentinel is required to enter the room where it has lost the fewest fights so far.

Recall also that the user should be able to quit the game any time.

4.2.1. Mutual Exclusion. The first feature, which requires the sentinel not to be Fighting while he is in Room1, is actually a special case of a more general

one, i.e. the prohibition of two states contained in different regions from being simultaneously active.

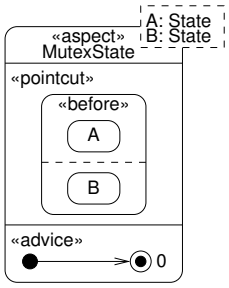


Figure 4.4: Aspect blocking every transition which would lead to an active state configuration containing A or B

This can be achieved by aspects that constrain the interaction between different regions, such as the very simple aspect MutexState given in Fig. 4.4. Its pointcut specifies the join points just before A or B gets active. Its advice tells the base machine not to finish this transition as long as finishing it would lead to a configuration that contains both A and B. (Label 0 means "suspend the transition until proceeding would not result in a state configuration that is matched by the pointcut", see Sect. 4.3.6). Instantiating this template by binding A with Room1 and binding B with Fighting thus models the feature that our sentinel must not become Fighting while he is in Room1. The correctness of this template is verified by model checking, see Sect. 5.8.3.

4.2.2. Synchronization. Synchronization of behaviors that are modeled in concurrent regions is also realized by mutual exclusions. To model feature 2 (changing room is only allowed when the sentinel is Patrolling or Fighting) of the sentinel, we need a slight variation of the MutexState aspect. Instead of inhibiting simultaneous activation of two states, we need in this case to prevent certain transitions from being fired. Figure 4.5 is a template for this purpose. It specifies all transitions where a certain state (in the lower region) C is active both before this transition is fired and after it. The join points specified are therefore those of a room change happening (in the upper region) with C being active. The advice, again, blocks the transition as long as C is active. Modeling feature 2 thus amounts to instantiating this aspect by binding A and B to every pair of states in the upper region of the base machine with $A \neq B$, and binding C to Idle.

4.2.3. Static aspects. HiLA allows the use of static (low-level, transformation) aspects, distinguished by the keywords ≪transformation pointcut≫ and ≪transformation advice≫. The syntax and semantics of static aspects were described in Chap. 3.

Restricting existing transitions in the base machine by imposing an additional guard is currently not supported by HiLA (but can be simulated by rather complex HiLA constructs, though). On the contrary, it is rather simple to model such restrictions with transformation aspects, as shown in Fig. 4.6. The ≪transformation pointcut≫ matches any substructure of the base machine consisting of state Source,

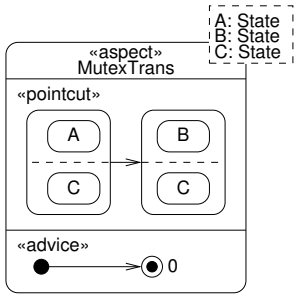


Figure 4.5: Aspect blocking every transition that would lead to a state configuration containing B or C, if the current active state configuration contains both A and C

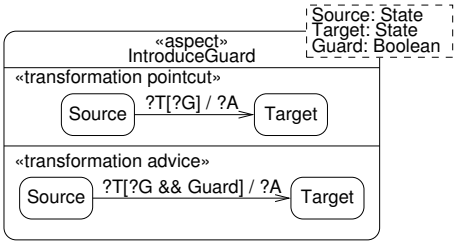


Figure 4.6: Transformation aspect

state Target, and a boolean expression Guard. The trigger, guard, and action of each matched transition is stored in the internal variables ?T, ?G and ?A, respectively. The matched part is replaced by the pattern contained in the ≪transformation advice≫. The result is that the guard of the selected transition is extended by an additional proposition. To model feature 3, we need a combination of (an instance of) Fig. 4.6 and a history property, see below.

4.2.4. History. Feature 3 is an example of history dependent features. HiLA's *history properties* allow us to model such features declaratively and highly modularly.

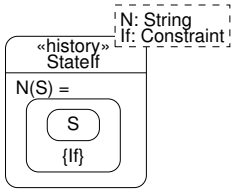


Figure 4.7: History property StateIf counting how often state State has been active with If valued `true`

In Fig. 4.7, a template StateIf of history properties is defined. It counts for each state S how often it has been active, with the constraint If being satisfied. This property is assigned the name N, which is a parameter of the template.

Template	Base	Binding
StateIf	Fig. 4.3b	N ↦a If ↦p@pre > p@post
IntroduceGuard	Fig. 4.3b	Source ↦Room1 Target ↦r ∈ {Room2, Room3} Guard ↦min(a(Succ(Source))) == a(Target)

Figure 4.8: Aspect instances modeling the feature that the sentinel always chooses the room with the least lost fights so far

Feature 3 is then modeled by the combination of an instance of StateIf and an instance of IntroduceGuard (Fig. 4.6), see Fig. 4.8. The instance of StateIf defines a history property a which counts for each state how often a fight was lost (p@pre > p@post) while the state was active.

The advice is introduced by the instance of IntroduceGuard, and makes use of two predefined functions: min takes a set M and yields the minimum element of M, Succ takes a state S and returns all successor states of S, i.e. all targets of transitions starting in S. Therefore min(a(Succ(Room1))) yields the minimum value of a for all successor states of Room1. The transition is thus only enabled when there is no other target room in which the sentinel has lost fewer fights.

Note that even though the transformation itself is described by a low-level aspect, the lucidity of this aspect depends to a large degree on the history property which is a declarative high-level construct.

4.2.5. Additional Events. While ≪begin≫ and ≪after≫ aspects introduced above specify join points of certain transitions being fired, ≪whilst≫ is used to specify join points of certain state constellations being matched by a set of states and a constraint, with a certain current event. In this sense, ≪whilst≫ defines additional reactions of the base machine to events to which originally no reactions were defined. Intuitively, such an aspect introduces new transitions to the base machine and advises them.

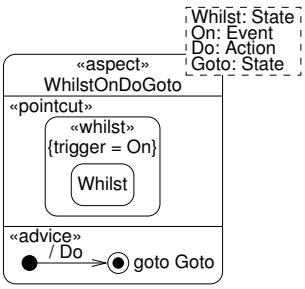


Figure 4.9: Aspect making each state contained in Whilst react to an additional event On by executing the action Do and then going to Goto

Figure 4.9 defines such a template. Its pointcut specifies the join points of any state configuration containing state Whilst being active and On being the current event. The very simple advice first carries out the action Do and then instructs the

base machine to goto state Goto. The feature of the sentinel game that the user should be able to quit any time can thus be simply modeled by instantiating this aspect by binding Whilst with Init and Running, On with event quit, Do with an empty action, and Goto with state Dying. In this example no constraint is defined, the default value true is used.

4.3. Abstract Syntax and Informal Semantics

The abstract syntax of our aspects is defined in a metamodel. Our metamodel is a conservative extension of the UML metamodel [57], which means that new, aspect-oriented concepts are defined as subclasses of UML metaclasses. Therefore our metamodel is "profileable", i.e. it can be mapped to a UML profile, and is thus fully UML compliant.

4.3.1. Aspect. All HILA aspects in this thesis are instances of the metaclass StateMachineAspect, which is a subclass of Aspect and, indirectly, UML NamedElement. It has a Pointcut and an Advice, both of which are defined to be special UML packages, since they are intended to contain other model elements. More particularly, the pointcut of a state machine aspect is called a state machine pointcut, and the advice of a state machine aspect is called a state machine advice (see Fig. 4.10).

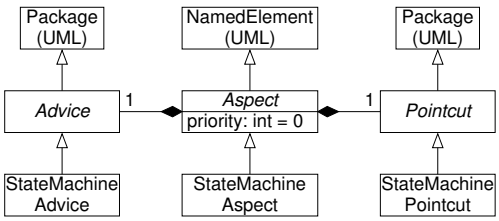


Figure 4.10: Metamodel: aspect

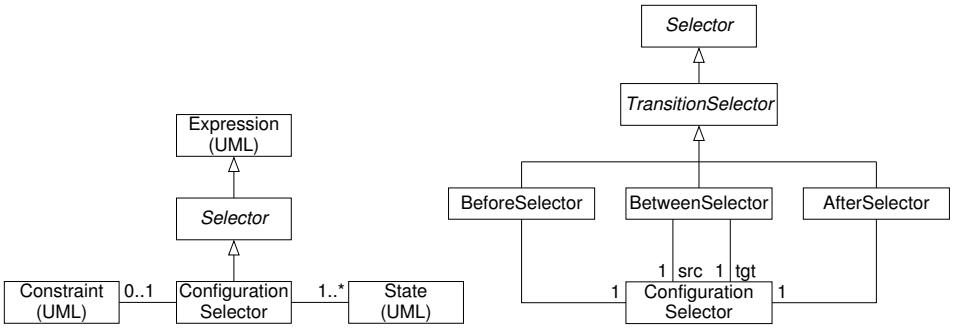
CONSTRAINT 4.1. The advice of a state machine aspect must be a state machine advice, and its pointcut must be a state machine pointcut.

```
context StateMachineAspect
   inv: advice.OCLIsKindOf(StateMachineAdvice) and
        pointcut.OCLIsKindOf(StateMachinePointcut)
```

Each Aspect object, hence each StateMachineAspect object can be assigned an integer priority. Its function will be explained in Sect. 4.3.8.

4.3.2. Configuration Selector. There are two auxiliary language constructs that we will need to construct our pointcuts. In this subsection, we define *configuration selector*s. The other, *transition selector*s, are defined in Sect. 4.3.3. Both selectors are subsumed under an abstract metaclass Selector, which is itself a subclass of UML Expression.

A configuration selector (see Fig. 4.11a) contains a set of states, and an optional constraint. It is used to specify constellations of state machines. It is presented as a rectangle with rounded corners, which contains states of the base machine,



(a) Configuration selector (b) Transition selector

Figure 4.11: Metamodel: selectors

separated by dashed lines. The constraint, if any, is given in curly braces. A constellation of the base machine is said to be *matched* by a configuration selector iff. all states contained in the selector are active and the constraint, if any, is satisfied.

CONSTRAINT 4.2. The states contained in a configuration selector must neither belong to the same region, nor contain each other.

```
context ConfigurationSelector
   inv: state -> forAll(s1, s2 | s1 <> s2 implies
                    s1.container <> s2.container and
                    not s1.contains(s2) and
                    not s2.contains(s1))
```

where `contains(b)` is a boolean function for states. It returns `b` is (directly or recursively) contained in the state:

```
context State
   def: contains(b): Boolean =
        if region -> size() = 0 then False
        else region -> exists(subvertex ->
               select(oclIsKindOf(State)) ->
                  exist(oclAsType(State).contains(b)))
```

4.3.3. Transition Selector. A *transition selector* is also a selector, as shown in Fig. 4.11b. A transition selector may be either a *before selector*, a *between selector*, or an *after selector*. A before selector contains one configuration descriptor, and is represented by a stereotype ≪before≫ within the configuration descriptor. It selects all the transitions of the base machine that would lead to a configuration containing any state contained in the configuration descriptor. An after selector also contains a configuration descriptor, and is represented by a stereotype ≪after≫ within the configuration descriptor. It selects all the transitions of the base machine that should be fired when any state configuration matched by the configuration descriptor has just become inactive. A between selector contains two configuration selectors, called src and tgt, and is represented by an arrow from src to tgt. It selects all the transitions of the base machine that should be fired when

any state constellation matched by src has just been active and that would lead to any state contained in tgt getting active.

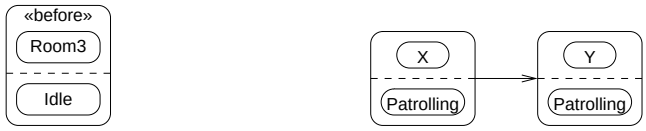
Recall that in HiLA, future valuations of environment variables are not considered in the pointcut (see Sect. 4.1.1). This is reflexed in the following constraints.

CONSTRAINT 4.3. The configuration selector of a before selector does not contain a constraint.

```
context BeforeSelector
   inv: config.constraint -> isEmpty()
```

CONSTRAINT 4.4. The target configuration selector of a between selector does not contain a constraint.

```
context BetweenSelector
   inv: tgt.constraint -> isEmpty()
```

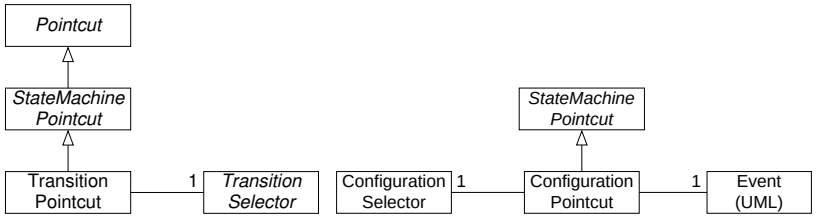


(a) Before selector (b) Between selector

Figure 4.12: Example: transition selectors

EXAMPLE 4.5 (Before selector). The before transition selector (sterotype ≪before≫) in Fig. 4.12a contains a configuration descriptor, which specifies all state configurations that contain Room3 or Idle. The transition selector thus selects all transitions leading to such configurations.

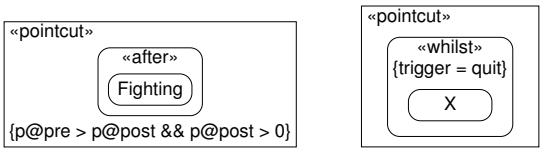
EXAMPLE 4.6 (Between selector). The between transition selector in Fig. 4.12b contains two configuration descriptors. It selects all transitions that are fired in a state configuration containing X and Patrolling, and would lead to a state configuration which contains Y or Patrolling.



(a) Transition pointcut (b) Configuration pointcut

Figure 4.13: Metamodel: pointcuts

4.3.4. Transition pointcut. In accordance to the two kinds of selectors, there are also two kinds of state machine pointcuts defined. In this subsection, we introduce *transition pointcut*. The other kind, *configuration pointcut*, is described in Sect. 4.3.5.



(a) Transition pointcut (b) Configuration pointcut

Figure 4.14: Example: pointcut

A transition pointcut contains a transition selector, see Fig. 4.13a. It specifies the points of time the transitions selected by its transition selector being fired, after the effect, if any, is executed.

EXAMPLE 4.7 (Transition pointcut). The pointcut in Fig. 4.14a contains a transition selector and an OCL constraint. It specifies the points of time in the base machine execution of any transition being fired just after any state configuration containing Fighting, where the value of the (contextual) property p has decreased while the Fighting was active (p@pre $>$ p@post) but stays positive (p@post > 0).

4.3.5. Configuration Pointcut. A configuration pointcut contains a configuration selector and a UML event (see Fig. 4.13b). It is distinguished by its configuration selector, presented by stereotype ≪whilst≫, along with the event, presented as tagged value trigger. A configuration pointcut cp specifies such join points that the current constellation of the base machine is matched by the selector, i.e. the active state configuration contains all the states of cp's states, cp's constraint, if any, is satisfied, and cp's event is the current event.

Intuitively, a configuration pointcut introduces new transitions to the base machine and selects the transitions.

EXAMPLE 4.8 (Configuration pointcut). Figure 4.14b shows a configuration pointcut. It specifies the points of time in the base machine's execution that quit is the current event to handle while state X is active. Since no explicit constraint is given, true is used.

4.3.6. Advice. A state machine aspect contains an advice, which is an *advice state machine*, a special kind of UML state machine (see Fig. 4.15). The advice may access to the properties of the base machine, and may have its own properties as well. This state machine is executed at points of time specified by the pointcut. The control flow returns to the base machine when the advice is finished, i.e. when a final state is arrived. Intuitively, an advice can be perceived as an extension of the transitions selected by the pointcut: the advised transition is interrupted, the advice executed, and then the base machine is resumed.

The difference between an advice state machine and a standard UML state machine is that in the former, final states contains a label, which is a set of states, and a constraint. We call this kind of final states *labeled final states*, see Fig. 4.16.

The label is a set of states, and is represented in the concrete syntax as goto X, X being the states. The label and the constraint are used to specify how to resume the execution of the base machine. When a final state with label goto X and constraint C is reached, the execution of the advice is finished, the base machine

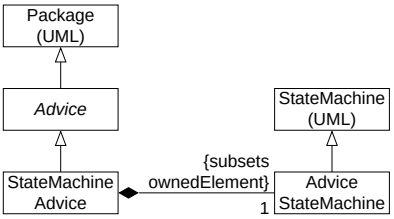


Figure 4.15: Metamodel: advice

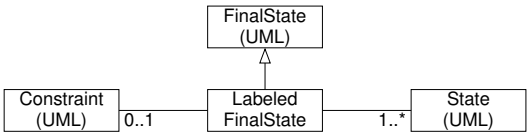


Figure 4.16: Metamodel: labeled final state

continues execution by activating the states contained in X, if C is satisfied. There are two generic labels predefined: src and tgt represent the source and the target of the advised transition, respectively. We require for each pair of of states contained in a label that they neither belong to the same region nor contain each other:

CONSTRAINT 4.9. The states contained in a label must neither belong to the same region, nor contain each other:

```
context LabeledFinalState
   inv: state -> forAll(s1, s2 | s1 <> s2 implies
                    s1.container <> s2.container and
                    not s1.contains(s2) and
                    not s2.contains(s1))
```

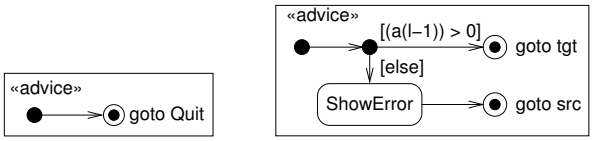
In addition, two generic constraints, 1 and 0, are also defined. Their semantics depends on the type of the pointcut. In a ≪before≫, ≪after≫, or ≪whilst≫ aspect, 1 is satisfied iff. activating the states contained in the label would result in a state configuration that matches the pointcut, and 0 is satisfied iff. 1 is not satisfied. In a transition aspect, the meaning of 1 and 0 is the same, except that instead of "the pointcut", it is now the tgt configuration of the pointcut to match.

Note that keyword 0 must not appear in an ≪after≫ or a ≪whilst≫ aspect. Since the configuration specified by the pointcut *has been already* active, there is nothing left to prevent from.

CONSTRAINT 4.10. All final states in an advice state machine are labeled final states.

```
context AdviceStateMachine
   inv: top.subvertex -> forAll (f |
          f.oclIsKindOf(FinalState)
          implies f.oclIsTypeOf(LabeledFinalState))
```

EXAMPLE 4.11. (Single exit) The advice in Fig. 4.17a does nothing but simply tells the base machine to change its control flow by "going to" a state Quit, which



(a) Single exit (b) Multiple exits

Figure 4.17: Example: advice

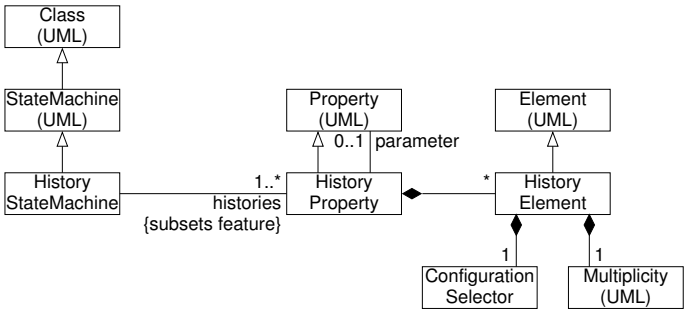


Figure 4.18: Metamodel: history

may or may not be the source or target of the advised transition. Since no explicit constraint is given, the default one, true, is used.

EXAMPLE 4.12. (Multiple final states) As in regular state machines, an advice may also have multiple final states. In Fig. 4.17b, the advice defines two possible exits. If the value of the variable a(l-1) is greater than zero, then the base machine should goto the target of the advised transition (final state labeled goto tgt), otherwise some error message should be shown (ShowError), and the base machine go back to the source of the advised transition (goto src). Again, no explicit constraint is given, the default is true.

4.3.7. History properties. In order to model history-based behavior modularly, a HILA aspect may contain a ≪history≫ compartment, in which *history properties* are defined. A history property contains one or more *history element*s, each of the history elements consisting of a configuration selector, a *multiplicity*. Moreover, history properties is defined as an array, if a parameter is given. The metamodel is given in Fig. 4.18.

CONSTRAINT 4.13. The parameter, if any, of a history property must not be a history property itself.

```
context HistoryProperty
   inv: not parameter.oclIsKindOf(HistoryProperty)
```

In the concrete syntax, each history element is represented by the configuration selector and the multiplicity, given in a pair of brackets. If a parameter is defined, it is specified by the keyword for, followed by the name of the parameter and the domain of the parameter. The history elements are separated by a line. The elements are used to match contiguous subsequences of the base machine's

execution history: a contiguous subsequence q of the execution history is matched by a history element, consisting of a configuration selector s and a multiplicity m, iff. q contains m state configurations that are matched by s. The value of a history property is the number of matches.

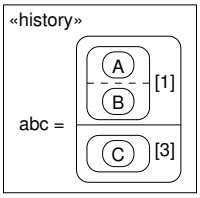
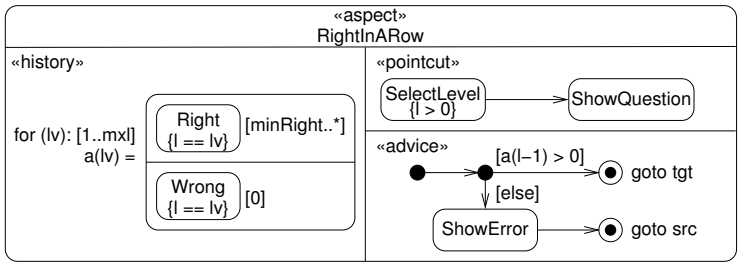
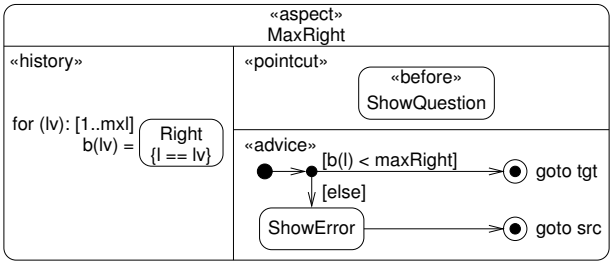


Figure 4.19: Example: history property

EXAMPLE 4.14 (History property). Figure 4.19 defines a history property abc. Its value is the number of the contiguous subsequences of the execution history that contain one (multiplicity [1]) state configuration including both state A and state B, as well as three ([3]) state configurations containing state C.



(a) *minRt*



(b) *maxRt*

Figure 4.20: Aspects for the elearning system (base machine: Fig. 2.3a)

To model the history-based features *minRt* and *maxRt* (p. 16) of the e-learning system, we use Fig. 2.3a as the base machine—which is also very simple in the sense that "the only form of communication among states is like in this example the activation of the subsequent state via a transition"—and define in Fig. 4.20 two aspects.[1]

[1] It is not a coincidence that we define two aspects to model two features!

EXAMPLE 4.15 (Aspect-oriented modeling of the *minRt* feature of the e-learning system). In Fig. 4.20a, the history property a is declared as an array of integers (keyword for), and defines for each positive natural number (parameter lv, domain [1..mxl], mxl being the highest possible level) if at this level (constraint {l == lv}), a contiguous subsequence of the execution history can be found in which Right has been at least minRight times active (multiplicity [minRight..*]) and Wrong has been zero times active (multiplicity [0]).

The pointcut selects all transitions from SelectedLevel to ShowQuestion when the current value of l is greater than zero. Actually, there is only one such transition in the base machine. When it is advised, the advice checks if the value for the level $l - 1$ stored in a is greater than 0. If yes, which means that the user did actually answer at least minRight questions correctly in a row on level $l - 1$, the machine should proceed as usual (label tgt), otherwise the advised transition is interrupted, and its source (label src) ShowQuestion activated. This way, the feature *minRt* of the e-learning system is modeled separately in an aspect. The modeling is declarative, its algorithmic implementation will be realized automatically in the weaving process. Therefore, the chance of the modeler making mistakes like in Fig. 2.3b is minimized.

EXAMPLE 4.16 (Aspect-oriented modeling of the *maxRt* feature of the e-learning system). In Fig. 4.20b, the history property b defines an array of integers, storing for each parameter lv how often Right has been active on this level. Every time when ShowQuestion is just about to become active, the advice is activated. If at level l Right has not been active too often ([b(l) $<$ maxRight]) yet, the transition is allowed, otherwise an error is shown, and the source of the advised target is activated again.

4.3.8. Priorities. As shown in Fig. 4.10, each state machine aspect may have a priority, which is an integer. The priority is used to define resolution strategies when a so called *resumption conflict* between aspects arises. In the e-learning example, the two aspects we defined, *minRt* and *maxRt*, may be actually conflicting: when the transition from SelectLevel to ShowQuestion is just about to be fired, the preconditions of both aspects are satisfied: the base machine is about to change from SelectLevel over to ShowQuestion, and ShowQuestion is just about to become active. Therefore, both aspects are executed. A conflict is therefore possible since the two aspects may define two different states to activate when the execution of the aspects are finished. For instance, if for a given l it holds that a(l-1) $>$ 0 but b(l) $<$ maxRight does not, then aspect RightInARow would allow the advised transition to be finished (goto tgt) while MaxRight would not.

When aspects are assigned priorities, the one with the highest priority rules the others. If no priority is given, the default value is the lowest possible priority, 0. If several aspects have the same (highest) priority, they have equal right: different resumption strategies indicated by different aspects would be considered as an error, the state machine would enter an "error" state.

Therefore, equally prioritized aspects are combined in an "and" relation: only when all of the aspects indicate the same resumption state, it is activated after the aspects' execution; if instead different resumption states are indicated, some error state is activated to indicate the conflict. On the other hand, aspects with different priorities are combined in an (asymmetric) "or" relation: as soon as the aspect with the highest priority defines explicitly a resumption state, it is activated; if not (recall

labels are optional, see Sect. 4.3.6), the resumption strategies indicated by the next highest prioritized aspects may be used.

EXAMPLE 4.17 (Equal priorities). In Fig. 4.20, no explicit priority is given, both of the aspects are default-prioritized. Therefore, when both of them are executed (just before the transition from SelectLevel to ShowQuestion is fired), their resumption strategies have equal rights. At runtime, when the execution of the aspects is finished and the same resumption state is indicated by both of them, then it is activated, otherwise the state machine would enter an error state.

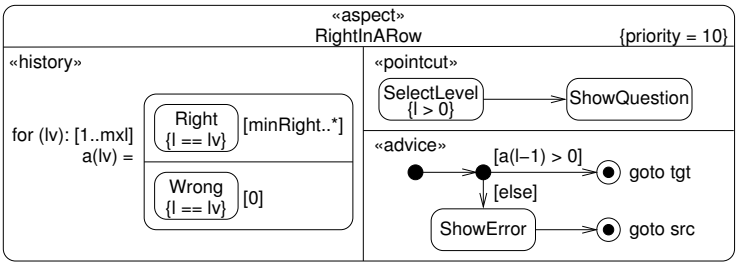


Figure 4.21: Example: priorities of aspects

EXAMPLE 4.18 (Different priorities). Suppose now aspect RightInARow was assigned a priority 10 (tagged value priority), as shown in Fig. 4.21. When used together with aspect MaxRight, this aspect is higher prioritized, since the implicit (default) priority of MaxRight is lower than any explicit priority. Therefore, only the resumption state of RightInARow is used at runtime, the resumption state of MaxRight will not be consulted.

Note that entering an error state when resumption states are conflicting is not necessarily the best choice for every software system. An extension of HILA to allow the modeler to define another behavior to be executed in case of conflicting resumption states is part of our future work.

4.4. Big Picture

Summarizing, Fig. 4.22 shows a big picture, containing the metaclasses introduced in the proceeding subsections and their relationships.

4.5. Discussion

HILA is a powerful language. By powerful, we do not mean it is very expressive. In fact, as stated earlier, each HILA aspect can also be "implemented" by a static, transformation aspect. HILA is powerful, because it is high-level, i.e. HILA aspects define directly modifications of the base machine's behavior, as opposed to indirectly via modifications of its syntax. This way, HILA aspects are more intuitive and less error-prone than static aspects.[2] For example, in Fig. 2.3b, the history property declaratively defines what should be counted: sequences containing only

[2]It is noteworthy that by separate modeling parts of the system behavior some overhead is produced for the understanding of the system, since the modeler now has to take care of a multitude of

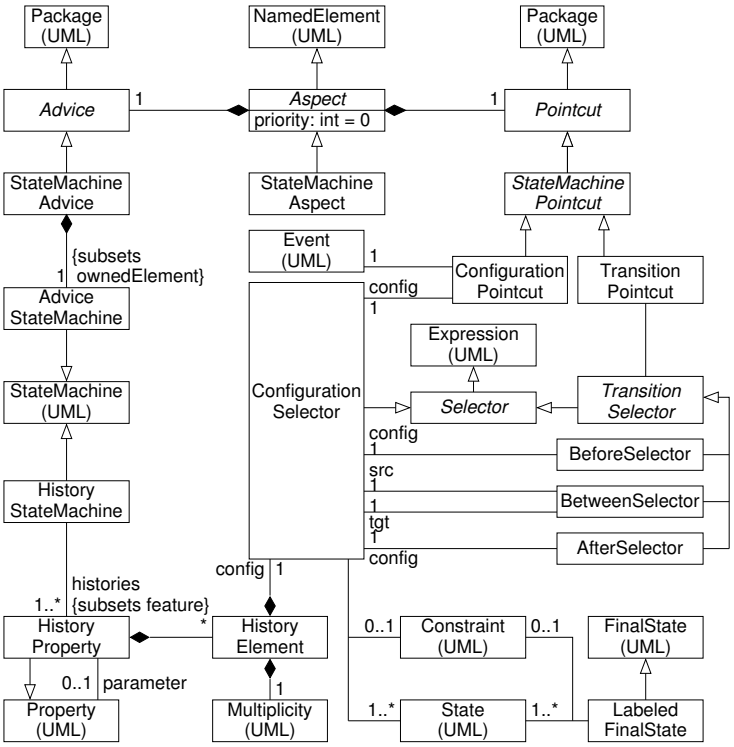


Figure 4.22: Metamodel: the big picture

Right and no Wrong. The whole imperative modeling as is required when using plain UML or static aspects is no longer necessary, the modeler does not even have a chance to make the same mistake as in Fig. 2.3b.

Feature coordination is also simpler in HiLA than is in the case of plain UML or static aspects, since in HiLA, it is not necessary for the modeler to design the coordination explicitly, instead, coordination, as well as implementing the highly declarative aspects of HiLA, is to be taken care of by the automatic weaving process, which is discussed in the following chapter.

aspects instead of only one single state machine. Moreover, although HiLA defines high-level, easy-to-understand constructs, the gotos may obscure the flow control. Another problem is interactions between the aspects, see Chap. 7. However, we believe that overall, HiLA can be considered helpful for reducing the complexity of state machines.

CHAPTER 5

Weaving

Contents

5.1. Main Idea **46**
5.2. Prerequisites **47**
5.2.1. Abstract syntax 47
5.2.2. Action language 48
5.2.3. Translation language 48
5.2.4. Auxiliary functions 49
5.2.5. Derived properties 50
5.3. Preprocessing **51**
5.3.1. Removing junctions 52
5.3.2. Removing target unstructured transitions 52
5.3.3. Removing fork vertices 54
5.3.4. Creating “before” section 56
5.3.5. Insert trace variables 57
5.3.6. Inserting Error States 58
5.3.7. Summary 58
5.4. Weaving History Properties **60**
5.4.1. Structure of the NFA 60
5.4.2. Simulating the NFA 61
5.4.3. Counting 62
5.5. Weaving Aspects **63**
5.5.1. aspect2Region 63
5.5.2. Weaving Configuration Aspects 67
5.5.3. Weaving Transition Aspects 67
5.6. Postprocessing **69**
5.7. Correctness **71**
5.8. Implementation **72**
5.8.1. Hugo/RT 72
5.8.2. Weaving 72
5.8.3. Example 73
5.9. Discussion **77**

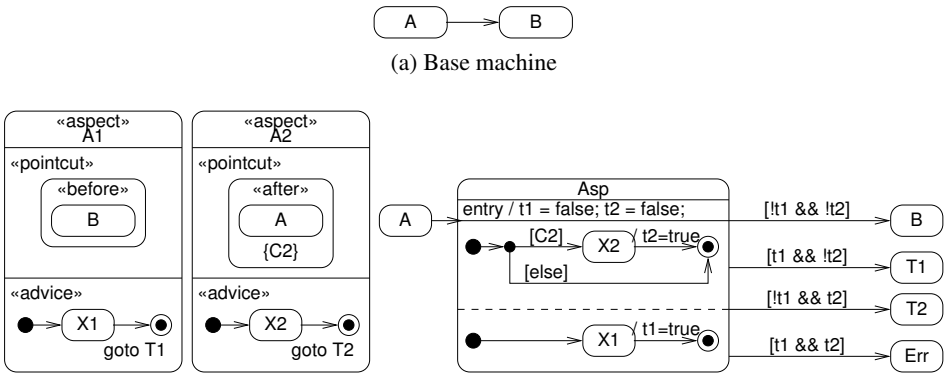
Weaving is the process of composing the base machine with the aspects. The weaving process modifies the base machine to include the behaviors modeled in the aspects, so that they are executed as the points of time specified by the pointcuts. It is the elaborate weaving process that implements the declarative HILA aspects by means of the more imperative elements of plain UML state machines.

As described in Sect. 3.3, static (transformation) aspects are woven by encoding the aspects in a graph grammar system. In this chapter, we present the weaving algorithms of dynamic HILA aspects.

Publication Notice. Section 5.4.1 was first published in [83].

5.1. Main Idea

HILA's aspects are woven by inserting the advice into transitions of the base machine. Put very simply, we implement "≪before≫ X" aspects by intercepting every transition leading to X, activating the advice of the aspect, and then proceeding to X; "≪after≫ X" is implemented by inserting the advice into every transition leaving X; and "≪whilst≫ X" by introducing a (self) transition both leaving and entering X, and then intercepting this transition. More details of how to determine the transitions to intercept and how to intercept them will be given later on.



(a) Base machine

(b) Aspects

(c) Result

Figure 5.1: Weaving: basic idea

A challenge to the weaving algorithm is posed by the so called "shared join points". That is, if one join point is selected by several pointcuts, there may arise some conflict.

For instance, consider the base machine given in Fig. 5.1a and the two aspects given in Fig. 5.1b. Whenever state B is just about to become active, aspect A1 activates state X1 and then instructs the base machine to go to T1. Whenever state A has just got inactive, aspect A2 activates state X2 and then tells the base machine to go to T2. Now what if state A has just become inactive and state B is just about to become active, as is the case in Fig. 5.1a? How to ensure that both aspects are executed? After the execution of the aspects, where should the base machine go to, T1 or T2?

In HILA, our answer to this problem is to exploit the concurrent nature of UML state machines, and weave aspects with shared pointcuts into different regions of one orthogonal state, which we call Asp. Thus, the aspects are executed simultaneously at runtime. A shared array T of boolean variables is also introduced to Asp. We refer to the elements of T as *resumption variables*, since they indicate how to resume the execution when the aspects are finished. The regions (the aspects) store in (elements of) T the states that should be activated after the execution of the aspect. For each possible target t to go to there is a resumption variable $\mathsf{T}_\mathsf{t} \in \mathsf{T}$, initialized with false. Aspect a instructing the base machine to go to state t is then implemented as an assignment $\mathsf{T}_\mathsf{t} \leftarrow \mathtt{true}$. Note this is also the only assignment that can be applied to T_t. This way, race conditions w.r.t. assignments to resumption variables are avoided, since for a given T_t, no matter how many such

assignments are made, no matter in which order they are made, the overall result is always $\mathsf{T_t}$ being set to true. After the execution of Asp, if more than one of the resumption variables have the value true, then a conflict has been detected, and an alarm is raised (state Err), otherwise, the transition corresponding to the target to go to will be activated. If none of the resumption variables is true, the base machine simply resumes the advised transition, i.e. activates its target to continue the execution.

The result of applying this idea to the above example is shown in Fig. 5.1c. The resumption variables are called t1 and t2. If both of them are true after the execution of Asp, state Err is activated. Otherwise, the base machine resumes according to the (only) variable that is actually true. If none of them is true, the default resumption strategy is followed, and the target of the advised transition, in this case state B, is activated. In the above example, however, this is not a real option, since in this concrete example, when Asp is active, in the lower region t1 will always be set to be true.

Note that activating state Err is the default, and not the only strategy for conflict handling. In fact, HILA allows the modeler to define priorities of aspects (see Sect. 4.3.8). We actually use the resumption strategy specified by the aspect with the highest priority of all conflicting aspects (not shown here, see Sect. 5.6 for details).

The actual implementation of the basic idea is shown in Algorithm 5.1: for a base machine SM, a set of history properties $\mathcal{H}$, sets of ≪whilst≫, ≪before≫, ≪after≫ and transition aspects W, B, A and M, we first preprocess SM to transform it to a canonical form, then we weave the $\mathcal{H}$, W, B, A and M in this order, and then wrap up the weaving in a postprocessing step. Note SM is modified in each of these steps.

Algorithm 5.1 Weaving

```
1: procedure WEAVING(SM, H, W, B, A, M)
2:     PREPROCESSING(SM)
3:     WEAVEHISTORYPROPERTIES(SM, H)
4:     WEAVECONFIGURATIONASPECTS(SM, W)
5:     WEAVETRANSITIONASPECTS(SM, B, A, M)
6:     POSTPROCESSING(SM)
7: end procedure
```

In the following, we first give the prerequisites of our weaving, and then describe the steps of the above algorithm in more detail.

5.2. Prerequisites

5.2.1. Abstract syntax. The abstract syntax of UML state machines and HILA aspects are defined by the metamodels in Fig. 2.2 and Fig. 4.22, respectively.

We refer to property p of object o as $p(o)$. If the name of a property is not explicitly given, we follow the common UML convention and use, independently of the multiplicity, the lower-cased name of its class as the property name.

Algorithm 5.2 Deep clone of states and regions

```
1:  function cloneVertex(v)
2:      v' ← clone(v)
3:      if isState(v) then
4:          for all r ∈ region(v) do
5:              region(v') ← region(v') ∪ {cloneRegion(r)}
6:          end for
7:      end if
8:      if class(v) = labeledfinal then
9:          label(v') ← clone(label(v))
10:     end if
11:     return v'
12: end function
13: function cloneRegion(r)
14:     r' ← new Region
15:     for all v ∈ subvertex(r) do
16:         subvertex(r') ← subvertex(r') ∪ {cloneVertex(v)}
17:     end for
18:     return r'
19: end function
```

5.2.2. Action language. For the target platform of our weaving, UML state machines, we assume an expression language Exp that includes at least boolean expressions (like true, false, $e_1 \wedge e_2$, etc.) and integer arithmetics. We further assume an action language Act based on Exp that at least includes a skip statement, assignments ($\leftarrow$), a case distinction (if... then ... else), a for (and a while) construct, and sequential concatenation of statements ($\cdot$). Variables of this language may be either boolean or integer. Exp and Act are used to describe the guards and the effects of the transitions, as well as the entry, exit, and do activities of the states. Furthermore, we assume a set of events $Event$ which includes $*$ denoting a completion event.

5.2.3. Translation language. The weaving algorithms are language independent, i.e. they can be implemented in every object-oriented programming language in which the following functions are implemented

- *Class function*: function $\mathrm{class}(o)$ returns for each object o its class in the sense of common object-oriented programming.
- *Constructors*: for each concrete metaclass C in Fig. 2.2 the constructor call new C returns a new instance of the class. After the constructor call, the single properties of the new instance are initialized with $\perp$, multiple properties with $\emptyset$. There are two shortcuts defined:
  - new Transition(r, g, e) returns a new Transition object T such that trigger$(T) = r$, guard$(T) = g$, and effect$(T) = e$.
  - new Transition(s, t, r, g, e), just as the three-parameter edition, additionally such that source$(T) = s$ and target$(T) = t$. After the constructor call, the properties incoming and outgoing of t and s, respectively, are set correctly, i.e. $T \in$ incoming(t) and $T \in$ outgoing(s).

- *Clone functions*: in addition to function $\mathrm{clone}(e)$, which returns a "shallow clone" of e as in object-oriented programming languages, we define functions $\mathrm{cloneVertex}(v)$ and $\mathrm{cloneRegion}(r)$ to return a "deep clone" of vertex v and region r, respectively (see Algorithm 5.2). We assume a function $\mathrm{clone}^{-1}(c)$ which returns the origin o of object c, if c was created as a clone of o. If c was not created as a clone, we write $\mathrm{clone}^{-1}(c) = \bot$.
- *Generators of action language elements*: we assume the following functions to generate elements of Act:
  - newVar, every invocation of $\mathrm{newVar}()$ returns a fresh variable in Exp.
  - Function $\mathrm{toUML}(a)$, which represents the implementation of a sequential algorithm a, consisting of assignments, case distinctions, for (or while) loops with boolean or integer variables or sets of booleans or integers, in Act.

5.2.4. Auxiliary functions. For convenience, we also define some auxiliary functions to be used in our weaving algorithms.

- For each vertex v we define
  - $\mathrm{isState}(v) := \mathrm{class}(v) \in \{\mathsf{State}, \mathsf{FinalState}\}$
  - $\mathrm{isSimple}(v) := \mathrm{isState}(v) \wedge \#\mathsf{region}(v) = 0$
  - $\mathrm{isComposite}(v) := \mathrm{isState}(v) \wedge \#\mathsf{region}(v) > 0$
  - $\mathrm{isFinal}(v) := \mathrm{class}(v) = \mathsf{FinalState}$
  - $\mathrm{isJoin}(v) := \mathrm{class}(v) = \mathsf{PseudoState} \wedge \mathit{kind}(v) = \mathsf{join}$
  - $\mathrm{isJunction}(v) := \mathrm{class}(v) = \mathsf{PseudoState} \wedge \mathit{kind}(v) = \mathsf{junction}$
  - $\mathrm{isFork}(v) := \mathrm{class}(v) = \mathsf{PseudoState} \wedge \mathit{kind}(v) = \mathsf{fork}$
  - $\mathrm{isInitial}(v) := \mathrm{class}(v) = \mathsf{PseudoState} \wedge \mathit{kind}(v) = \mathsf{initial}$
- For each set S we write
  - $\mathrm{add}(S, e)$ to mean $S \leftarrow S \cup \{e\}$,
  - $\mathrm{remove}(S, e)$ to mean $S \leftarrow S \setminus \{e\}$,
  - $\mathrm{remove}(S, E)$ to mean $S \leftarrow S \setminus E$.
- For each region r we define $\mathsf{subvertex}^+(r)$ to be the transitive closure of $\mathsf{subvertex}(r)$ (see Fig. 2.2).
- For each state s, $\mathrm{isComposite}(s)$, we define
  - $\mathsf{subvertex}(s) := \bigcup_{r \in \mathsf{region}(s)} \mathsf{subvertex}(r)$
  - $\mathsf{subvertex}^*(s) := \bigcup_{r \in \mathsf{region}(s)} \mathsf{subvertex}^+(r)$
  - $\mathsf{subvertex}^+(s) := \mathsf{subvertex}^*(s) \cup \{s\}$.
- For each simple state s, we define $\mathsf{subvertex}^*(s) := \{s\}$.
- For each region r we define $\mathrm{initial}(r) \in \mathsf{subvertex}(r)$ to mean the (only one) initial vertex. If there is no initial vertex contained in r, we write $\mathrm{initial}(r) = \bot$. For a state machine SM, we write $\mathrm{initial}(r)$ to mean $\mathrm{initial}(\mathsf{top}(\mathsf{SM}))$.
- For a state machine SM, we define

$$\mathsf{V}(\mathsf{SM}) := \mathsf{subvertex}^+(\mathsf{top}(\mathsf{SM}))$$
$$\mathrm{S}(\mathsf{SM}) := \{v \in \mathsf{V}(\mathsf{SM}) \mid \mathrm{isState}(v)\}$$
$$\mathrm{S}_{\mathsf{simple}}(\mathsf{SM}) := \{s \mid s \in \mathrm{S}(\mathsf{SM}) \wedge \mathrm{isSimple}(s)\}$$
$$\mathrm{S}_{\mathsf{comp}}(\mathsf{SM}) := \mathrm{S}(\mathsf{SM}) \setminus \mathrm{S}_{\mathsf{simple}}(\mathsf{SM}).$$

- For a state machine SM, we define
$$\mathsf{J_c}(\mathsf{SM}) := \{v \in \mathsf{V}(\mathsf{SM}) \mid \mathrm{isJunction}(v)\}$$
$$\mathsf{J_n}(\mathsf{SM}) := \{v \in \mathsf{V}(\mathsf{SM}) \mid \mathrm{isJoin}(v)\}$$
$$\mathsf{F_k}(\mathsf{SM}) := \{v \in \mathsf{V}(\mathsf{SM}) \mid \mathrm{isFork}(v)\}$$
$$\mathsf{I}(\mathsf{SM}) := \{v \in \mathsf{V}(\mathsf{SM}) \mid \mathrm{isInitial}(v)\}$$
$$\mathrm{T}(\mathsf{SM}) := \bigcup_{v \in \mathsf{V}(\mathsf{SM})} (\mathsf{incoming}(v) \cup \mathsf{outgoing}(v)).$$
- For two vertices $v, v' \in \mathsf{V}(\mathsf{SM})$, we define $\mathrm{LCA}(v, v')$ to be the *least common ancestor* of v and v', i.e. $\mathrm{LCA}(v, v')$ is the region r, such that (1) $(v, v') \subseteq \mathsf{subvertex}^+(r)$, and (2) for each $r' \neq r$, $(v, v') \subseteq \mathsf{subvertex}^+(r')$ implies $\mathsf{subvertex}^+(r) \subset \mathsf{subvertex}^+(r')$.
- For a state machine SM, we define $\mathrm{removeTransition}(\mathsf{SM}, t)$ to remove the transition t from SM. If we assume in our programming language explicit garbage collection is not necessary (such as in the case of Java), this can be simply implemented by removing t from the set of outgoing transitions of t's source and from the set of incoming transitions of t's target (see also Sect. 5.2.5), defined as $\mathrm{remove}(\mathsf{outgoing}(\mathsf{source}(t)), t)$; $\mathrm{remove}(\mathsf{incoming}(\mathsf{target}(t)), t)$. We also define $\mathrm{removeTransition}(T)$ to remove all transitions contained in T.
- For a state machine SM, we define $\mathrm{removeVertex}(\mathsf{SM}, v)$ to remove the vertex v from SM. If we assume in our programming language explicit garbage collection is not necessary (such as in the case of Java), this can be simply implemented by removing v from the region containing v, defined as $\mathrm{remove}(\mathsf{subvertex}(\mathsf{container}(v)), v)$ (see also Sect. 5.2.5). We further define $\mathrm{removeVertex}(V)$ to remove all vertices contained in V.
- For each region r, class C which is a subclass of Vertex, and a variable v, we write $v = \mathrm{insertVertex}(r, \mathsf{C})$ for the combination of creating a new object of class C and inserting it into region r, defined as $v \leftarrow \mathrm{new}\ \mathsf{C}$; $\mathrm{add}(\mathsf{subvertex}(r), v)$. The return value may also be omitted.
- For each region r, pseudostate kind k, and and a variable v, we write $v = \mathrm{insertVertex}(r, k)$ for the combination of creating a pseudostate of the kind k and inserting it into region r, defined as $v \leftarrow \mathrm{new}\ \mathsf{Pseudostate}$; $\mathit{kind}(v) \leftarrow k$; $\mathrm{add}(\mathsf{subvertex}(r), v)$. The return value may also be omitted.
- For vertices s, s', trigger t, constraint g, and action a, we also refer to $\mathrm{new}\ \mathsf{Transition}(s, s', t, g, a)$ as $\mathrm{insertTransition}(s, s', t, g, a)$.
- In the UML, an initial vertex *init* has exactly one outgoing transition[1]. We call this transition the *initial transition*. We refer to the initial transition of a region r as $\mathrm{inTr(r)}$, if r contains an initial vertex. For a state machine SM we define $\mathrm{inTr}(\mathsf{SM}) := \mathrm{inTr}(\mathsf{top}(\mathsf{SM}))$.

5.2.5. Derived properties. To make it easy to keep the state machine consistent, we regard some properties defined in the metamodel as "derived", i.e. their

[1] Actually, the UML Specification requires only that "an initial vertex can have at most one outgoing transition" [57, p. 541]. Since an initial vertex without outgoing transitions would not make any sense, we conclude that an initial vertex must have exactly one outgoing transition.

values are derived from other properties, and cannot be assigned directly. The properties and their derivation are given in the following:

- target and source of Transition. Given a state machine SM and a transition t, $\mathsf{target}(t)$ is calculated as the (unique) vertex $v \in \mathsf{V(SM)}$ such that $t \in \mathsf{incoming}(v)$, $\mathsf{source}(t)$ as the (unique) vertex $v \in \mathsf{V(SM)}$ such that $t \in \mathsf{outgoing}(v)$.
- container of Vertex. Given a vertex v, its $\mathsf{container}(v)$ is calculated as the (unique) region $r \in \bigcup_{s \in \mathrm{S(SM)}} \mathsf{region}(s)$ such that $v \in \mathsf{subvertex}(r)$.

The uniqueness of the source and the target of a transition t is provided by restriction of writing access of the vertices: throughout our algorithms, we avoid setting their values by assignment, but use instead the functions $\mathrm{setTarget}(t, v)$ and $\mathrm{setSource}(t, v)$, which are defined in the following, to set the target and source, respectively, of t to vertex v.

procedure SETTARGET(t, v)
  $v' \leftarrow \mathsf{target}(t)$
  $\mathsf{incoming}(v') \leftarrow \mathsf{incoming}(v') \setminus \{t\}$
  $\mathsf{incoming}(v) \leftarrow \mathsf{incoming}(v) \cup \{t\}$
end procedure

procedure SETSOURCE(t, v)
  $v' \leftarrow \mathsf{source}(t)$
  $\mathsf{outgoing}(v') \leftarrow \mathsf{outgoing}(v') \setminus \{t\}$
  $\mathsf{outgoing}(v) \leftarrow \mathsf{outgoing}(v) \cup \{t\}$
end procedure

Similarly, we do not set the container property of a vertex directly, but indirectly by modification of the set of subvertices contained in the container.

5.3. Preprocessing

Before weaving HiLA aspects into the base machine, we first need to preprocess the latter. The purpose of preprocessing the base machine is to transform it into a canonical form, and to make it as simple as possible to determine the applicable aspects at a certain transition.

The preprocessing consists of six steps, as shown in Algorithm 5.3. Since each of the steps is semantic preserving, the overall preprocessing is also semantic preserving. In the following, the six transformation steps, as well as the semantic preservation, are explained in more detail.

Algorithm 5.3 Preprocessing

```
1: procedure PREPROCESSING(SM)
2:     REMOVEJUNCTIONS(SM)
3:     REMOVETARGETUNSTRUCTUREDTRANSITIONS(SM)
4:     REMOVEFORKVERTICES(SM)
5:     CREATEBEFORESECTIONS(SM)
6:     INSERTTRACEVARIABLES(SM)
7:     INSERTERRORSTATES(SM)
8: end procedure
```

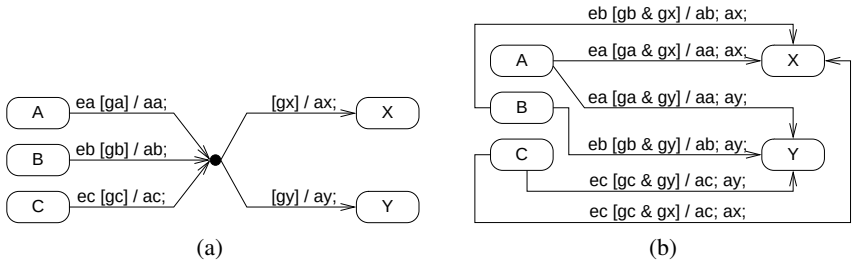


Figure 5.2: Example: eliminating a junction vertex

5.3.1. Removing junctions. Junctions are "semantic-free", they are used to "chain together multiple transitions" [57, p. 542]. That is, a chain of multiple transitions, connected by junctions, is semantically equivalent to a single transition.

In the first preprocessing step, we remove (almost, see below) all junctions in the base machine SM by transforming each such transition chain to a semantically equivalent single transition. This way we simplify the structure of the base machine by reducing the number of transitions contained in it. The purpose of this preprocessing step is to simplify the determination the proceeding and the following states of a given state by shortening the transition paths.

To this end, we simply merge the transitions connected by (almost) each junction. We replace each pair (t_1, t_2) of transitions where there exists a junction j such that $j = \mathsf{target}(t_1) = \mathsf{source}(t_2)$ by a transition t with $\mathsf{source}(t) = \mathsf{source}(t_1)$, $\mathsf{target}(t) = \mathsf{target}(t_2)$, $\mathsf{trigger}(t) = \mathsf{trigger}(t_1)$, $\mathsf{guard}(t) = \mathsf{guard}(t_1) \wedge \mathsf{guard}(t_2)$, and $\mathsf{effect}(t) = \mathsf{effect}(t_1) \cdot \mathsf{effect}(t_2)$. Note that according to [57, p. 573] t_2 must not have a trigger.

EXAMPLE 5.1 (Removing junction vertices). In Fig. 5.2a three transitions lead to the junction, two transitions leave it. This preprocessing step replaces them transitions by six other transitions, as shown in Fig. 5.2b.

As stated above, we remove only *almost* every junction. The only junctions that we do not remove are the followers of the initial vertices (in each region), if they are junctions. The reason why they cannot be removed is simply that they are necessary to model certain behaviors, e.g. when target unstructured transitions are removed, see Sect. 5.3.2.

The algorithm is given in Algorithm 5.4. Note we require that in SM there is no cycle consisting of such transitions whose source and target are both junctions. That is, we require $\mathsf{source}(t_1) \neq \mathsf{target}(t_k)$ for every $\{t_1, t_2, \ldots, t_k\} \subseteq \mathrm{T}(\mathsf{SM}), k \in \mathbb{N}$ such that $\mathrm{isJunction}(\mathsf{source}(t_i)) \wedge \mathrm{isJunction}(\mathsf{target}(t_i))$, $1 \leq i \leq k$, and $\mathsf{target}(t_i) = \mathsf{source}(t_{i-1}), 1 < i \leq k$.

After this step, the base machine contains no junctions any more, except the targets of the initial transitions of the regions, if these targets are junctions.

5.3.2. Removing target unstructured transitions. A transition t is called structured, if its source and target are direct subvertices of the same region, otherwise its called unstructured, i.e. t is structured iff. $\mathrm{LCA}(\mathsf{source}(t), \mathsf{target}(t)) = \mathsf{container}(\mathsf{source}(t)) = \mathsf{container}(\mathsf{target}(t))$. We say t is *target unstructured*, if $\mathrm{LCA}(\mathsf{source}(t), \mathsf{target}(t)) \neq \mathsf{container}(\mathsf{target}(t))$, and t is *source unstructured*, if

Algorithm 5.4 Remove junctions

```
1: procedure REMOVEJUNCTIONS(SM)
2:         ▷ precondition: there is no cycle consisting of only transitions whose target and source are both junctions
3:     for all j ∈ J_c(SM) do
4:         if target(inTr(container(j))) ≠ j then
5:             T_1 ← incoming(j)
6:             T_2 ← outgoing(j)
7:             for all (t_1, t_2) ∈ T_1 × T_2 do
8:                 insertTransition(source(t_1), target(t_2), trigger(t_1), guard(t_1) ∧ guard(t_2), (effect(t_1) · effect(t_2)))
9:             end for
10:            remove(T(SM), (T_1 ∪ T_2))
11:            remove(V(SM), j)
12:        end if
13:    end for
14: end procedure
```

$\mathrm{LCA}(\mathsf{source}(t), \mathsf{target}(t)) \neq \mathsf{container}(\mathsf{source}(t))$. Obviously a transition can be both target unstructured and source unstructured.

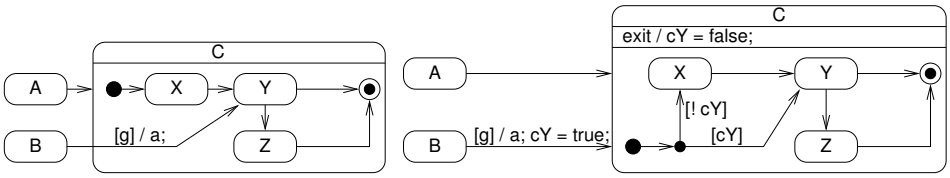
We remove in this step the target unstructured transitions in a given base machine SM. The purpose of this step is to uniform the activation of composite states. In general, a composite state S can be activated in two (semantically equivalent) ways:

(1) as the result of firing a (target structured) transition t leading to S. In this case, there must be an initial vertex $i_r \in \mathsf{subvertex}(r)$ for each $r \in \mathsf{region}(S)$, otherwise the state machine is not well-formed (see p. 12). After t is fired, the initial transitions $\mathrm{inTr}(r)$ for each $r \in \mathsf{region}(S)$ are fired,

(2) as the result of firing a (target unstructured) transition t' leading to some substate s of S. In this case, the execution of SM continues at s after the firing of t'. Additionally, if S contains more than one regions, the regions not containing s are started by firing their initial transitions.

This preprocessing step replaces the second form by the first one. The basic idea is to replace the target of t' by S, and, on the new transition, make variable assignments to store the original target. When the initial transitions of S's regions are fired, the variables are used to distinguish the "right" state to activate.

EXAMPLE 5.2 (Removing target unstructured transitions). In Fig. 5.3a, the transition from B to Y is target unstructured, since the least common ancestor of B and Y is not the region containing Y. The result of applying this preprocessing step to Fig. 5.3a is given in Fig. 5.3b: the problematic transition changes its target to C, and hence becomes target structured. The transition now also contains an action, which sets variable cY to true. When C is activated, its initial junction distinguishes two cases in dependence of cY which is the next state to activate. When the state machine is started, variable cY is initialized to be false (not shown in Fig. 5.3). When C is left, cY is set back to false to ensure the correct execution next time C gets active.



(a) Before preprocessing (b) After preprocessing

Figure 5.3: Example: removing target unstructured transitions

The algorithm implementing this step is given in Algorithm 5.5. For each target unstructured transition t, a unique boolean variable $\mathsf{cnt}_{\mathsf{target}(t)}$ is created (in line 5), which is initialized with false when the state machine is started (line 6). The effect of t is extended by an assignment of a new variable which indicates $\mathsf{target}(t)$ in variable cnt_r (line 7), and its target is set to be the container state cs of its original target (line 9). The exit action of cs is extended (line 10) to ensure that $\mathsf{cnt}_{\mathsf{target}(t)}$ has the value false when cs is activated again.

If the region does not contain an initial vertex originally, one is inserted into the region (lines 12–15). If the region does not contain an initial junction, one is inserted (lines 20–24). From the initial junction to the target of the problematic transition a new transition (which we call t') is inserted to simulate t (lines 15, 22, and 31). Note all the existing transitions from the initial junction are extended by an additional guard (line 29) which prevents them from being fired when t' is supposed to be fired.

After this step, every transition in the state machine is target structured, and every region r contains an initial vertex. The target of the (exactly one) transition leaving the initial vertex, which we notate as $\mathrm{inTr}(r)$, is always a junction. $\mathrm{inTr}(r)$ is the only transition entering this junction. The fact that every transition is target structured makes it easier to determine when ≪after≫ aspects should be executed, see Sect. 5.5.3.

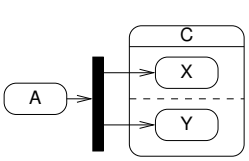
Note, however, that if the original state machine contains a fork pseudostate f, according to the UML, all transitions outgoing f "must target states in different regions of an orthogonal state" [57, p. 541]. This means that all these transitions are target unstructured, and that after this step, these transitions will be leading to the state containing their original targets. Obviously this breaks the above constraint (see Fig. 5.4 for an example). This violation will be reconciled in the next step.

5.3.3. Removing fork vertices. Transitions outgoing fork vertices have been modified while target unstructured transitions were modified (Sect. 5.3.2). The purpose of removing all fork vertices now is to mitigate the aforementioned violation of the well-formedness constraint. According to the UML Specification [57, p. 541] each fork vertex must have exactly one incoming transition. We remove each fork vertex in the state machine and all the transitions leaving it (all of the transitions lead to some substate of an orthogonal state), and replace the target of its incoming transition by the orthogonal state. Effects, if any, of the outgoing transitions are merged with those of the incoming transition. Recall that "a fork segment must not have guards or triggers" [57, p. 573].

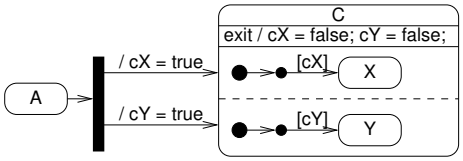
Algorithm 5.5 Remove target unstructured transitions

```
1: procedure REMOVETARGETUNSTRUCTUREDTRANSITIONS(SM)
2:     while ∃t ∈ T(SM) · [LCA(source(t), target(t)) ≠ container(target(t))] do
3:         t' ← target(t)
4:         r ← container(t')
5:         cnt_t' ← newVar
6:         effect(inTr(SM)) ← effect(inTr(SM)) · toUML(cnt_t' ← false)
7:         effect(t) ← (effect(t)) · toUML(cnt_t' ← true)              ▷ new effect
8:         cs ← state(r)
9:         setTarget(t, cs)                                            ▷ new target
10:        exit(cs) ← exit(cs) · toUML(cnt_t' ← false)
11:        if initial(r) = ⊥ then
12:            i ← insertVertex(r, initial)
13:            j ← insertVertex(r, junction)   ▷ junction not necessary, but still add one, to keep the algorithm simple
14:            insertTransition(i, j, ⊥, true, skip)
15:            insertTransition(j, t', ⊥, toUML(cnt_t'), skip)
16:        else
17:            i ← initial(r)
18:            it ← inTr(r)
19:            if ¬isJunction(target(it)) then                         ▷ new junction needed
20:                j ← insertVertex(r, junction)
21:                insertTransition(i, j, ⊥, true, skip)
22:                insertTransition(j, t', ⊥, toUML(cnt_t'), skip)
23:                setSource(it, j)
24:                guard(it) ← guard(it) ∧ toUML(¬cnt_t')
25:            else
26:                j ← target(it)
27:                ▷ make sure that the existing transitions won't be fired when the new transition is inserted
28:                for all t_old ∈ T(SM) · [source(t_old) = j] do
29:                    guard(t_old) ← guard(t_old) ∧ toUML(¬cnt_t')
30:                end for
31:                insertTransition(j, target(t), ⊥, toUML(cnt_t'), skip)
32:            end if
33:        end if
34:    end while
35: end procedure
```

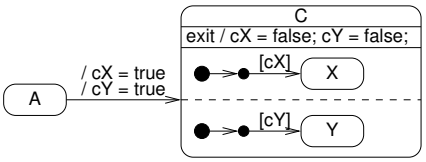
EXAMPLE 5.3 (Removing fork vertices). Figure 5.4a shows part of a state machine, containing a fork vertex. Note the two transitions leaving the fork vertex are both target unstructured. First, these transitions are transformed according to the procedure described in Sect. 5.3.2 to be target-structured, the results is shown in Fig. 5.4b, which is not (yet) well-formed w.r.t. [57]. Applying the procedure described in this section, Fig. 5.4b is transformed to Fig. 5.4c, where the fork and all its outgoing transitions are removed, and the incoming transition is extended by the effects of the (now removed) outgoing transitions.



(a) Original situation



(b) Target unstructured transitions removed (not well-formed w.r.t. [57]!)



(c) Fork removed

Figure 5.4: Example: removing fork vertices

Algorithm 5.6 Remove fork vertices

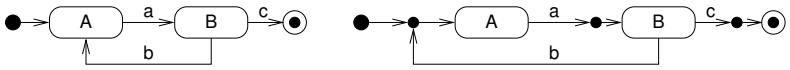
```
1:  procedure REMOVEFORKVERTICES(SM)
2:      for all frk ∈ F_k(SM) do
3:          t_1 ← t ∈ T(SM), target(t) = frk                    ▷ exactly one
4:          t_2 ← t ∈ T(SM), source(t) = frk                    ▷ anyone
5:          setTarget(t_1, target(t_2))                         ▷ update the target
6:          effect(t_1) ← effect(t_1) · (•_{source(t)=frk} effect(t))   ▷ update the effect
7:          removeTransition(SM, {t | source(t) = frk})
8:      end for
9:      removeVertex(SM, F_k(SM))
10: end procedure
```

The algorithm is given in Algorithm 5.6. The target of the incoming transition is updated in line 5; its effect is updated in line 6, where we use a bullet (•) to refer to the sequential concatenation of a set of actions in an arbitrary order. Since in the case of removing forks the effects to concatenate are assignments of pair-wise distinct variables, it does not matter in which order they are concatenated.

After this step, the state machine contains no fork vertices any more. The violation of well-formedness caused by Sect. 5.3.2 is reconciled.

5.3.4. Creating "before" section. The purpose of creating a "before" section before each state is to simplify the weaving of ≪before≫ aspects: for each state (including final states) we unify all its incoming transitions, so that syntactically there is only one transition leading to each state. This way ≪before≫ aspects only need to be woven once, at one place.

To this end, we introduce for each state s a new junction $\mathsf{jBefore}(s)$ into the region of s, redirect all incoming transitions of s to $\mathsf{jBefore}(s)$, and insert a new transition from $\mathsf{jBefore}(s)$ to s. Due to the "semantic-freedom" [57, p. 542] of junctions, this operation does not change the semantics of the original state machine.



(a) Original situation (b) After inserting before sections

Figure 5.5: Example: creating "before" section

EXAMPLE 5.4 (Creating before sections). Three before sections are needed to preprocess the state machine given in Fig. 5.5a: for the states A and B, and the final state (final states are also states). The result of applying the preprocessing step "Creating before sections" is given in Fig. 5.5b. Note the before sections of B and the final state are in this example not really necessary, since in the original state machine there is only one transition leading to B resp. the final state. Our algorithm does not check this situation, and simply creates for each state a before section. Therefore, in the following preprocessing and weaving steps, there is always exactly one transition leading to each state, and this transition always originates from a junction vertex.

Algorithm 5.7 Create before sections

```
1: procedure CREATEBEFORESECTION(SM)
2:     for all s ∈ S(SM) do                          ▷ including final states
3:         j ← insertVertex(container(s), junction)
4:         for all t ∈ T(SM) · [target(t) = s] do
5:             setTarget(t, j)
6:         end for
7:         insertTransition(j, s, ⊥, true, skip)
8:     end for
9: end procedure
```

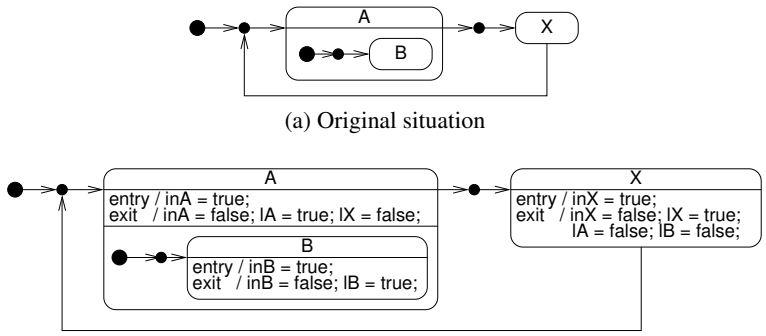
The (rather simple) algorithm for this preprocessing step is given in Algorithm 5.7. After this step, there is exactly one transition leading to each state s of the state machine. This transition always originates from a junction vertex, which we call jBefore(s). All ≪before≫ aspects should be woven to this transition. On the other side, this transition is the only one leaving jBefore(s), its target is always a state.

5.3.5. Insert trace variables. One of HiLA's highlights is its support for cross-region aspects and history-based aspects. As a preparation for weaving such aspects, we extend the entry and exit actions of SM's states to make assignments to indicate which states are currently active and which ones have just become inactive.

Tracing the currently active states is simple. We just define for each state s a unique variable in_s, and set in_s to be true and false in the entry and exit action of s, respectively. At runtime, the values of these variables indicate which states (in different regions) are currently active.

Tracing the last active state is slightly more complex. We introduce for each state s another variable l_s, and set l_s to be true in the exit action of s, indicating that s "has just become inactive". Furthermore, we also have to extend the exit

action of each state, that is directly reachable from s, by an action to set $l_{s'}$ to be false for each s' such that $s' \in \mathsf{subvertex}^*(s)$.



(a) Original situation

(b) After inserting trace variables

Figure 5.6: Example: inserting trace variables

EXAMPLE 5.5 (Inserting trace variables). The state machine given in Fig. 5.6a is transformed by this step to Fig. 5.6b. At runtime, when B is entered, we have inA = true and inB = true, and, if the last active state was X, also lX = true. When A (and thus B) is left, X gets active, we have inX = true, lA = true and lB = true.

The algorithm for this step is shown in Algorithm 5.8. Note in lines 14–22 we exploit the fact that after the proceeding steps each transition outgoing a state either leads to a junction, which is the source of the before section of another state, or a join vertex, from which the only outgoing transition then leads to the source of the before section of another state. The assignments of the in variables, indicating "I am active", are inserted in lines 10 and 11. The l variables, indicating the last active states, are set in the loop from line 12 to 26.

5.3.6. Inserting Error States. When resumption conflicts are detected, the state machine is supposed to enter an error state (see Sect. 5.1). In the current version of HILA, we consider this error to be an internal error of the region containing the aspect state, and insert for each region r a state $\mathsf{Err}(r)$ for the indication of resumption conflicts having occurred "somewhere in r".

The algorithm for inserting error states is straight forward. We first visit all states that are originally contained in SM, add an error state to each of its regions, and then insert an error state for the top region of SM, see Algorithm 5.9. The error states are not yet connected to the rest of the state machine. This is only done in the postprocessing step, see Sect. 5.6.

5.3.7. Summary. Overall, after the preprocessing steps, our state machines show the following properties:

- For each state s that is not the error state of some region, i.e. s is an "original" state contained in the base machine, there is exactly one transition entering it. The source of this transition is always a junction, which we call $\mathsf{jBefore}(s)$. There is exactly one transition leaving $\mathsf{jBefore}(s)$, which always leads to s.

Algorithm 5.8 Insert trace variables

```
1: procedure INSERTTRACEVARIABLES(SM)
2:     for all s ∈ S(SM) do
3:         in_s ← newVar
4:         l_s ← newVar
5:         effect(inTr(SM)) ← effect(inTr(SM)) · toUML(in_s ← false)
6:         effect(inTr(SM)) ← effect(inTr(SM)) · toUML(l_s ← false)
7:     end for
8:     for all s ∈ S(SM) do
9:         entry(s) ← entry(s) · toUML(in_s ← true)
10:        exit(s) ← exit(s) · toUML(in ← false)
11:        exit(s) ← exit(s) · toUML(l_s ← true)
12:        for all t ∈ outgoing(s) do
13:            j ← target(t)
14:            if isJunction(j) then                ▷ j = jBefore(s') for some state s'
15:                t' ← t'' such that source(t'') = j                ▷ only one!
16:                s' ← target(t')
17:            else                                 ▷ invariant: j is a join vertex
18:                t' ← t'' such that source(t'') = j                ▷ only one!
19:                j' ← target(t')                  ▷ j' = jBefore(s') for some state s'
20:                t'' ← t''' such that source(t''') = j'            ▷ only one!
21:                s' ← target(t'')
22:            end if
23:            for all s_0 · [s_0 ∈ subvertex*(s)] do
24:                exit(s') ← exit(s') · toUML(l_{s_0} ← false)
25:            end for
26:        end for
27:    end for
28: end procedure
```

Algorithm 5.9

```
1: procedure INSERTERRORSTATES(SM)
2:     for all s ∈ S(SM) do
3:         for all r ∈ region(s) do
4:             Err(r) ← insertVertex(r, state)
5:         end for
6:     end for
7:     Err(top(SM)) ← insertVertex(top(SM)), state)
8: end procedure
```

- It holds that jBefore(s) and s are always in the same region, that is, container(jBefore(s)) = container(s).
- All transitions are target structured. There may, however, exist source unstructured transitions.
- For each transition t leaving a state, its target may be either a junction (jBefore(s'), where s' is also a state), or a join vertex.

- If a transition t is source unstructured, there exists exactly one state s such that $\mathsf{source}(t) \in \mathsf{subvertex}^+(s)$ and $\mathrm{LCA}(s, \mathsf{target}(t)) = \mathsf{container}(s) = \mathsf{container}(\mathsf{target}(t))$. We write $\mathsf{source}_{\mathsf{struct}}(t) := s$.
- If a transition t is source structured, we write $\mathsf{source}_{\mathsf{struct}}(t) := \mathsf{source}(t)$.
- For each state s which is not the error state of some region, two boolean variables are introduced: in_s, indicating if s is currently active, and l_s, indicating if s has just become inactive.
- In each region r, there is an initial vertex $\mathrm{initial}(r)$. The (exactly one) transition $\mathrm{inTr}(r)$ leaving $\mathrm{initial}(r)$ leads to a junction, which we refer to as $j^i(r)$. The transition $\mathrm{inTr}(r)$ is the only transition entering $j^i(r)$. Each transition leaving $j^i(r)$ leads to another junction $j = \mathsf{jBefore}(s)$, where s is a state.
- Each junction is either $j^i(r)$ of some region r, or $\mathsf{jBefore}(s)$ for some state s, but never both.
- In each region r, there is a state $\mathsf{Err}(r)$ which indicates that some resumption conflict has just occurred in r. Error states are not yet connected to other vertices. They will remain isolated, i.e. unconnected until the post-processing step of the weaving, which is described in Sect. 5.6. For a given state machine SM, we write $\mathsf{Err}(\mathsf{SM})$ to mean the set of all its error states: $\mathsf{Err}(\mathsf{SM}) := \{\mathsf{Err}(r) \mid r \in \mathsf{region}(s) \text{ for some } s \in \mathrm{S}(\mathsf{SM})\}$.

As will be shown in the following sections, these properties play an important role in our weaving algorithms.

5.4. Weaving History Properties

History properties are implemented by a non-deterministic finite automaton, which performs one execution step once a "history relevant" state gets active. A state s is history relevant w.r.t. a history property p if and only if s is contained in the configuration selector of one of p's history elements. Let state machine SM and aspect $\mathcal{A}$ containing the set of history properties $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ be given. We write $\mathsf{hrel}(\mathsf{SM}, h_i)$ for the set of states that are relevant for keeping track of the history of active state configurations in order to keep the history property h_i up to date. Obviously we have $\mathsf{hrel}(\mathsf{SM}, h_i) = \bigcup_{e \in \mathsf{elem}(h_i)} \mathsf{config}(e)$, where $\mathsf{elem}(h_i)$ is the set of the history elements of a h_i, and $\mathsf{config}(e)$ is the configuration descriptor of a history element. Furthermore, we define $\mathsf{hrel}(\mathsf{SM}, \mathcal{A}) = \bigcup_{h \in \mathcal{H}} \mathsf{hrel}(\mathsf{SM}, h)$.

EXAMPLE 5.6 (History relevant states). Let SM be the state machine given in Fig. 2.3a, and $\mathcal{A}$ be the aspect in Fig. 4.20a, then $\mathsf{hrel}(\mathsf{SM}, \mathcal{A})$ is the set containing the states Right and Wrong.

5.4.1. Structure of the NFA. We first define an NFA for a pair of a state machine SM and a history property p, which we call $NFA(\mathsf{SM}, p)$. Given a state machine SM, weaving a set of history properties $P = \{p_i\}$ (contained in a set of aspects) to SM is performed by simulating all $NFA(\mathsf{SM}, p_i)$ in the base machine.

We need some auxiliary functions in the construction of the NFA: if the multiplicity of an element e is of the form $n_1..n_2$ we call $\min(e) := n_1$ the *minimal* and $\max(e) := n_2$ the *maximal number of occurrences* of that element. If the multiplicity of e is of the form $n..*$ we define $\min(e) := \max(e) := n$. In this case we define $\mathsf{infinite?}(e) = \mathsf{true}$, otherwise $\mathsf{infinite?}(e) = \mathsf{false}$.

To distinguish states of the finite automaton from states of the state machine we refer to the former as *hstate*s. Given a history property p, we number his history elements from 0 to n and write e_i for the history element with number i. We define an hstate for each tuple $(a_0, \ldots, a_n)$ with $0 \leq a_i \leq \max(e_i)$ and refer to it as $\mathit{hstate}(a_o, \ldots, a_n)$.

The NFA is supposed to react to the events of the base machine activating and deactivating a configuration as defined by one of the history elements of p, and decide thus whether the current execution history is "accepted" by the pattern. To this end, we insert a transition with label e_i from each hstate of the form $\mathit{hstate}(a_0, \ldots, a_i, \ldots, a_n)$ to $\mathit{hstate}(a_0, \ldots, a_i + 1, \ldots, a_n)$ if $a_i < \max(e_i)$, a self-transition with label e_i for hstates of the form $\mathit{hstate}(a_0, \ldots, a_n)$ if $a_i = \max(e_i) \wedge \mathsf{infinite?}(e) = \mathsf{true}$, and a transition from $\mathit{hstate}(a_0, \ldots, a_n)$ to the origin $\mathit{hstate}(0, \ldots, 0)$ if $a_i = \max(e_i) \wedge \mathsf{infinite?}(e) = \mathsf{false}$. Finally, we insert (nondeterministic) transitions $h_0 \xrightarrow{s} h_0$ for all states $s \in \mathsf{hrel}(M, p_i)$. This way, it is ensured that the automaton keeps track of all sequences of states in its history. The accepting states are all hstates $\mathit{hstate}(a_0, \ldots, a_n)$ with $\min(e_i) \leq a_i \leq \max(e_i)$ for all i with $0 \leq i \leq n$.

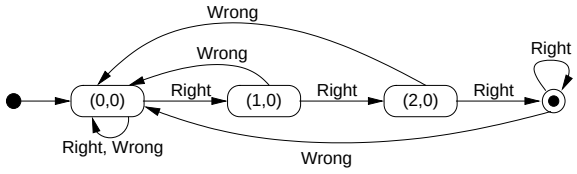


Figure 5.7: Example: NFA for history property a defined in Fig. 4.20a

EXAMPLE 5.7 (NFA). Figure 5.7 shows the NFA for the history property a defined in Fig. 4.20a.

5.4.2. Simulating the NFA. The NFA is not actually constructed by the weaving. Instead, its behavior is simulated by the base machine. In fact, the base machine needs to simulate one such NFA for each history property.

Given a state machine SM and a set of history properties $\{p_1, \ldots, p_n\}$, we refer to the elements of $\mathsf{elem}(p_i)$ as $e_{i,j}, 1 \leq j \leq \#(\mathsf{elem}(p_i))$ for $1 \leq i \leq n$. For each $1 \leq i_0 \leq n$ we define a set of boolean variables $B = \{b_{i_0,i_1,\ldots,i_K}\}$, with $K = \#(\mathsf{elem}(p_{i_0}))$ and $0 \leq i_m \leq \max(e_{i_0,i_m})$ for $1 \leq m \leq K$. A variable $b_{i_0,i_1,\ldots,i_K}$ is true iff. in the virtual NFA for the history property p_{i_0} the $\mathit{hstate}(i_1, \ldots, i_K)$ is active.

To implement the history property in a base machine SM, we extend the exit behavior of each state $s \in \mathrm{S}(\mathsf{SM})$ by an action A_s. The function of A_s is to check if there exists a history property p_i and a history element of it, $e_{i,j}$, such that the complete configuration of $e_{i,j}$ is active while s is active. If this is the case, and $\mathsf{constraint}(e_{i,j})$ is also satisfied, we simulate a step of the NFA when s is left by first checking the variables in B to find the currently active *hstate*s and then updating the values of the B-variables according to the transitions now fired in the NFA.

Determining if the configuration of a history element e is active is achieved by testing if all states of $\mathsf{config}(e)$, that are different than s, are active.

5.4.3. Counting. Finally, counting the occurrences of such contiguous subsequences that are specified by a history property p is implemented by yet another extension of the exit actions to check if the NFA is in an accepting state (i.e. whether the corresponding boolean variable is true), and, if so, to increase the history property by one.

For weaving history properties, we define an index for each element of $\mathcal{H}$, and write $\mathcal{H} = \{h_1, \ldots, h_n\}$. We also define an index of each $h \in \mathcal{H}$ and note $\mathsf{elem}(h) = \{e_h^1, \ldots, e_h^{k_h}\}$, where $k_h = \#(\mathsf{elem}(h))$. Furthermore, the function idx returns the index of a history property or a history element: we define $\mathrm{idx}(h_x) := x$, $\mathrm{idx}(e_h^x) := x$.

The overall algorithm weaving a set $\mathcal{H}$ of history properties to state machine SM is shown in Algorithm. 5.10. For each history property h we use a variable v_h to store the variable of SM to represent h. Later on, for each $e \in \mathsf{elem}(h)$ and each $s \in \mathsf{config}(e)$, the exit action of s is extended with an action to check if the NFA should advance one step, and if yes, simulate the step.

This algorithm is given in Algorithm 5.11. First it is checked if all other states contained in the current history element are active (line 2). If this is the case, an execution step of the NFA is made: the currently active NFA states are stored in the boolean variables named b, where $b_{x,i_{x_1},\ldots,i_{x_k}} = \mathsf{true}$ indicates that in the NFA for history property h_x, currently the NFA state labeled $i_{x_1}, \ldots, i_{x_k}$ is active. The NFA states following this state are then activated according to Sect. 5.4.1. Since in our NFA, the original state $(0, \ldots, 0)$ is always active, we set the variable representing it to be true (line 14). Finally, it is tested if the NFA has entered an accepting state (line 15), and, if so, the value of v_h is increased by one.

If a history property is parameterized, we consider it as a set of a number of history properties, distinguished by the value of the parameter. We repeat the above procedure and define (and simulate) an NFA for each value from the domain of its

Algorithm 5.10 Weave history properties

```
1:  procedure WEAVEHISTORYPROPERTIES(SM, H)
2:      for all h ∈ H do
3:          v_h ← newVar
4:          for all i_1, i_2, ..., i_k, k = #(elem(h)), and 0 ≤ i_m ≤ max(e^m), 1 ≤
    m ≤ k do
5:              b_{idx(h),i_1,i_2,...,i_k} ← newVar
6:              add(property(SM), b_{idx(h),i_1,i_2,...,i_k})
7:          end for
8:      end for
9:      for all h ∈ H do
10:         for all e ∈ elem(h) do
11:             for all s ∈ config(e) do
12:                 exit(s) ← exit(s) · toUML(oneStep(h, e, s))        ▷ see
    Algorithm 5.11
13:             end for
14:         end for
15:     end for
16: end procedure
```

parameter. For example, the history property a in Fig. 4.20a is woven as a set of NFAs, one for each lv.

Algorithm 5.11 Simulating One Step of the NFA

```
1:  procedure ONESTEP(h, e, s)
2:      if (⋀_{s'∈config(e),s'≠s} isActive(s')) ∧ constraint(e) then   ▷ one step of the NFA
3:          for all b_{idx(h),i_1,i_2,...,i_k}, k = #(elem(h)) and 0 ≤ i_m ≤ max(e_h^m) for each m ∈ 1,...,k do
4:              if b_{idx(h),i_1,i_2,...,i_k} then
5:                  if i_{idx(e)} < max(e) then
6:                      b_{idx(h),i_1,i_2,...,(i_{idx(e)}+1),...,i_k} ← true
7:                      b_{idx(h),i_1,i_2,...,i_k} ← false
8:                  else if ¬ infinite?(e) then                      ▷ i_{idx(e)} = max(e)
9:                      b_{idx(h),0,...,0} ← true
10:                     b_{idx(h),i_1,i_2,...,i_k} ← false
11:                 end if
12:             end if
13:         end for
14:         b_{idx(h),0,...,0} ← true
15:         if ⋀_{m∈1,...,#(elem(h))} min(e_h^m) ≤ i_m ≤ max(e_h^m) ∧ b_{idx(h),i_1,i_2,...,i_k} then
16:             v_h ← v_h + 1
17:         end if
18:     end if
19: end procedure
```

5.5. Weaving Aspects

In HILA, each aspect is woven into a region of an orthogonal state. In this section, we first describe how this region is constructed, then we give the algorithms introducing the states containing these regions to the base machine.

5.5.1. aspect2Region. The transformation of an aspect $\mathfrak{a}$ to a region $R_{\mathfrak{a}}$(s) to advise the transition entering state s is described in Algorithms 5.12–5.17. At run time, when $R_{\mathfrak{a}}(s)$ gets active, it first checks whether the precondition of $\mathfrak{a}$ is satisfied, and, if this is the case, executes advice ($\mathfrak{a}$). On every transition t where $\mathfrak{a}$ is applicable, an instance of $R_{\mathfrak{a}}(\mathsf{target}(t))$ is created. Algorithm 5.12 shows how an instance of $R_{\mathfrak{a}}(s)$ is constructed, if $\mathfrak{a}$ is applicable just before state s is about to become active. The constructed region will be inserted into the "aspect state" to splice the (exactly one, see Sect. 5.3.4) transition leading to s.

As shown in Algorithm 5.12, $R_{\mathfrak{a}}$(s) contains basically a clone of each (direct or recursive) subvertex of advice($\mathfrak{a}$) and the corresponding transitions (Algorithm 5.13). Additionally, $R_{\mathfrak{a}}(s)$ also contains a junction j (Algorithm 5.14) and additional transitions around j: the initial transition t_1 is modified to be leading to j; t_2 and $t_3(r)$ ensure that the advice is executed if the precondition of the aspect is satisfied, and otherwise the advice is skipped. The value of $t_3(r)$ will also be used in the next step, where assignments are inserted (in Algorithm 5.15) as effects

Algorithm 5.12 Region implementing an advice

```
1: function RegionOfAspect(SM, 𝔞, s)
2:     r ← new Region
3:     InsertClones(𝔞, r)                          ▷ see Algorithm 5.13
4:     InsertJunction(𝔞, r, s)                     ▷ see Algorithm 5.14
5:     InsertGotoVariables(SM, r)                  ▷ see Algorithm 5.15
6:     CollectGotoVariables(𝔞, r, s)               ▷ see Algorithm 5.17
7:     return r
8: end function
```

Algorithm 5.13 Insert clones of vertices and transitions

```
1: procedure InsertClones(𝔞, r)
2:     for all v ∈ V(𝔞) do
3:         add(subvertex(r), cloneVertex(v))
4:     end for
5:     for all v ∈ subvertex⁺(r) do
6:         o ← clone⁻¹(v)                          ▷ o ≠ ⊥
7:         for all t ∈ outgoing(o) do
8:             insertTransition(v, v′, trigger(t), guard(t), effect(t)),   where clone⁻¹(v′) = target(t)
9:         end for
10:    end for
11: end procedure
```

Algorithm 5.14 Insert a junction to determine if the advice should be executed

```
1: procedure InsertJunction(𝔞, r, s)
2:     j ← insertVertex(r, Junction)
3:     t_𝔞 ← inTr(advice(𝔞))
4:     t_1 ← inTr(r)          ▷ since 𝔞 contains an initial vertex, so does r
5:     setTarget(t_1, j)                           ▷ redirect inTr(r) to j
6:     effect(t_1) ← skip
7:     t_2 ← insertTransition(j, v, ⊥, toUML(pre^s_𝔞), effect(t_𝔞)),   where clone⁻¹(v) = target(t_𝔞) ▷ proceed to advice only when precondition satisfied
8:     f ← f′ where f′ ∈ subvertex(r) ∧ class(f′) = labeledfinal   ▷ any one
9:     t_3(r) ← insertTransition(j, f, ⊥, toUML(¬pre^s_𝔞), effect(t_𝔞))   ▷ go to a final state if precondition not satisfied
10: end procedure
```

of transitions leading to the final states of $R_{\mathfrak{a}}(s)$ to set resumption variables (see Sect. 5.1). Obviously, transition $t_3(r)$ should not set a resumption variable, since it is only a bypass of the aspect. On the other hand, the effect of every other transition leading to a (labeled) final state in the region is extended by two assignments (lines 12–14, see also Algorithm 5.16), setting the resumption variable to true, and stores (in variable $\mathsf{pr}(s)$) the priority of the aspect setting this resumption variable. Note that according to the assignment in Algorithm 5.16, writing access to $\mathsf{gt}(s)$ always set its value to true, this way, race conditions w.r.t. these variables are avoided. Since we assume that the firing of a transition is atomic (see Sect. 2.1.4)

and in particular the execution of the effect cannot be interrupted, and the writing access in lines 5.16, lines 3–5 only increases the value of $\mathsf{pr}(s)$, race conditions are also avoided w.r.t. the priorities. If we do not assume the atomicity of transition firing, a slightly more complex model would be necessary, using boolean variables to indicate if a certain priority is set for a certain resumption state.

Algorithm 5.15 Insert goto variables

```
1:  procedure INSERTGOTOVARIABLES(SM, r)
2:      for all f ∈ subvertex(r) ∧ class(f) = labeledfinal do
3:          s' ← label(f) ∩ subvertex(container(s))          ▷ at most one!
4:          if s' ≠ ⊥ then
5:              if gt(s') ∉ property(SM) then
6:                  gt(s') ← newVar
7:              end if
8:              if pr(s') ∉ property(SM) then
9:                  pr(s') ← newVar
10:             end if
11:             p ← priority(a)
12:             for all t ∈ incoming(f), t ≠ t3(r) do
13:                 effect(t) ← effect(t) · toUML(ASSIGNVARIABLE(s', p))   ▷
    see Algorithm 5.16
14:             end for
15:         end if
16:     end for
17: end procedure
```

Algorithm 5.16 Assign goto and priority variables

```
1:  procedure ASSIGNVARIABLE(s, p)
2:      gt(s) ← true
3:      if pr(s) < p then
4:          pr(s) ← p
5:      end if
6:  end procedure
```

Algorithm 5.17 Collect goto variables

```
1:  procedure COLLECTGOTOVARIABLES(a, r, s)
2:      gtVars(r) ← ∅
3:      prVars(r) ← ∅
4:      for all f ∈ subvertex(r), class(f) = labeledfinal do
5:          s' ← label(f) ∩ subvertex(container(s))          ▷ only one!
6:          gtVars(r) ← gtVars(r) ∪ {gt(s')}
7:          prVars(r) ← prVars(r) ∪ {pr(s')}
8:      end for
9:  end procedure
```

Since an advice may contain several (labeled) final states, and each of them may contain a label to indicate a resumption state, there may be several resumption strategies defined by the advice. Therefore a region r implementing an aspect may also need several resumption variables, and for each a priority variable. These are collected (Algorithm 5.17) in the set variables $\mathrm{gtVars}(r)$ and $\mathrm{prVars}(r)$. Later on, the sets will be merged to determine all the resumption variables used in the aspect state.

The value of function $pre^s_{\mathfrak{a}}$ (see Algorithm 5.14) is true iff. the precondition of aspect $\mathfrak{a}$ is satisfied just before state s is entered. More formally, the function is defined as follows:

$$pre^s_{\mathfrak{a}} = \begin{cases} \mathsf{true}, & \text{if } kind(\mathfrak{a}) = \mathsf{before} \wedge s \in \mathsf{config}(\mathfrak{a}) \\ \mathsf{true} & \text{if } kind(\mathfrak{a}) = \mathsf{whilst} \wedge s \in \mathsf{config}(\mathfrak{a}) \wedge [l_s] \\ & \wedge \forall v \in \mathsf{config}(\mathfrak{a}) \cdot [l_v \vee in_v] \\ & \wedge \mathsf{constraint}(\mathsf{pointcut}(\mathfrak{a})) \\ \mathsf{true}, & \text{if } kind(\mathfrak{a}) = \mathsf{after} \\ & \wedge \mathsf{constraint}(\mathsf{pointcut}(\mathfrak{a})) \\ & \wedge \exists s' \in \mathrm{pred}(s) \cdot [s' \in \mathsf{config}(\mathfrak{a}) \wedge l_{s'}] \\ & \wedge \forall v \in \mathsf{config}(\mathfrak{a}) \cdot [l_v \vee in_v] \\ \mathsf{true}, & \text{if } kind(\mathfrak{a}) = \mathsf{between} \\ & \wedge \mathsf{constraint}(\mathsf{src}(\mathfrak{a})) \wedge s \in \mathsf{tgt}(\mathfrak{a}) \\ & \wedge \exists s' \in \mathrm{pred}(s) \cdot [s' \in \mathsf{src}(\mathfrak{a}) \wedge l_{s'}] \\ & \wedge \forall v \in \mathsf{src}(\mathfrak{a}) \cdot [l_v \vee in_v] \\ \mathsf{false}, & \text{otherwise} \end{cases}$$

The function $\mathrm{pred}(s)$ gives the set of all states "preceding" s. It is the union of the substates of the $\mathsf{source}_{\mathsf{struct}}$ of all transitions leading to $\mathsf{jBefore}(s)$, possibly via a join vertex, and is defined as $\mathrm{pred}(s) := \bigcup_{t \in \mathsf{to}(s)} \mathsf{subvertex}^*(\mathsf{source}_{\mathsf{struct}}(t))$, where $\mathsf{to}(s) := \{t \mid \mathsf{target}(t) = \mathsf{jBefore}(s)\} \cup \{t \mid \exists t' \cdot [\mathrm{isJoin}(\mathsf{target}(t_1)) \wedge \mathsf{source}(t_2) = \mathsf{target}(t_1) \wedge \mathsf{target}(t_2) = \mathsf{jBefore}(s)]\}$.

Recall that a «before» aspect should be executed as soon as at least one of the states contained in its config is about to be entered. On the other hand, in the case of an «after» aspect $\mathfrak{a}$, it is necessary for $pre^s_{\mathfrak{a}}$ to check 1) for any given state $s' \in \mathsf{src}(\mathfrak{a})$, either s' has just been left or it is still active and 2) at least one of these states has just been left. The precondition of a «between» is simply a conjunction of those of «before» and «after».

It is worth noting that if necessary, the @pre values of variables are stored by an entry action. That is, any construct of the form x@pre, x being a property of the state machine, is considered as a variable xBefore, and we extend each state contained in the configuration by an entry action xBefore = x. For example, in our weaving, the constraint given in Fig. 4.14a is actually implemented by an entry action pBefore = p of state Fighting and a "normal" constraint pBefore > p && p >0 in the construction of $pre^s_{\mathfrak{a}}$. Note also that in the actual implementation, it is not necessary to explicitly check if after the advised transition being fired some "right" state is going to be active (i.e. whether $s \in \mathsf{config}(\mathfrak{a})$ or $s \in \mathsf{tgt}(\mathfrak{a})$ is satisfied), since this is implicitly guaranteed, see Algorithms 5.18 and 5.19.

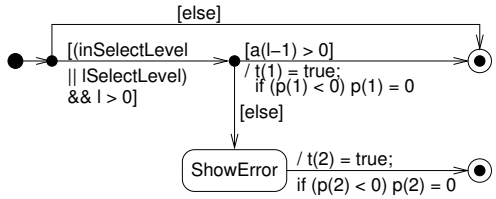


Figure 5.8: Example: region implementing aspect RightInARow (Fig. 4.20a)

EXAMPLE 5.8 (Region implementing aspect RightInARow (Fig. 4.20a)). Figure 5.8 shows the region implementing the advice of aspect RightInARow defined in Fig. 4.20a. When the region (i.e. the containing state) gets active, it is first checked if the value of l is greater than zero and if the state SelectLevel is active or has just been inactive. If this condition is not satisfied, the rest of the region is not executed, instead, a final state (it may be any one) is activated. On the other hand, if the precondition holds, the behavior as specified by the advice is executed. In this case, a variable, t(1) or t(2), depending on which final state actually terminates the region execution, is set to true, indicating how the execution of the base machine should be resumed, see Fig. 5.1c. Priority assignments are also implemented, although in this case they are actually not necessary, since the aspect has the default priority of 0 which is also the lowest priority, and the premise of the if statements can never be satisfied in this special case. Note we do not explicitly check if ShowQuestion is about to be active, since this is implicitly guaranteed by weaving the aspects on the "right" transition.

5.5.2. Weaving Configuration Aspects. Aspects with a ≪whilst≫ pointcut are referred to as ≪whilst≫ aspects, or, since the pointcut contains a configuration selector, *configuration aspects*. These aspects are woven first, before those with transition pointcuts.

We call W applicable for a state s if $s \in \mathsf{config}(\mathsf{W})$. For a set W of whilst aspects we write $W_s := \{p \in W \mid s \in \mathsf{config}(p)\}$. Furthermore, for an event e_0, we define $W_{s,e_0} := \{p \in W_s \mid \mathsf{trigger}(p) = e_0\}$. Apparently it holds that $W_s = \bigcup_{e_0 \cdot [\exists x \in W \text{ such that } \mathsf{trigger}(x) = e_0]} W_{s,e_0}$.

We weave ≪whilst≫ aspects by introducing additional elements to the base machine, see Algorithm 5.18: for each state s and each event e_0 such that $W_{s,e_0} \neq \emptyset$, we introduce a composite state Aspect (line 7), with $\mathsf{region}(\mathsf{Aspect}) = \{R_{\mathfrak{a}}(s) \mid \mathfrak{a} \in W_{s,e_0}\}$ (line 10), where $R_{\mathfrak{a}}$(s) is the region implementing $\mathfrak{a}$, see Sect. 5.5.1.

State Aspect is connected to s by a pair of transitions: one (which we call t) from s to Aspect, with trigger e, an guard true, and an empty effect (line 12); another (which we call t') from Aspect to $\mathsf{jBefore}(s)$, with no explicit trigger, guard or effect (line 13). This way, when the base machine is in state s, all applicable advices are executed simultaneously upon event e.

Note we define (in line 4) a variable $\mathsf{Asp}_w(s)$ to store all Aspect states, implementing the ≪whilst≫ aspects applicable to s. This variable will be used in the postprocessing step (Sect. 5.6).

5.5.3. Weaving Transition Aspects. After the preprocessing step "insert before sections" (Sect. 5.3.4), each state in the base machine has only one incoming

Algorithm 5.18 Weaving configuration aspects

```
1:  procedure WEAVINGCONFIGURATIONASPECTS(SM, W)
2:      S_orig(SM) ← S(SM) \ Err(SM)        ▷ store the "original" states of SM for later use, see Algorithms 5.19 and 5.20.
3:      for all s ∈ S_orig do
4:          Asp_w(s) ← ∅
5:          W_{s,e0} ← {w ∈ W | s ∈ config(w) ∧ e0 = trigger(w)}
6:          for all e0, W_{s,e0} ≠ ∅ do
7:              Aspect ← insertVertex(container(s), state)
8:              add(Asp_w(s), Aspect)
9:              for all a ∈ W_{s,e0} do
10:                 add(region(Aspect), REGIONOFASPECT(SM, a, s))        ▷ see Algorithm 5.12
11:             end for
12:             insertTransition(s, Aspect, e0, true, skip)
13:             insertTransition(Aspect, jBefore(s), ∗, true, skip)
14:         end for
15:     end for
16: end procedure
```

transition, whose source is always a junction. All ≪before≫, ≪after≫ and ≪between≫ aspects are woven into an orthogonal state splicing this transition.

For each state s and a set of aspects A, we find out all $a \in A$ that are applicable to s. Obviously, a ≪before≫ aspect b is applicable to s iff. $s \in \mathsf{config}(\mathsf{pointcut}(b))$. An ≪after≫ aspect a is applicable to s iff. s is preceded by a state s', and s' is contained in config(pointcut(a)). A ≪between≫ aspect w is applicable to s iff. $s \in \mathsf{tgt}(\mathsf{pointcut}(w))$ and s is preceded by a state s', and s' is contained in src(pointcut(a)).

We then weave all transition aspects, that are applicable to the same state, into a concurrent state Aspect, with a region for each aspect. This process is described in Algorithm 5.19. For SM and given sets of ≪before≫ aspects B, ≪after≫ aspects A and ≪between≫ aspects M, we first calculate the applicable aspects of s (lines 3–6). If there is any applicable aspect, we create state Aspect, insert it into the same region of s, connect it to $\mathsf{jBefore}(s)$ and s (lines 7–15), and then insert for each applicable aspect $\mathfrak{a}$ a region $R_{\mathfrak{a}}(s)$ into Aspect (lines 14). Recall that we refer to the source of the before section of a state s, which is always a junction (see Sect. 5.3.4), as $\mathsf{jBefore}(s)$. That is, it holds that $\exists_1 t \cdot [\mathsf{jBefore}(s) = \mathsf{source}(t) \wedge \mathsf{target}(t) = s]$.

Note that being applicable is only a necessary condition for aspect $\mathfrak{a}$ to be executed, but not a sufficient one. In the case of ≪after≫ and ≪between≫ aspects, it still has to be checked if the complete configuration has just been active and if the constraint, if any, of the pointcut is satisfied. These checks are done within the region implementing the aspect, see Sect. 5.5.1.

The transition created in line 12 is still subject to postprocessing, where the labels of the (labeled) final states of the aspects are implemented. Note also that variable $\mathsf{Asp}_t(s)$ (line 10) is used to return the orthogonal state implementing all transition aspects applicable to s. This value will be used in the following section. While there may be several such states implementing all ≪whilst≫ aspects for a

Algorithm 5.19 Weaving transition aspects

```
1:  procedure WEAVINGTRANSITIONASPECTS(SM, B, A, M)
2:      for all s ∈ S_orig(SM) do ▷ consider only the states "originally" contained in SM
3:          B_s ← {b ∈ B | s ∈ config(b)}
4:          A_s ← {a ∈ A | ∃s' ∈ pred(s) · [s' ∈ config(a)]}
5:          M_s ← {m ∈ A | s ∈ tgt(m) ∧ ∃s' ∈ pred(s) · [s' ∈ src(m)]}
6:          ALL_s ← B_s ∪ A_s ∪ M_s
7:          if ALL_s ≠ ∅ then
8:              t ← t' ∈ T(SM), source(t') = jBefore(s), target(t') = s ▷ exactly one!
9:              Aspect ← insertVertex(region(s), state)
10:             Asp_t(s) = Aspect
11:             setTarget(t, Aspect)
12:             insertTransition(Aspect, s, ∗, true, skip)   ▷ guard still subject to modification, see Algorithm 5.20
13:             for all b ∈ ALL_s do
14:                 add( region(Aspect), REGIONOFASPECT(SM, b, s))   ▷ see Algorithm 5.12
15:             end for
16:         end if
17:     end for
18: end procedure
```

given state s (see Sect. 5.5.2), i.e. one for each trigger, all transition aspects of s are included in a single state $\mathsf{Asp}_t(s)$.

5.6. Postprocessing

The weaving process so far introduces the Aspect state where the advices of applicable aspects are executed in parallel. The variables indicating the target of gotos are also set. Still necessary are transitions that are activated in dependence of these variables so that the gotos are actually done by the state machine.

This is done in Algorithm 5.20: for each (labeled) final state f of each aspect state Aspect, first, the state out of the same region of Aspect contained by $\mathsf{label}(f)$ is calculated and stored in variable g (line 8), this is the state to which a transition from Aspect is needed. If the label does not contain a state of the region, then no such state can be found ($g = \bot$). In this case, the default resumption strategy is followed, that is, the target of the advised transition is activated after the execution of Aspect (line 10).

A resumption strategy of going to state s is implemented by a transition leading to $\mathsf{jBefore}(s)$ (line 15) to ensure that whenever s is about to become active all ≪before≫ s aspects are executed, except when the aspect is implemented on the "before" section of s. In this case, the ≪before≫ s aspects have already been executed, and their results are already considered in g, so we introduce a transition that leads to s (line 13).

Function $\mathsf{GT}(\mathsf{Aspect})$ returns the target to go to: if no target is explicitly defined by a region of Aspect, then $\bot$ is returned, and Algorithm 5.20 interprets it

Algorithm 5.20 Postprocessing

1: **procedure** POSTPROCESSING(SM)
2: **for all** $(s \in \mathrm{S}_{\mathrm{orig}}(\mathsf{SM})$ **do** ▷ original states only
3: **for all** $\mathsf{Aspect} \in \mathsf{Asp}_w(s) \cup \{\mathsf{Asp}_t(s)\}$ **do**
4: $\mathrm{add}(\,\mathsf{defer}(\mathsf{Aspect}), *)$ ▷ declare the completion event as "deferrable"
5: $trDefault \leftarrow t \in \mathrm{T}(\mathsf{SM})$ where $\mathsf{source}(t) = \mathsf{Aspect} \wedge \mathsf{target}(t) = s$ or $\mathsf{source}(t) = \mathsf{Aspect} \wedge \mathsf{target}(t) = \mathsf{jBefore}(s)$ ▷ exactly one; this is the default transition
6: $\mathsf{guard}(trDefault) \leftarrow \mathrm{toUML}(\bigwedge \mathsf{GT}(\mathsf{Aspect}) \neq s \bigcup_{r \in \mathsf{region}(\mathsf{Aspect})} \mathrm{gtVars}(r)) \wedge \mathsf{guard}(trDefault)$ ▷ protect the default transition from being affected by the ones which are introduced from line 7 on.
7: **for all** $f \in \bigcup_{r \in \mathsf{region}(\mathsf{Aspect})} \mathsf{subvertex}(r)$, where $\mathrm{class}(f) = \mathsf{LabeledFinalState}$ **do**
8: $g \leftarrow \mathsf{label}(f) \cap \mathsf{subvertex}(\mathsf{container}(\mathsf{Aspect}))$ ▷ at most one!
9: **if** $g = \bot$ **then**
10: $g \leftarrow s$ ▷ default
11: **end if**
12: **if** $g = s \wedge \mathsf{Aspect} = \mathsf{Asp}_t(s)$ **then**
13: $v \leftarrow g$
14: **else**
15: $v \leftarrow \mathsf{jBefore}(g)$
16: **end if**
17: $t \leftarrow t' \in \mathrm{T}(\mathsf{SM}), \mathsf{source}(t') = \mathsf{Aspect}, \mathsf{target}(t') = g$ ▷ at most one!
18: **if** $t = \bot$ **then**
19: $\mathrm{insertTransition}(\mathsf{Aspect}, v, *, \mathsf{constraint}(f) \wedge \mathrm{toUML}(\mathsf{GT}(\mathsf{Aspect}) = g), \mathsf{skip})$
20: **else**
21: $\mathsf{guard}(t) \leftarrow \mathsf{guard}(t) \vee (\mathsf{constraint}(f) \wedge \mathrm{toUML}(\mathrm{GT}(\mathsf{Aspect}) = g))$
22: **end if**
23: **end for**
24: WEAVECONFLICTHANDLING(Aspect)
25: **end for**
26: **end for**
27: **end procedure**

(in line 10) as the default target (the target of the advised transition); if exactly one target is defined, then this is the target to go to after the execution of Aspect. A conflict is determined if at least two disjunct targets are defined. In this case, the current implementation raises an exception by indicating the base machine to enter the Err state (Algorithm 5.21, line 3). Other conflict resolutions can be easily realized by another implementation of Algorithm 5.21.

GT(Aspect) is defined as follows: if none of the resumption variables is true, then $\bot$ is returned; if on the highest priority level, exactly one resumption variable

Algorithm 5.21 Weaving conflict handling

```
1: procedure WEAVECONFLICTHANDLING(Aspect)
2:     r ← container(Aspect)
3:     insertTransition(Aspect, Err(r), ∗, GT(Aspect) = err, skip)
4: end procedure
```

is true, then this variable is returned; otherwise, i.e. if at least two different resumption variables on the highest priority level are true, then err is returned, indicating the state machine to enter an exception state or to handle the exception.

$$\mathsf{GT}(\mathsf{Aspect}) = \begin{cases} \bot, & \text{if } \neg\exists s \in \bigcup_{r\in\mathsf{region(Aspect)}} \mathrm{gtVars}(r) \cdot [\mathsf{gt}(s)] \\ g, & \text{if } \mathsf{gt}(g), g \in \bigcup_{r\in\mathsf{region(Aspect)}} \mathrm{gtVars}(r), \\ & \text{and } \forall s \in \bigcup_{r\in\mathsf{region(Aspect)}} \mathrm{gtVars}(r) \\ & \quad \cdot[(s \neq g \wedge \mathsf{pr}(s) \geq \mathsf{pr}(g)) \implies \neg\mathsf{gt}(s)] \\ \mathsf{err}, & \text{if } \exists g_1, g_2 \in \bigcup_{r\in\mathsf{region(Aspect)}} \mathrm{gtVars}(r) \\ & \quad \cdot[g_1 \neq g_2 \wedge \mathsf{pr}(g_1) = \mathsf{pr}(g_2) \wedge \mathsf{gt}(g_1) \wedge \mathsf{gt}(g_2) \wedge \\ & \quad \forall s \in \bigcup_{r\in\mathsf{region(Aspect)}} \mathrm{gtVars}(r) \\ & \quad \cdot[\mathsf{pr}(s) > \mathsf{pr}(g) \implies \neg\mathsf{gt}(s)]] \end{cases}$$

5.7. Correctness

Our weaving algorithms implement the informal semantics given in Sect. 4.3 correctly, we show this in the following theorem:

THEOREM 5.9 (Correctness of the weaving algorithms). *Given a state machine* SM *and a set* A *of aspects. Let the result of weaving the aspects contained in* A *to* SM *be* SM_A*. If* SM *and* SM_A *are in the same constellation, that is, if the active state configuration and the environment (the valuation of the variables) are the same in both* SM *and* SM_A*, then in the next execution step of* SM *some aspect* $\mathfrak{a} \in A$ *is activated iff. in the next execution step of* SM_A *some aspect state containing a region implementing* $\mathfrak{a}$ *is activated.*

PROOF. We first prove that when an aspect $\mathfrak{a}$ is activated by SM, then, in SM_A, an aspect state containing a region implementing $\mathfrak{a}$ is also activated. If $\mathfrak{a}$ is a ≪whilst≫ aspect, its activation is the result of SM being in a certain constellation (active state configuration and variable valuations) and handling a certain event. According to Algorithm 5.18, there is a transition t' in SM_A that should be fired in the same constellation upon the same event. Since t' leads to an aspect state containing (among others) an instance of $\mathfrak{a}$, this instance is activated in the next step. If $\mathfrak{a}$ is a transition aspect, then the activation of $\mathfrak{a}$ can be only caused by a change of the constellation of SM as a consequence of the firing of some transition $t \in \mathrm{T}(\mathsf{SM})$. Let t's corresponding transition in SM_A be t_A. Since SM and SM_A have the same constellation, t_A is now fired in SM_A. According to Algorithm 5.19, t_A leads to an aspect state Aspect, containing an instance of every aspect (in particular, $\mathfrak{a}$) that is applicable to t. Due to the same environment of SM and SM_A, the precondition of $\mathfrak{a}$ is also satisfied in SM_A. Therefore, the instance of $\mathfrak{a}$ in SM_A is activated.

On the other hand, suppose an instance R of some aspect $\mathfrak{a}$ is activated in SM_A. This means that $\mathfrak{a}$'s precondition is satisfied and that there is a transition t_A leading to some aspect state Aspect containing R is fired. Due to the same environment of SM and SM_A, the precondition is also satisfied in SM. Moreover, if $\mathfrak{a}$ is a ≪whilst≫ aspect, $\mathsf{trigger}(\mathsf{pointcut}(\mathfrak{a}))$ is the current event; if $\mathfrak{a}$ is a transition aspect, t_A's corresponding transition, t, should be fired in SM. In both cases, it holds that $\mathfrak{a}$ is activated. □

5.8. Implementation

Modeling with HILA and weaving of aspects were implemented prototypically in the tool Hugo/HILA, an extension of the UML translator and model checker Hugo/RT [45]. Aspects are given in an extension of the UTE format[2] and woven according to the algorithms described above. The weaving result is then output in UTE again.

5.8.1. Hugo/RT. Hugo is a translator and model checker for UML 1.x [56] state machines. It closes the gap between the state machine model, enhanced by the temporal language specification of the properties to verify, and the model checkers SPIN [36] and Uppaal [49]. Currently, Hugo/RT supports only UML 1.x.

Model. The input model, consisting of UML state machines, is specified in the ArgoUML[3] format .zargo, the MagicDraw[4], or Hugo/RT's (proprietary) textual format UTE. The state machines are then translated to kripke structures [13] that are accepted by the aforementioned model checkers. Simply spoken, Hugo/RT breaks down the hierarchical structure of UML state machines, calculates the (compound) transitions to fire, and updates the variables according the the UML Specification [56]. The translation details are beyond the scope of the thesis, the interested reader is referred to [44].

Specification. For the specification of system properties, both linear temporal logic (LTL) and computation tree logic (CTL) [48] can be used. Note that in Hugo/RT the operators G (always), F (eventually) and U (until) are defined on UML states instead of states of the transition system. Hugo/RT translates property specifications containing these operators transparently from the view of the user into temporal logic formulae w.r.t. the underlying kripke structure.

Moreover, Hugo/RT also supports the OCL statement inState, returning if a given state is active. This statement can currently only be used in a temporal logical specification, but not in the state machine. We therefore still need the preprocessing step "inserting trace variables" (Sect. 5.3.5) to make the states "know" which other states are also active.

Note, however, that Hugo/RT does not support the deferment of completion events. In Hugo/HILA we use "defer *" to defer completion events.

5.8.2. Weaving. We extended Hugo/RT by the functionality of reading HILA aspects and weaving them to a base machine. In fact, we extended the UTE format and its Hugo/RT parser to cover aspect definitions as well. Our extension also

[2] http://www.pst.ifi.lmu.de/projekte/hugo/#UTE
[3] http://www.argouml.org
[4] http://www.magicdraw.com/

allows the base machine to be given in UML 2.2.[5] The weaving algorithms, as presented above, are then applied to compose the aspects together with the base machine. The result of the weaving is output in a text file, also in the UTE format, containing elements of plain UML state machines only.

The implementation of the function toUML (p. 49) is straight forward, the required language elements of the expression language *Exp* and the action language *Act* (see Sect. 5.2.2), except for for or while loops, can be directly translated into UTE, see Table 5.1:

In Algorithms	toUML
addition, subtraction	+, -
boolean operations	&&, ||, !
case distinction	if . . . then . . . else

Table 5.1: Translation of the action language

Loops are not supported by UTE directly. We therefore implement loops by "brute-force": the weaver, instead of the weaving result, loops over all possible values of the loop variables, in each iteration the content of the loop is generated once, with the current value of the loop variable. This is possible because all the loops included in the algorithms that should be implemented in UTE have the form "for all $s \in S$", where S is determinable by statical analysis and does not change over the iterations.

Hugo/HILA weaves a set of aspects to a given base machine and outputs the result in another state machine, written in Hugo/RT's UTE format. The weaving result can then be model checked. Examples are given in the following Sect. 5.8.3 and Sect. 8.3.

5.8.3. Example. Figure 5.9 defines on the left hand side a state machine in UML 2.2, which contains a concurrent state X with two parallel regions. Each of the both regions contains two simple states. On the right hand side, an aspect in UTE format is shown. The aspect is actually an instance of the template given in Fig. 4.4, A being instantiated with A2 and B with B2.

Weaving Result. The result of weaving the aspect to the base machine, both of which are shown in Fig. 5.9 (in UML 1.x), is given in the following.

```
1  behaviour {
2    states {
3      initial INIT;
4      concurrent X {
5        entry in_X = true;
6        exit  last_X = true; in_X = false;
7              rv_0 = false; rv_1 = false;
8        composite a {
9          simple A1{
10           entry in_X_a_A1 = true;
```

[5] The differences of UML 2.2 [57] state machines and UML 1.x [56] state machines are only syntactic. We only had to define a wrapper to translate the UML 2.2 syntax to UML 1.x syntax.

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

```
aspect BeforeAspect {
  before config {
    state X.a.A2;
    state X.b.B2;
  }
  advice {
    states {
      initial AI;
      labeledfinal AaF {
        goto 0
      }
    }
    transitions {
      AI -> AaF{}
    }
  }
}
```

Figure 5.9: Example: weaving of a mutual exclusion aspect

```
11            exit  last_X_a_A2 = false;
12                  last_X_a_A1 = true;
13                  in_X_a_A1 = false;
14          }
15          simple A2{
16            entry in_X_a_A2 = true;
17            exit  last_X_a_A1 = false;
18                  last_X_a_A2 = true;
19                  in_X_a_A2 = false;
20          }
21          initial init;
22          junction _init_junction;
23          junction jBefore_A1;
24          junction jBefore_A2;
25          composite aspect_A2 {
26            defer *;
27            entry goto_X_a_A2 = false; pr_X_a_A2 = 0;
28            initial AI;
29            final AaF;
```

```
30            junction jAI;
31          }
32        }
33        composite b {
34          simple B1{
35            entry in_X_b_B1 = true;
36            exit  last_X_b_B2 = false;
37                  last_X_b_B1 = true;
38                  in_X_b_B1 = false;
39          }
40          simple B2{
41            entry in_X_b_B2 = true;
42            exit  last_X_b_B1 = false;
43                  last_X_b_B2 = true;
44                  in_X_b_B2 = false;
45          }
46          initial init;
47          junction _init_junction;
48          junction jBefore_B1;
49          junction jBefore_B2;
50          composite aspect_B2 {
51            defer *;
52            entry goto_X_b_B2 = false; pr_X_b_B2 = 0;
53            initial AI;
54            final AaF;
55            junction jAI;
56          }
57        }
58      }
59      junction jBefore_X;
60    }
61    transitions {
62      INIT -> jBefore_X {
63        effect rv_0 = true; rv_1 = true;}
64      jBefore_X -> X {}
65
66      X.a.init -> X.a._init_junction {}
67      X.a._init_junction -> X.a.jBefore_A1 {
68        guard rv_0;}
69      X.a._init_junction -> X.a.jBefore_A1 {
70        guard !rv_0;}
71      X.a.jBefore_A1 -> X.a.A1 {}
72      X.a.A1 -> X.a.jBefore_A2 {}
73
74      X.a.jBefore_A2 -> X.a.aspect_A2 {}
75
76      X.a.aspect_A2.AI -> X.a.aspect_A2.jAI {}
77      X.a.aspect_A2.jAI -> X.a.aspect_A2.AaF {
```

```
      effect goto_X_a_A2 = true;
             if (pr_X_a_A2 < 0) pr_X_a_A2=0;}
    X.a.aspect_A2.jAI -> X.a.aspect_A2.AaF {
      guard false;}
    X.a.aspect_A2 -> X.a.A2 {
      guard  in_X_b_B2 != true && goto_X_a_A2;}
    X.a.aspect_A2 -> X.a.A2 {
      guard !goto_X_a_A2;}

    X.a.A2 -> X.a.jBefore_A1 {}

    X.b.init -> X.b._init_junction {}
    X.b._init_junction -> X.b.jBefore_B1 {
      guard rv_1;}
    X.b._init_junction -> X.b.jBefore_B1 {
      guard !rv_1;}
    X.b.jBefore_B1 -> X.b.B1 {}
    X.b.B1 -> X.b.jBefore_B2 {}

    X.b.jBefore_B2 -> X.b.aspect_B2 {}

    X.b.aspect_B2.AI -> X.b.aspect_B2.jAI {}
    X.b.aspect_B2.jAI -> X.b.aspect_B2.AaF {
      effect goto_X_b_B2 = true;
             if (pr_X_b_B2 < 0) pr_X_b_B2=0;}
    X.b.aspect_B2.jAI -> X.b.aspect_B2.AaF {
      guard false;}
    X.b.aspect_B2 -> X.b.B2 {
      guard in_X_a_A2 != true && goto_X_b_B2;}
    X.b.aspect_B2 -> X.b.B2 {
      guard !goto_X_b_B2;}

    X.b.B2 -> X.b.jBefore_B1 {}
   }
}
```

Hugo/HILA also automatically generates statements of Hugo/RT for the initialization of the trace variables:

```
signature {
      attr in_X: boolean = false;
      attr last_X: boolean = false;
      attr in_X_a_A1: boolean = false;
      attr last_X_a_A1: boolean = false;
      attr in_X_a_A2: boolean = false;
      attr last_X_a_A2: boolean = false;
      attr in_X_b_B1: boolean = false;
      attr last_X_b_B1: boolean = false;
      attr in_X_b_B2: boolean = false;
```

```
    attr last_X_b_B2: boolean = false;
    attr goto_X_a_A2: boolean = false;
    attr goto_X_b_B2: boolean = false;
    attr rv_0: boolean = false;
    attr rv_1: boolean = false;
    attr pr_X_a_A2: int = 0;
    attr pr_X_b_B2: int = 0;
  }
```

Validation. The weaving result can be model checked by Hugo/HILA. In fact, the mutual exclusion of A2 and B2 is actually delivered by the aspect, i.e. the property `F (inState(X.a.A2) and inState(X.b.B2))` is falsified by Hugo/RT. On the other hand, (`A2|B2`) is the only state configuration excluded by the aspect. For example, the configuration (`A2|B1`) is still reachable, i.e. the property `F (inState(X.a.A2) and inState(X.b.B1))` is verified.

5.9. Discussion

Our weaving algorithms implement the informal semantics of HILA aspects presented in Chap. 4. The basic idea is to find out which aspects should be executed just before a certain state gets activated, and to weave these aspects into orthogonal regions of an aspect state spliced into the (exactly one) transition leading to that state. This way, all the aspects are executed in parallel, and the risk of inconsistency caused by shared joinpoints is minimized. Conflicts of resumption strategies are detected at runtime. Moreover, aspects can also be assigned priorities to eliminate resumption conflicts.

The "remaining risk" of conflicts is handled by a conflict handler generated by Algorithmus 5.21. Currently, the state machine enters a "local" error state in the region where the conflict is determined. An alternative would be a "global" error state for the whole state machine. Although this way we need only one error state and the weaving result is more clean, our solution as described above localizes the exception and allows the other parts, where no exception occurred, can be executed as usual. In any case, separating the generation of the conflict handler (in Algorithm 5.21) makes it possible to implement other conflict handling strategies easily.

In the algorithms presented above, statements "goto s" are actually implemented as transitions leading to the junction jBefore(s) except when s is the target of the advised transition and the resumption strategy is set by a transition aspect. This guarantees that before s becoming active, it is always checked if the precondition of any other aspect would be satisfied when s got active. Therefore, ≪before≫ aspects are *violation resistant* in the sense that, as long as its constraint is valuated true, an aspect ≪before≫ S will really be executed every time S is about to get active, no matter what other aspects are also defined for the base machine.

On the contrary, ≪after≫ aspects are not violation resistant, since there are situations that after configuration S has just been active, an ≪after≫ S aspect is not executed even if its constraint is satisfied, since S was left as the result of the execution of some ≪whilst≫ S aspect.

Theoretically, both ≪before≫ and ≪after≫ aspects may be implemented in both ways: violation resistant or not. However, in the case of ≪before≫ aspects, we

deem the violation resistant implementation to be the only desirable one. In concurrent systems, mutual exclusion is one of the most common and most important features. We therefore need a language construct which allows us to elegantly specify mutual exclusion, indepedently from other features that may or may not be modeled in aspects. On the other hand, the current implementation of ≪after≫ aspects is not the only choice. A violation resistant implementation may also make sense. The realization is straight forward. It is only necessary to extend Algorithm 5.18 and insert the ≪after≫ aspects into $\mathsf{Asp}_w(s)$.

Obviously ≪whilst≫ aspects are violation resistant, i.e. they are always executed whenever the specified configuration is active, the specified trigger is the current event, and the constraint is satisfied. The reason is that the joinpoints of ≪whilst≫ aspects are disjunct to those of ≪before≫ and ≪after≫ aspects, and ≪whilst≫ aspects catching the same joinpoints (same configuration, same trigger) are woven into parallel regions and executed in parallel.

As stated in Sect. 4.1, HILA aspects can be regarded as event action rules: an aspect is a statement in the form of "when proposition P is satisfied, then do action A". In this sense, the violation resistance of ≪before≫ and ≪whilst≫ aspects as discussed above guarantees that whenever P is satisfied A is always executed. However, it is still possible for P to be made unsatisfiable by other aspects. After presenting the formal semantics of HILA aspects in the next chapter, where, in particular, the violation resistance of ≪before≫ and ≪whilst≫ aspects is discussed on a more formal basis, we discuss in Chap. 7 under which circumstances the precondition P may be made unsatisfiable.

Part 3

HiLA d'Ivoire

CHAPTER 6

Formal Semantics

Contents

6.1. Abstract Transition Systems **81**
6.2. UML State Machines **82**
6.3. History Properties **84**
6.4. Structural Extension by ≪whilst≫Aspects **85**
6.5. Behavior Extension **86**
6.5.1. Pointcut 86
6.5.2. Aspect Instance 86
6.5.3. Advice 88
6.6. Weaving and Semantics **90**
6.7. Discussion **92**

We present a formal semantics of executing state machines with history properties and aspects in the style of structural operational rules, with reflective execution of aspects. In order to simplify the semantic rules and to concentrate on history properties and aspects, we do not include a detailed semantical discussion on the run-to-completion semantics of UML state machines, but rather rely on an abstract transition selection function which ensures that, in accordance with the UML specification [57], a maximal consistent, prioritized, conflict-free set of transitions in a state machine for executing an event is chosen (for possible semantical choices see, e.g. [71, 44]). The concurrency of the state machine is semantically represented by sets of transitions operating concurrently on a set of states for a single execution step.

6.1. Abstract Transition Systems

We use transition systems to model state machines. In our transition systems, it is not specified concretely how to determine which transitions should be fired. This is modeled by an abstract "transition selection" function: given a configuration of a transition system, a function θ delivers the transitions to fire. In this sense, our transition systems are abstract. Only a concrete θ will make such a transition system concrete and executable.

For modeling our abstract transition systems, we assume a set of events $\mathscr{E}$ which comprises an empty event $*$; a language of propositions $\mathscr{P}$, from which in particular guards are drawn, and which has the boolean constants true and false and a negation operator $\neg$; and a language of actions $\mathscr{A}$ which includes a skip-operation ε and shows a sequential composition operator ;. A transition system is executed in an environment from a domain $\mathscr{N}$ which contains abstract representations of the valuation of contextual attributes and of an event pool for the transition system. For $\eta \in \mathscr{N}$, $e \in \mathscr{E}$ and $g \in \mathscr{P}$, we write $\eta \models e[g]$ to mean that e is the current

event in the environment η and the proposition g evaluates to true. For $\eta, \eta' \in \mathscr{N}$, $e \in \mathscr{E}$ and $a \in \mathscr{A}$, we write $\eta, \eta' \models e/a$ to mean that e is the current event in environment η and that η evolves to η' by consuming e and executing a.

A transition system M is given by a set $S(M)$ of states, a set $T(M)$ of transitions, an initial state $i_M \in S(M)$ and a set $F_M \subseteq S(M)$ of final states. A transition $t \in T(M)$ is given by a source state $\sigma_t \in \mathrm{S}(M)$, a triggering event $e_t \in \mathscr{E}$, a guarding proposition $g_t \in \mathscr{P}$, an effect action $a_t \in \mathscr{A}$, and a target state $\sigma'_t \in \mathrm{S}(M)$.

For set S of states and set T of transitions we define $\sigma_T := \{\sigma_t \mid t \in T\}$ and $\sigma'_T := \{\sigma'_t \mid t \in T\}$. We also define $\sigma'(S,T) := (S \setminus \sigma_T) \cup \sigma'_T$ and $\mathcal{R}_*(T) := T \cup \{t \mid \sigma_t \in \sigma'_T \wedge e_t = *\}$, and write $\mathcal{R}^+_*(T)$ to mean the transitive closure of $\mathcal{R}_*(T)$. Moreover, we define $\mathcal{R}_*(S) := (S \setminus \sigma_T) \cup \sigma'_T$, where $T = \{t \mid \sigma_t \in S \wedge e_t = *\}$, and write $\mathcal{R}^+_*(S)$ to mean the transitive closure of $\mathcal{R}_*(S)$.

We write $\mathbf{A}(T)$ for the possible sequential compositions of the actions of the transitions in T, i.e. $\mathbf{A}(T) = \{a_{t_1}\,;\ldots;\,a_{t_n} \mid \{t_1,\ldots,t_n\} = T\}$. Each transition system is equipped with a transition selection function

$$\theta_M : \wp(S_M) \times \mathscr{N} \times \mathscr{E} \to \wp(T_M)$$

such that $\theta_M(S,\eta,e)$ represents the active transition set for executing the event e in environment η starting in the state configuration S. We require that if $T = \theta_M(S,\eta,e)$ then for all $t \in T$: (i) $\sigma_t \in S$; (ii) $e_t = e$; (iii) $\eta \models e[g_t]$.

With these preliminaries, the execution of a state machine M can be captured by the rule

$$\text{(mch)} \qquad \langle (M,\eta),(M,\eta')\rangle \models S \xrightarrow{e:T} S',$$
$$\text{if } T = \mathcal{R}^+_*(\theta_M(S,\eta,e)),\ a \in \mathbf{A}(T),\ \eta,\eta' \models e/a \text{ and } S' = \mathcal{R}^+_*(\sigma'(S,T)).$$

stating that the transition system M starting in environment η with states $S \subseteq \mathrm{S}(M)$ being active moves, by firing a suitable set of transitions T from $\mathrm{T}(M)$ for some event e, to environment η' and state configuration S' consuming event e and executing the action a which is an interleaving of all actions executed by transitions in T. T includes all transitions that are selected by θ and all their "following" transitions with empty event $*$.

6.2. UML State Machines

UML state machines are also represented by abstract transition systems, where we assume that the concrete implementation of θ returns, as required by the UML Specification [57], the maximal consistent, prioritized, conflict free set of transitions, without specifying the θ implementation concretely. In the following, we describe how a state machine SM is represented by such a transition system.

We assume that our UML state machines have been normalized in our preprocessing process (Sect. 5.3), but without before sections or trace variables inserted. In particular, such a state machine SM contains the following pseudostates:

- an initial vertex in each region,
- the junctions following the initial vertices (see Sect. 5.3.2),
- join vertices in the original state machine (join vertices are not handled by the preprocessing).

The corresponding transition system $\mathsf{TS}_{\mathsf{SM}}$ shares the sets of events, propositions and the action language with SM. In the following, we show the construction

of the sets of states and transitions of $\mathsf{TS}_{\mathsf{SM}}$. The basic idea, put very simply, is to represent SM's simple states and initial vertices[1] by states of $\mathsf{TS}_{\mathsf{SM}}$, and to include in $\mathsf{TS}_{\mathsf{SM}}$ also "transit states" to represent the entrance and exits of composite states. For each composite state s there is a state representing s being entered, and for each pair (s,t) of a composite state s and a transition t leaving s there is a state representing s being left as a result of t being fired.

More precisely, $S(\mathsf{TS}_{\mathsf{SM}})$ is the smallest set that satisfies the following:

(a) For each $x \in \mathsf{I}(\mathsf{SM}) \cup \mathrm{S}(\mathsf{SM})$ there is a unique state $\mathsf{s}_x \in S(\mathsf{TS}_{\mathsf{SM}})$. We require $\mathsf{s}_x \neq \mathsf{s}_y$ if $x \neq y$, and write $\mathsf{s}^{-1}(\mathsf{s}_x) := x$.

(b) For each transition t with $s \in \mathrm{S}_{\mathsf{comp}}(\mathsf{SM})$ where $s := \mathsf{source}_{\mathsf{struct}}(t)$, there is a state $\mathsf{s}_{s,t} \in S(\mathsf{TS}_{\mathsf{SM}})$. We further define S_s to be the set of all states matching $\mathsf{s}_{s,\_}$, i.e. $\mathsf{S}_s := \{\mathsf{s}_{s,t} \mid \mathsf{source}_{\mathsf{struct}}(t) = s\}$.

$T(\mathsf{TS}_{\mathsf{SM}})$ is the smallest set which satisfies the following:

(1) For each source structured transition t such that $\mathsf{source}(t) \in \mathsf{I}(\mathsf{SM}) \cup \mathrm{S}_{\mathsf{simple}}(\mathsf{SM})$ and $\mathsf{target}(t) \in \mathrm{S}(\mathsf{SM})$ there is a transition $\mathsf{t}_t \in T(\mathsf{TS}_{\mathsf{SM}})$ with $\sigma_{\mathsf{t}_t} = \mathsf{s}_{\mathsf{source}(t)}$, $\sigma'_{\mathsf{t}_t} = \mathsf{s}_{\mathsf{target}(t)}$, $e_{\mathsf{t}_t} = \mathsf{trigger}(t)$, $g_{\mathsf{t}_t} = \mathsf{guard}(t)$, and $a_{\mathsf{t}_t} = \mathsf{effect}(t)$.

(2) For each source structured transition t with $s \in \mathrm{S}_{\mathsf{comp}}(\mathsf{SM})$, where $s := \mathsf{source}_{\mathsf{struct}}(t) = \mathsf{source}(t)$, there is a transition $\mathsf{t}_{s,t} \in T(\mathsf{TS}_{\mathsf{SM}})$ with $\sigma_{\mathsf{t}_{s,t}} = \mathsf{s}_{s,t}$, $e_{\mathsf{t}_{s,t}} = \mathsf{trigger}(t)$, $g_{\mathsf{t}_{s,t}} = \mathsf{guard}(t)$, $a_{\mathsf{t}_{s,t}} = \mathsf{effect}(t)$ and $\sigma'_{\mathsf{t}_{s,t}} = x$, where $x := \mathsf{s}_{\mathsf{target}(t)}$.

(3) For each source unstructured transition t with $s \in \mathrm{S}_{\mathsf{comp}}(\mathsf{SM})$, where $s := \mathsf{source}_{\mathsf{struct}}(t) \neq \mathsf{source}(t)$, there is a transition $\mathsf{t}_{s,t} \in T(\mathsf{TS}_{\mathsf{SM}})$ with $\sigma_{\mathsf{t}_{s,t}} = \mathsf{s}_{s,t}$, $e_{\mathsf{t}_{s,t}} = *$, $g_{\mathsf{t}_{s,t}} = \mathsf{true}$, $a_{\mathsf{t}_{s,t}} = \mathsf{skip}$ and $\sigma'_{\mathsf{t}_{s,t}} = x$, where $x := \mathsf{s}_{\mathsf{target}(t)}$ if $\mathrm{isState}(\mathsf{target}(t))$ and $x := \mathsf{s}_{\mathsf{target}(t')}$ if $\mathrm{isJoin}(\mathsf{target}(t))$ and $t' \in \mathrm{T}(\mathsf{SM}) \wedge \mathsf{source}(t') = \mathsf{target}(t)$.

(4) For each pair (s,i) where the following holds: $s \in \mathrm{S}_{\mathsf{comp}}(\mathsf{SM})$, $i \in \mathsf{I}(\mathsf{SM})$, and $\mathsf{state}(\mathsf{container}(i)) = s$, there is a transition $\mathsf{t}_{s,i}$ with $\sigma_{\mathsf{t}_{s,i}} = \mathsf{s}_s$, $\sigma'_{\mathsf{t}_{s,i}} = \mathsf{s}_i$, $e_{\mathsf{t}_{s,i}} = *$, $g_{\mathsf{t}_{s,i}} = \mathsf{true}$, and $a_{\mathsf{t}_{s,i}} = \mathsf{skip}$.

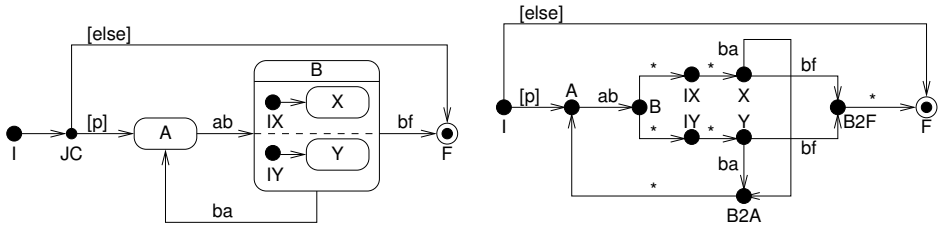
(5) For each pair (s,t) where it holds $\mathsf{source}_{\mathsf{struct}}(t) \in \mathrm{S}_{\mathsf{comp}}(\mathsf{SM})$ and $s \in \mathrm{S}_{\mathsf{simple}}(\mathsf{SM}) \cap \mathsf{subvertex}^+(\mathsf{source}_{\mathsf{struct}}(t))$, there is a transition $\mathsf{t}_{s,t}$ where $\sigma_{\mathsf{t}_{s,t}} = \mathsf{s}_s$, $\sigma'_{\mathsf{t}_{s,t}} = \mathsf{s}_{\mathsf{source}_{\mathsf{struct}}(t),t}$, $e_{\mathsf{t}_{s,t}} = \mathsf{trigger}(t)$, guard $a_{\mathsf{t}_{s,t}} = \mathsf{guard}(t)$ and $a_{\mathsf{t}_{s,t}} = \mathsf{effect}(t)$.

(6) For each pair (t_1,t_2) with $\mathsf{target}(t_1) = \mathsf{source}(t_2)$, $\mathrm{isJunction}(j)$, $j := \mathsf{target}(t_1)$, and $j = \mathrm{inTr}(r)$ for some region r, there is a transition $t \in T(\mathsf{TS}_{\mathsf{SM}})$, with $\sigma_t = \mathsf{s}_{\mathsf{source}(t_1)}$, $\sigma'_t = \mathsf{s}_{\mathsf{target}(t_2)}$, $e_t = *$, $g_t = \mathsf{guard}(t_2)$, and $a_t = \mathsf{effect}(t_2)$.

The behavior of state machine SM is then specified by the rule (mch) with $M = \mathsf{TS}_{\mathsf{SM}}$. Note in the representation of UML state machines the transition selection function θ remains unspecified. We just require that θ respect the regions of SM and that the transitions selected by θ represent a maximal consistent, prioritized, conflict-free set of transitions of SM.

EXAMPLE 6.1 (Representation of UML state machines as transition systems). Figure 6.1a shows a UML state machine. For illustration purposes, each of the

[1] Junction and join vertices are ignored, the transitions entering and leaving junctions or joins are connected. See rules 3 and 6 of the construction of $T(\mathsf{TS}_{\mathsf{SM}})$.



(a) UML state machine (b) Semantics as a transition system

Figure 6.1: Example: UML state machine as transition system

initial and junction vertices was given a name and notated beneath the vertex. Its semantics as a transition system is given in Fig. 6.1b. Note that

- the junction JC is not represented in the transition system,
- the non-simple state B is not represented directly, but indirectly as follows:
  - state B (in Fig. 6.1a) is active iff. at least one of the states X and Y in Fig. 6.1b is active,
  - entering the UML state B (in Fig. 6.1a) is represented explicitly by an "entrance state" (B) in Fig. 6.1b,
  - exiting the UML state B (in Fig. 6.1a) is represented by several "exit states", one for each transition leaving B. In Fig. 6.1b, state B2F represents B being left as the result of the transition to the final state F being triggered by an event bf, and state B2A represents B being left as the result of the transition to state A being triggered by an event ba,
  - in Fig. 6.1b, the states B, IX, IY are only "transit" states: as soon as they get active, the outgoing *-transitions are activated.

6.3. History Properties

A history state machine (M, H) consists of a state machine M and a sequence of pairs of (c_i, en_i), where c_i is a state configuration and en_i is the environment, i.e. the valuation of all (non-history) properties of M at the moment where c_i is the current active state configuration. When M is executed, H is updated after each execution step. Semantically, this can be described as

(hist)
$$\frac{\langle (M,\eta),(M,\eta')\rangle \models S \xrightarrow{e:T} S'}{\langle (M,H,\eta),(M,H',\eta')\rangle \models S \xrightarrow{e:T} S'} \quad \text{where } H' = H \cdot (S',\eta')$$

Determining the value of a history property p thus amounts to using pattern matching p on H according to the multiplicity constraints of p. Adding a history property to a state machine does not, in itself, enable any new behavior for the augmented state machine, since rule (hist) does not permit exploitation of the stored history. The reason for introducing history properties is that we can then use them in aspects to modify the behavior of the base state machine.

Note that by virtue of the rules, history state machines can be viewed as basic state machines. Thus, for applying aspects to state machines, we need not differentiate between basic state machines and history state machines.

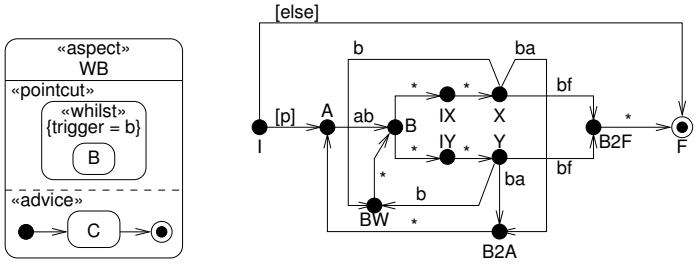
6.4. Structural Extension by ≪whilst≫ Aspects

Semantically, ≪whilst≫ aspects define a syntactical extension of the base machine as well as modifications of its dynamic behavior. In this section, we describe the syntactical extension defined by ≪whilst≫. Put very simply, a ≪whilst≫ aspect w extends the base machine by a transition, which reacts to $\mathsf{trigger}(w)$, for each state contained in $\mathsf{config}(w)$. The dynamic semantics of ≪whilst≫ aspects will be handled together with other dynamic HILA aspects in the following Sect. 6.5.

Given a transition system $\mathsf{TS_{SM}}$, representing a UML state machine SM, the application of a ≪whilst≫ aspect w to $\mathsf{TS_{SM}}$, notated by $\mathsf{TS^w_{SM}}$, is a transition system with $S(\mathsf{TS^w_{SM}}) = S(\mathsf{TS_{SM}}) \cup S'$ and $T(\mathsf{TS^w_{SM}}) = T(\mathsf{TS_{SM}}) \cup T^w_{\mathsf{SM}}$ where S' and T^w_{SM} are the smallest sets which satisfy the following:

- For each $s \in \mathsf{config}(w) \wedge \mathrm{isSimple}(s)$, there is a transition $\mathsf{t}_s \in T^w_{\mathsf{SM}}$ such that $\sigma_{\mathsf{t}_s} = \sigma'_{\mathsf{t}_s} = \mathsf{s}_s$, $e_t = \mathsf{trigger}(w)$, $g_{\mathsf{t}_s} = \mathsf{constraint}(w)$, and $a_{\mathsf{t}_s} = \mathsf{skip}$.
- For each $s \in \mathsf{config}(w) \wedge \mathrm{isComposite}(s)$, there is a state $\mathsf{w}_{s,w} \in S'$, and a transition $\mathsf{t}_{s,w} \in T^w_{\mathsf{SM}}$ such that $\sigma_{\mathsf{t}_{s,w}} = \mathsf{w}_{s,w}$, $\sigma'_{\mathsf{t}_{s,w}} = \mathsf{s}_s$, $e_{\mathsf{t}_{s,w}} = *$, $g_{\mathsf{t}_{s,w}} = \mathsf{true}$, $a_{\mathsf{t}_{s,w}} = \mathsf{skip}$. We further define W_s to be the set of all states matching $\mathsf{w}_{s,\_}$, i.e. $\mathsf{W}_s := \{\mathsf{w}_{s,t} \mid \mathsf{source}(t) = s\}$.
- For each $s \in \mathsf{config}(w) \wedge \mathrm{isComposite}(s)$, let S be the set of all simple states (directly or recursively) contained in s, i.e. $S := \{s' \mid s' \in \mathsf{subvertex}^+(s) \wedge \mathrm{isSimple}(s')\}$, there is for each $s' \in S$ a transition $\mathsf{t}_{s',w} \in T^w_{\mathsf{SM}}$ with $\sigma_{\mathsf{t}_{s',w}} = \mathsf{s}_{s'}$, $\sigma'_{\mathsf{t}_{s',w}} = \mathsf{w}_{s,w}$, $e_{\mathsf{t}_{s',w}} = \mathsf{trigger}(w)$, $g_{\mathsf{t}_{s',w}} = \mathsf{constraint}(w)$, and $a_{\mathsf{t}_{s',w}} = \mathsf{skip}$.

Given two ≪whilst≫ aspects w_1 and w_2, we notate the result of first extending $\mathsf{TS_{SM}}$ by w_1 and then by w_2 as $\mathsf{TS^{w_1,w_2}_{SM}}$, and the result of first extending $\mathsf{TS_{SM}}$ by w_2 and then by w_1 as $\mathsf{TS^{w_2,w_1}_{SM}}$. Obviously it holds that $\mathsf{TS^{w_1,w_2}_{SM}} = \mathsf{TS^{w_2,w_1}_{SM}}$. Given a set W of ≪whilst≫ aspects, we write $\mathsf{TS^W_{SM}}$ to refer to the result of extending $\mathsf{TS_{SM}}$ by all aspects contained in W in an arbitrary order.



(a) Whilst aspect (b) Extended transition system

Figure 6.2: Example: structural extension by ≪whilst≫ aspects

EXAMPLE 6.2 (Structural extension by ≪whilst≫ aspects). Figure 6.2a contains a ≪whilst≫ aspect WB. It defines, besides a behavior extension as will be described below, a structural extension of the base machine. When WB is applied to the base machine SM given in Fig. 6.1a (i.e. the transition system shown in Fig. 6.1b), the transition system is extended to Fig. 6.2b, where new transitions with trigger b leaving X and Y and leading to state BW, as well as a transition from BW to B, are introduced. Note the advice is not woven into the transition system. In HILA, advices are not understood as syntactical modifications.

6.5. Behavior Extension

To define the semantics of aspects we extend the rule (mch) to operate on a pair consisting of a transition system and a set of (concrete) aspect instances which we will proceed to define.

6.5.1. Pointcut. The pointcut of an aspect determines on which transitions of the base machine the aspect may be executed. On these transitions it must be checked at runtime whether the advice of the aspect should be executed or not.

Given a base machine SM (and its transition system $\mathsf{TS}_{\mathsf{SM}}$) and an aspect $\mathfrak{a}$, the semantics of $\mathsf{pointcut}(\mathfrak{a})$ is a set $\mathsf{PC}_{\mathfrak{a}}$ of transitions. We first define some auxiliary functions:

- For a set of states $X \subseteq \mathrm{S}(\mathsf{SM})$,
  $\mathsf{Simple}(X) := \{s \in X \mid \mathrm{isSimple}(s)\}$,
- For a set of states $X \subseteq \mathrm{S}(\mathsf{SM})$,
  $\mathsf{M}_{\mathsf{before}}(X) := \bigcup_{s \in X} \mathsf{s}_s$.
- For a state $s \in \mathrm{S}(\mathsf{SM})$,
  $m^{\mathsf{tr}}_{\mathsf{out}}(s) := \bigcup_{s \in \mathsf{subvertex}^*(s')} \mathsf{S}_{s'}$.
- For a state $s \in \mathrm{S}(\mathsf{SM})$,
  $m^{\mathsf{wh}}_{\mathsf{out}}(s) := \bigcup_{s \in \mathsf{subvertex}^*(s')} \mathsf{W}_{s'}$.
- For a set of states $X \subseteq \mathrm{S}(\mathsf{SM})$,
  $\mathsf{T}_{\mathsf{before}}(X) := \{t \in T(\mathsf{TS}_{\mathsf{SM}}) \mid \sigma'_t \in \mathsf{M}_{\mathsf{before}}(X)\}$.
- For a set of states $X \subseteq \mathrm{S}(\mathsf{SM})$,
  $\mathsf{T}_{\mathsf{after}}(X) :=$
  $\{t \in T(\mathsf{TS}_{\mathsf{SM}}) \mid (\sigma_t \in \bigcup_{s \in \mathsf{Simple}(X)} \{\mathsf{s}_s\} \wedge \sigma'_t \notin \bigcup_{s \in \mathsf{Simple}(X)} m^{\mathsf{tr}}_{\mathsf{out}}(s))$
  $\vee\ \sigma_t \in \bigcup_{s \in X} m^{\mathsf{tr}}_{\mathsf{out}}(s)\}$.
- For a set of states $X \subseteq \mathrm{S}(\mathsf{SM})$,
  $\mathsf{T}_{\mathsf{whilst}}(X) :=$
  $\{t \in T(\mathsf{TS}_{\mathsf{SM}}) \mid (\sigma_t \in \bigcup_{s \in \mathsf{Simple}(X)} \{\mathsf{s}_s\} \wedge \sigma'_t \notin \bigcup_{s \in \mathsf{Simple}(X)} m^{\mathsf{wh}}_{\mathsf{out}}(s))$
  $\vee\ \sigma_t \in \bigcup_{s \in X} m^{\mathsf{wh}}_{\mathsf{out}}(s)\}$.

$\mathsf{PC}_{\mathfrak{a}}$ is a set of transitions, defined as follows:

$$\mathsf{PC}_{\mathfrak{a}} := \begin{cases} \mathsf{T}_{\mathsf{before}}(\mathsf{config}(\mathfrak{a})), & kind(\mathfrak{a}) = \mathsf{before} \\ \mathsf{T}_{\mathsf{after}}(\mathsf{config}(\mathfrak{a})), & kind(\mathfrak{a}) = \mathsf{after} \\ \mathsf{T}_{\mathsf{before}}(\mathsf{tgt}(\mathfrak{a})) \cap \mathsf{T}_{\mathsf{after}}(\mathsf{src}(\mathfrak{a})), & kind(\mathfrak{a}) = \mathsf{between} \\ \mathsf{T}_{\mathsf{whilst}}(\mathsf{config}(\mathfrak{a})), & kind(\mathfrak{a}) = \mathsf{whilst} \end{cases}$$

6.5.2. Aspect Instance. Let $\mathcal{A}$ be the set of all aspects. Several instances of an aspect may be active simultaneously; to avoid confusing these instances we rename the states of each aspect before activating it. Therefore, we assume an

infinite set of states, $\mathcal{S}$, and an infinite set of variables $\mathcal{V}$. We assume the two sets are disjoint from all other states and variables, and that each invocation of the operations $\mathrm{new}(\mathcal{S})$ and $\mathrm{new}(\mathcal{V})$ returns a fresh element of $\mathcal{S}$ and $\mathcal{V}$, respectively. For an aspect $\mathfrak{a} \in \mathcal{A}$ we write $\mathfrak{a}^{\mathcal{S}}$ for the state machine obtained by replacing each state in $\mathsf{TS}_{\mathsf{advice}(\mathfrak{a})}$ with a fresh state, $\mathrm{new}(\mathcal{S})$, and all transitions in $\mathsf{TS}_{\mathsf{advice}(\mathfrak{a})}$ with equivalent transitions between the new states.

Given a state machine SM (and its transition system representation $\mathsf{TS}_{\mathsf{SM}}$), a state $z \in S(\mathsf{TS}_{\mathsf{SM}})$ and an aspect $\mathfrak{a}$, we define $\mathbf{c}(z, \mathfrak{a})$ to be an instantiation of aspect $\mathfrak{a}$ *guarding* state z, which is a pair $\mathbf{c}(z, \mathfrak{a}) := (v, \mathsf{T})$, where $v \in \mathcal{V}$ and T is a transition system representing a state machine. We also refer to the first projection of $\mathbf{c}(z, \mathfrak{a})$ as $\mathbf{s_f}(\mathbf{c}(z, \mathfrak{a}))$. The name of v and the structure of T are defined by the constructor $\mathrm{new}_c(z, \mathfrak{a}) := (\mathrm{new}(\mathcal{V}), \mathsf{subst}(\mathsf{TS}_{\mathfrak{a}^{\mathcal{S}}}, z))$, where $\mathsf{TS}_{\mathfrak{a}^{\mathcal{S}}}$ is the transition system representation of $\mathfrak{a}^{\mathcal{S}}$, and $\mathsf{subst}(\mathsf{TS}_{\mathfrak{a}^{\mathcal{S}}}, z)$ is the state machine obtained by extending the effect of each transition entering a final state by an assignment of $\mathbf{s_f}(\mathbf{c}(z, \mathfrak{a}))$. That is, we define $R_{\mathbf{c}(z)} \subseteq S(\mathsf{TS}_{\mathsf{SM}})$ to be the set of (transition system) states representing the (UML) states in the same region of the state represented by z: $R_{\mathbf{c}(z)} := \{\mathsf{s}_s \mid s \in \mathrm{S}(\mathsf{SM}) \wedge \mathsf{region}(s) = \mathsf{region}(\mathsf{s}^{-1}(z)\}$. Then $\mathsf{subst}(\mathsf{TS}_{\mathfrak{a}^{\mathcal{S}}}, z)$ is defined as $\mathsf{subst}(\mathsf{TS}_{\mathfrak{a}^{\mathcal{S}}}, z) := \mathfrak{a}^{\mathcal{S}}[a_t/a_t \cdot \mathbf{s_f}(\mathbf{c}(z, \mathfrak{a})) \leftarrow \mathsf{M}_{\mathsf{before}}(\mathsf{label}(\mathsf{s}^{-1}(\sigma'_t))) \cap R_{\mathbf{c}(z)}]$ for each transition t such that $\mathrm{isFinal}(\mathsf{s}^{-1}(\sigma'_t))$. Since in a label, there is at most one state in each region (constraint 4.3.6), the set $\mathsf{M}_{\mathsf{before}}(\mathsf{label}(\mathsf{s}^{-1}(\sigma'_t))) \cap R_{\mathbf{c}(z)}$ contains at most one element. This element is then assigned to $\mathbf{s_f}(\mathbf{c}(z, \mathfrak{a}))$. If the set is empty, we say $\mathbf{s_f}(\mathbf{c}(z, \mathfrak{a})) = \epsilon$.

Furthermore, we define $\mathsf{z}(\mathbf{c}(z, \mathfrak{a})) := z$, $\mathsf{ap}(\mathbf{c}(z, \mathfrak{a})) := \mathfrak{a}$, $S(\mathbf{c}(z, \mathfrak{a})) := S(\mathsf{subst}(\mathfrak{a}^{\mathcal{S}}, z))$. We also define $i_{\mathbf{c}(z,\mathfrak{a})}$ to be the instantiation of the initial state of the advice of $\mathfrak{a}$, and $F_{\mathbf{c}(z,\mathfrak{a})}$ to be the set of the instances of the final states of the advice of $\mathfrak{a}$.

For a given set C of aspect instances, we refer to the subset of C including all transition aspects guarding state z as $\mathsf{trans}_{C,z} := \{\mathbf{c} \in C \mid \mathsf{z}(\mathbf{c}) = z \wedge kind(\mathsf{ap}(\mathbf{c})) \in \{\mathsf{before}, \mathsf{after}, \mathsf{between}\}\}$, and the subset containing all configuration aspects guarding z as $\mathsf{whilst}_{C,z} := \{\mathbf{c} \in C \mid \mathsf{z}(\mathbf{c}) = z \wedge kind(\mathsf{ap}(\mathbf{c})) = \mathsf{whilst}\}$. For a given event ev, we define $\mathsf{whilst}^{ev}_{C,z} := \{w \in \mathsf{whilst}_{C,z} \mid \mathsf{trigger}(w) = ev\}$. Obviously, it holds $\mathsf{whilst}_{C,z} = \bigcup_{ev} \mathsf{whilst}^{ev}_{C,z}$ and $\mathsf{whilst}^{ev_1}_{C,z} \neq \mathsf{whilst}^{ev_2}_{C,z}$ for each pair of $ev_1 \neq ev_2$. We also call each $\mathsf{trans}_{C,z}$ and each $\mathsf{whilst}^{ev}_{C,z}$ an aspect group.

We write $\mathsf{ctnr}(z)$ to mean $\mathsf{container}(\mathsf{s}^{-1}(z))$ and define for any set of aspect instances Γ_z such that $\forall \gamma \in \Gamma_z \cdot [\mathsf{z}(\gamma) = z]$

$$\mathbf{s_f}(\Gamma_z) := \begin{cases} z & \text{if } (\forall \gamma \in \Gamma_z) \cdot [\mathbf{s_f}(\gamma) = \epsilon], \\ s & \text{if } (\exists \gamma \in \Gamma_z) \cdot [\mathbf{s_f}(\gamma) = s] \wedge \\ & \quad (\forall \gamma' \in \Gamma_z, \mathsf{priority}(\mathsf{ap}(\gamma')) > \mathsf{priority}(\mathsf{ap}(\gamma))\cdot \\ & \quad\quad [\mathbf{s_f}(\gamma') = \epsilon]) \wedge \\ & \quad (\forall c' \in \Gamma_z, \mathsf{priority}(\mathsf{ap}(\gamma')) = \mathsf{priority}(\mathsf{ap}(\gamma))\cdot \\ & \quad\quad [\mathbf{s_f}(\gamma') = s \vee \mathbf{s_f}(\gamma') = \epsilon]]), \\ \mathsf{Err}(\mathsf{ctnr}(z)) & \text{otherwise.} \end{cases}$$

For each such Γ, if $\#(\Gamma) = n$, we also refer to the elements of Γ as $\mathbf{c}^1_\Gamma, \ldots, \mathbf{c}^n_\Gamma$.

6.5.3. Advice. With the above preparations, we are now in the position to define the semantics of how the advices are executed, and, after their termination, how the base machine execution is resumed.

In general, an advised state machine is executed with a set of aspect instances. Given a state machine SM, a finite set $A = \{A_1, \ldots, A_k\}$ of HILA aspects, where $W \subseteq A$ maximal with $\forall w \in W \cdot [kind(w) = \mathsf{whilst}]$, we define M to be the transition system representing SM, structurally enhanced by the $\ll\mathsf{whilst}\gg$ aspects: $M := \mathsf{TS}^W_{\mathsf{SM}}$.

The execution of the advised state machine then amounts to the execution of M with a set of aspect instances, as specified by

$$\langle (M, C, \eta), (M, C', \eta') \rangle \models S \xrightarrow{e:T} S' \tag{asp}$$

where M is the state machine, C is the set of the currently active aspect instances, and η is the environment. T, the transitions to fire, is defined as

$$T := (T_{inner} \setminus T_{rem}) \cup T_{add} \cup T_{res}$$

The transitions contained in the T_{inner} are "inner" transitions within the base machine or one of the aspects. That is, no transition contained in this component leaves one state in the base machine and leads to one in an aspect or vice versa. Its first component T_{base} is the set of base machine transitions that would be fired if no aspect were applied to the base machine. The second is the union of all transitions being fired in the aspects (i.e. their advices).

$$T_{inner} := T_{base} \cup \Big(\bigcup_{\mathbf{c} \in C} \theta_{\mathbf{c}}(S \cap S(\mathbf{c}), \eta, e) \Big), T_{base} := \theta_M(S \cap S(M), \eta, e)$$

T_{rem} is a subset of T_{base}, containing the transitions that are advised by the aspects and therefore not fired

$$T_{rem} := \{t \mid t \in T_{base} \wedge \mathbb{A}(t, S, \eta) \neq \emptyset\}$$

where

$$\mathbb{A}(t, S, \eta) := \{\mathfrak{a} \mid t \in \mathsf{PC}_{\mathfrak{a}} \wedge \mathsf{compl}(S, \mathfrak{a}) \wedge \eta \models \mathsf{constraint}(\mathfrak{a})\}$$

The proposition $\mathsf{compl}(S, \mathfrak{a})$ is used to check whether all the states contained in the configuration of the pointcut are (or have just been) active. It is defined as follows

$$\mathsf{compl}(S, \mathfrak{a}) = \begin{cases} \mathsf{true} & kind(\mathfrak{a}) = \mathsf{before} \\ S \supseteq \mathsf{config}(\mathfrak{a}) & kind(\mathfrak{a}) = \mathsf{after} \\ S \supseteq \mathsf{src}(\mathfrak{a}) & kind(\mathfrak{a}) = \mathsf{between} \\ S \supseteq \mathsf{config}(\mathfrak{a}) & kind(\mathfrak{a}) = \mathsf{whilst} \end{cases}$$

T_{add} contains all transitions activating an aspect (instance), defined as

$$T_{add} = \bigcup_{t \in T_{rem}} \{\sigma_t \xrightarrow{e:\varepsilon} i_c \mid c \in CA(t)\}$$

where $CA(t)$ is the set of aspect instances generated by the advices applicable on transition t:

$$CA(t) = \{\mathrm{new}_c(\sigma'_t, \mathfrak{a}) \mid \mathfrak{a} \in \mathbb{A}(t, S, \eta)\}$$

C_{add}, the set of all generated aspect instances that should be executed now, is then the union of all CA:

$$C_{add} = \bigcup_{t \in T_{rem}} CA(t)$$

We define $E \subseteq S(\mathsf{TS_{SM}})$ to be the maximal set of states such that for each $e \in E$, it holds that for each $1 \leq i \leq \#(\mathsf{trans}_{C,e})$ there is an f_e^i such that $f_e^i \in F_{\mathbf{c}^i_{\mathsf{trans}_{C,e}}} \cap S$, which means that for every $e \in E$, all aspects contained in $\mathsf{trans}_{C,e}$ are finished. We further define $G \subseteq S(\mathsf{TS_{SM}}) \times \mathscr{E}$ to be the maximal set of state-event-pairs such that for each $g = (z, e) \in G$, and for each $1 \leq i \leq \#(\mathsf{whilst}^e_{C,z})$ there is an $f^i_{z,e}$ such that $f^i_{z,e} \in F_{\mathbf{c}^i_{\mathsf{whilst}^e_{C,z}}} \cap S$, which means that for every pair $(z, e) \in G$, all aspects contained in $\mathsf{whilst}^e_{C,z}$ are finished.

The set of the transitions from these aspects back to the base machine (or to the initial state of some other aspect's advice) is defined as $T_{res} := T_E \cup T_G$, where $T_E := \bigcup_{z \in E, 1 \leq i \leq \#(\mathsf{trans}_{C,z})} \{f^i_z \xrightarrow{*:\epsilon} t \mid t \in \rho(\mathsf{trans}_{C,z})\}$ and $T_G := \bigcup_{(z,e) \in G, 1 \leq i \leq \#(\mathsf{whilst}^e_{C,z})} \{f^i_{z,e} \xrightarrow{*:\epsilon} t \mid t \in \tau(\mathsf{whilst}^e_{C,z})\}$. The functions $\rho(\mathsf{trans}_{C,z})$ and $\tau(\mathsf{whilst}^e_{C,z})$ are defined as follows:

$$\rho(\mathsf{trans}_{C,z}) := \begin{cases} \{\mathbf{s_f}(\mathsf{trans}_{C,z})\}, & \mathbf{s_f}(\mathsf{trans}_{C,z}) \in \{z, \mathsf{Err}\} \\ \{\mathbf{s_f}(\mathsf{trans}_{C,z})\}, & \mathbf{s_f}(\mathsf{trans}_{C,z}) \notin \{z, \mathsf{Err}\} \wedge \\ & \nexists \mathfrak{a} \cdot [kind(\mathfrak{a}) = \mathsf{before} \wedge \\ & \quad \mathbf{s_f}(\mathsf{trans}_{C,z}) \in \mathsf{M_{before}}(\mathsf{config}(\mathfrak{a}))] \\ \{i_c \mid c \in \mathsf{B}\}, & \mathbf{s_f}(\mathsf{trans}_{C,z}) \notin \{z, \mathsf{Err}\} \wedge \\ & \mathsf{B} := \{\mathfrak{a} \mid kind(\mathfrak{a}) = \mathsf{before} \wedge \\ & \quad \mathbf{s_f}(\mathsf{trans}_{C,z}) \in \mathsf{M_{before}}(\mathsf{config}(\mathfrak{a}))\} \neq \emptyset \end{cases}$$

$$\tau(\mathsf{whilst}^e_{C,z}) := \begin{cases} \{\mathbf{s_f}(\mathsf{whilst}^e_{C,z})\}, & \mathbf{s_f}(\mathsf{whilst}^e_{C,z}) = \mathsf{Err} \\ \{\mathbf{s_f}(\mathsf{whilst}^e_{C,z})\}, & \mathbf{s_f}(\mathsf{whilst}^e_{C,z}) \neq \mathsf{Err} \wedge \\ & \nexists \mathfrak{a} \cdot [kind(\mathfrak{a}) = \mathsf{before} \wedge \\ & \quad \mathbf{s_f}(\mathsf{whilst}^e_{C,z}) \in \mathsf{M_{before}}(\mathsf{config}(\mathfrak{a}))] \\ \{i_c \mid c \in B\}, & \mathbf{s_f}(\mathsf{whilst}^e_{C,z}) = \mathsf{Err} \\ & \mathsf{B} := \{\mathfrak{a} \mid kind(\mathfrak{a}) = \mathsf{before} \wedge \\ & \quad \mathbf{s_f}(\mathsf{whilst}^e_{C,z}) \in \mathsf{M_{before}}(\mathsf{config}(\mathfrak{a}))\} \neq \emptyset \end{cases}$$

We define $C_E = \bigcup_{z \in E} \mathsf{trans}_{C,z}$, $C_G = \bigcup_{(z,e) \in G} \mathsf{whilst}^e_{C,z}$ and calculate C', η' and S' as follows:

$$C' = (C \cup C_{add}) \setminus (C_E \cup C_G)$$
$$\eta, \eta' \models e/a, a \in \mathbf{A}(T),$$
$$S' = (S \setminus (\sigma_T)) \cup \sigma'_T.$$

Since C_E and C_G are subsets of C, it holds that $C_{add} \cap C_E = C_{add} \cap C_G = \emptyset$.

6.6. Weaving and Semantics

We show that the result of our weaving process (see Chap. 5) is a state machine that exhibits the semantics defined in this chapter. To this end, we first show that the formal semantics does describe the behavior of a state machine which is enhanced by a set of aspects.

Given a state machine SM and an aspect $\mathfrak{a}$, the advice of $\mathfrak{a}$ will be executed in the next step iff. in SM's transition system, $\mathsf{TS}_{\mathsf{SM}}$, an instance of $\mathfrak{a}$ is activated in the next execution step. More precisely, we prove the following theorem

THEOREM 6.3 (Equivalence between state machine with aspects and transition system). *Given a state machine* SM*, a set* A *of aspects. Assume* SM *has been preprocessed, but without the before sections inserted. Let the semantical representation of* SM *(and the aspects) be* $\mathsf{TS}_{\mathsf{SM}}$ *and the current active state configuration of* $\mathsf{TS}_{\mathsf{SM}}$ *be* S*. Assume* SM *and* $\mathsf{TS}_{\mathsf{SM}}$ *have the same environment, i.e. the variables have the same values, and the same active state configuration, i.e.* $S = \{\mathsf{s}_x \mid x \in \mathrm{S}_{\mathsf{simple}}(\mathsf{SM}), x \text{ is active}\}$*. Then an aspect* $\mathfrak{a}$ *is activated by* SM *iff. its instance is activated by* $\mathsf{TS}_{\mathsf{SM}}$*.*

PROOF. We first prove that when an aspect $\mathfrak{a}$ is activated by SM, then an instance of $\mathfrak{a}$ is also activated in $\mathsf{TS}_{\mathsf{SM}}$, that is, in the execution step, as specified by the rule (asp), see p. 88, the set $C' \setminus C$ contains an instance of $\mathfrak{a}$.

Consider the following cases:

(1) $kind(\mathfrak{a}) = \mathsf{before}$. In this case, $\mathfrak{a}$ getting active means that a transition $t \in \mathrm{T}(\mathsf{SM})$ is activated where $\mathsf{target}(t) = s$ for some $s \in \mathsf{config}(\mathfrak{a})$, which means that
   - $\exists q \in T(\mathsf{TS}_{\mathsf{SM}})$ such that $\sigma'_q = \mathsf{s}_s$, $e_q = \mathsf{trigger}(t)$, and $g_q = \mathsf{guard}(t)$.

   Therefore it holds that $q \in \mathsf{PC}_{\mathfrak{a}} \wedge \mathfrak{a} \in \mathbb{A}(q, S, \eta)$.

(2) $kind(\mathfrak{a}) = \mathsf{after}$, in this case, $\mathfrak{a}$ getting active means that $S \supseteq \mathsf{config}(\mathfrak{a})$, and a transition $t' \in \mathrm{T}(\mathsf{SM})$ is activated where $\mathsf{target}(t') = s$ for some $s \in \mathrm{S}(\mathsf{SM})$, and $\mathsf{source}(t') \in \mathrm{S}(\mathsf{SM}) \vee \mathrm{isJoin}(\mathsf{source}(t'))$ (in the first case we define $t := t'$, in the second case we define t to be any r such that $\mathsf{target}(r) = \mathsf{source}(t')$). In both cases we define $x := \mathsf{source}(t)$.
   - If $x \in \mathsf{config}(\mathfrak{a})$ and t is source structured, then $\exists q \in T(\mathsf{TS}_{\mathsf{SM}})$ such that $\sigma'_q = \mathsf{s}_s$, $e_q = \mathsf{trigger}(t)$, $g_q = \mathsf{guard}(t)$, and $\sigma_q = \mathsf{s}_x$ (if $\mathrm{isSimple}(x)$) or $\sigma_q = \mathsf{s}_{x,t}$ (if $\mathrm{isComposite}(x)$),
   - if $x \in \mathsf{config}(\mathfrak{a})$ and t is source unstructured, then $\exists q \in T(\mathsf{TS}_{\mathsf{SM}})$ such that $\sigma_q = \mathsf{t}_{\mathsf{source}_{\mathsf{struct}}(t),t}$, $\sigma'_q = \mathsf{s}_s$, $e_q = \mathsf{trigger}(t)$, and $g_q = \mathsf{guard}(t)$,
   - if $x \notin \mathsf{config}(\mathfrak{a})$, then $\exists y \in \mathsf{config}(\mathfrak{a}) \cap \mathsf{subvertex}^*(\mathsf{source}_{\mathsf{struct}}(t))$. Since $x \neq y$, t must be source unstructured. Therefore $\exists q_1, q \in T(\mathsf{TS}_{\mathsf{SM}})$ such that $\sigma_q = \mathsf{s}_{\mathsf{source}_{\mathsf{struct}}(t),t}$, $\sigma'_q = \mathsf{s}_s$, $\mathsf{trigger}(q) = *$,

$g_q = \bot$; $\sigma'_{q_1} = \sigma_q$, $\mathsf{trigger}(q_1) = \mathsf{trigger}(t)$, $\mathsf{guard}(q_1) = \mathsf{guard}(t)$, and $\sigma_{q_1} = z$ for any $z \in \mathsf{subvertex}^*(y) \cap \mathrm{S}_{\mathsf{simple}}(\mathsf{SM})$.

Therefore it holds that $q \in \mathsf{PC}_{\mathfrak{a}} \wedge \mathfrak{a} \in \mathbb{A}(q, S, \eta)$.

(3) $kind(\mathfrak{a}) = \mathsf{between}$, in this case, $\mathfrak{a}$ getting active means that $S \supseteq \mathsf{src}(\mathfrak{a})$, and a transition $t' \in \mathrm{T}(\mathsf{SM})$ is activated where $\mathsf{target}(t') = s$ for some $s \in \mathsf{tgt}(\mathfrak{a})$, and $\mathsf{source}(t') \in \mathrm{S}(\mathsf{SM}) \vee \mathrm{isJoin}(\mathsf{source}(t'))$ (in the first case we define $t := t'$, in the second case we define t to be any r such that $\mathsf{target}(r) = \mathsf{source}(t')$). In both cases we define $x := \mathsf{source}(t)$.

- If $x \in \mathsf{src}(\mathfrak{a})$ and t is source structured, then $\exists q \in T(\mathsf{TS}_{\mathsf{SM}})$ such that $\sigma'_q = \mathsf{s}_s$, $e_q = \mathsf{trigger}(t)$, $g_q = \mathsf{guard}(t)$, and $\sigma_q = \mathsf{s}_x$ (if $\mathrm{isSimple}(x)$) or $\sigma_q = \mathsf{s}_{x,t}$ (if $\mathrm{isComposite}(x)$),
- if $x \in \mathsf{src}(\mathfrak{a})$ and t is source unstructured, then $\exists q \in T(\mathsf{TS}_{\mathsf{SM}})$ such that $\sigma_q = \mathsf{t}_{\mathsf{source}_{\mathsf{struct}}(t),t}$, $\sigma'_q = \mathsf{s}_s$, $e_q = \mathsf{trigger}(t)$, and $g_q = \mathsf{guard}(t)$,
- if $x \notin \mathsf{src}(\mathfrak{a})$, then $\exists y \in \mathsf{config}(\mathfrak{a}) \cap \mathsf{subvertex}^*(\mathsf{source}_{\mathsf{struct}}(t))$. Since $x \neq y$, t must be source unstructured. Therefore $\exists q_1, q \in T(\mathsf{TS}_{\mathsf{SM}})$ such that $\sigma_q = \mathsf{s}_{\mathsf{source}_{\mathsf{struct}}(t),t}$, $\sigma'_q = \mathsf{s}_s$, $\mathsf{trigger}(q) = *$, $g_q = \bot$; $\sigma'_{q_1} = \sigma_q$, $\mathsf{trigger}(q_1) = \mathsf{trigger}(t)$, $\mathsf{guard}(q_1) = \mathsf{guard}(t)$, and $\sigma_{q_1} = z$ for any $z \in \mathsf{subvertex}^*(y) \cap \mathrm{S}_{\mathsf{simple}}(\mathsf{SM})$.

Therefore it holds that $q \in \mathsf{PC}_{\mathfrak{a}} \wedge \mathfrak{a} \in \mathbb{A}(q, S, \eta)$.

(4) $kind(\mathfrak{a}) = \mathsf{whilst}$, in this case, $\mathfrak{a}$ getting active means that $S \supseteq \mathsf{config}(\mathfrak{a})$ and $\mathsf{trigger}(\mathfrak{a})$ is the current event. For any $x \in \mathsf{config}(\mathfrak{a})$, it holds

- either $\exists q \in T(\mathsf{TS}_{\mathsf{SM}})$ such that $\sigma_q = \mathsf{s}_x$, $\sigma'_q = \mathsf{s}_s$, $e_q = \mathsf{trigger}(\mathfrak{a})$, and $g_q = \mathsf{constraint}(\mathfrak{a})$ (when x is a simple state),
- or $\exists q_1, q \in T(\mathsf{TS}_{\mathsf{SM}})$ such that $\sigma_{q_1} = \mathsf{s}_z$ for any $z \in \mathsf{subvertex}^*(x)$, $\sigma'_{q_1} = \sigma_q = \mathsf{w}_{x,\mathfrak{a}}$, $\sigma'_q = \mathsf{s}_x$, $\mathsf{trigger}(q_1) = \mathsf{trigger}(\mathfrak{a})$, $\mathsf{trigger}(q) = *$, $g_{q_1} = \mathsf{constraint}(\mathfrak{a})$ and $g_q = \bot$ (when x is a composite state).

Therefore it holds that $q \in \mathsf{PC}_{\mathfrak{a}} \wedge \mathfrak{a} \in \mathbb{A}(q, S, \eta)$.

Since in all four cases, we have $q \in \mathsf{PC}_{\mathfrak{a}}$ and therefore $q \in T_{rem}$, the set C_{add} contains a new instance of $\mathfrak{a}$, i.e. $C_{add} \setminus C$ contains a new instance of $\mathfrak{a}$. Since $C_{add} \cap C_E = \emptyset$, it holds that C' also contains this instance.

On the other hand, suppose C' contains a new instance ι of $\mathfrak{a}$, ι is also contained in C_{add}, i.e. $\iota = \mathrm{new}_c(\sigma'_q, \mathfrak{a})$ for some $q \in T(\mathsf{TS}_{\mathsf{SM}}))$. Since $q \in T_{rem}$, it holds that $q \in \mathsf{PC}_{\mathfrak{a}}$.

Therefore, the precondition of $\mathfrak{a}$ is satisfied, $\mathfrak{a}$ is activated by SM. □

Now we are in the position to show that the weaving result does exhibit the semantics defined in the chapter.

THEOREM 6.4 (Equivalence between weaving result and transition system). *Given a state machine* SM *and a set* A *of aspects. Let the weaving result be* SA*, and the semantical representation be* $\mathsf{TS}_{\mathsf{SM}}$*. An aspect is activated in* SA *iff. an instance of this aspect is activated in* $\mathsf{TS}_{\mathsf{SM}}$.

PROOF. According to Theorem 5.9, SA is equivalent to the combination of SM and A. According to Theorem 6.3 $\mathsf{TS}_{\mathsf{SM}}$ is also equivalent to the combination of SM and A. Therefore, SA is equivalent to $\mathsf{TS}_{\mathsf{SM}}$. □

6.7. Discussion

As stated before, we did not specify in our semantics the details of how to determine which transitions are selected to be fired in an execution step of UML state machines, but applied the abstract function θ. Thanks to this abstraction we can better focus the description of the semantics on how the aspects are entered and left. The advices of the aspects, as independent units, are executed in the same way as a UML state machine.

As opposed to static aspects (see Chap. 3), HILA aspects are dynamic, i.e. they are defined as modification of the execution of the base machine when certain (runtime) conditions are satisfied. Due to the dynamic characteristics, the advices of HILA aspects are also represented in the formal semantics as independent units. Apart from being executed when certain conditions, as defined by the pointcuts, are satisfied at runtime of the transition system, these units are independent of the base machine.

Since the plain UML state machine does not support this kind of independent units, HILA aspects, including both pointcuts and advices, are implemented as a part of an overall state machine, when they are woven together with the base machine (see Chap. 5). Despite this difference, we could prove the equivalence between the weaving result and the semantical representation.

This equivalence is actually intuitive, since the weaving result and the transition system exhibit many similarities:

- In the weaving result, all transition aspects guarding state z are woven into parallel regions of an orthogonal state, which is spliced into the (only one) transition leading to z. At runtime, these aspects are activated and executed in parallel. In the transition system, these aspects are represented by aspect instances contained in an aspect group, and are activated and executed in parallel.
- In the weaving result, all configuration aspects with the same trigger and guarding the same state are woven into parallel regions of an orthogonal state. At runtime, these aspects are also activated and executed in parallel. In the transition system, these aspects are represented by aspect instances contained in an aspect group, and are activated and executed in parallel.
- In the weaving result, the execution of (instances of) aspects in the orthogonal state is only finished when the execution of each of the aspects is finished and each aspect has set a (possibly default) resumption strategy. In the transition system, first the aspect instances in the aspect group are finished. Only when all of them are finished, the instances become eligible to be removed from the set of current active aspect instances.
- In the weaving result, a resumption strategy goto s, which is set by transition aspects, s being a (non-error) state, is implemented as a transition leading to s if s is the target of the advised transition, or otherwise, as a transition to the junction jBefore(s). That is, when an aspect $\mathfrak{a}_1$ guarding s' is finished, if the resumption strategy is goto $s \neq s'$, and activating s would satisfy the precondition of another, ≪before≫ aspect $\mathfrak{a}_2$ (s is contained in config($\mathfrak{a}_2$), and the constraint of the pointcut of $\mathfrak{a}_2$ is satisfied), then $\mathfrak{a}_2$ would be executed first, concurrently with other aspects whose preconditions are also satisfied. In the transition system, this is also the result of the function ρ, see page 89.

- In the weaving result, a resumption strategy goto s, which is set by configuration aspects, s being a (non-error) state, is implemented as a transition leading to jBefore(s), even if s is the target of the advised transition. That is, when an aspect $\mathfrak{a}_1$ guarding s' is finished, if the resumption strategy is goto s, and activating s would satisfy the precondition of another, ≪before≫ aspect $\mathfrak{a}_2$ (s is contained in config($\mathfrak{a}_2$), and the constraint of the pointcut of $\mathfrak{a}_2$ is satisfied), then $\mathfrak{a}_2$ would be executed first, concurrently with other aspects whose preconditions are also satisfied. In the transition system, this is also the result of the function τ, see page 89.
- In the weaving result, when a ≪whilst≫ S aspect is executed, an ≪after≫ S aspect would not be executed. In the transition system, this is reflected by the fact that in the calculation T_{rem}, only transitions contained in the base machine are considered, see page 88. Neither in the weaving result nor in the transition system is it guaranteed that an ≪after≫ S aspect is executed whenever S gets inactive, since a ≪whilst≫ S aspect might circumvent the ≪after≫ aspect.
- In the weaving result, if aspects (actually, regions implementing aspect) contained in the same orthogonal aspect state indicate different resumption strategies, the base machine enters an exception state Error. In the transition system, this is reflected by the calculation of $\mathbf{s_f}(\Gamma)$ (page 87), $\rho(\mathsf{trans}_{C,z})$, and $\rho(\mathsf{whilst}^e_{C,z})$ (page 89).

CHAPTER 7

Interaction of aspects

Contents

7.1. Change of State Reachability **97**
7.1.1. Reachability in UML state machines 97
7.1.2. Reachability reduction by aspects 97
7.1.3. Reachability enhancement 98
7.2. Conflict Detection **99**
7.2.1. Resumption conflicts 99
7.2.2. Rule violation conflicts 99
7.2.3. Reachability conflict 100
7.3. Discussion **101**

The previous chapters showed how HILA provides valuable help to achieve a clean separation of concerns by modeling parts of the system behavior in aspects, which are developed separately from the base machine and separately from each other.

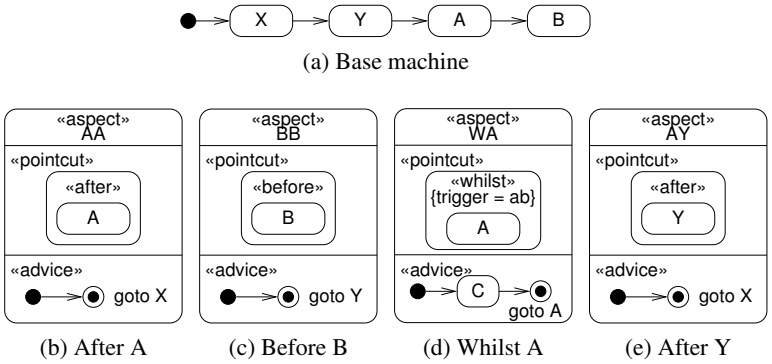
However, the aspects may still have some interactions with each other. For example, since different aspects, which are applicable to the same transition, are woven as concurrent regions and executed in parallel, race conditions can always arise when assignments of shared variables are done. Such race conditions, however, are not investigated in the research of this thesis, since their detection is not statically decidable. Moreover, it may be argued that shared variables in different aspects are often a hint of unclean separation of concerns, and that in a more logical design shared variables between aspects are not necessary.

In the following, we assume that for a given base machine, its HILA aspects do not share variables with each other and do not change the values of variables they share with the base machine. Moreover, we assume that the sets of events that the aspects react to are pair-wise disjunct, and that an aspect does not send events that others react to.

Under these circumstances there is no race conditions between HILA aspects w.r.t. variable assignments. However, since aspects may change the control flow of the base machine, there may still exist interference between aspects w.r.t. the control flow of the complete system behavior. HILA aspects may show interference of the following types:

- Two aspects that are executed in parallel (in different regions of the orthogonal Aspect state) may specify different states to activate after the execution of the advices. We call this kind of conflicts *resumption conflicts*.

- An ≪after≫ aspect stating "whenever a constellation of the base machine has just become inactive, where all the states contained in the configuration were active and the constraint was satisfied" defined by an ≪after≫ aspect may be violated, since the execution of a ≪whilst≫ aspect may have caused this situation, but according to our weaving algorithms and the formal semantics, the ≪after≫ aspect is not executed, see Chaps. 5 and 6. We call this kind of conflicts *rule violation conflicts*.
- An aspect may be correct if applied as the only aspect to the base machine, but, if it is applied together with other aspects, may never be executed since the other aspects make it impossible for its precondition to be ever satisfied. We call this kind of conflicts *reachability conflicts*.



(a) Base machine

(b) After A (c) Before B (d) Whilst A (e) After Y

Figure 7.1: Conflicts between HILA Aspects

EXAMPLE 7.1 (Conflicts). Consider the base machine given in Fig. 7.1a and its aspects shown in Fig. 7.1b–Fig. 7.1e. While the behavior defined in the aspects, if applied in isolation, is rather simple, there exist subtle interferences between the aspects when several of them are applied.

- When aspects AA and BB are applied, then there arises a resumption conflict whenever state A is left. In this situation, the preconditions of both of the aspects are satisfied: A has just been active, and the base machine is just about to enter B. The execution of the two advices in parallel results in conflicting resumption strategies: aspect AA requires to activate state X, and aspect BB requires to activate state Y.
- When aspects WA and AA are applied, then there arises a rule violation conflict whenever state A is active and event ab is the current event. In this situation, WA is executed, state C of the advice is activated. Although it is also true that "state A has just been active", aspect AA is, according to our weaving and the formal semantics, not executed.
- When aspects AY and AA or BB are applied, then there arises a reachability conflict. Every time after Y has been active, AY is executed and instructs the base machine to go to X. Therefore, state A and B will no longer get a chance to be active, and consequently, aspects AA or BB no longer be executed.

The first two kinds of conflicts have been discussed in Chaps. 5 and 6. In the following, we study reachability conflicts in more detail, and then give some guide lines of how to detect (all three kinds of) conflicts between HILA aspects.

7.1. Change of State Reachability

Reachability is often a very important property of states in state-based models like UML state machines of transition systems. We define

- A state configuration c of a UML state machine SM is *reachable from configuration* c' iff. there is a possible execution of SM in which first c', and then c gets active at least once. A state configuration c of a UML state machine SM is *reachable* iff. there is a possible execution of SM in which c is active at least once.
- A state s of a UML state machine SM is *reachable from state* s' iff. there is a possible execution of SM in which first s', and then s gets active at least once. A state s of a UML state machine SM is *reachable* iff. there is a possible execution of SM in which first s gets active at least once.
- A HILA aspect $\mathfrak{a}$ is *reachable* w.r.t. a UML state machine SM and a set of aspects $A \ni \mathfrak{a}$ iff. there is a possible execution of the weaving result of SM and all aspects of A, in which the advice of $\mathfrak{a}$ is executed at least once.

Obviously, aspects have repercussions on the reachability of the base machine's states. The goto statements of the advices can make both states that are originally reachable in the base machine now unreachable and states that are originally unreachable (under some circumstances) now reachable. In the following, we examine these two possibilities in more detail.

7.1.1. Reachability in UML state machines. For a given UML state machine SM, we recall some obvious facts about the reachability of SM's states.

- if a state s is unreachable, then for each state configuration S, $s \in S$ implies S is unreachable;
- for a state s, if each state $s' \in \mathcal{P}(s)$ is unreachable, so is s, where $\mathcal{P}(s)$ is the set of all predecessor states of s. In particular, if a state s' is unreachable, then for each state $s = \mathsf{target}(t), t \in \mathsf{outgoing}(s')$, $\mathcal{P}(s) = \{s'\}$ implies s is unreachable;
- if a state configuration S is unreachable, then for each state configuration S', $S \subseteq S'$ implies S' is unreachable;

Albeit very simple, these facts will be important for the discussion in the rest of this chapter.

7.1.2. Reachability reduction by aspects. Some aspects make certain states or state configurations of the base machine unreachable.

≪**before**≫. The strongest form of reachability reduction is achieved by ≪before≫ aspects. Let X be a state configuration. If an aspect with pointcut ≪before≫ X does not allow X to be activated after the execution, then X will never be active when the weaving result is executed.

The reachability reduction caused by aspects with pointcut ≪before≫ X is "strong" in the sense that if any other aspect $\mathfrak{a}$ specifies goto X as its resumption strategy, this is implemented as "goto the junction before X" (see Sect. 5.6). When

$\mathfrak{a}$ is finished, X is not activated directly, but it is first checked if any aspect prohibits the activation of X. Therefore, a reduction of reachability by a ≪before≫ aspect cannot be overridden by any other aspect (by reachability enhancement, see Sect. 7.1.3).

EXAMPLE 7.2 (Reachability reduction by ≪before≫ aspects). When aspect BB (Fig. 7.1c) is applied to the base machine (Fig. 7.1a), then state B is effectively prevented from ever getting active, no matter which resumption strategies are defined in other aspects.

≪**between**≫ and ≪**after**≫. Just like ≪before≫ aspects, ≪between≫ and ≪after≫ aspects may also restrain reachability of states from the base machine, although it is rather hard even to make a single state unreachable.

In a system consisting of a base machine and a set of aspects, if some state or state configuration is made unreachable by ≪between≫ and ≪after≫ aspects only, then the unreachability may be overridden due to the effect of reachability enhancement when an additional aspect is applied.

EXAMPLE 7.3 (Reachability reduction by ≪after≫ aspects). When aspect AY (Fig. 7.1e) is applied to the base machine (Fig. 7.1a), every time just after Y being active, X, instead of A, gets activated. Since the transition from Y to A is the only way for A to get active, aspect AY actually makes state A unreachable. Note, however, this reachability reduction is "weak" in the sense that it may be overridden by another aspect. An example is given below.

≪**whilst**≫. Since ≪whilst≫ aspects do not disable any existing transition from the base machine, they obviously do not reduce reachability.

7.1.3. Reachability enhancement. An aspect can also make states of the base machine "more reachable", since goto statements introduce new transitions. More concretely, an aspect may make a state s reachable from the source of the advised transition, if there is a goto s in the advice and, as discussed above, s is not prohibited by any aspect with a pointcut of ≪before≫ s.

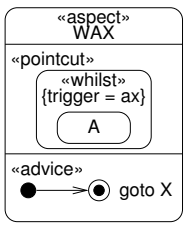


Figure 7.2: Example: reachability enhancement

EXAMPLE 7.4 (Reachability enhancement). In the base machine (Fig. 7.1a), state X is unreachable from A. Applying aspect WAX (Fig. 7.2) to the base machine would make X reachable again: when state A is active, and the current event is ax, then X gets active.

7.2. Conflict Detection

7.2.1. Resumption conflicts. As shown in Chap. 5, HILA's weaving is designed in such a way that conflicts are detected at runtime. At runtime, if two aspects do define different targets to go to after their execution, then, by default, an Err state is activated, indicating the system is in an error status. Moreover, the HILA modeler may define a priority between aspects, indicating which aspect should rule the other in case of conflicts.

Since in general, it is not possible to statically determine which labeled final states are executed (and therefore which labels are used as resumption strategy), simulation is the most effective way to detect such conflicts. Test cases or model checking may be very useful to discover resumption conflicts. Formal methods to guarantee conflict freedom of systems with infinitely many configurations would be an interesting piece of future work.

Some simple rules, however, may be used to reduce the number of needed test cases. They are listed in the following, with a brief explanation for each:

- Resumption conflicts are only possible at those joinpoints where several aspects are applicable (no *shared* joinpoint, no resumption conflict).
- Resumption conflicts are only possible between aspects with the same priority (otherwise the resumption strategy with the highest priority simply wins).
- Resumption conflicts are only possible between aspects with different labels (no difference, no conflict).
- Resumption conflicts are only possible when the shared joinpoint can be matched by a constellation of the base machine.
- A resumption conflict occurs when different resumption strategies are actually set (by different aspects) .

While the first three conditions can be checked by static analysis, checking the last two conditions generally requires execution of the complete system (generated by the weaving process).

EXAMPLE 7.5 (Detection of resumption conflicts). Of the aspects Fig. 7.1b through Fig. 7.1e with respect to the base machine (Fig. 7.1a), only Fig. 7.1b and Fig. 7.1c have shared joinpoints. Since the two aspects are assigned the same priority (both have the default priority, which is defined to be 0), they have to be examined in more detail. Since the aspects, when executed in parallel, may actually specify different resumption strategies (X and Y). The aspects are therefore candidates for resumption conflict and subject to further examination. Since in this case the advices of the aspects are very simple, it can be easily seen that they actually cause a resumption conflict. In general, conflict detection requires validation by simulation such as testing or model checking.

7.2.2. Rule violation conflicts. Rule violation conflicts are conflicts that although the precondition of some ≪after≫ aspect is satisfied, the advice is not executed.

- Rule violation conflicts are only possible between a ≪whilst≫ and an ≪after≫ aspect.
- Rule violation conflicts between ≪whilst≫ aspect w and ≪after≫ aspect a are only possible when $W \subseteq A$ where $W := \mathsf{config}(\mathsf{pointcut}(w))$

and $A := \mathsf{config}(\mathsf{pointcut}(a))$ (only when w is executed a rule violation conflict can occur).

- Given ≪whilst≫ aspect w and ≪after≫ aspect a, if $W \subseteq A$ where $W := \mathsf{config}(\mathsf{pointcut}(w))$ and $A := \mathsf{config}(\mathsf{pointcut}(a))$, then a rule violation conflict occurs when w is executed.

The first two conditions can be checked by static analysis. Checking the last one generally requires execution of the complete system.

EXAMPLE 7.6 (Detection of violation conflicts). Considering the aspects AA (Fig. 7.1b) and WA (Fig. 7.1d), we have $W = A = \{\mathsf{A}\}$. The first two conditions are satisfied. When the state A is active, and ab is the current event, WA is executed, the state machine enters a (new) state C. That is, although A has just been left, the aspect with "≪after≫ A" is not executed.

7.2.3. Reachability conflict. An aspect is made unreachable by others when its precondition is made unsatisfiable, in particular, when the state configuration of the pointcut is made unreachable. However, checking if a state configuration is disallowed by an aspect is not an easy task. In general, an advice may contain several final states, each with a label. It is not possible to decide by statical analysis which of the final states will terminate the execution of the advice (or even if the advice will terminate).

To investigate the problem of reachability conflict detection, we first define the following simple, sufficient conditions which ensure that for a given aspect a and a given state configuration X, when the execution of a is finished, then the active state configuration is different than X;

- either X is explicitly disallowed by the keyword 0 (see Sect. 4.3.6), the execution of a (actually, its advice) is terminated by a final state with a label 0 and X is the configuration specified in the pointcut (or the target configuration, if a is a ≪between≫ aspect),
- or all the final states of a have labels that are to be activated by resumption, and for each L of these labels it holds $\exists r \in \bigcup_{s \in \mathrm{S}(\mathsf{SM})} \mathsf{region}(s) \cdot [l, x \neq \emptyset \wedge l \neq x]$ where $l = L \cap \mathsf{subvertex}(r)$ and $x = X \cap \mathsf{subvertex}(r)$. That is, if any region contains a state $s \in X$ and in this region another state $s' \neq s$ is specified for the state machine to go to, then X can not be active after the execution of a (after the execution s' will be active, and s, which is in the same region, cannot be active, therefore X cannot be active). Note that according to constraint 4.3.6 a label may contain at most one state in a region

We then induce unreachability of state configurations from the unreachability of states:

- If an aspect B has a pointcut ≪before≫ s, where s is a state, and disallows the base machine to activate s after its execution (by satisfying one of the two conditions above), then every state configuration $S \ni s$ is also made unreachable.
- If an aspect B has a pointcut ≪before≫ S, where S is a state configuration, and disallows the base machine to activate S after its execution (by satisfying one of the two conditions above), then every state configuration $S' \supseteq S$ is made unreachable.

For detecting reachability conflicts, static analysis is of more help than in the case of resumption conflicts. More concretely, using the rules given above, we can summarize the following rules:

- If an aspect B has a pointcut $\ll$before$\gg$ S, which makes configuration S unreachable, then B makes all aspects with $\ll$after$\gg$ S, $\ll$whilst$\gg$ S or $\ll$between$\gg$ S and S' strictly unreachable, i.e. these aspects will never be executed.
- Assume there exists an aspect b with $\ll$before$\gg$ x, which makes state x unreachable. Let state $X = \{x_1, \ldots, x_n\}$, $x_i \neq x$, to be all states that are *only from* x reachable, i.e. in each execution of the state machine where $x' \in X$ is active at least once, x has been active before x'. Then aspects containing these states are made unreachable by b. However, reachability can be reconstituted since $x_1, \ldots, x_n$ may be made reachable again if other aspects are introduced later.
- For a state s, if for each $p \in \mathcal{P}(s)$, there exists an aspect with a pointcut "$\ll$between$\gg$ p and s" or "$\ll$after$\gg$ p", the advice does not allow s to be activated (see above), and there is no other aspect with a goto label to instruct the base machine to enter s, then s is made unreachable.

Note that in the last rule it is necessary to check if "there is no other aspect with a goto label to instruct the base machine to enter s" since, as opposed to the case of $\ll$before$\gg$, for aspects with a pointcut $\ll$between$\gg$ x or $\ll$after$\gg$ y ($x, y \in \mathcal{P}(s)$) are not executed when s is to be activated as a result of the resumption from another aspect.

EXAMPLE 7.7 (Detection of reachability conflicts). In the base machine shown in Fig. 7.1a, the only possibility for state A to get active is to be activated by the transition from Y to A. Therefore, states A and B are made unreachable by aspect AY. These two states may be made reachable again if other aspects (with a resumption strategy of goto A or goto B) are applied.

7.3. Discussion

Aspect interference is an intrinsic problem of aspect-oriented approaches to software development. Race conditions are a very common problem in parallel systems, which, in general, cannot be checked without running the system. For this reason we had to concentrate on systems that are free of shared variables. Even in this context, the following questions have to be answered, before an approach can be helpful in practice:

(1) When multiple aspects are applicable, how to ensure that all of them are executed? In which order are they executed?
(2) Is it possible that an aspect works fine in isolation, but does not when used together with other aspects?

This chapter gave HiLA's answers to these questions. We weave aspects that are simultaneously applicable into concurrent regions and (all of them) are executed in parallel. When the absence of shared variables and of common signals is also assumed, the only possible interference between such aspects is that of different resumption targets. HiLA's weaving makes it possible to raise an alarm in case such conflicts occurred; it is also possible for the modeler to prioritize aspects to resolve resumption conflicts.

To the second question above, we examined conditions under which the effect of one aspect may be overridden by others, or aspects may be made unreachable by others. The simple static analysis rules given above may be used to warn the modeler that conflicts are detected.

Aspects being made unreachable by others is a semantical conflict that does not amount to confluence of a system of syntactical rewriting. Detection of such conflicts is not directly supported by static aspect approaches.

Part 4

HiLA du Monde

CHAPTER 8

Case Study

Contents

8.1. Overview and Static Structure **106**
8.2. Modeling the Behavior of the CCCMS **109**
8.2.1. Modeling the Main Success Scenario of Use Case 1 109
8.2.2. Modeling the Extensions of Use Case 1 112
8.2.3. Modeling Use Case 10 114
8.3. Validation of the Model **117**
8.4. Discussion **119**
8.4.1. The HiLA Approach to Modeling 119
8.4.2. The HiLA Language 121

The practical applicability of HiLA was validated by modeling a car crash crisis management system (CCCMS), which is defined in [43].

The development of the static design model is rather traditional, and takes into account both the analysis model and the requirements of the dynamic models. In modeling the dynamic behavior of CCCMS, we follow a simple methodology of first deriving a base state machine from the main success scenario of the use case and then modeling the extensions with HiLA aspects. This way, we achieve a clean separation of extensions (more generally: concerns) as separate aspects operating on a single base state machine. As long as the behavior of the interaction can be expressed using high-level aspects, this separation of concerns is possible, even when the different concerns interact with each other.

The systematic transition between analysis and behavioral design offers several advantages: it provides a concrete method to derive a system design (with state machines) from use cases; the resulting state machines are easy to understand since they correspond directly to requirements; and the approach provides excellent traceability from requirements to behavioral model and *vice versa*, which simplifies the validation that all requirements are addressed by the design and simplifies subsequent changes to system requirements. The HiLA language thus not only reduces the complexity of the dynamic models, but also makes it possible to systematically generate the state machine models from use cases and to achieve a high traceability between requirements and design models, which generally is not easy when working with plain UML state machine. Our methodology is a similar technique to the one described by Jacobson's rendering of use case slices as aspects [37].

While the part of CCCMS we modeled may be considered smaller than most real applications, it is complex enough to demonstrate the advantages that HiLA models and the HiLA approach offer over traditional UML-based approaches to behavioral modeling.

In this chapter, we first give an overview of our CCCMS modeling, show its static structure, and demonstrate how Use Cases 1 and 10 are modeled in HILA. Then we show how model checking can be used to validate our models. The modeling of the other use cases is given in Appendix A. Finally, we make a critical evaluation of HILA against the background of modeling the CCCMS.

Publication Notice. The main content of this chapter is a reprint of [35]. Only some diagrams have been modified due to new definition of some HILA elements since the publication of [35].

8.1. Overview and Static Structure

The overall flow of events in resolving a car crash crisis is described in Use Case 1 "Resolve Crisis" (and its included use cases): Upon a crisis, a coordinator first gathers crisis information; the CCCMS recommends missions based on this information and the coordinator selects missions accordingly. For each mission internal and external resources are selected, and these resources execute their mission. When execution has finished, the coordinator closes the crisis.

In our model we directly represent this main course of actions by handling the different phases of a car crash crisis resolution in a chain of separate objects which reflect the respective states of the CCCMS. A crisis is reported to an instance of class System which just represents the phase of waiting for an incident. A System then creates a Crisis. In this Crisis the coordinator gathers the necessary crisis information: the witness reports and other crisis details. From this information a Crisis creates an Adviser which supervises the remainder of the crisis resolution. An Adviser recommends appropriate missions to the coordinator, accepts a selection of these missions, creates Mission objects, allocates the necessary resources to each Mission, delegates mission execution to the resources, and collects changes to the mission.

This division of labor has two driving factors: On the one hand, we take the stance that each use case has a primary interaction object; this accounts in particular for moving from System, the primary interaction object for the overall use case 1 "Resolve Crisis" to Crisis that handles the included use case 2 "Capture Witness Report". On the other hand, inside use cases several tasks have to be done in parallel, where the necessary degree of concurrency is not known up-front. This concurrency justifies, e.g. the creation of several Mission objects which all have to be executed in parallel.

A domain model of the CCCMS can thus be derived from the use case descriptions using rather conventional techniques (see, e.g. [12, 62]) obeying both the use case structuring and the required parallelism. The overall static structure, enriching the domain model by particular associations between the entities and operations as well as receptions for these entities again follows straightforwardly from an analysis of the requirements;[1] in both cases it is mainly enough to concentrate on the primary (success) scenarios. We therefore forego a detailed account of the design steps of the static structure but merely show the resulting class diagrams in Fig. 8.1, Fig. 8.2, and Fig. 8.3.

[1] As customary we do not make class constructors explicit.

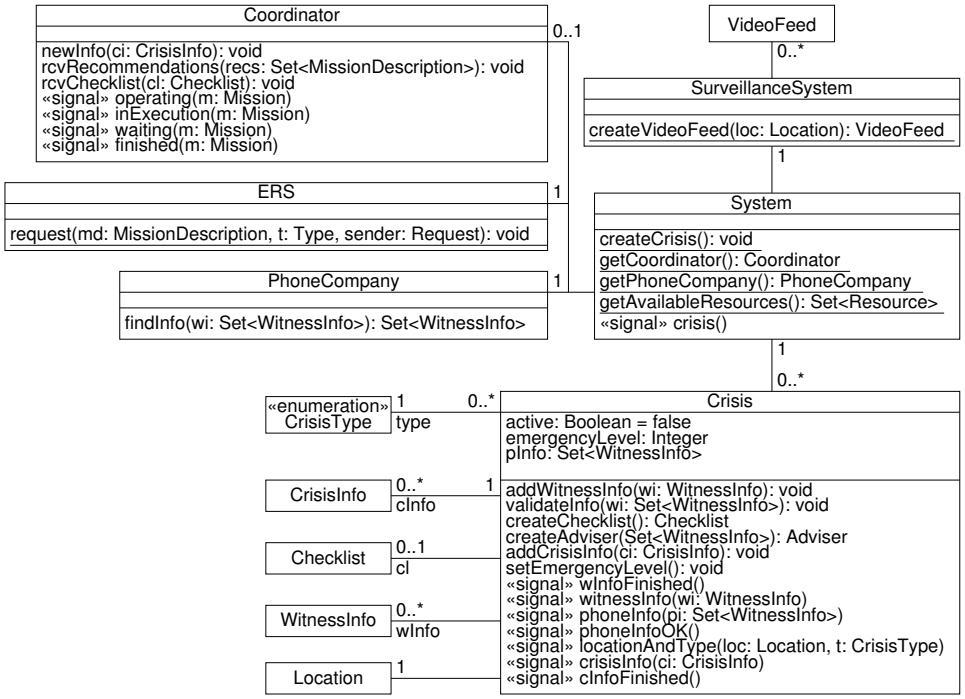


Figure 8.1: Class diagram: around System

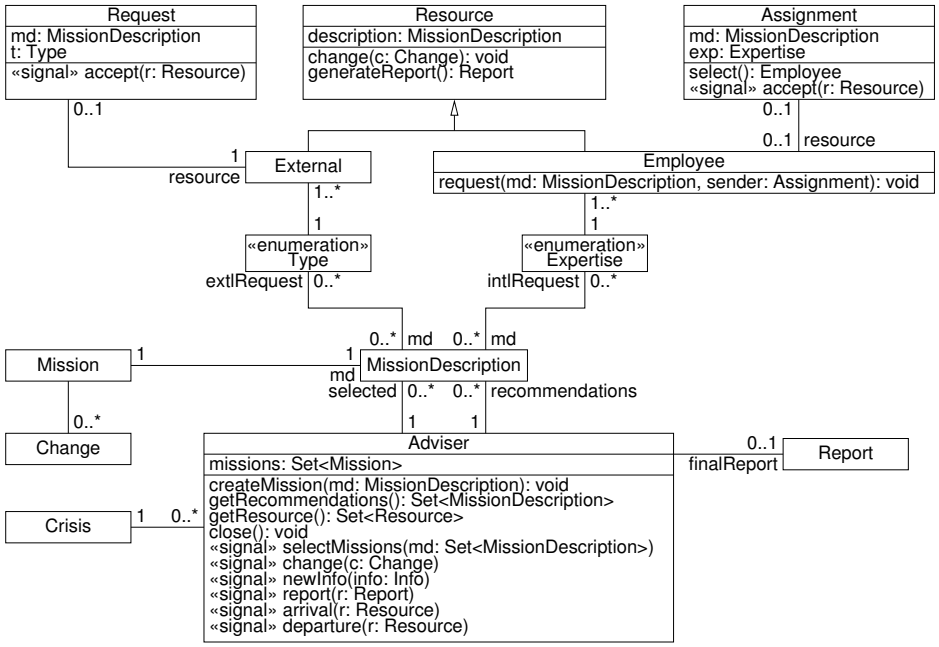


Figure 8.2: Class diagram: around Adviser

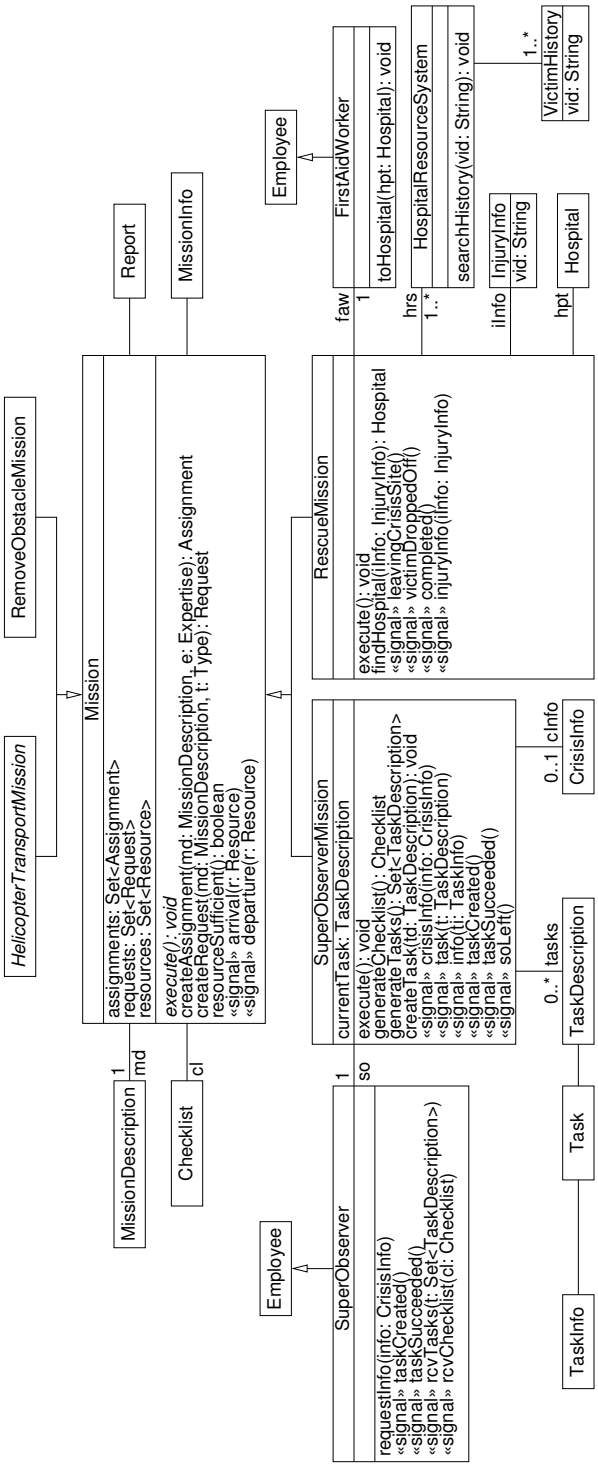
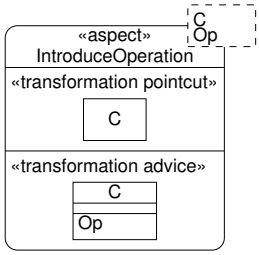
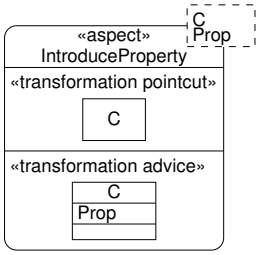


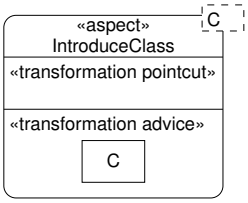
Figure 8.3: Class diagram: around Mission



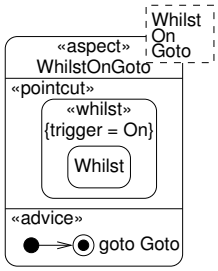
(a) Extend class C with operation Op



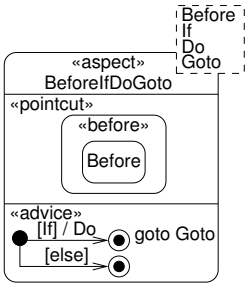
(b) Extend class C with property Prop



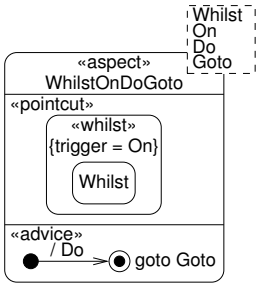
(c) Introduce class C to the system



(d) When state Whilst is active, if On is the current event, go to state Goto



(e) Before activating state Before, if condition If is true, execute the action Do and go to state Goto, otherwise, do nothing



(f) When state Whilst is active, if the current event is On, execute the action Do, and go to state Goto

Figure 8.4: Aspect templates used in the CCCMS case study

8.2. Modeling the Behavior of the CCCMS

In the following we describe in detail how we modeled two use cases of the CCCMS. In the modeling, both static and dynamic aspects are used, the former for extending the static structure of the system, the latter to model the use case extensions. A considerable part of the aspects we used are instantiations of the simple aspect templates given in Fig. 8.4.

8.2.1. Modeling the Main Success Scenario of Use Case 1. The main success scenario of use case 1 "Resolve Crisis" is modeled in the following, where we first cite the textual description of each step from [43] and then show how the step is modeled.

The starting point of a crisis management process is System; its behavior is modeled in Fig. 8.5: a System is idle until it receives a crisis notification (event crisis), upon which it creates a Crisis. The crisis then assumes the responsibility to manage the crisis by being the primary point of interaction with Coordinator, while the system is ready for other crisis notifications. It is the transition from System to Crisis where the handling of "Resolve Crisis" really starts.

1. Coordinator *captures witness report (UC 2).*

We do not detail the behavior of use case 2 "Capture Witness Report" here, which is handled by Crisis; see Appendix A.1. In particular, after successful termination of this step the information of Crisis will be up to date and an Adviser being attached

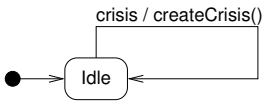


Figure 8.5: Class System: base machine

to the Crisis for handling the remaining steps for crisis resolution will have taken over.

2. System *recommends to* Coordinator *the missions that are to be executed based on the current information about the crisis and resources.*
3. Coordinator *selects one or more missions recommended by the system.*

For these steps it is the Adviser who represents the *System*. The Adviser, see

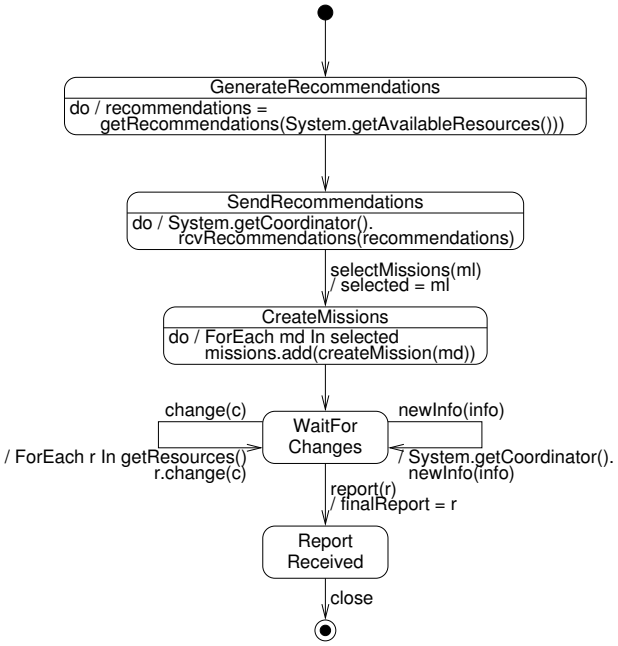


Figure 8.6: Class Adviser: base machine

Fig. 8.6, generates a set of recommendations (in state GenerateRecommendations where we omit the details of how getRecommendations proceeds) and passes the recommendations on to the Coordinator (in state SendRecommendations). An event selectMissions causes the Adviser to store the descriptions of the selected missions in selected. In CreateMissions it creates a new Mission for the description of each selected mission, and adds it to the set missions. (We abstract from the detail of how to decide whether a "super observer mission", a "rescue mission", a "helicopter transport mission", or a "remove obstacle mission" should be created for a given mission description, but hide this in the operation createMission.)

The base state machine for Mission is shown in Fig. 8.7. The next steps from 4 to 11 are executed *for each mission in parallel:*

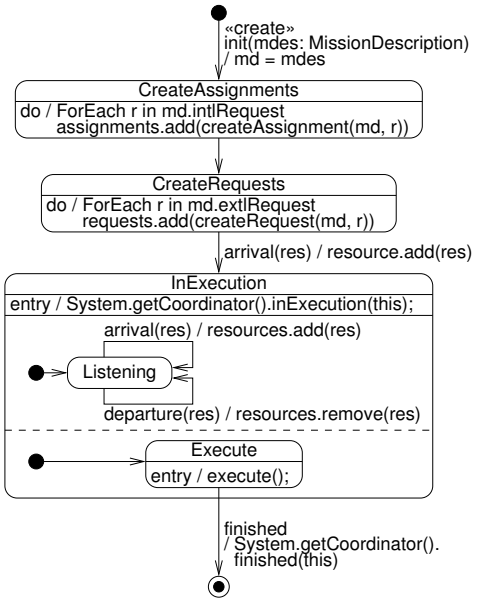


Figure 8.7: Class Mission

4. *For each internal resource required by a selected mission,* System *assigns an internal resource (UC 3).*
5. *For each external resource required by a selected mission,* System *requests an external resource (UC 4).*

Each Mission creates first Assignment objects (in state CreateAssignments) and then Request objects (in state CreateRequests) to allocate internal and external resources, respectively.

6. Resource *notifies* System *of arrival at mission location.*
7. Resource *executes the mission (UC 5).*
8. Resource *notifies* System *of departure from mission location.*

Upon arrival of a resource, the Mission starts execution (InExecution). The details of how a Mission executes (asynchronously calling its abstract method execute) depends on the type of Mission; see Appendices A.5 and A.6. Meanwhile, the mission also keeps track of the arrived and departed resources: in state Listening it waits for the resources to report their arrivals and departures, and updates its attribute resources accordingly. We suppose some resource sends (according to its mission description) to the Mission a finished signal to stop its execution when the mission is accomplished.

9. *In parallel to steps 6–8,* Coordinator *receives updates on the mission status from* System.

Each Mission continuously informs the coordinator when it is in execution or finished. Note that the missions are now the representatives of the overall CCCMS, i.e., the *System*.

10. *In parallel to steps 6–8,* System *informs* Resource *of relevant changes to mission/crisis information.*

This part of informational action is done by the Adviser (see Fig. 8.6). After creating the missions, the adviser gets ready for change notifications (WaitForChanges), and simply passes received change information on to the resources.

11. Resource *submits the final mission report to* System.

If the Adviser receives a final report (by the event report), it stops waiting for change notifications, and waits for the coordinator to close this crisis resolution session. (We assume here that there is exactly one such final report, although there may be many resources.)

12. *In parallel to steps 4–8,* Coordinator *receives new information about the crisis from* System.

When the Adviser receives any new information (by newInfo), it passes the information on to the coordinator.

13. Coordinator *closes the file for the crisis resolution.*

After receiving the final report, the adviser waits for the coordinator to close the file (in state ReportReceived), and then terminates (final state).

8.2.2. Modeling the Extensions of Use Case 1. Now we model extensions of use case 1 "Resolve Crisis" by HiLA aspects. Several times we not only have to extend the behavioral part of the CCCMS, but also have to extend first the underlying static structure. As for the main success scenario, we start with a citation of the extension from the use case description [43] and then describe the model. In fact, most of the extensions require only rather simple aspects and we will make ample use of instantiations of the templates given in Fig. 8.4 as well as for static structures. We mainly use a tabular format for presenting these instantiations succinctly; in these tables x ↦ y stands for ≪bind≫ x -> y.

1a. Coordinator *is not logged in.*
  1a.1. Coordinator *authenticates with* System *(UC 10).*
  1a.2. *Use case continues with step 1.*

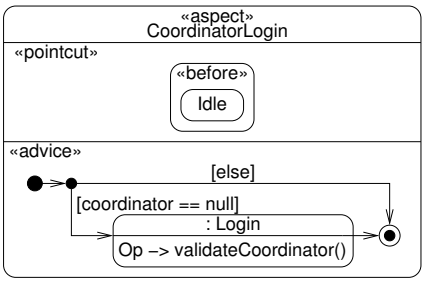


Figure 8.8: UC 1, extension 1a: ensuring that the Coordinator is logged in.

Template	Base	Binding
IntroduceOperation	Fig. 8.1	C ↦ System Op ↦ validateCoordinator(u: String, i: String): void

Table 8.1: UC 1, extension 1a: Introducing a new operation to class System

When the coordinator is not logged in Use Case 10 has to be executed. We therefore add dynamic aspect CoordinatorLogin (see Fig. 8.8) and its static counterpart (see Tab. 8.1): Before the Idle state of System becomes active, the system checks whether the coordinator is not logged in (coordinator == null). If this is the case the Login sub-state machine is triggered and use case 10 "Authenticate User" (see Sect. 8.2.3) steps in.

4a. *Internal resource is not available after step 4.*
    4a.1. System *requests an external resource instead (i.e., use continues in parallel with step 5).*

We have to expect a signal assignmentFailed in state Listening of class Mission (see Fig. 8.7) and to create a new request for an external resource on reception of this signal. We assume that for each kind of expertise there is a type of external resources as substitution, and call this type the externalType of the expertise. The necessary extensions can be completely covered by instantiating our templates, see Tab. 8.2.

Template	**Base**	**Binding**
IntroduceOperation	Fig. 8.3	C ↦ Mission Op ↦ ≪signal≫ assignmentFailed
IntroduceOperation	Fig. 8.3	C ↦ Mission Op ↦ createRequest(t: Type)
IntroduceProperty	Fig. 8.3	C ↦ Mission Prop ↦ externalType: Type[1]
WhilstOnDoGoto	Fig. 8.7	Whilst ↦ Listening On ↦ assignmentFailed Do ↦ createRequest(exp.externalType) Goto ↦ Listening

Table 8.2: UC 1, extension 4a: Template instantiations

5a. *External resource is not available after step 5.*
    5a.1. *Use continues in parallel with step 2.*

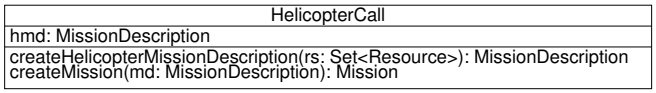
The fact that an external resource is unavailable is determined by the extensions of UC 4 "Request External Resource", where two exceptional responses of the external resource are introduced: partial approval or denial, see Appendix A.3.2. We define an instance of aspect WhilstOnDoGoto for (the state machine of) the main success scenario of UC 4, which is given in Fig. A.2b, to inform the Mission and the System that the resource is unavailable (event denial), as well as to ask the Crisis object to create another Adviser instance for the use case to "continue in parallel with step 2". The new adviser will have the knowledge of the unavailable resource and will recommend different missions to the coordinator than the current one did, details are hidden in the static operation System.getAvailableResources.

6a. System *determines that the crisis location is unreachable by standard transportation means, but reachable by helicopter.*
    6a.1. System *informs the Coordinator about the problem.*
    6a.2. Coordinator *instructs* System *to execute a helicopter transport mission (UC 9).*
    6a.3. *Use case continues with step 6.*

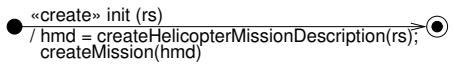
Template	Base	Binding
IntroduceOperation	Fig. 8.3	C ↦ Mission Op ↦ ≪signal≫ resourceUnavailable
IntroduceOperation	Fig. 8.1	C ↦ System Op ↦ ≪signal≫ missionFailed(m: Mission)
WhilstOnDoGoto	Fig. A.2b	Whilst ↦ WaitForAcceptance On ↦ denial Do ↦ mission.resourceUnavailable; System.missionFailed(mission); md.adviser.crisis.createAdviser()

Table 8.3: UC 1, extension 5a: Template instantiations

We introduce two additional transitions to Fig. 8.7 by instantiating WhilstOnDoGoto twice: one instance passes on the request to the coordinator (System.getCoordinator().needHelicopter()) that the resource is requiring a helicopter (helicopterRequired) when the mission is waiting for the resources to arrive at the mission location (state WaitForArrival). The other reacts to the coordinator's instruction startHelicopterMission and creates a new instance of HelicopterCall to start the helicopter transport mission.



(a) Class diagram



(b) State machine

Figure 8.9: Class HelicopterCall

The class HelicopterCall is modeled in Fig. 8.9. A HelicopterCall objects creates a mission description, in which a helicopter transport mission to transport a set of resources to a certain location is described, and creates a mission according to this description. The necessary signals and properties are introduced in Tab. 8.4.

All the remaining extensions of UC 1 follow the same patterns. We summarize the necessary instantiations in Table 8.5. In particular, the fact that a resource is "unable to contact *System*" (in extensions 6b and 8a) is not explicitly modeled, we simply model the consequence of this fact, i.e., that "*SuperObserver* notifies *System*". The use case continuing "in parallel with step 2" is modeled by asking the Crisis object to create a new Adviser. If parallelism is not required (like in extension 7b), we simply go to state RecommendMissions to generate new missions to recommend to the coordinator.

8.2.3. Modeling Use Case 10. The main success scenario of this Use Case 10 according to [43] reads as follows:

1. System *prompts* CMSEmployee *for login id and password.*
2. CMSEmployee *enters login id and password into* System.
3. System *validates the login information.*

Template	Base	Binding
IntroduceOperation	Fig. 8.1	C ↦ Coordinator Op ↦ ≪signal≫ needHelicopter(r: Resource)
IntroduceOperation	Fig. 8.3	C ↦ Mission Op ↦ ≪signal≫ helicopterRequired(r: Resource)
IntroduceOperation	Fig. 8.3	C ↦ Mission Op ↦ ≪signal≫ startHelicopterMission(r: Set⟨Resource⟩)
IntroduceClass	Fig. 8.3	C ↦ HelicopterCall (Fig. 8.9)
WhilstOnDoGoto	Fig. 8.7	Whilst ↦ WaitForArrival On ↦ helicopterRequired(r) Do ↦ System.getCoordinator().needHelicopter(r) Goto ↦ WaitForArrival
WhilstOnDoGoto	Fig. 8.7	Whilst ↦ WaitForArrival On ↦ startHelicopterMission(r: Set⟨Resource⟩) Do ↦ createHelicopterCall(r) Goto ↦ WaitForArrival

Table 8.4: UC 1, extension 6a: Template instantiations

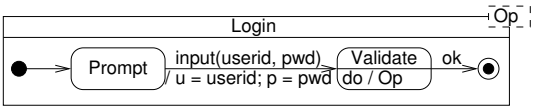
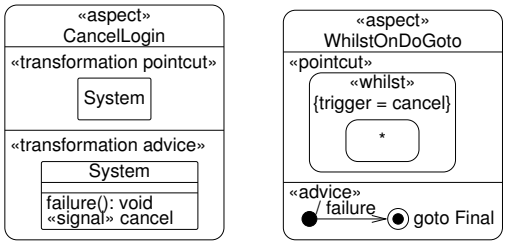


Figure 8.10: Base sub-state machine for Login

This use case is modeled in see Fig. 8.10. Since the use case is used twice (in use cases 1 and 3) the login procedure is rendered as a sub-state machine that takes the validation procedure as template parameter. The class System contains receptions for entering the user id and password and for accepting a successful login attempt. The behavior of System is a direct reflection of the use case description: first the user is prompted to input his credentials (state Prompt), which are then validated (Validate). If the credentials are correct, a signal ok is created by the validation mechanism (not modeled here), upon which the use ends in success. The parameter Op is supposed to be instantiated with an appropriate validation method which sends a signal ok when the validation the credentials are correct.

The first extension is specified as follows:

2a. CMSEmployee *cancels the authentication process. Use case ends in failure.*



(a) Static aspect (b) Dynamic aspect

Figure 8.11: Aspects for extension 2a of Use Case 10

Ext.	Template	Base	Binding
$6b$	IntroduceOperation	Fig. 8.2	C ↦ c, c ∈ {Assignment, Request} Op ↦≪signal≫ soNotifyArrival(r: Resource)
	WhilstOnDoGoto	Fig. 8.7	Whilst ↦ WaitForArrival On ↦ soNotifyArrival(r) Do ↦ arrived.add(r) Goto ↦ OneArrived
$6c$	IntroduceOperation	Fig. 8.2	C ↦ Resource Op ↦≪signal≫ updateRequired()
	WhilstOnDoGoto	Fig. 8.7	Whilst ↦ WaitForArrival On ↦ after t time Do ↦ resource.updateRequired() Goto ↦ WaitForArrival
$7a$	IntroduceOperation	Fig. 8.2	C ↦ Adviser Op ↦ ≪signal≫ moreMissionsRequired
	WhilstOnDoGoto	Fig. 8.6	Whilst ↦ WaitForChanges On ↦ moreMissionsRequired Do ↦ crisis.createAdviser() Goto ↦ WaitForChanges
$7b$	IntroduceOperation	Fig. 8.2	C ↦ Adviser Op ↦ missionFailed(m: Mission)
	WhilstOnGoto	Fig. 8.6	Whilst ↦ WaitForChanges On ↦ missionFailed(m) Goto ↦ RecommendMissions
$8a$	IntroduceOperation	Fig. 8.2	C ↦ c, c ∈ Assignment, Request} Op ↦≪signal≫ soNotifyDeparture(r: Resource)
	WhilstOnDoGoto	Fig. 8.7	Whilst ↦ WaitForDeparture On ↦ soNotifyDeparture(r) Do ↦ left.add(r) Goto ↦ OneLeft
$8b$	IntroduceOperation	Fig. 8.2	C ↦ c, c ∈ {Assignment, Request} Op ↦≪signal≫ delayReasonRequired
	WhilstOnDoGoto	Fig. 8.7	Whilst ↦ WaitForDeparture On ↦ after t time Do ↦ resource.delayReasonRequired() Goto ↦ WaitForDeparture
$9a, 12a$	IntroduceOperation	Fig. 8.2	C ↦ Adviser Op ↦ ≪signal≫ changeRequired
	WhilstOnDoGoto	Fig. 8.6	Whilst ↦ WaitForChanges On ↦ changeRequired Do ↦ crisis.createAdviser Goto ↦ Final
$11a$	WhilstOnGoto	Fig. 8.6	Whilst ↦ WaitForChanges On ↦ after t time Goto ↦ Final

Table 8.5: UC 1: extensions

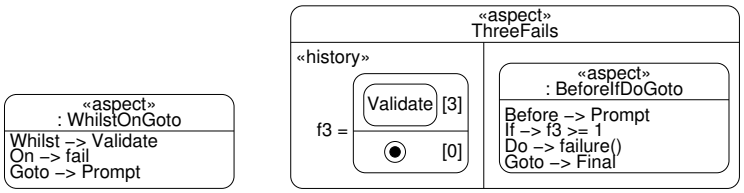
As is the general strategy in our modeling approach this extension is represented by aspects; in this case a combination of one static and one dynamic aspect, as shown in Fig. 8.11. The pointcut of the dynamic aspect in Fig. 8.11b matches ≪whilst≫ the base state machine is in any (*) state configuration and the event cancel occurs; it advises the base state machine to execute the ≪advice≫: perform failure as an effect and go to the Final state of the base machine. The static aspect in Fig. 8.11a adds the reception cancel and the operation failure (and also the implementation of failure) to System.

The second and last extension of use case 10 is handling failed authentication attempts:

3a. System *fails to authenticate the* CMSEmployee.
    3a.1. *Use case continues at step 1.*

3a.1a. CMSEmployee *performed three consecutive failed attempts.*
3a.1a.1. *Use case ends in failure.*

Modeling such behavior requires the ability to track the execution history of the state machine. This is what the history properties of HILA are designed for.



(a) Aspect fail (b) Aspect 3fail

Figure 8.12: Dynamic aspects for extension 3a of Use Case 10

The dynamic aspect in Fig. 8.12a uses the template notation: a failed authentication attempt leads back to state Prompt. The dynamic aspect in Fig. 8.12b is additionally equipped with a ≪history≫ property. The history variable f3 stores the number of such subsequences in the execution history in which state Validate was active three (multiplicity [3]) times without the final state being active (multiplicity [0]). The history variable f3 is used in the enclosed aspect template instance to ensure that after three consecutive failed authentication attempts (f3 $>=$ 1) the machine terminates.

8.3. Validation of the Model

Since result of weaving HILA aspects with the base machine is another plain UML state machine, it is possible to validate HILA aspects by formally prove the correctness of the weaving result, be it theorem proving (like PVS [7] or KIV [10]) or model checking (like UMC [30] or Hugo/RT [45]). Section 5.8.3 contains an example of model checking the result of mutual exclusion aspects. We now validate the result of weaving the HILA aspects of the (simplified) Use Case 10 "Authenticate User", see Sect. 8.2.3 by model checking.

According to our method, we start out with the base state machine in Fig. 8.10, ignoring the operation Op and just concentrating on the basic login steps of the main success scenario: prompting for user id/password, accepting an input, and validating it; only a successful validation resulting in an ok signal is reflected in this model (in the following we ignore how the input provided by the user is stored). The use case extensions, represented by the aspects in Figs. 8.11 and 8.12, require the possibility of canceling the login procedure and the handling of a failing validation. In particular, it has to be possible to attempt to login at least, but also at most, three times unsuccessfully. Thus the (woven) state machine should exhibit the following property, stated in linear temporal logic (LTL):

```
F (inState(Prompt) and F (not inState(Prompt) and
F (inState(Prompt) and F (not inState(Prompt) and
F (inState(Prompt)))))
```

The temporal modality F is to be read as "eventually" or "it is the case in the future"; thus it should be possible that the state machine first goes to state Prompt,

then to some other state, then to Prompt again, then to some other state, and finally to Prompt again. Note that the property only refers to states in the base state machine. Obviously, the desired behavior is not possible in the original base state machine in Fig. 8.10, and this is also confirmed by Hugo/RT: Hugo/RT translates the state machine and the assertion into the input language of a back-end model checker, in this case SPIN. SPIN then verifies that there is no possible run of the state machine with the prescribed sequence of being in Prompt and being not in Prompt.

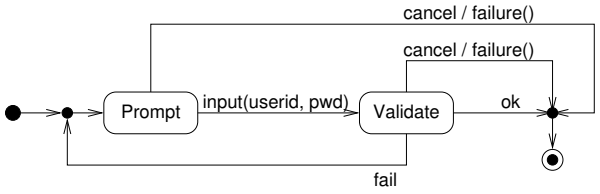


Figure 8.13: Weaving result for sub-machine login from Fig. 8.10 and the aspects in Fig. 8.11b, Fig. 8.12a (unnecessary jBefore junctions removed)

The result of weaving in the aspects in Fig. 8.11b and Fig. 8.12a is shown in Fig. 8.13. Canceling a login process and the failing of a login attempt are represented rather straightforwardly by additional transitions (triggered by cancel and fail). In this resulting state machine the property stated above is possible, as confirmed by Hugo/RT and SPIN; the following property, however, stating the possibility to try to login four times, is also satisfied:

```
F (inState(Prompt) and F (not inState(Prompt) and
F (inState(Prompt) and F (not inState(Prompt) and
F (inState(Prompt) and F (not inState(Prompt) and
F (inState(Prompt)))))))
```

By employing a history property, the aspect in Fig. 8.12b ensures that at most three failing attempts in a row can be made. The weaving result is shown in

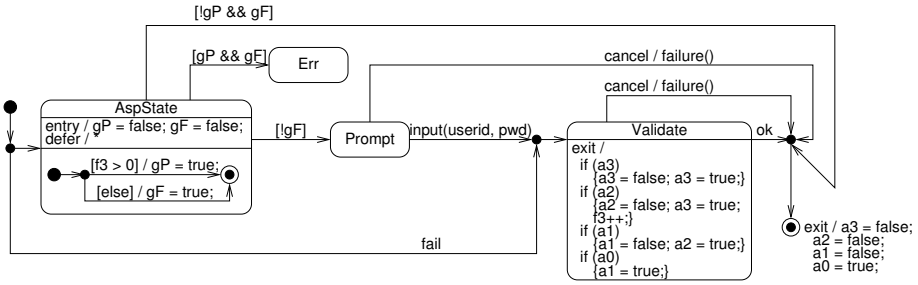


Figure 8.14: Weaving result for sub-machine login from Fig. 8.10 and Figs. 8.11b and 8.12 (unnecessary jBefore junctions removed)

Fig. 8.14.[2] Note `a0` is initialized with 0 (not shown in the diagram). Now Hugo/RT

[2]For simplicity, we removed the jBefore junctions and priority handling. There is only one aspect involved in AspState.

(and SPIN) confirm that it is possible to have three unsuccessful attempts to login, but no more attempts are then possible.

It is in general not only useful to check that certain properties are indeed enforced by applying aspects, but also, conversely, that certain properties are being preserved. A simple example for preservation of properties is that the base state machine can terminate:

```
F (inState(Final))
```

Hugo/RT and SPIN verify that this property also holds for the woven state machine (in fact this could also be checked by inspection; however, here more is true: all executions of the woven state machine inevitably lead to the Final state).

As demonstrated for the authentication use case, we can currently only check the result of the weaving process; it would be desirable to compositionally validate the base state machine and the aspects in a dynamic fashion, without having to take into account the weaving result which can be rather complex and sometimes slightly unintuitive.

8.4. Discussion

In the following, we will evaluate how well the HiLA approach to modeling worked for the CCCMS case study, which language features of HiLA we used in the modeling task, and how well they addressed the issues presented by the domain.

8.4.1. The HiLA Approach to Modeling. Overall, we were pleasantly surprised how well our simple approach of modeling the main success scenario in a base machine and the extensions withaspects worked for the CCCMS case study, in particular since the use cases were developed without special consideration for, and most likely even without knowledge of, HiLA. In general, each base state machine corresponds to a single use case and each use case extension is modeled by one or more aspects; each aspect belongs to a single use case, and the behavior of most use case extensions could be modeled with high-level aspects. This affords excellent correspondences between requirements and design, albeit at a certain increase in the number of interacting model elements. In practical applications less strict adherence to this method might be advisable since it can reduce the composition complexity. Tool support for interactively switching on and off aspect weaving would be very helpful.

Many structural patterns repeatedly appear in different use case extensions of the CCCMS case study, e.g. "while waiting in some state S, if event e happens, do something not foreseen in the base use case." Since HiLA provides a highly expressive template language for defining aspects, most of the extensions can be concisely summarized in tabular form, see Tables 8.1–8.5. We employ a regular naming scheme for templates and reuse these templates in most of our HiLA models. With some experience it becomes therefore easy to see which behavioral modifications are required by the listed extensions. In effect, we use the template language of HiLA for tailoring aspects to different contexts and base state machines, and thereby achieve a high degree of aspect reuse.

While modeling the CCCMS we sometimes had to deviate from the simple, systematic approach described in the previous paragraph. These deviations were necessary to accommodate the unbounded parallelism that is present throughout

the case study: state machines themselves can only provide a statically fixed number of parallel regions, the dynamic "spawning" of new parallel regions cannot be represented in the state machine formalism. Therefore we have to model dynamically created concurrent regions as concurrent objects; each object corresponds to one parallel "thread" of execution, the behavior of the thread is given by the object's state machine.

This pattern for managing unbounded parallelism is the reason why the active part of the system is represented by different objects over time and, consequently, some base use cases are modeled by several state machines. For example, the single System instance creates a new instance of Crisis for every crisis report received by the system, the state machines of the concurrently executing Crisis instances are then responsible for handling all simultaneously active crises.

A similar situation can be observed in use case extensions. Except for a certain pattern of extensions for UC 1, each extension is described by a set of aspects that can be modeled without modification to the base state machine and without knowledge of other extensions. However, the case is not so simple for extensions 5a.1, 7a.1, 7b.1, 9a.1 and 12a.1 of UC 1: step 2 of the base use case ("system recommends missions to coordinator") is specified as a straightforward, serial part of the main work flow and would therefore be modeled as part of the base state machine. In contrast to the sequential behavior of the base use case, the extensions specify "use case continues in parallel with step 2" and thereby lead to unbounded parallelism in the base state machine. We therefore have to model the recommendation, selection and change monitoring of missions in a new class Adviser and start a concurrent Adviser instance every time new missions have to be created.

We point out one other potential pitfall that did not arise in the CCCMS scenario: some modeling shortcuts, such as replacing several linearly connected states with a single default transition, are not applicable when working with the proposed HILA approach. For example, in UC 2, step 2a.3 ("system validates information received from the phone company"), which is represented by the transition from Validate to OK in Fig. 8.10, it is tempting not to use an explicit event phoneInfoOK and a transition into the subsequent state OK but to model success by a completion transition from Validate to the join (as we are focusing on the main success scenario). However, indulging this temptation greatly complicates the aspect that introduces the additional possibility that the phone company does not match the witness info, as required by extension 5a.

The success of an approach that relates behavioral design models as closely to use cases as HILA depends heavily on the quality of the requirements analysis. The CCCMS illustrates that it is not necessary to develop requirements models in a specialized manner in order to profit from the abstraction mechanisms provided by the HiLA language: for large parts of the CCCMS the relation between HiLA models and requirements is immediately apparent, and even the necessary deviations from a "pure" approach exhibit a large degree of regularity and are easy to understand; the application of these patterns to other use cases is straightforward. Nevertheless, the correspondence between requirements and design models could have been further improved by writing use cases in a way that takes into account the limitations that state machines place on parallelism, i.e., by separating all situations exhibiting unrestricted parallelism into separate use cases.

Another issue that arises when going from an informal description, such as use cases, to an executable formalism like state machines is that there may be imprecisions or possibilities for misunderstandings in the informal text. The CCCMS case study was for the most part free from such imprecisions, which shows the care that went into its creation. Nevertheless there were a few isolated examples, where the descriptions of different use cases do not seem to match exactly, e.g. UC 1, extension 5a is triggered when an external "resource is not available after step 5." UC 4, which is referenced by step 5, may either end "in success," "in degraded success," or "in failure." It is, however, not made clear whether any of these conditions is the same as a resource not being available, and if so, how "degraded success" should be handled in UC 1.

8.4.2. The HiLA Language. The requirements of the CCCMS could be satisfied with relatively basic language features of HiLA: we used a number of high-level aspects and aspect templates for state machines, as well as some low-level aspects either to introduce new classes or to add operations and properties to existing classes.

This case study shows the value of HiLA's ability to efficiently cope with scenarios that require more sophisticated handling of execution history, see Sect. 8.2.3.

The HiLA language allows modelers to easily define templates that represent commonly used aspects for their scenarios. This can be seen in this case study: roughly 80% of the aspects needed to model the use case extensions of the CCCMS can be expressed as instantiations of a small number of aspect templates. This expressivity of the modeling language is not without risks: it is tempting to overuse templates which can quickly lead to inscrutable models, in which (parameterized) aspects no longer correspond to single requirements, and where changes to a single aspect template may have unforeseen consequences throughout the model.

Certain aspects, e.g. aspects that need to introduce new guards into existing transitions, can only be modeled by graph transformations of state machine models. No such example is necessary for the CCCMS, but had we modeled UC 2 as described above (in Sect. 8.4.1 on p. 120) such an introduction would have been necessary. Applying several graph transformations to a base state machine raises concerns about the confluence of the transformations and therefore the well-formedness of the final result [75]. This is a problem that our approach shares with all other approaches that make use of graph transformations. However, as can be seen in this case study, we can model many scenarios without resorting to this mechanism. Note that we use static aspects for class diagrams, but only to introduce new classes, methods or properties. In these cases confluence is normally not problematic and well-formedness of the result easily checked.

When only high-level aspects are applied to state machines, the problem becomes much less pressing: in most cases the concurrent execution of advices reduces the number of spurious conflicts between aspects; in particular there are no conflicts when several mutually independent aspects are applied to the same transition. Cases where several aspects interfere in HiLA generally represent a real conflict between different behaviors that has to be resolved by the modeler. Moreover, conflicting modifications of control flow by several concurrently active aspects can faithfully be detected at run time and a conservative static approximation can point out all potential conflicts of this kind at design time. Still, there are interactions

which we currently do not detect reliably, e.g. consumption of an event that is deferred by the base state machine or another aspect. While these situations appear much less frequently than (spurious) interactions between graph transformations it is our intention to improve the HiLA tools to detect and warn about this last class of indeterministic behavior.

The HiLA language is amenable to testing and verification. Templates allow us to test the behavior of aspects by applying them to simple base state machines, and by applying compositions of several aspects simultaneously. Moreover, since HiLA is integrated with the Hugo/RT model translation tools for state machines, it is straightforward to apply model-checking techniques to HiLA models, and therefore to validate models against behavioral specifications. Currently this is only possible after weaving and therefore non-compositional. Independent verification of individual aspects remains a challenge and is a subject for future research.

CHAPTER 9

Related Work

Contents

9.1. Event Condition Action Systems and Programming Languages 123
9.2. Modeling Languages Supporting Static Aspects 123
9.3. Modeling Languages Supporting Dynamic Aspects 124
9.4. Aspect Interference 125

Aspect-oriented Software Development has been a very active research topic for more than a dozen years. HILA is related to a large number of publications.

Publication Notice. The main content of this chapter has been published in [35]. The discussion about ECA systems is new.

9.1. Event Condition Action Systems and Programming Languages

The idea of defining rules in the form of "when X happens do Y" is not young. In fact, HILA can also be viewed as a special form of the well-known *Event Condition Action* paradigm (ECA). Since HILA is defined to be an extension of UML state machines, the implementation of HILA aspects has to take into account many of the subtlenesses of the UML state machine semantics, see Chap. 5.

In prevalent ECA systems like Drools [9] the X is often only dependent of an event and a condition and not, as opposed to HILA, any system state. Consequently, ECA rules are usually applied in systems where the current state is irrelevant for the determination of the action to execute, such as information systems, see e.g. [58, 60]. In comparison, HILA is defined to enhance the modularity of UML state machine models, which are usually used to model state-based behaviors.

Languages constructs, whose semantics was defined using run time information, are also available since the very beginning of main stream aspect-oriented programming, see e.g. cflow of AspectJ [39]. More expressive constructs are designed in other languages like JAsCo [69, 70] Object Teams [34] and Arachne [22], all of which support the definition of (parts) of execution traces as pointcuts. HILA's history properties are actually reminiscent of the *stateful* or *trace-based* aspects in these languages. An overview of dynamic aspect-oriented programming languages can be found in [28].

9.2. Modeling Languages Supporting Static Aspects

As stated before, in most aspect-oriented modeling approaches aspects are understood as model transformations: an aspect defines some modification of the elements of the base model. In [61, 66], primitive directives of model modifications are defined. A static aspect can be viewed as a combination of such directives.

Theme/UML [14] is one of the best known static approaches. It models different features in different models (called themes) and uses UML templates to define common behavior of several themes. Like in HILA, Theme/UML does not include a pointcut language, reusability is achieved by defining templates. Theme/UML does not provide a method of conflict detection, but simply proposes to have an expert to supervise the aspects and to resolve conflicts if necessary.

Other approaches, e.g. [66, 75, 79] or, applying the idea of aspect-orientation for requirements engineering, [6], include a selection language for pointcut definition. The advice is reused when the pointcut, a pattern, is matched by several parts of the base model and the advice (the transformation of the base model) is woven everywhere the pointcut is matched. The semantics of aspects in these approaches is defined using graph transformation [24], the approaches are also implemented as graph transformation, see [53, 72, 74]. Detection of (syntactical) conflicts between transformations (i.e. aspects) then amounts to confluence checking in the graph transformation system. Detection of semantical conflicts does not seem directly supported. A case study of applying static aspects to the development of software product lines is given in [38].

While static aspects are helpful for separating the system behavior in several parts, they do not provide decisive aid for reducing the complexity of software design models, since model transformations are low-level instruments, and the modeler using static aspects (which are in actually model transformations) still has to define the system behavior in every detail. For an example, see Sect. 3.5.

On the other side, static aspects are generally more expressive than high-level approaches including HILA can be, since the weaving of every high-level aspect can also be formulated as static aspects. As shown in Chap. 8, HILA is often used in combination with static aspects, which, e.g., extends the static structure of the base model.

9.3. Modeling Languages Supporting Dynamic Aspects

The pointcut language JPDD [31] facilitates the definition of trace-based pointcuts. In comparison, HILA also allows the modeler to define synchronization related pointcuts. Moreover, an elaborate weaving algorithm is necessary in HILA to implement the advices of the aspects.

Altisen et al. [4, 63] propose aspects for Mealy automata. Pointcuts are also defined as automata. In comparison, the implementation of HILA's history properties using automata is transparent to the modeler, which means the modeling of history-based behaviors is more declarative hence easier. Moreover, since HILA is defined as an extension of UML state machines, the weaving algorithm of our approach is also much more elaborate, mainly due to the richer language constructs of the UML. Because of the wider acceptance of the UML, HILA is supposed to be more tightly connected to common practice.

Considering UML state machines, Mahoney et al. [50] 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. Modeling of history-based features is not covered by this approach.

State-based aspects in reactive systems are also supported by the Motorola WEAVR tool [17, 18, 84]. Their aspects can be applied to the modeling approach Telelogic TAU[1], which supports flat, "transition-centric" state machines. In comparison, our approach is also applicable to UML state machines, where in general concurrent threads are contained, and concerns such as thread synchronization increase the difficulty of correct and modular modeling. We believe that the aspect language presented in this paper, and the verification tools available because of its integration into Hugo/RT, provide valuable help to address these problems.

To the author's knowledge, techniques for conflict detection and resolution are not provided by these approaches.

9.4. Aspect Interference

Aspect interference is an intrinsic problem of aspect-oriented techniques. It has been addressed in a large amount of publications, at least for aspect-oriented programming languages (for an overview, see [2, 47]).

Techniques interference detection proposed so far include detecting shared fields addressed by read and write operations [65], a formal foundation of AOP languages using the logical approach Conditional Program Transformations [47], and graph transformations [2]. These approaches focus on sequential programming languages. In comparison, our approaches exploits the concurrency of state machines and weaves aspects into parallel regions to mitigate the problem of join points being changed or made unreachable by other aspects. Notations of precedence declaration in order to resolve conflicts are proposed in, e.g. [42, 55, 61, 84].

Weaving into parallel constructs is also proposed in [21], where an approach to concurrent event-based AOP (CEAOP) is defined. Concurrent aspects can be translated into Finite Sequential Processes and checked with the LTSA model-checker. Many similarities exist between CEAOP and the work presented in this paper; however the two approaches take complementary viewpoints in the sense that our work is primarily concerned with a state-based view of AOP that allows, e.g., the definition of mutual exclusion in state machines, whereas the CEAOP is mostly concerned with aspects over event sequences. CEAOP provides operators to combine aspects, e.g., by executing different aspects in sequence or in parallel; our approach is more restricted since our aspects are always executed in parallel. Furthermore, pointcuts in CEAOP are actually similar to sequences of pointcuts according to the usual definition, and pieces of advice are executed at different points of this sequence. This makes it easy to define stateful aspects. While our history mechanism can also be used to define these kinds of aspects, the definition has to be given in several parts and is more cumbersome than in CEAOP. On the other hand, the history mechanism in our approach can take into account values of context variables which significantly increases the expressive power; it seems that this possibility does currently not exist in CEAOP.

Considering modeling languages, since static aspects are implemented by graph transformation [11], syntactical conflicts between aspects amount to inconfluence of the underlying rewriting system that can be detected automatically by graph

[1] http://www.telelogic.com/products/tau/index.cfm

transformation tools like Attributed Graph Grammar (AGG)[2] [75]. However, detection of other kinds of interference, such as conflicting resumption strategies, is not directly supported.

[2] http://tfs.cs.tu-berlin.de/agg/

CHAPTER 10

Conclusions and Future Work

Contents

10.1. Summary 127
10.2. Future Work 128

We conclude this thesis by giving some summarizing remarks on HILA and sketch some future work.

10.1. Summary

We have presented our approach to aspect-oriented modeling using the High-level Aspects (HILA), an aspect-oriented extension of UML state machines [57]. The most distinguishing feature of HILA's modeling language is its high abstraction level. That is, HILA aspects do have a dynamic semantics, whereas in other prevalent approaches of aspect-oriented state machines the aspects are defined as syntactical modification of elements of of the base model. The semantics of HILA is given in a structural operational manner, and implemented in the weaving algorithms.

The high abstraction level considerably simplifies the complexity of the models. With HILA, the modeler only needs to define *what* is supposed to be done by an aspect instead of having to design in every detail *how* to do it. We showed examples of how simple it is to model history-based features and synchronization of parallel systems in HILA, which is generally not an easy task using plain UML state machines or using static aspects.

The high degree of separation of concerns of HILA models enables the use of an interesting design method in which the alignment of use cases (including extensions) and design models (base machines and aspects) is very clear. This way we achieve a high traceability between requirements and design artefacts.

HILA is supported tool Hugo/HILA, which reads textual definition of the base machine and HILA aspects, and accomplishes weaving according to the algorithms described in this thesis. Model checking HILA aspects, also performed by Hugo/HILA, helps detect modeling mistakes at an early stage of the development process. Reusability of HILA aspects is greatly enhanced by defining templates to model recurring situations. Our experiences made in modeling the CCCMS case study give rise to the assumption that the Pareto Principle is also valid here, i.e. when modeling a real system, about 80% of the necessary HILA aspects can be defined as instantiations of simple aspect templates.

Apart from the CCCMS case study, HILA has been applied to model adaptive systems [80].

10.2. Future Work

The results of this thesis can be extended in several directions.

It is not surprising that a high-level language like HILA does not exhibit an as high expressive power as low level, i.e. transformational aspect-oriented languages do. We plan to extend HILA to cover behaviors that are currently hard to model. For example, synchronization of parallel regions is currently implemented by waiting passively until firing the advised transition would no longer break any desired mutual exclusion. Language constructs indicating another implementation strategy, e.g. actively instructing the base machine to get out of the other,, problematic states may be also useful, see [81]. Another desirable language feature would be the synchronization between concurrent objects (see [35] for an example) or to allow the modeler to distinguish between applying advice at the start of the transition execution, i.e., before the effect of the transition takes place, or at the end of the transition (after its effect). Similarly, more complex annotations than goto can be defined for final states. However these extensions potentially complicate the weaving process and the semantics of aspects; since we have not yet found it necessary to use them in practical applications, we have refrained from adding them to the language.

We defined a structural operational semantics of HILA. It provides a basis to implement the weaving algorithms. Interesting are also other formal semantics, with which we could reason about HILA aspects more directly. In particular, we are currently developing an algebraic semantics of HILA using the framework Maude [15].

In the software development process, HILA is used in the phase of analysis and design. It follows pretty well aspect-oriented requirements engineering aspects, such as AoURN [54], and may provide a starting point for aspect-oriented implementation using approaches like RAM [41]. Integrating HILA with other aspect-oriented software development approaches is also part of our future work, see [3]. Note, however, that the HILA methodology of mapping main success scenarios to base machines and use case extensions to aspects does not require aspect-oriented requirements descriptions. Moreover, this mapping may be even valuable to derive an aspect-oriented design from conventional, object-oriented requirement artefacts. We therefore plan to extend the techniques described in [77] to generate base machines and aspects from use cases in the syntax of Restricted Use Case Modeling (RUCM, [78]) automatically.

The idea of high-level aspects is not only applicable to state machines. We also plan to define high-level aspects for activity diagrams and interaction diagrams of the UML.

APPENDIX A

Remaining Use Cases of the CCCMS

Contents

A.1. Use Case 2: Capture Witness Report **129**
A.1.1. Main success scenario 130
A.1.2. Extensions 130
A.2. Use Case 3: Assign Internal Resource **130**
A.2.1. Main success scenario 132
A.2.2. Extensions 132
A.3. Use Case 4: Request External Resource **133**
A.3.1. Main success scenario 135
A.3.2. Extensions 135
A.4. Use Case 5: Execute Mission **135**
A.5. Use Case 6: Execute SuperObserver Mission **135**
A.5.1. Main success scenario 136
A.5.2. Extensions 136
A.6. Use Case 7: Execute Rescue Mission **137**
A.6.1. Main success scenario 138
A.6.2. Extensions 138
A.7. Use Case 8: Execute Helicopter Transport Mission **139**
A.8. Use Case 9: Execute Remove Obstacle Mission **139**

For each of the other use cases of the CCCMS, we first cite its textual description given in [43], then we give its modeling with HILA.

Publication Notice. This appendix was first published in [35].

A.1. Use Case 2: Capture Witness Report

Description in [43]:

Main Success Scenario:
Coordinator requests Witness to provide his identification.

1. *Coordinator provides witness information to System as reported by the witness.*
2. *Coordinator informs System of location and type of crisis as reported by the witness.*

In parallel to steps 2–4:

2a.1 System contacts PhoneCompany to verify witness information.
2a.2 PhoneCompany sends address/phone information to System.
2a.3 System validates information received from the PhoneCompany.

3. *System provides Coordinator with a crisis-focused checklist.*
4. *Coordinator provides crisis information to System as reported by the witness.*

5. System assigns an initial emergency level to the crisis and sets the crisis status to active.
Use case ends in success.

Extensions:

1a,2a. The call is disconnected. The base use case terminates.
In parallel to steps 3-4, if the crisis location is covered by camera surveillance:
3a.1. System requests video feed from SurveillanceSystem.
3a.2. SurveillanceSystem starts sending video feed to System.
3a.3. System starts displaying video feed for Coordinator.
4a. The call is disconnected.
4a.1 Use case continues at step 5 without crisis information.
5a. PhoneCompany information does not match information received from Witness.
5a.1 The base use case is terminated.
5b. Camera vision of the location is perfect, but Coordinator cannot confirm the situation that the witness describes or the Coordinator determines that the witness is calling in a fake crisis.
5b.1 The base use case is terminated.

A.1.1. Main success scenario. Modeled in Fig. A.1. The phone information of the witness as found by the phone company is stored in pInfo. It is then compared with witnessInfo (hidden in the operation validateInfo). State Validate is only left when validateInfo determines that the phone information is OK (event phoneInfoOK). The transitions to the join vertex and then to AssignLevel are only enabled when both OK and CrisisInfoReceived are active.

A.1.2. Extensions. The extensions of use case 2 are modeled by instantiating the aspect templates given in Fig. 8.4. The bindings are given in Table A.1. We model that "*Coordinator* cannot confirm the situation" by an additional signal deny. Moreover, we point out that multiple instances of IntroduceOperation are defined to extend class Crisis by signal disconnect ($1a, 2a, 4a$). The reason is that we propose to model the extensions separately from each other, and while modeling one extension we therefore assume no knowledge of other extensions. According to our definition of transformation aspects (see [79]), only one instance of the signal is actually introduced to the class.

A.2. Use Case 3: Assign Internal Resource

Description in [43]:

Main Success Scenario:

System selects an appropriate CMSEmployee based on the mission type, the emergency level, location and requested expertise. In very urgent cases, steps 1 and 2 can be performed for several CMSEmployees concurrently, until one of the contacted employees accepts the mis- sion.

1. System sends CMSEmployee mission information.
2. CMSEmployee informs System that he accepts the mission.
Use case ends in success.

Extensions:

1a. CMSEmployee is not logged in.

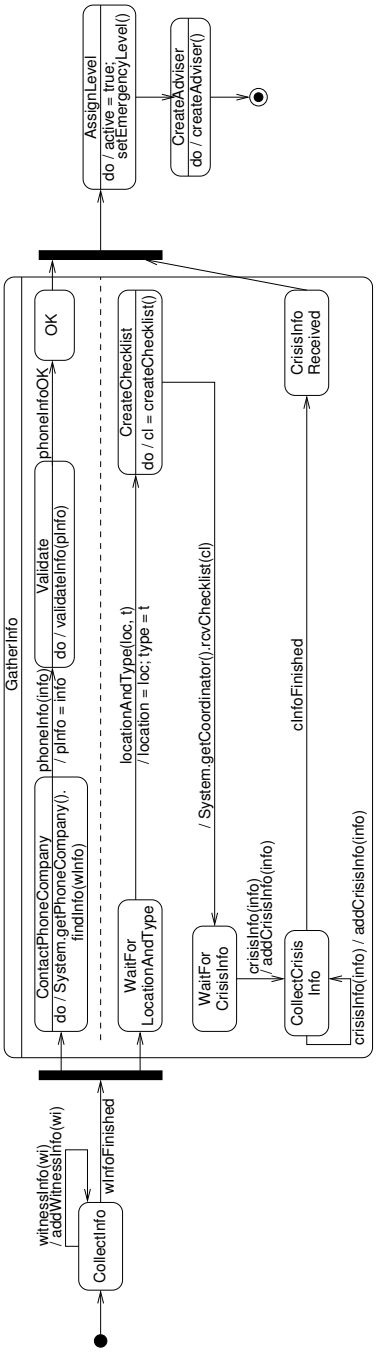


Figure A.1: UC 2: Capture Witness Report (Class Crisis: base machine)

1a.1 System requests the CMSEmployee to login.
1a.2 CMSEmployee authenticates with System (UC 10).
1a.3 Use case continues at step 1.

1b. CMSEmployee is unavailable or unresponsive.
1b.1 System selects the next appropriate CMSEmployee.

Ext.	Template	Base	Binding
$1a, 2a$	IntroduceOperation	Fig. 8.1	C ↦ Crisis Op ↦ ≪signal≫ disconnect
	WhilstOnGoto	Fig. A.1	Whilst ↦ s, s ∈ {CollectInfo, WaitForLocationAndType} On ↦ disconnect Goto ↦ Final
$3a$	IntroduceOperation	Fig. 8.1	C ↦ SurveillanceSystem Op ↦ ≪static≫ covers(loc: Location): Boolean
	IntroduceOperation	Fig. 8.1	C ↦ Coordinator Op ↦ rcvVideoFeed(vf: VideoFeed): void
	AfterIfDo	Fig. A.1	After ↦ CreateChecklist If ↦ SurveillanceSystem.covers(location) Do ↦ System.getCoordinator.rcvVideoFeed(SurveillanceSystem.createVideoFeed(location))
$4a$	IntroduceOperation	Fig. 8.1	C ↦ Crisis Op ↦ ≪signal≫ disconnect
	WhilstOnGoto	Fig. A.1	Whilst ↦ CollectCrisisInfo On ↦ disconnect Goto ↦ CrisisInfoReceived
$5a$	IntroduceOperation	Fig. 8.1	C ↦ Crisis Op ↦ ≪signal≫ phoneInfoWrong
	WhilstOnGoto	Fig. A.1	Whilst ↦ Validate On ↦ phoneInfoWrong Goto ↦ Final
$5b$	IntroduceOperation	Fig. 8.1	C ↦ Crisis Op ↦ ≪signal≫ s, s ∈ {fake, deny}
	WhilstOnGoto	Fig. A.1	Whilst ↦ GatherInfo On ↦ s, s ∈ {fake, deny} Goto ↦ Final

Table A.1: Modeling of extensions of UC 2

1b.2 Use case continues at step 1.

1b.1a No other CMSEmployee is available. Use case ends in failure.

2a. CMSEmployee informs System that he cannot accept the mission.

2a.1 System selects the next appropriate CMSEmployee.

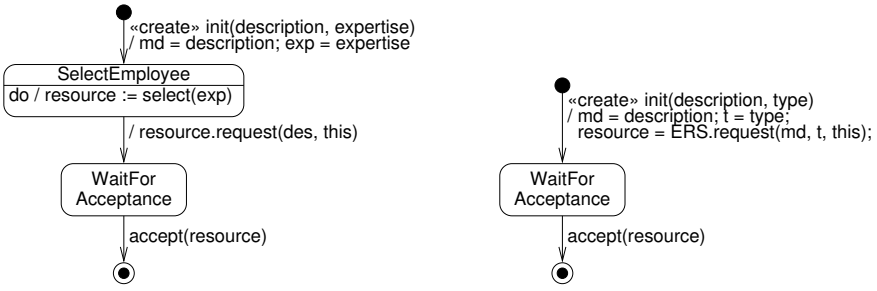
2a.2 Use case continues at step 1.

2a.2a No other CMSEmployee is available. Use case ends in failure.

A.2.1. Main success scenario. Modeled in Fig. A.2a. We assume that all "internal resources" are employees. The Assignment object passes on the mission description it received from the adviser for its own creation to the most appropriate employee (selected in SelectEmployee), as well as a reference of itself (this), for the employee to send an acceptance notification (accept).

A.2.2. Extensions. The feature "*in very urgent cases, steps 1 and 2 can be performed for several* CMSEmployees *concurrently, until one of the contacted employees accepts the mission*" is not formulated as an extension in [43]. However, we consider this an exceptional feature and decided to model it in a separate aspect.

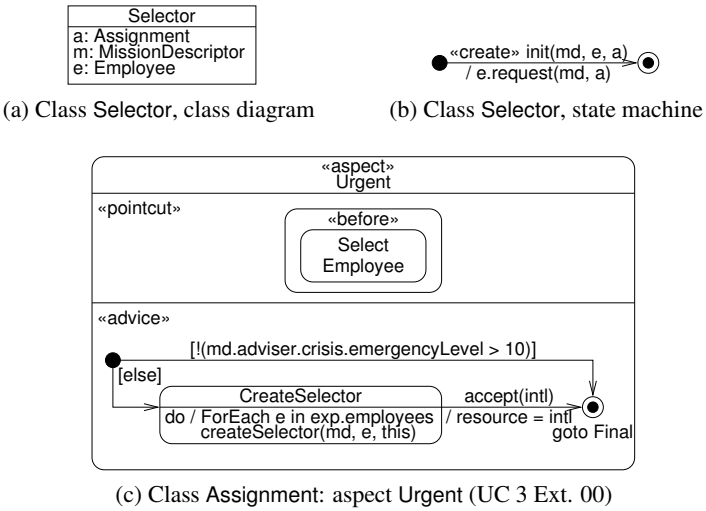
This feature is modeled by first introducing the class Selector (by instantiating the aspect IntroduceClass), which is modeled in Fig. A.3a and A.3b, and then using the aspect Urgent (Fig. A.3c) to alternate the "normal" behavior specified in the main success scenario. If the emergency level is greater than 10, which we assume is the threshold for the crisis to be "very urgent", then, instead of selecting the most appropriate employee (≪before≫ SelectEmployee), a Selector object is created for each employee with the desired expertise (exp.employees). Each of



(a) Class Assignment: base machine (b) Class Request: base machine

Figure A.2: Classes for resource allocation

these selectors then sends in parallel a request to "its" employee. As soon as any employee intl accepts the request (accept), it is assigned the mission (resource = intl), and the base machine terminates (goto Final).



(a) Class Selector, class diagram (b) Class Selector, state machine

(c) Class Assignment: aspect Urgent (UC 3 Ext. 00)

Figure A.3: Modeling of UC 3, Ext. 00

Extension $1a$ is modeled by the aspect in Fig. A.4. Again, we instantiate the template Login given in Fig. 8.10 on page 115. The needed operations are introduced by instantiations of aspect templates as given in Table A.2. The extensions 1b and 2a are modeled by instantiating the template given in Fig. A.5 with T -> t, t $\in$ {reject, after t time}, and the extension of the static structure are also given in Table A.2.

A.3. Use Case 4: Request External Resource

Description in [43]:

Main Success Scenario:

1. System sends mission request to ERS, along with mission-specific information.

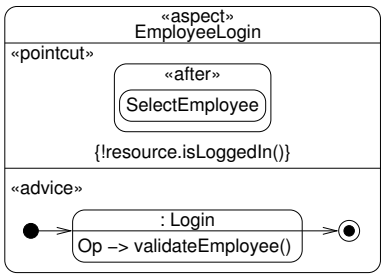


Figure A.4: UC 3, extension 1a

Ext.	Template	Base	Binding
$1a$	IntroduceProperty	Fig. 8.2	C ↦ Assignment Prop ↦ u: String
	IntroduceProperty	Fig. 8.2	C ↦ Assignment Prop ↦ p: String
	IntroduceOperation	Fig. 8.2	C ↦ Assignment Op ↦ ≪signal≫ input(u: String, i: String)
	IntroduceOperation	Fig. 8.2	C ↦ Assignment Op ↦ validateEmployee(u: String, i: String): void
	IntroduceOperation	Fig. 8.2	C ↦ Employee Op ↦ isLoggedIn(): Boolean
$1b, 2a$	IntroduceProperty	Fig. 8.2	C ↦ Assignment Prop ↦ failure: Boolean = false
	IntroduceOperation	Fig. 8.2	C ↦ Assignment Op ↦ otherEMSEmployeeAvailable(): Boolean
	IntroduceOperation	Fig. 8.2	C ↦ Assignment Op ↦ ≪signal≫ reject

Table A.2: Introducing static elements for modeling UC 3

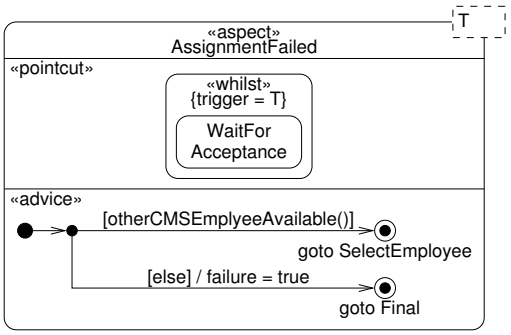


Figure A.5: UC 3, ext. 1b, 2a: bind T -> t, t ∈ {reject, timeout}

2. ERS informs System that request can be processed. Use case ends in success.

Extensions:

2a. ERS notifies System that it partially approves request for resources. Use case ends in degraded success.

2b. ERS notifies System that it cannot service the request. Use case ends in failure.

A.3.1. Main success scenario. Modeled in Fig. A.2b. The Request object passes the mission description, the desired type of external resource, as well as a reference (this), to the *ExternalRequestSystem* (ERS), and waits for some resource to accept.

A.3.2. Extensions. Again, the extensions are modeled with instances of aspect templates (Fig. 8.4) with bindings defined in Table A.3. Exit codes are introduced to indicate different results of this use case; two WhilstOnDoGoto aspects set the right exit code upon the corresponding response from the resource.

Ext.	Template	Base	Binding
$2a, 2b$	IntroduceClass	Fig. 8.2	C ↦ ExitCode = success (Fig. A.6)
	IntroduceProperty	Fig. 8.2	C ↦ Request Prop ↦ exitCode: ExitCode
	IntroduceOperation	Fig. 8.2	C ↦ Assignment Op ↦ s, s ∈ { ≪signal≫ partialApproval, ≪signal≫ denial }
	WhilstOnDoGoto	Fig. A.2b	Whilst ↦ WaitForAcceptance On ↦ partialApproval Do ↦ exitCode = degradedApproval Goto ↦ Final
	WhilstOnDoGoto	Fig. A.2b	Whilst ↦ WaitForAcceptance On ↦ denial Do ↦ exitCode = failure Goto ↦ Final

Table A.3: Aspect instantiations modeling UC 4

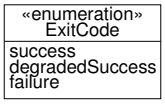


Figure A.6: Class ExitCode

A.4. Use Case 5: Execute Mission

Not modeled, for this is an abstract use case.

A.5. Use Case 6: Execute SuperObserver Mission

Description in [43]:

Main success scenario:
SuperObserver is at the crisis location.

1. *System sends a crisis-specific checklist to SuperObserver.*
2. *SuperObserver feeds System with crisis information.*
3. *System suggests crisis-specific missions to SuperObserver.*

Steps 4–8 is repeated as many times as needed.

4. *SuperObserver notifies System of the type of mission he wants to create.*
5. *System sends a mission-specific information request to SuperObserver.*
6. *SuperObserver sends mission-specific information1 to System.*
7. *System acknowledges the mission creation to SuperObserver.*

8. System informs SuperObserver that mission was completed successfully.
9. SuperObserver judges that his presence is no longer needed at the crisis location.
Use case ends in success.

Extensions:

7a. Mission cannot be created and replacement missions are possible.
7a.1 System suggests replacement missions to SuperObserver.
7a.2 Use case continues with step 4.
7b. Mission cannot be created and no replacement missions are possible.
7b.1 System suggests notifying the NationalCrisisCenter.
7b.2 Use case continues with step 4.
8a. Mission failed.
8a.1 System informs SuperObserver and Coordinator about mission failure.
8a.2 Use case continues with step 4.

A.5.1. Main success scenario. Modeled in Fig. A.7. In the use case description [43] *SuperObserver* is supposed to select *missions* to execute. Since the term *Mission* is already in use, we assume that it is (sub-)*tasks* that should be suggested to and then selected by *SuperObserver* for execution.

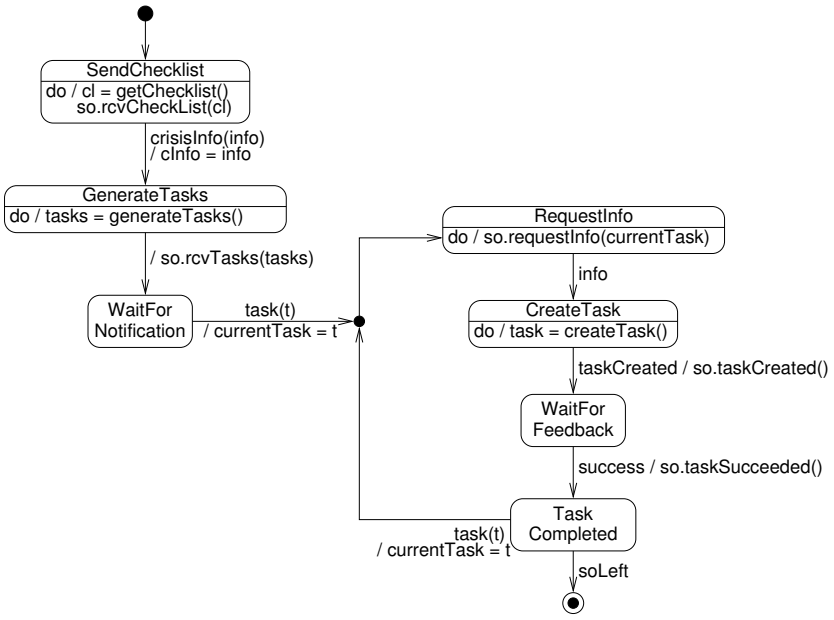


Figure A.7: SuperObserverMission.execute: base machine

A.5.2. Extensions. Extensions 7a and 7b are modeled in Fig. A.8 with introduction of static elements as defined in Table A.4. Extension 8a is modeled by the instantiations only, also given in Table A.4.

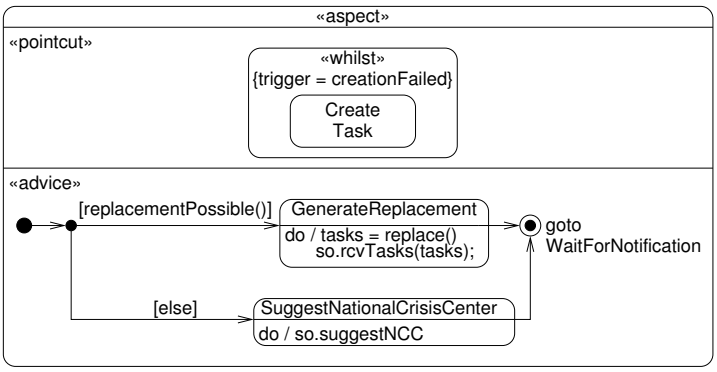


Figure A.8: Aspect for UC 6, extensions 7a/b

Ext.	Template	Base	Binding
$7a, 7b$	IntroduceOperation	Fig. 8.3	C ↦ SuperObserverMission Op ↦ ≪signal≫ creationFailed()
	IntroduceOperation	Fig. 8.3	C ↦ SuperObserverMission Op ↦ replacementPossible(): Boolean
	IntroduceOperation	Fig. 8.3	C ↦ SuperObserverMission Op ↦ replace(): Set<TaskDescription>
	IntroduceOperation	Fig. 8.3	C ↦ SuperObserver Op ↦ ≪signal≫ suggestNCC
$8a$	IntroduceOperation	Fig. 8.3	C ↦ SuperObserverMission Op ↦ ≪signal≫ taskFailed(t: Task)
	IntroduceOperation	Fig. 8.3	C ↦ SuperObserver Op ↦ ≪signal≫ taskFailed(t: Task)
	IntroduceOperation	Fig. 8.1	C ↦ Coordinator Op ↦ ≪signal≫ taskFailed(t: Task)
	WhilstOnDoGoto	Fig. A.7	Whilst ↦ WaitForFeedback On ↦ taskFailed(task) Do ↦ so.taskFailed(task); System.getCoordinator.taskFailed(task) Goto ↦ TaskCompleted

Table A.4: UC 6, aspects to introduce static elements used by the aspects

A.6. Use Case 7: Execute Rescue Mission

Description in [43]:

Main Success Scenario: FirstAidWorker is at the crisis location.

1. *FirstAidWorker transmits injury information of victim to System.*

Steps 2 and 3 are optional.

2. *FirstAidWorker determines victim's identity and communicates it to System.*
3. *System requests victim's medical history information from all connected HospitalResourceSystems.*

FirstAidWorker administers first aid procedures to victim.

4. *System instructs FirstAidWorker to bring the victim to the most appropriate hospital.*
5. *FirstAidWorker notifies System that he is leaving the crisis site.*
6. *FirstAidWorker notifies System that he has dropped off the victim at the hospital.*
7. *FirstAidWorker informs System that he has completed his mission.*
   Use case ends in success.

Extensions:

4a. HospitalResourceSystem transmits victim's medical history information to System.

4a.1 System notifies FirstAidWorker of medical history of the victim relevant to his injury.

4a.2 Use case continues at step 4.

A.6.1. Main success scenario. Modeled in Fig. A.9. The "optional" steps that *FirstAidWorker* determines victim's identity upon which the system requests the victim's medical history information from all connected HospitalResourceSystems are modeled by the iInfo.vid != null branch after WaitForInjuryInfo, where vid stands for victim ID.

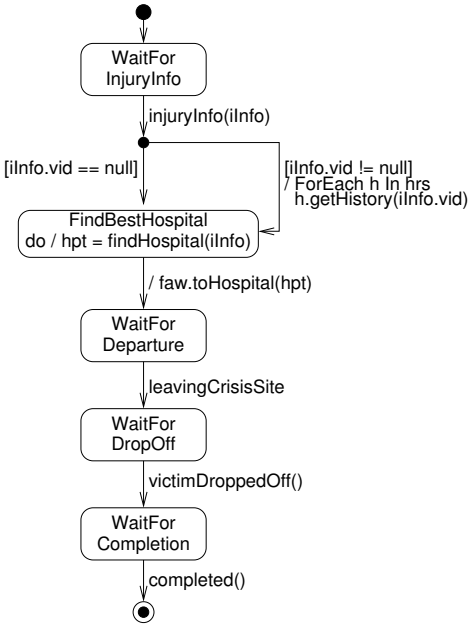


Figure A.9: RescueMission.execute: base machine

A.6.2. Extensions. The (only) extension is modeled by the instantiations of aspect templates as described in Table A.5.

Template	Base	Binding
IntroduceOperation	Fig. 8.3	C ↦ FirstAidWorker Op ↦ victimHistory(his: History)
IntroduceOperation	Fig. 8.3	C ↦ RescueMission Op ↦ ≪signal≫ victimHistory(his: History)
WhilstOnDoGoto	Fig. A.9	Whilst ↦ FindBestHospital On ↦ victimHistory(his) Do ↦ faw.victimHistory(his) Goto ↦ FindBestHospital

Table A.5: Use Case 7, extension 4a

A.7. Use Case 8: Execute Helicopter Transport Mission

Not modeled, since its scenarios are not described in the case study.

A.8. Use Case 9: Execute Remove Obstacle Mission

Not modeled, since its scenarios are not described in the case study.

Bibliography

[1] http://www.aosd.net.

[2] Mehmet Aksit, Arend Rensink, and Tom Staijen. A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points. In *Proc. 8th Int. Conf. Aspect-Oriented Software Development (AOSD'09)*, pages 39–50, 2009.

[3] Mauricio Alférez, Nuno Amálio, Selim Ciraci, Franck Fleurey, Robert France, Jörg Kienzle, Jacques Klein, Max Kramer, Sebastien Mosser, Gunter Mussbacher, Ella Roubtsova, and Gefei Zhang. Aspect-Oriented Model Development at Different Levels of Abstraction. 2011. Submitted.

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

[5] Scott W. Ambler. Introduction to UML 2 State Machine Diagrams. http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm. 2008-11-03.

[6] João Araújo, Jon Whittle, and Dae-Kyoo Kim. Modeling and Composing Scenario-Based Requirements with Aspects. In *Proc. 12th IEEE Int. Conf. Requirements Engineering (RE'04)*, pages 58–67. IEEE, 2004.

[7] Tamarah Arons, Jozef Hooman, Hillel Kugler, Amir Pnueli, and Mark van der Zwaag. Deductive Verification of UML Models in TLPVS. In Baar et al. [8], pages 335–349.

[8] Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors. *Proc. 7th Int. Conf. Unified Modeling Language (UML'04)*, volume 3273 of *Lect. Notes Comp. Sci.* Springer, 2004.

[9] Michl Bali. *Drools: Jboss Rules 5.0 Developer's Guide*. Packt Publishing, 2009.

[10] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive Verification of UML State Machines. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *Proc. 6th Int. Conf. Formal Methods and Software Engineering (ICFEM'04)*, volume 3308 of *Lect. Notes Comp. Sci.*, pages 434–448. Springer, 2004.

[11] Luciano Baresi and Reiko Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. 1st Int. Conf. Graph Transformation*, volume 2505 of *Lect. Notes Comp. Sci.*, pages 402–429. Springer, 2002.

[12] Michael Blaha and James Rumbaugh. *Object-Oriented Modeling and Design with UML*. Prentice Hall, 2nd edition, 2004.

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[14] Siobhán Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design: the Theme Approach*. Addison-Wesley, 2005.

[15] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude: A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lect. Notes Comp. Sci.* Springer, 2007.

[16] Constantinos Constantinides, Therapon Skotiniotis, and Maximilian Störzer. AOP Considered Harmful. In *Eur. Interactive Wsh. Aspects in Software (EWAS'04)*, 2004.

[17] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Joinpoint Inference from Behavioral Specification to Implementation. In Erik Ernst, editor, *Proc. 21st Eur. Conf. Oriented Programming (ECOOP'07)*, volume 4609 of *Lect. Notes Comp. Sci.*, pages 476–500. Springer, 2007.

[18] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Stateful Aspects: The Case for Aspect-Oriented Modeling. In *10th Int. Wsh. Aspect-Oriented Modeling (AOM'07)*, Vancouver, 2007.

[19] Edsger W. Dijkstra. On the Role of Scientific Thought, 1974. Online accessible under http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF, 2009-12-04.

[20] Peter Dolog. *Engineering Adaptive Web Applications*. PhD thesis, Universität Hannover, 2006.

[21] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent Aspects. In *Proc. 5st Int. Conf. Generative Programming and Component Engineering (GPCE'06)*, pages 79–88. ACM, 2006.

[22] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An Expressive Aspect Language for System Applications with Arachne. In Mezini and Tarr [52], pages 27–38.

[23] Doron Drusinsky. *Modeling and Verification Using UML Statecharts*. Elsevier, 2006.

[24] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[25] Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors. *Proc. 10th Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *Lect. Notes Comp. Sci.* Springer, 2007.

[26] José Fiadeiro. *Categories for Software Engineering*. Springer, 2004.

[27] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[28] Robert E. Filman, Michael Haupt, and Robert Hirschfeld, editors. *Proc. 2nd Dynamic Aspects Wsh. (DAW'05)*. Technical Report 05.01. Research Institute for Advanced Computer Science, 2005.

[29] Martin Fowler. *UML Distilled*. Addison-Wesley, 3rd edition, 2003.

[30] Stefania Gnesi. Formal Specification and Verification of Complex Systems. In *Proc. 8th Int. Wsh. Formal Methods for Industrial Critical Systems (FMICS'03)*, volume 80 of *Electr. Notes Theor. Comput. Sci.*, 2003.

[31] Stefan Hanenberg, Dominik Stein, and Rainer Unland. From Aspect-Oriented Design to Aspect-Oriented Programs: Tool-Supported Translation of JPDDs into Code. In Brian M. Barry and Oege de Moor, editors, *Proc. 6th Int. Conf. Aspect-Oriented Software Development*, pages 49–62. ACM, 2007.

[32] David Harel. Statecharts: A Visual Formulation for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[33] Florian Heidenreich and Henrik Lochmann. Using Graph-Rewriting for Model Weaving in the Context of Aspect-Oriented Product Line Engineering. In *Proc. 1st Int. Wsh. Aspect-Oriented Product Line Engineering (AOPLE'06)*, 2006.

[34] Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In Mehmet Aksit, Mira Mezini, and Rainer Unland, editors, *Rev. Papers Int. Conf. NetObjectDays (NODe'02)*, volume 2591 of *Lect. Notes Comp. Sci.*, pages 248–264. Springer, 2003.

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

[36] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[37] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2005.

[38] Praveen K. Jayaraman, Jon Whittle, Ahmed M. Elkhodary, and Hassan Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In Engels et al. [25], pages 151–165.

[39] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proc. 15th Eur. Conf. Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lect. Notes Comp. Sci.*, pages 327–353, 2001.

[40] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proc. 11th Eur. Conf. Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lect. Notes Comp. Sci.*, pages 220–242. Springer, 1997.

[41] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-Oriented Multi-View Modeling. In *Proc. 8th Int. Conf. Aspect-Oriented Software Development (AOSD'09)*, pages 87–98. ACM, 2009.

[42] Jörg Kienzle and Samuel Gélineau. AO Challenge — Implementing the ACID Properties for Transactional Objects. In Robert E. Filman, editor, *Proc. 5th Int. Conf. Aspect-Oriented Software Development (AOSD'06)*, pages 202–213. ACM, 2006.

[43] Jörg Kienzle, Nicolas Guelfi, and Sadaf Mustafiz. Crisis Management Systems: A Case Study for Aspect-Oriented Modeling. *Trans. Aspect-Oriented Software Development (TAOSD)*, 7:1–22, 2010.

[44] Alexander Knapp. Semantics of UML State Machines. Technical Report 0408, Institut für Informatik, Ludwig-Maximilians-Universität München, 2004.

[45] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model Checking Timed UML State Machines and Collaborations. In Werner Damm and Ernst Rüdiger Olderog, editors, *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 2469 of *Lect. Notes Comp. Sci.*, pages 395–416. Springer, 2002.

[46] Alexander Knapp and Gefei Zhang. Model Transformations for Integrating and Validating Web Application Models. In Heinrich C. Mayr and Ruth Breu, editors, *Proc. Modellierung 2006 (MOD'06)*, volume P-82 of *Lect. Notes Informatics*, pages 115–128. Gesellschaft für Informatik, 2006.

[47] Günter Kniesel. Detection and Resolution of Weaving Interactions. *Trans. Aspect-Oriented Software Development (TAOSD)*, 5:135–186, 2009.

[48] Fred Kröger and Stefan Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. Springer, 2008.

[49] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *J. Software Tools for Technology Transfer (STTT)*, 1(1–2):134–152, 1997.

[50] Mark Mahoney, Atef Bader, Tzilla Elrad, and Omar Aldawud. Using Aspects to Abstract and Modularize Statecharts. In *Proc. 5th Int. Wsh. Aspect-Oriented Modeling*, Lisboa, 2004.

[51] Peter Marwedel. *Embedded System Design*. Springer, 2005.

[52] Mira Mezini and Peri L. Tarr, editors. *Proc. 4th Int. Conf. Aspect-Oriented Software Development (AOSD'05)*. ACM Press, 2005.

[53] Brice Morin, Jacques Klein, Olivier Barais, and Jean-Marc Jézéquel. A Generic Weaver for Supporting Product Lines. In *Proc. 13th Int. Wsh. Software Architectures and Mobility (EA'08)*, pages 11–18. ACM, 2008.

[54] Gunter Mussbacher and Daniel Amyot. Extending the User Requirements Notation with Aspect-oriented Concepts. In Rick Reed, Attila Bilgic, and Reinhard Gotzhein, editors, *Proc. 14th Int. SDL Forum (SDL'09)*, volume 5719 of *Lect. Notes Comp. Sci.*, pages 115–132. Springer, 2009.

[55] István Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing Aspects at Shared Join Points. In Robert Hirschfeld, Ryszard Kowalczyk, Andreas Polze, and Mathias Weske, editors, *Proc. Net.ObjectDays (NODe'05)*, volume P-69 of *Lect. Notes Informatics*, pages 19–38. Gesellschaft für Informatik, 2005.

[56] Object Management Group. Unified Modeling Language Specification, Version 1.5. Specification, OMG, 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[57] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.2. OMG Available Specification, OMG, 2009. http://www.omg.org/spec/UML/2.2/Superstructure.

[58] Alexandra Poulovassilis, George Papamarkos, and Peter T. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *Rev. Sel. Papers Wsh. EDBT'06*, volume 4254 of *Lect. Notes Comp. Sci.*, pages 855–864. Springer, 2006.

[59] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing Safe State Machines from Esterel. In *Proc. ACM SIGPLAN/SIGBED Conf. Language, Compilers, and Tool Support for Embedded Systems (LCTES'06)*, pages 113–124. ACM, 2006.

[60] Ying Qiao, Kang Zhong, Hongan Wang, and Xiang Li. Developing Event-Condition-Action Rules in Real-Time Active Database. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *SAC*, pages 511–516. ACM, 2007.

[61] Rughu Reddy, Sudipto Ghosh, Robert B. France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg. Directives for Composing Aspect-Oriented Design Class Models. In Awais Rashid and Mehmet Aksit, editors, *Trans. Aspect-Oriented Software Development I*, volume 3880 of *Lect. Notes Comp. Sci.*, pages 75–105. Springer, 2006.

[62] Ian Sommerville. *Software Engineering*. Addison-Wesley, 8th edition, 2007.

[63] David Stauch. Modifying Contracts with Larissa Aspects. *Electr. Notes Theor. Comput. Sci.*, 203(4):125–140, 2008.

[64] Friedrich Steimann. The Paradoxical Success of Aspect-Oriented Programming. In Peri L. Tarr and William R. Cook, editors, *Proc. 21st ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 481–497. ACM, 2006.

[65] Maximilian Störzer, Florian Forster, and Robin Sterr. Detecting Precedence-Related Advice Interference. In *Proc. 21st IEEE/ACM Int. Conf. Automated Software Engineering (ASE'06)*, pages 317–322. IEEE, 2006.

[66] Greg Straw, Geri Georg, Eunjee Song, Sudipto Ghosh, Robert France, and James M. Bieman. Model Composition Directives. In Baar et al. [8], pages 84–97.

[67] Gabriele Taentzer. AGG: A Graph Transformaton Environment for Modeling and Validation of Software. In John L. Pfaltz, Manfred Nagl, and Boris 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, 2004.

[68] Klaas van den Berg, José María Conejero, and Juan Hernández. Analysis of Crosscutting in Early Software Development Phases Based on Traceability. *Trans. Aspect-Oriented Software Development (TAOSD)*, 3:73–104, 2007.

[69] Wim Vanderperren, Davy Suvée, María Agustina Cibrán, and Bruno De Fraine. Stateful Aspects in JAsCo. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Proc. 4th Int. Wsh. Software Composition (SC'05)*, volume 3628 of *Lect. Notes Comp. Sci.*, pages 167–181. Springer, 2005.

[70] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive Programming in JAsCo. In Mezini and Tarr [52], pages 75–86.

[71] Michael von der Beeck. Formalization of UML-Statecharts. In Martin Gogolla and Cris Kobryn, editors, *Proc. 4th Int. Conf. Unified Modeling Language (UML'01)*, volume 2185 of *Lect. Notes Comp. Sci.*, pages 406–421. Springer, 2001.

[72] Susanne Wagner. Entwicklung Eines Modellierungswerkzeugs für Aspekt-orientierte Klassendiagramme. Project thesis, Ludwig-Maximilians-Universität München, 2006. In German.

[73] Stephan Weißleder, Dehla Sokenou, and Bernd-Holger Schlingloff. Reusing State Machines for Automatic Test Generation in Product Lines. In *Proc. 1st Int. Wsh. Model-based Testing in Practice (MoTiP'08)*, 2008.

[74] Jon Whittle, Praveen K. Jayaraman, Ahmed M. Elkhodary, Ana Moreira, and João Araújo. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. *Trans. Aspect-Oriented Software Development (TAOSD)*, 6:191–237, 2009.

[75] Jon Whittle, Ana Moreira, João Araújo, Praveen K. Jayaraman, Ahmed M. Elkhodary, and Rasheed Rabbi. An Expressive Aspect Composition Language for UML State Diagrams. In Engels et al. [25], pages 514–528.

[76] Marco Winckler and Philippe A. Palanque. StateWebCharts: A Formal Description Technique Dedicated to Navigation Modelling of Web Applications. In Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha, editors, *Proc. 10th Int. Wsh. Design Specification and Verification of Interactive Systems (DS-VIS'03)*, volume 2844 of *Lect. Notes Comp. Sci.*, pages 61–76. Springer, 2003.

[77] Tao Yue. *Automatically Deriving a UML Analysis Model from a Use Case Model*. PhD thesis, Carleton University, 2009.

[78] Tao Yue, Lionel C. Briand, and Yvan Labiche. A Use Case Modeling Approach to Facilitate the Transition Towards Analysis Models: Concepts and Empirical Evaluation. Technical Report SCE-09-05, Carleton University, 2009.

[79] Gefei Zhang. Towards Aspect-Oriented Class Diagrams. In *Proc. 12th Asia-Pacific Software Engineering Conf. (APSEC'05)*, pages 763–768. IEEE, 2005.

[80] Gefei Zhang. Aspect-Oriented Modeling of Adaptive Web Applications with HiLA. In Gabriele Kotsis, David Taniar, Eric Pardede, and Ismail Khalil, editors, *Proc. 7th Int. Conf. Advances in Mobile Computing & Multimedia (MoMM'09)*, pages 331–335. ACM, 2009.

[81] Gefei Zhang. Aspect-Oriented UI Modeling with State Machines. In Jan Van den Bergh, Stefan Sauer, Kai Breiner, Heinrich Hußmann, Gerrit Meixner, and Andreas Pleuß, editors, *Proc. 5th Int. Wsh. Model-Driven Development of Advanced User Interfaces*, pages 45–48. CEUR, 2010.

[82] Gefei Zhang and Matthias Hölzl. HiLA: High-Level Aspects for UML-State Machines. In *Proc. 14th Wsh. Aspect-Oriented Modeling (AOM@MoDELS'09)*, 2009.

[83] Gefei Zhang, Matthias Hölzl, and Alexander Knapp. Enhancing UML State Machines with Aspects. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proc. 10th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *Lect. Notes. Comp. Sci.*, pages 529–543. Springer, 2007.

[84] Jing Zhang, Thomas Cottenier, Aswin van den Berg, and Jeff Gray. Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. *Journal of Object Technology*, 6(7):89–108, 2007.

Index

Act, 48
advice, 5, 20, 30, 38, 87
≪after≫, 29, 68, 77
$\mathsf{Asp}_w(s)$,$\mathsf{Asp}_t(s)$, 67
aspect, 3, 5, 63
aspect group, 87, 92
aspect instance, 86
aspect state, 63, 93

≪before≫, 29, 68
"before" section, 56, 68, 77
≪between≫, 29, 68

clone, clone^{-1}, 49
composite state, 11, 50, 83
compound transition, 12
configuration aspect, 67
configuration pointcut, 38
configuration selector, 35
consistency checking, 23, 77, 95
CTL, 72

defer, 11, 70
dynamic aspect, 6

Err, 47, 58
e-learning, 15
error state, 43, 58
Exp, 48

graph grammar, 21

history property, 30, 33, 40, 60, 84
Hugo/HILA, 72

in_s, 57

jBefore, 56
join point, 5

l_s, 57
$\text{LCA}(v, v')$, 50
low level, 19
LTL, 72

model checking, 77, 117
mutual exclusion, 31, 73

pointcut, 5, 20, 29, 86
pre^s_{a}, 66
precondition, 66
priority, 42, 66

ρ, 89, 93
resumption, 47, 66, 93

s, s^{-1}, $\mathsf{s}_{s,t}$, 83
S, 50
$\mathsf{source}_{\mathsf{struct}}(t)$, 60
semantics, 12, 81, 90
sentinel, 9, 14, 30
simple state, 11, 83
SPIN, 72, 117
state configuration, 12
static aspect, 6, 23, 32
structured transition, 52
subvertex, 49
synchronization, 32

τ, 89, 93
T, 50
toUML, 49, 73
tracing currently active state, 57
tracing last active state, 57
transition aspect, 68
transition pointcut, 38
transition selector, 36

UML State Machine, 4, 82
Uppaal, 72
UTE, 72

V, 50

W_s, 85
weaving, 3, 21, 45, 72, 90
≪whilst≫, 29, 34, 67, 78, 85