Modeling the Car Crash Crisis Management System Using HiLA

Matthias Hölzl[1], Alexander Knapp[2], and Gefei Zhang[1]

[1] Ludwig-Maximilians-Universität München
{matthias.hoelzl, gefei.zhang}@pst.ifi.lmu.de
[2] Universität Augsburg
knapp@informatik.uni-augsburg.de

Abstract. An aspect-oriented modeling approach to the Car Crash Crisis Management System (CCCMS) using the High-Level Aspect (HiLA) language is described. HiLA is a language for expressing aspects for UML static structures and UML state machines. In particular, HiLA supports both a static graph transformational and a dynamic approach of applying aspects. Furthermore, it facilitates methodologically turning use case descriptions into state machines: for each main success scenario, a base state machine is developed; all extensions to this main success scenario are covered by aspects. Overall, the static structure of the CCCMS is modeled in 43 classes, the main success scenarios in 13 base machines, the use case extensions in 47 static and 31 dynamic aspects, most of which are instantiations of simple aspect templates.

1 Introduction

The UML has become the customary modeling language for many software development projects; state machines are among the most popular techniques for describing reactive and interactive systems [10]. Therefore, an approach based on UML state machines suggests itself for modeling the behavior of the Car Crash Crisis Management System (CCCMS). However, experience shows that state machine models become complex even for relatively simple systems. Often this complexity is not inherent in the problem itself or even in its description as state-based system, but it is induced by a lack of abstraction mechanisms in UML state machines. This becomes particularly obvious for systems exhibiting mutually interacting use cases: many of the interrelations between different use cases can only be modeled by introducing global state or by pervasive modifications of the state machine structure.

To address these problems, we have developed the HiLA approach: the HiLA language is an extension of UML state machines with aspects; it supports both high-level aspects which are based on the dynamic execution behavior of state machines, and low-level aspects defined as graph transformations of the static structure of models. HiLA allows the modeler to represent different use cases (or, more generally, different concerns) as separate aspects operating on a single base state machine. This separation of concerns is possible even if the different concerns interact, as long as the behavior of the interaction can be expressed using high-level aspects.

By closely aligning state machine models with use case descriptions, the HiLA language not only reduces the complexity of the dynamic models, but also gives rise to a modeling technique that is not easily applicable when working with standard UML state machines: the state machine models can systematically be generated from the analysis results, in particular from use cases. To achieve this, we model each base use case with a state machine, and we model each use case extension as a set of aspects that are applied to one or more base machines. This is a similar technique to the one described by Jacobson's rendering of use case slices as aspects [15]. The development of the static design model is more traditional, and takes into account both the analysis model and the requirements of the dynamic models.

The systematic transition between analysis and behavioral design offers several advantages: it provides a concrete method to move from use cases to state machines; 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.

While the part of the CCCMS we modeled is much 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. However, as we will further elaborate in Sect. 5, HiLA is designed to address certain modeling problems, e.g., complex mutual exclusion requirements or history-dependent behaviors, which are not needed to model the CCCMS case study.

For our exposition, we assume that the reader is familiar with the syntax and semantics of UML in general (see [6] for an introduction). In particular, we make use of UML's template mechanism (see [23, Sect. 17.5] for its specification).

The remainder of this paper is structured as follows: Sect. 2 briefly reviews UML state machines and gives an overview of the HiLA language. The modeling of the CCCMS in HiLA is detailed in Sect. 3; Sect. 4 demonstrates how the resulting HiLA model can be validated. The method of applying HiLA to the CCCMS case study, and the HiLA language itself are evaluated in Sect. 5. Related work is discussed in Sect. 6. We conclude by summarizing our approach in Sect. 7.

2 A Brief Overview of HiLA

Our modeling language of choice is the UML [23], the lingua franca of object-oriented analysis and design. The static structure of our CCCMS is modeled by class diagrams, its dynamic behavior by state machines, enhanced with HiLA. In the following, we review the basic concepts of UML state machines (see, e.g., [10]) and introduce HiLA.

2.1 UML State Machines

A UML state machine provides a behavioral view of its context. Figure 1 shows a state machine of a process which contains two parallel threads. After creation, the process is first initialized in the state Init, then the two threads run in parallel in the state Running, until they receive the events astop and bstop while they are in the states A3 and B3,

respectively. In this case, each thread waits for the other to receive its stop signal in waiting states A4 and B4, respectively, before the threads terminate conjointly.

In more detail, a UML state machine consists of *regions* which contain *vertices* and *transitions* between vertices. A vertex is either a *state*, where the state machine may dwell in and which may show hierarchically contained regions, or a *pseudo state* regulating how transitions are compound in execution. Transitions are triggered by *events* and describe, by leaving and entering states, the possible state changes of the state machine. The events are drawn from an *event pool* associated with the state machine, which receives events from its own or from different state machines. The *context* of a state machine, a UML classifier, describes the features, in particular the attributes, which may be used and manipulated during execution.

A state of a state machine is *simple*, if it contains no regions (such as Init and all states contained in Running in Fig. 1); a state is *composite*, if it contains at least one region; a composite state is said to be *orthogonal*, if it contains more than one region, visually separated by dashed lines (such as Running). Each state may show an *entry* behavior (like actB2 in B2), an *exit* behavior (like actA2 in A2), which are executed on activating and deactivating the state, respectively; a state may also show a *do activity* (like in Init) which is executed while the state machine sojourns in this state. Transitions are triggered by events (a12, a23), show guards (condB), and specify actions to be executed when a transition is fired (actA23). Completion transitions (transition leaving Init) are triggered by an implicit *completion event* emitted when a state completes all its internal activities. Events may be *deferred* (as a12 in Init), i.e., put back into the event pool, if they are not to be handled currently but only later on. By executing a transition, its source state is left and its target state entered; transitions, however, may also be declared to be *internal* to a state (b3 / actB3), thus skipping the activation–deactivation scheme. An *initial* pseudo state, depicted as a filled circle, represents the starting point for the execution of a region. A *final* state, depicted as a circle with a filled circle inside, represents the completion of its containing region; if all regions of a state machine are completed, the state machine terminates. *Junction* pseudo states, also depicted as filled circles (see lower region of Running), allow for case distinctions. Transitions to and from different regions of an orthogonal composite state can be synchronized by *fork* and *join* pseudo states, presented as bars. For simplicity, we omit the other pseudo state kinds (entry and exit points, shallow and deep history, and choice).

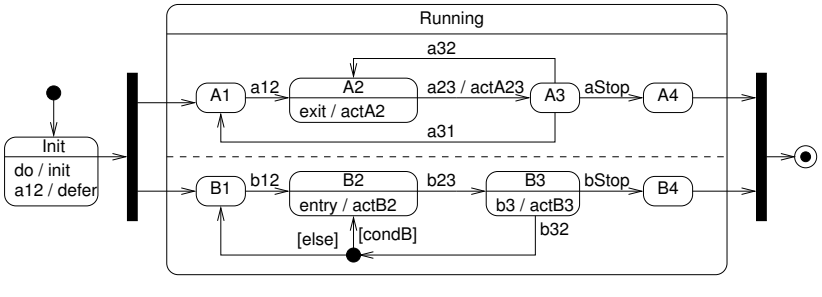


Fig. 1. State machine of a process containing two parallel threads

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

2.2 HiLA

An initial version of the High-Level Aspects (HiLA) language, an aspect-oriented extension of UML state machines, was defined in [34]. This section contains an introduction to a further development of HiLA. We first illustrate the main features of HiLA by means of use case 10 "Authenticate User" of the CCCMS, and then give a brief overview of the general syntax and informal semantics of HiLA.

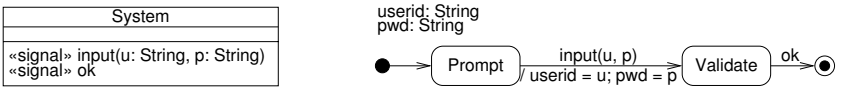
Our models consist of a UML *basic static structure* and a set of UML *base state machines*. The basic static structure usually contains one or more classes; each base state machine is attached to one of these classes and specifies its behavior.[1]

HiLA offers two kinds of aspects to modify such a model: *Static* (or *low-level*, *transformational) aspects* directly specify a model transformation of the basic static structure or of the base state machines. *Dynamic (high-level) aspects* only apply to state machines and specify additional or alternative behavior to be executed at certain "appropriate" points of time in the base machine's execution. These points of time are when a transition is being fired with the state machine in certain state configurations, or if firing the transition would lead to certain state configurations. The transitions are specified by the *pointcut* of the dynamic aspect, the behavior to execute by its *advice*. Dynamic aspects have been introduced in order to avoid certain intricacies arising when relying on model transformations only, e.g., the notorious confluence problem [30] which is mitigated in HiLA by the concurrent execution of all applying aspects [33].

[1] In particular, all classes with state machines are active.

We illustrate the static and dynamic aspect language of HiLA by means of their application to use case 10 of the CCCMS. The main success scenario of this use case according to the case study specification [17] 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.*



(a) Static structure for use case 10 (simplified)

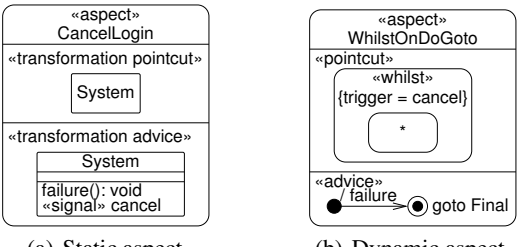
(b) Base state machine for System from Fig. 2(a) (simplified)

Fig. 2. Use Case 10, main success scenario

Let us assume that the static structure for this use case looks as in Fig. 2(a) and that the main success scenario is modeled by the state machine in Fig. 2(b).[2] 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 first extension is specified as follows:

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



(a) Static aspect

(b) Dynamic aspect

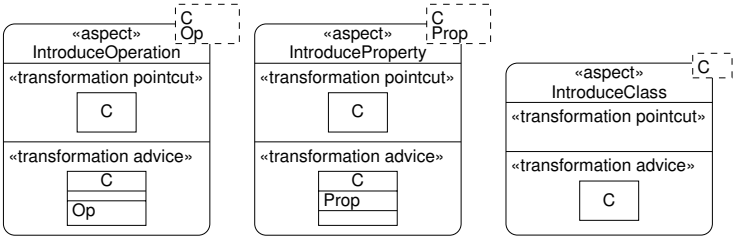
Fig. 3. Aspects for extension 2a of Use Case 10

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

[2] The real situation is given in Figs. 8 and 29.

Fig. 3. The pointcut of the dynamic aspect in Fig. 3(b) 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. 3(a) adds the reception cancel and the operation failure (and also the implementation of failure) to System.

In fact, both the low-level static and the high-level dynamic aspects represent instances of rather general patterns which occur widely in practice and in particular in the CCCMS. To provide a more compact an conspicuous notation, we abstract these and similar aspects into templates, employing the general UML mechanism to parameterize constructs.



(a) Extend class C with operation Op (b) Extend class C with property Prop (c) Introduce class C to the system

Fig. 4. Low-level (static) aspect templates

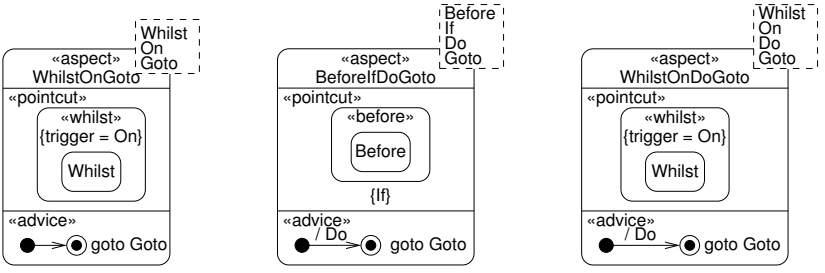
The templates used for low-level aspects are shown in Fig. 4. Template IntroduceOperation in Fig. 4(a) introduces a new operation Op into an existing class C; template IntroduceProperty in Fig. 4(b) introduces a property; and template IntroduceClass in Fig. 4(c) represents the creation of a new class. (All these aspects have no effect when the element to be introduced already exists.) The static aspect in Fig. 3(a) can therefore be rendered simply as shown in Tab. 1.

Table 1. Use Case 10, extension 2a: Fig. 3(a) as template instantiations

Template	**Binding**
IntroduceOperation	C ↦ System Op ↦ ≪signal≫ cancel
IntroduceOperation	C ↦ System Op ↦ failure(): void

Templates to abbreviate commonly used concepts are even more useful for high-level aspects as illustrated by Fig. 5(a)–(c). These templates all adhere to the following form: Whilst a certain state is active[3], On occurrence of a certain trigger, If a certain condition holds, Do a specified action and then Goto a specified state. The regular naming

[3] or Before a certain state becomes active, or After a certain state was active.



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

(b) Before activating state Before, if condition If is true, execute the action Do and go to state Goto

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

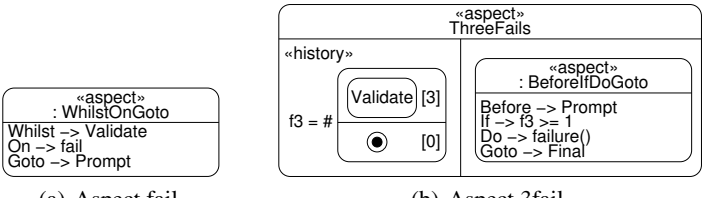
Fig. 5. High-level aspect templates

scheme ensures that the templates are easy to remember and to use; convenient variants and variations are readily formed. Figure 3(b) is an instance of WhilstOnDoGoto with Whilst $\mapsto$ *, On $\mapsto$ cancel, Do $\mapsto$ failure and Goto $\mapsto$ Final (instantiating a template parameter X with * is meant as an abbreviation for instantiating X with all states in the base machine).

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. For this purpose HiLA offers *history properties*.



(a) Aspect fail

(b) Aspect 3fail

Fig. 6. Dynamic aspects for extension 3a of Use Case 10

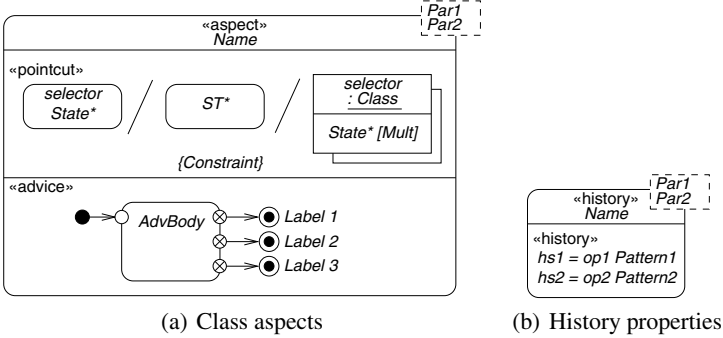
The dynamic aspect in Fig. 6(a) uses the template notation: a failed authentication attempt leads back to state Prompt. (From here on we will employ template notation whenever adequate.) However, the dynamic aspect in Fig. 6(b) is additionally equipped

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.

2.3 General Concrete Syntax and Informal Semantics of Dynamic Aspects

The general concrete syntax of high-level aspects is shown in Fig. 7; the *oblique* identifiers are place holders. *State** is a set of states. The *selector* may be either ≪before≫, ≪after≫ or ≪whilst≫. A pointcut labeled with ≪after≫ selects all transitions leaving any state in *State** while the state machine is in a state configuration containing all states in *State**. Similarly, a pointcut labeled with ≪before≫ selects each transition T entering any state contained in *State**, but only in active state configurations where after taking T all states in *State** will become active. A pointcut labeled with ≪whilst≫ always has an annotation Trigger = e; conceptually it selects the compound transition from *State** to *State** with trigger e, but if this transition does not already exist it is created by the pointcut. Alternatively, the selector may be empty and a set of states and transitions *ST** may be specified. In this case, the pointcut matches all (existing) transitions in *ST**. Note that for any given event and environment, a ≪whilst≫ aspect is only enabled when no existing transition is enabled for this event and environment; moreover, only ≪whilst≫ aspects can cause transitions in situations where no originally existing transition reacts to the given event and environment. Pointcuts with selector ≪before≫ or ≪after≫, or with empty selector only match existing transitions. The pointcut may contain a constraint. In this case, the advice is only executed when the constraint is satisfied at the time of matching the pointcut.

The third case shown in Fig. 7(a) takes into account the active state configurations of all state machines belonging to one of the specified *Class*es; *Mult*, *selector* and *State* are functions of *Class*. If *Mult* is not specified, then *Mult* := *. In the case the *selector* is ≪before≫, the pointcut selects a transition in the currently active state machine if that transition would cause *Mult* instances of each specified *Class* to be in active state



(a) Class aspects (b) History properties

Fig. 7. Concrete syntax of HiLA aspects

configuration *State** with satisfied *Constraint*. If the *selector* is ≪after≫, then the pointcut matches transitions that are about to be fired, while there are *Mult* instances of each specified *Class* in active state configuration *State** with satisfied *Constraint*.

In general, the advice of a dynamic aspect is a state machine, where the final states may be labeled. When an advised transition is taken, the advices of all aspects that advise this transition are executed concurrently. When all advices have reached a final state, execution of the base machine is resumed, normally with a compound transition from the final states of the advices to the target state of the advised transition. The labels on final states can modify this resumption transition: if one or more final states are labeled goto T and all other states are unlabeled the target of the resumption transition is the state T; if concurrently active final states are labeled goto T and goto T' with T $\neq$ T' the state machine is erroneous. Note that this does not preclude final states with different labels, but in an error-free state machine they may not be active simultaneously.

In HiLA, properties of the execution history are specified by *history properties*. The basic idea behind history properties is that they provide a declarative way to specify the value of a variable that depends on the execution history of the state machine; other aspects can then use that value in their pointcut or advice to enable history-sensitive behavior. History properties are defined in *history aspects*, which are identified by the label ≪history≫, as shown in Fig. 7(b). A history property contains a name, followed by an operator and a pattern consisting of active state configurations and guards. The pattern matches contiguous subsequences of the execution history. The value of the history property is the result of applying the function defined by the operator to the subsequences selected by the pattern. In particular, operator # ("number of") returns the number of matches of the pattern by the base machine's execution history where the guards evaluate to true. The pattern language is regular, as presented in [34], patterns can be concatenated by an arrow (→), as presented in [32].

3 Modeling the CCCMS

We model the domain of the CCCMS in UML class diagrams and its behavior in UML state machines, "the most popular language for modeling reactive components" [10]. The method that we follow in the modeling task of the state machines is to first derive a base state machine from the main success scenarios of the use cases and then to enrich these bases by HiLA aspects for all the extensions of the primary scenario. In our description, we concentrate on use case 1 of the CCCMS.

Use case 1 "Resolve Crisis" (and its included use cases) describes the overall flow of events in resolving a car crash crisis: On 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.

3.1 Overview and Static Structure

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

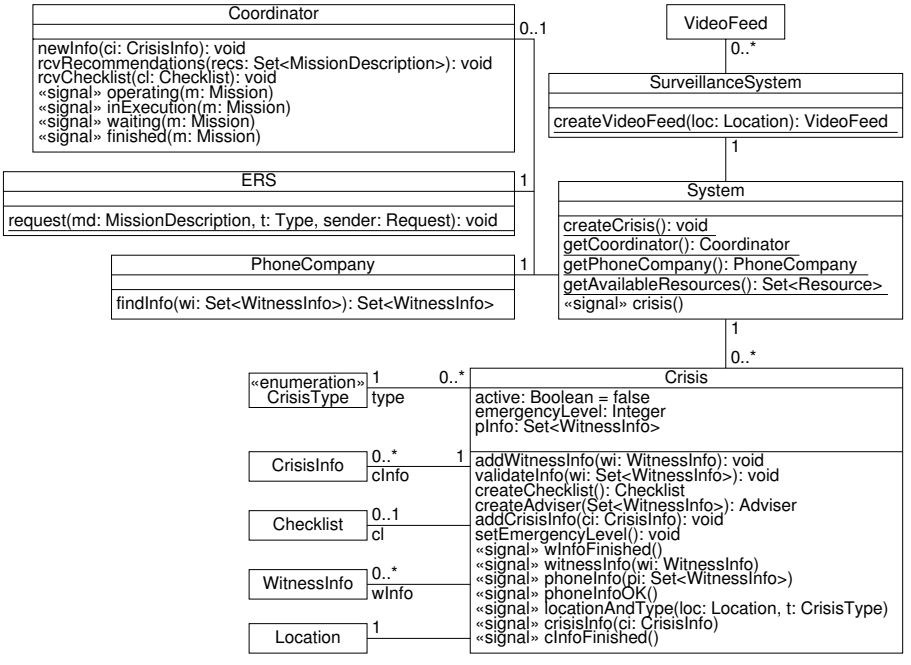


Fig. 8. Class diagram: around System

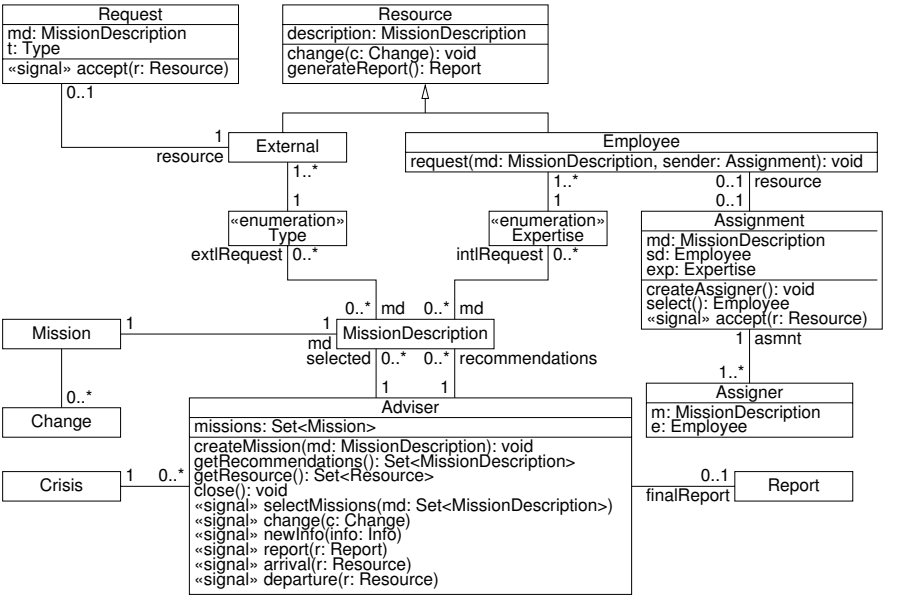


Fig. 9. Class diagram: around Adviser

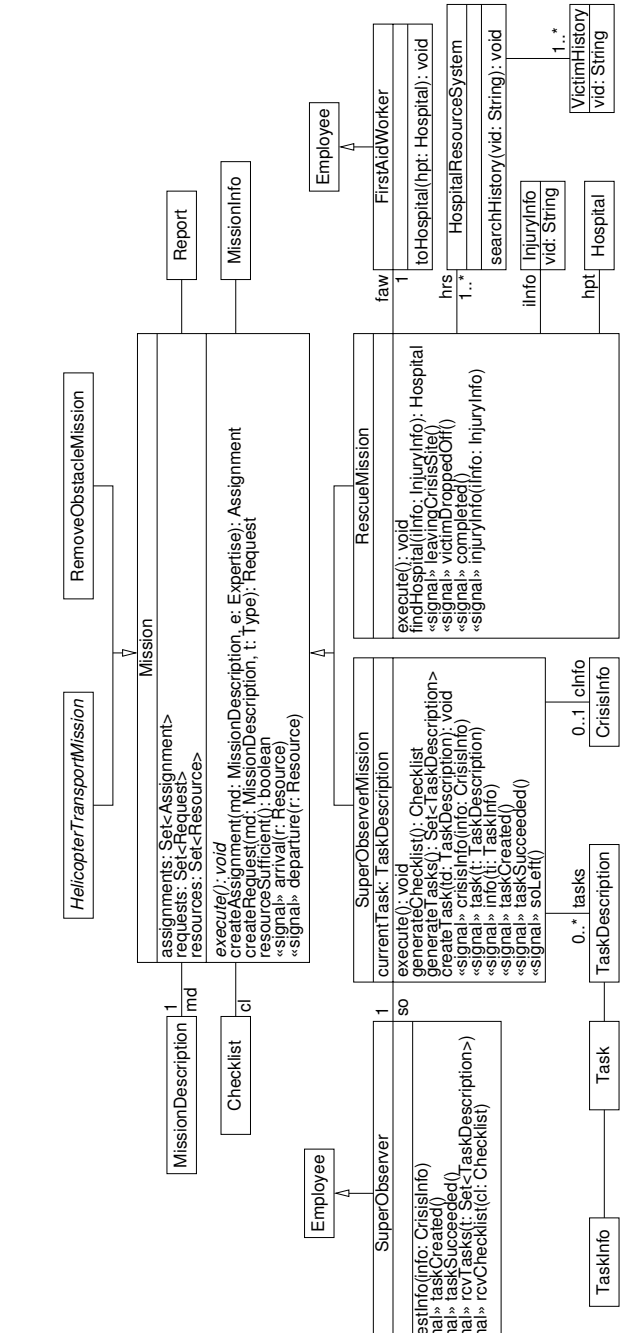


Fig. 10. Class diagram: around Mission

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., [5,26]) 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;[4] 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, Fig. 9, and Fig. 10.

3.2 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 the case study report [17] and then show how the step is modeled.

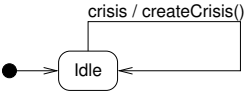


Fig. 11. Class System: base machine

The starting point of a crisis management process is System; its behavior is modeled in Fig. 11: a System is idle until it receives a crisis notification (event crisis), upon which it creates a Crisis. The Crisis instance 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.

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

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 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.*

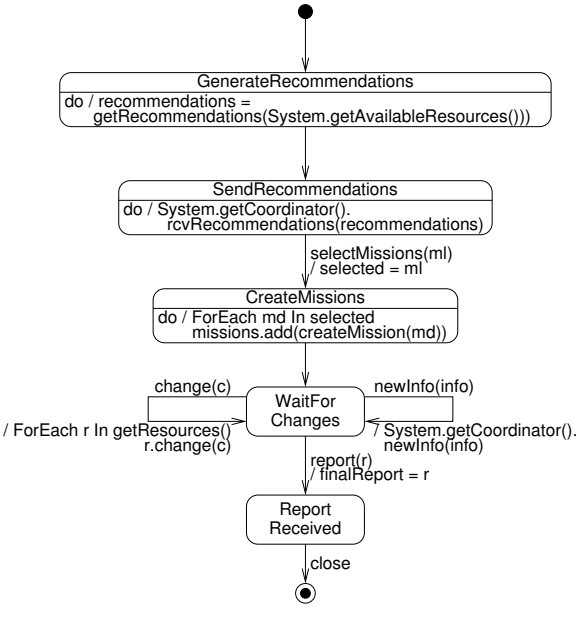


Fig. 12. Class Adviser: base machine

For these steps, it is the Adviser who represents the *System*. The Adviser, see Fig. 12, 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. 13. The next steps from 4 to 11 are executed *for each mission in parallel:*

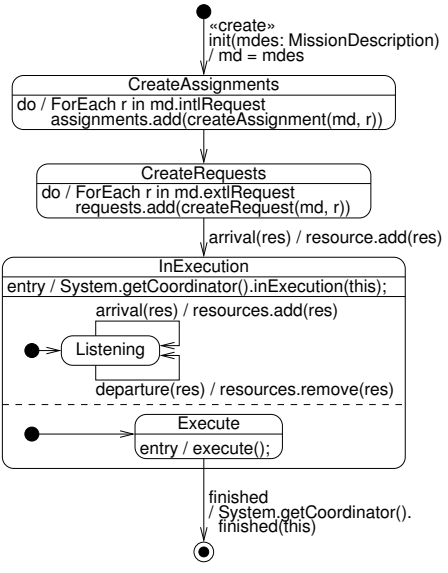


Fig. 13. Class Mission: base machine

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) depend 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. 12). After creating the missions, the adviser gets ready for change notifications (WaitForChanges), and simply passes received change information onto 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 onto 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).

3.3 Modeling the Extensions of Use Case 1

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 case study report [17] 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. 5 for state machines and in Fig. 4 for static structures. We mainly use a tabular format for presenting these instantiations succinctly; in these tables x $\mapsto$ y stands for $\ll$bind$\gg$ x -> y.

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

Table 2. Use Case 1, extension 1a: Introducing a new operation to class System

Template	Base	Binding
IntroduceOperation	Fig. 8	C $\mapsto$ System Op $\mapsto$ validateCoordinator(u: String, i: String): void

When the coordinator is not logged in, use case 10 has to be executed. We therefore add dynamic aspect CoordinatorLogin (see Fig. 14) and its static counterpart (see Tab. 2):

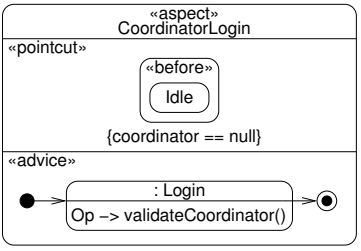


Fig. 14. Use Case 1, extension 1a: Ensuring the Coordinator is logged in

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 App. A.9) 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. 13) 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. 3.

Table 3. Use Case 1, extension 4a: Template instantiations

Template	Base	Binding
IntroduceOperation	Fig. 10	C ↦ Mission Op ↦ ≪signal≫ assignmentFailed
IntroduceOperation	Fig. 10	C ↦ Mission Op ↦ createRequest(t: Type)
IntroduceProperty	Fig. 10	C ↦ Mission Prop ↦ externalType: Type[1]
WhilstOnDoGoto	Fig. 13	Whilst ↦ Listening On ↦ assignmentFailed Do ↦ createRequest(exp.externalType) Goto ↦ Listening

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 App. A.3. We define an instance of aspect WhilstOnDoGoto for (the state machine of) the main success scenario of UC 4, which is given in Fig. 25, 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

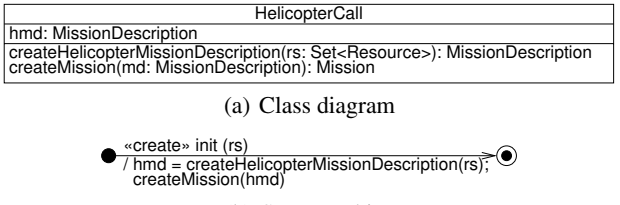
instance for the use case to "continue in parallel with step 2". The new adviser will have the knowledge of the unavailable resource and will recommend different missions to the coordinator than the current one did, details are hidden in the static operation System.getAvailableResources.

Table 4. Use Case 1, extension 5a: Template instantiations

Template	Base	Binding
IntroduceOperation	Fig. 10	C ↦ Mission Op ↦ ≪signal≫ resourceUnavailable
IntroduceOperation	Fig. 8	C ↦ System Op ↦ ≪signal≫ missionFailed(m: Mission)
WhilstOnDoGoto	Fig. 25	Whilst ↦ WaitForAcceptance On ↦ denial Do ↦ mission.resourceUnavailable; System.missionFailed(mission); md.adviser.crisis.createAdviser()

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.*

We introduce two additional transitions to Fig. 13 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

Fig. 15. Class HelicopterCall

The class HelicopterCall is modeled in Fig. 15. A HelicopterCall objects create 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. 5.

All the remaining extensions of UC 1 follow the same patterns. We summarize the necessary instantiations in Table 6. In particular, the fact that a resource is "unable to

Table 5. Use Case 1, extension 6a: Template instantiations

Template	Base	Binding
IntroduceOperation	Fig. 8	C ↦ Coordinator Op ↦ ≪signal≫ needHelicopter(r: Resource)
IntroduceOperation	Fig. 10	C ↦ Mission Op ↦ ≪signal≫ helicopterRequired(r: Resource)
IntroduceOperation	Fig. 10	C ↦ Mission Op ↦ ≪signal≫ startHelicopterMission(r: Set⟨Resource⟩)
IntroduceClass	Fig. 10	C ↦ HelicopterCall (Fig. 15)
WhilstOnDoGoto	Fig. 13	Whilst ↦ WaitForArrival On ↦ helicopterRequired(r) Do ↦ System.getCoordinator().needHelicopter(r) Goto ↦ WaitForArrival
WhilstOnDoGoto	Fig. 13	Whilst ↦ WaitForArrival On ↦ startHelicopterMission(r: Set⟨Resource⟩) Do ↦ createHelicopterCall(r) Goto ↦ WaitForArrival

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.

Table 6. Use Case 1: other extensions

Ext.	Template	Base	Binding
$6b$	IntroduceOperation	Fig. 9	C ↦ c, c ∈ {Assignment, Request} Op ↦≪signal≫ soNotifyArrival(r: Resource)
	WhilstOnDoGoto	Fig. 13	Whilst ↦ WaitForArrival On ↦ soNotifyArrival(r) Do ↦ arrived.add(r) Goto ↦ OneArrived
$6c$	IntroduceOperation	Fig. 9	C ↦ Resource Op ↦≪signal≫ updateRequired()
	WhilstOnDoGoto	Fig. 13	Whilst ↦ WaitForArrival On ↦ after t time Do ↦ resource.updateRequired() Goto ↦ WaitForArrival
$7a$	IntroduceOperation	Fig. 9	C ↦ Adviser Op ↦ ≪signal≫ moreMissionsRequired
	WhilstOnDoGoto	Fig. 12	Whilst ↦ WaitForChanges On ↦ moreMissionsRequired Do ↦ crisis.createAdviser() Goto ↦ WaitForChanges
$7b$	IntroduceOperation	Fig. 9	C ↦ Adviser Op ↦ missionFailed(m: Mission)
	WhilstOnGoto	Fig. 12	Whilst ↦ WaitForChanges On ↦ missionFailed(m) Goto ↦ RecommendMissions
$8a$	IntroduceOperation	Fig. 9	C ↦ c, c ∈ Assignment, Request} Op ↦≪signal≫ soNotifyDeparture(r: Resource)
	WhilstOnDoGoto	Fig. 13	Whilst ↦ WaitForDeparture On ↦ soNotifyDeparture(r) Do ↦ left.add(r) Goto ↦ OneLeft

Table 6. (*continued*)

Ext.	Template	Base	Binding
$8b$	IntroduceOperation	Fig. 9	C ↦ c, c ∈ {Assignment, Request} Op ↦≪signal≫ delayReasonRequired
	WhilstOnDoGoto	Fig. 13	Whilst ↦ WaitForDeparture On ↦ after t time Do ↦ resource.delayReasonRequired() Goto ↦ WaitForDeparture
$9a, 12a$	IntroduceOperation	Fig. 9	C ↦ Adviser Op ↦ ≪signal≫ changeRequired
	WhilstOnDoGoto	Fig. 12	Whilst ↦ WaitForChanges On ↦ changeRequired Do ↦ crisis.createAdviser Goto ↦ Final
$11a$	WhilstOnGoto	Fig. 12	Whilst ↦ WaitForChanges On ↦ after t time Goto ↦ Final

3.4 Modeling System Monitoring

In a crisis management system like the CCCMS, monitoring is often indespensible. HiLA, with its support for aspect-oriented modeling of history-based and cross-state-machine features, provides valuable help in high-level modeling of system monitoring. In the following, we demonstrate how HiLA simplifies monitoring modeling by means of two examples, derived vom Use Case 9 "Execute Remove Obstacle Mission". This use case is originally not specified in [17]. We concretize it for illustration purposes as follows:

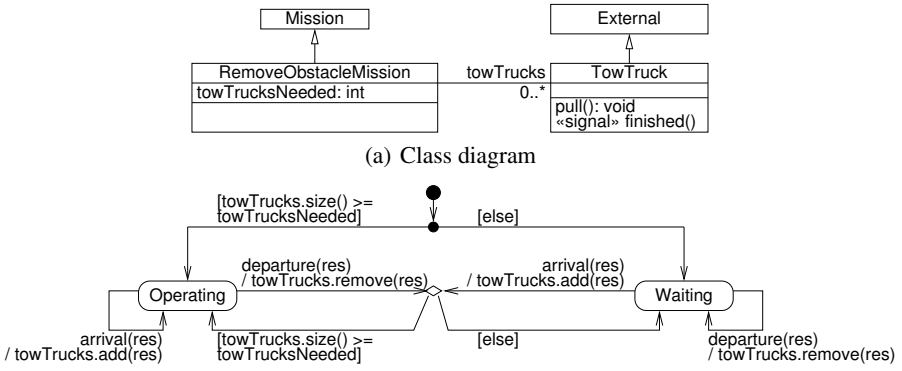
Use Case "Execute Remove Obstacle Mission"

1. Tow truck *notifies* System *of arrival at mission location.*
2. Tow truck *informs* System *that the obstacle has been removed.*
3. *Use case ends in success.*

The static structure of the class RemoveObstacleMission is given in Fig. 16(a), the state machine for the above main success scenario in Fig. 16(b). The mission keeps track of the tow trucks currently on-site in towTrucks, and is either Operating or Waiting depending on if these are sufficient to pull the obstacle (towTrucks.size() >= towTrucksNeeded).

History-based Monitoring. Monitoring features often require reactions on some special run of the system. HiLA's history properties, in particular, their concatenations, can be very helpful for this purpose. For example, the aspect modeled in Fig. 17 introduces a monitoring feature to the base machine defined in Fig. 16(b): if a mission switches too frequently between Operating and Waiting resources are arriving and leaving too frequently to allow smooth execution of the allocated tasks. This indicates a problem and therefore the management system should be alerted.

To achieve this, a history property a is defined as the number of the occurrences of the pattern containing four transitions between Operating and Waiting (as indicated



(a) Class diagram

(b) Base machine

Fig. 16. Class RemoveObstacleMission

by the arrow between these states and the multiplicity 4). The pointcut of the aspects matches points of time just before Operating gets active; if such an occurrence can be found in the execution history (a>=1), an alarm is raised.

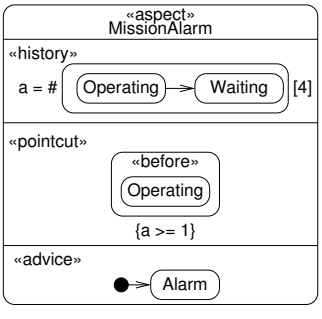


Fig. 17. Aspect raising an alarm if the Mission switches very often between Operating and Waiting, to be applied to Fig. 16(b)

Cross-object Monitoring. There are also many situations where we are more concerned with the interaction between multiple objects than with the behavior of a single object. For example, it may be required that if there are too many tow trucks, an alarm should be raised. We first define a state machine for tow trucks, see Fig. 18: it first approaches to the obstacle to remove, and then pulls it. The static structure of tow trucks was given in Fig. 16(a).

Now we model the monitoring feature. Because of the importance of synchronization and mutual exclusion behaviors, such as needed for monitoring tow trucks, HiLA supports them with a compact notation: Fig. 19(a) shows an aspect template that recognizes situations immediately before M or more objects of type C are about to be simultaneously in state A with the condition If being true. The condition If may be specific in each

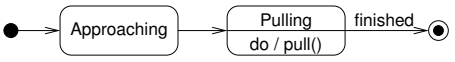
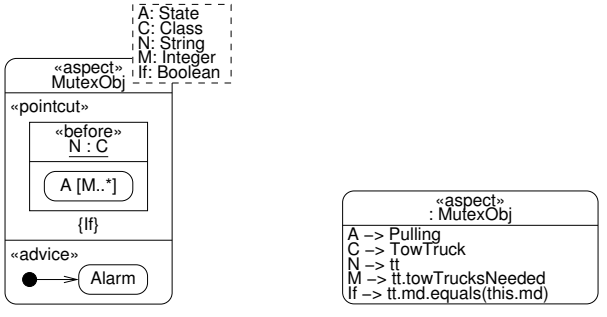


Fig. 18. Class TowTruck: base machine

instance of M, which we call N. If this situation was about to happen, the transitions that would lead into the undesirable state are not taken and instead transitions into a new state Alarm are executed. The instantiations needed to implement the monitoring feature are given in Fig. 19(b).



(a) Template for coordinating multiple state machines

(b) Instantiations for monitoring tow trucks

Fig. 19. HiLA modeling of monitoring

There are several noteworthy points about these kinds of pointcuts:

- The change into the undesirable state configuration with at least M active instances of type C may be caused by a single object or by multiple objects transitioning in a synchronized way; all these cases are covered by the pointcut in Fig. 19.
- The semantics of this pointcut is rather difficult to express in traditional UML, even when using OCL constraints. OCL does not offer access to the event queue and therefore it is not possible to easily specify the situation that *would happen* if the next event was dispatched without aspects being applied.
- While HiLA specifies the semantics of aspects between state machines it does not force a particular implementation strategy on users. For example, a simulated system might rely on knowledge of global state to check whether the pointcut applies, while a distributed system might use semaphores or other synchronization primitives to implement these constraints.

4 Validation of the Model

The use of UML state machines for modeling reactive and interactive systems has led in particular to a considerable amount of work on their formal analysis, be it by theorem

proving (like PVS [3] or KIV [4]) or model checking (like UMC [12] or Hugo/RT [18]). Through its weaving process, HiLA inherits these formal methods: The application of aspects to a UML state machine results in another UML state machine which can be analyzed by standard tools. We exemplify the possible validation of (the weaving of) HiLA models by means of the (simplified) Use Case 10 "Authenticate User," see Sect. 2.2, and the model checking component of Hugo/RT.

According to our method, we start out with the base state machine in Fig. 2(b) that just reflects 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. 3 and 6, 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. 2(b), 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 [14]. 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.

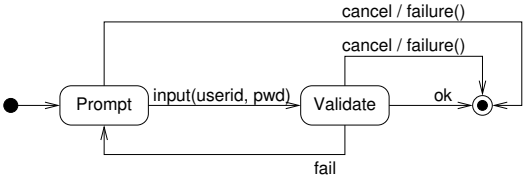


Fig. 20. Weaving result for sub-machine login from Fig. 2(b) and the aspects in Fig. 3(b), Fig. 6(a)

The result of weaving in the aspects in Fig. 3(b) and Fig. 6(a) is shown in Fig. 20. 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. 6(b) ensures that at most three failing attempts in a row can be made. The weaving result is shown in Fig. 21. The history property f3 needs an additional counter variable v counting the failing validations in a row. This counter v is reset on canceling and also on a successful login. If three failing login attempts have been counted, the history property f3 is set, and on every possible entry into Prompt this history property is checked, prohibiting entry if it is set. Now, Hugo/RT and SPIN confirm that on the one hand it is possible to have three unsuccessful attempts to login, but no more attempts are then possible.

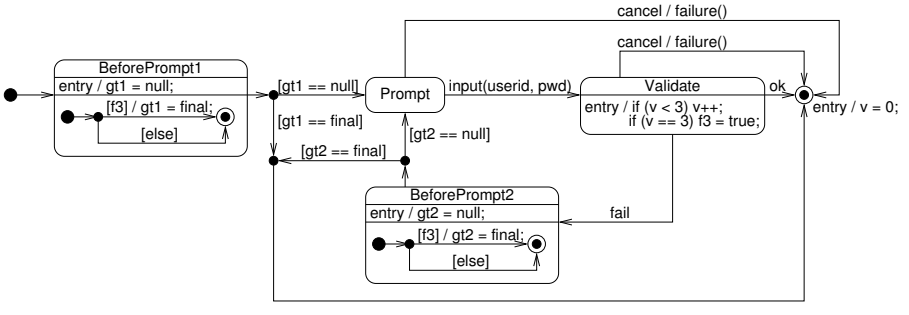


Fig. 21. Weaving result for sub-machine login from Fig. 2(b) and Figs. 3(b) and 6

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 rather 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 take into account the weaving result which can be rather complex and sometimes slightly unintuitive.

5 Evaluation of the HiLA Approach

In the following sections, 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.

5.1 The HiLA Approach to Modeling

In our opinion, one of the main advantages of aspect-oriented methods is that they allow a closer correspondence between requirements, model and executable code. Therefore, one important criterion for evaluating HiLA is how well our proposed approach could be applied to the requirements specified in the CCCMS case study and how directly individual state machines and aspects can be traced to use cases and use case extensions.

Overall, we were pleasantly surprised how well our proposed approach 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 traceability from requirements to 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 rather 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 2–6). 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. Note, however, that even in the case of unbounded parallelism HiLA still provides the necessary expressiveness of composition by featuring cross-object pointcuts, as demonstrated in Sect.3.4.

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 the responsibles for handling all simultaneously active crises.

A similar situation presents itself for 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. 29, 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 requirement 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.

5.2 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 to either introduce new classes or to add operations or properties to existing classes.

HiLA was developed with particular emphasis on concurrent systems that require complex synchronization and mutual exclusion [34]. The original CCCMS scenario relies on central control. We therefore supplemented one of the under-specified use cases, UC 9 "Execute Remove Obstacle Mission," in order to demonstrate HiLA's expressivity of composition, in particular its features to control and limit parallelism. In addition, the CCCMS exhibits only very simple history-based behavior; again HiLA is designed to efficiently cope with scenarios that require more sophisticated handling of execution histories and we have provided examples in Sect. 2.2 and 3.4.

The HiLA language allows modelers to easily define templates that represent commonly used aspects for their scenarios. This can be seen in Sect. 3: 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. 5.1 on p. 258) 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 [30]. 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. Monitors, see Sect. 3.4, can be used to

specify test goals in the same language in which the model is written. 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 is done and therefore not compositional. Independent verification of individual aspects remains a challenge and is a subject for future research.

HiLA can easily be extended, for example, by allowing 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.

6 Related Work

The idea of high-level aspects was first seen in dynamic aspect-oriented programming languages such as JAsCo [28] and Object Teams [13]. Using history properties to quantify over the execution history is reminiscent to the trace aspects of Arachne [9].

Considering state machines, aspects for Mealy automata are propsed by [2]. In comparison, the weaving algorithm of HiLA is much more elaborate, mainly due to the richer language constructs of the UML. Because of the wider acceptance of the UML, our approach is also more tightly connected to common practice. State-based aspects in reactive systems are also supported by JPDD [25] and Telelogic TAU [35], both of which do not rely soly on graph transformation, but facilitate specification of high-level pointcuts in flat state machines. In comparison, HiLA was designed to be applicable to parallel and hierarchical 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 HiLA provides valuable help to address these problems.

Addressing the complete UML, Theme/UML [7] models different features in different models (called themes) and uses UML templates to define common behavior of several themes. General composers such as GeKo [21] and MATA [29] can be used for weaving. The aspects of these approaches are low level, which means modeling nontrivial features often requires rather complex aspects, and comprehension of the aspects amounts to understanding the weaving result.

Aspect interference is an intrinsic problem of aspect-oriented techniques. It has been addressed in a large amount of publications, for an overview see [1,19]. Notations of precedence declaration are proposed in, e.g., [16,22,24,35]. Techniques to detect interference proposed so far include detecting shared fields addressed by read and write operations [27], a formal foundation of AOP languages using Conditional Program Transformations [19], and graph transformations [1,30]. These approaches focus on sequential programming languages. In comparison, HiLA exploits the concurrency of state machines and weaves aspects into parallel regions to solve the problem of join points being changed by other aspects or the result of weaving depending on the weaving order.

Weaving into parallel constructs is also proposed in [8], 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.

Besides crisis management systems, HiLA was also applied to model adaptive web applications and computer games, see [32,33].

7 Conclusions

We have presented the HiLA approach for aspect-oriented modeling and applied it to the CCCMS case study. The expressiveness of HiLA has allowed us to methodologically transform the use case descriptions of the CCCMS into state machines; each main success scenario is modeled by a base state machine, each extension of this scenario is represented by a set of aspects. This method affords a high degree of traceability from the resulting model to the requirements. Additionally, since we rely exclusively on state machine models for the behavior, we can apply well-known formal methods, like model checking, for validating system properties.

Acknowledgements. We thank the reviewers for their insightful and thorough comments. This work has been partially sponsored by the DFG project MAEWA (WI 841/7–2) and the EU project SENSORIA (IST-2 005-016004).

References

1. Aksit, M., Rensink, A., Staijen, T.: A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points. In: Proc. 8th Int. Conf. Aspect-Oriented Software Development (AOSD 2009), pp. 39–50 (2009)
2. Altisen, K., Maraninchi, F., Stauch, D.: Aspect-Oriented Programming for Reactive Systems: Larissa, a Proposal in the Synchronous Framework. Sci. Comp. Prog. 63(3), 297–320 (2006)
3. Arons, T., Hooman, J., Kugler, H.-J., Pnueli, A., van der Zwaag, M.B.: Deductive Verification of UML Models in TLPVS. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 335–349. Springer, Heidelberg (2004)

4. Balser, M., Bäumler, S., Knapp, A., Reif, W., Thums, A.: Interactive Verification of UML State Machines. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 434–448. Springer, Heidelberg (2004)
5. Blaha, M., Rumbaugh, J.: Object-Oriented Modeling and Design with UML, 2nd edn. Prentice Hall, Englewood Cliffs (2004)
6. Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Reading (2005)
7. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. Addison-Wesley, Reading (2005)
8. Douence, R., Le Botlan, D., Noyé, J., Südholt, M.: Concurrent Aspects. In: Proc. 5th Int. Conf. Generative Programming and Component Engineering (GPCE 2006), pp. 79–88. ACM, New York (2006)
9. Douence, R., Fritz, T., Loriant, N., Menaud, J.-M., Ségura-Devillechaise, M., Südholt, M.: An Expressive Aspect Language for System Applications with Arachne. In: Mezini, Tarr (eds.) [20], pp. 27–38
10. Drusinsky, D.: Modeling and Verification Using UML Statecharts. Elsevier, Amsterdam (2006)
11. Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.): MODELS 2007. LNCS, vol. 4735. Springer, Heidelberg (2007)
12. Gnesi, S.: Formal Specification and Verification of Complex Systems. In: Proc. 8th Int. Wsh. Formal Methods for Industrial Critical Systems (FMICS 2003). Electr. Notes Theor. Comput. Sci., vol. 80 (2003)
13. Herrmann, S.: Object Teams: Improving Modularity for Crosscutting Collaborations. In: Aksit, M., Mezini, M., Unland, R. (eds.) NODe 2002. LNCS, vol. 2591, pp. 248–264. Springer, Heidelberg (2003)
14. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2003)
15. Jacobson, I., Ng, P.-W.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley, Reading (2004)
16. Kienzle, J., Gélineau, S.: AO Challenge — Implementing the ACID Properties for Transactional Objects. In: Filman, R.E. (ed.) Proc. 5th Int. Conf. Aspect-Oriented Software Development (AOSD 2006), pp. 202–213. ACM, New York (2006)
17. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis Management Systems: A Case Study for Aspect-Oriented Modeling. Trans. Aspect-Oriented Software Development (TAOSD) 7, 1–22 (2010)
18. Knapp, A., Merz, S., Rauh, C.: Model Checking Timed UML State Machines and Collaborations. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 395–416. Springer, Heidelberg (2002)
19. Kniesel, G.: Detection and Resolution of Weaving Interactions. In: Rashid, A., Ossher, H. (eds.) Transactions on Aspect-Oriented Software Development V. LNCS, vol. 5490, pp. 135–186. Springer, Heidelberg (2009)
20. Mezini, M., Tarr, P.L. (eds.): Proc. 4th Int. Conf. Aspect-Oriented Software Development (AOSD 2005). ACM, New York (2005)
21. Morin, B., Klein, J., Barais, O., Jézéquel, J.-M.: A Generic Weaver for Supporting Product Lines. In: Proc. 13th Int. Wsh. Software Architectures and Mobility (EA 2008), pp. 11–18. ACM, New York (2008)
22. Nagy, I., Bergmans, L., Aksit, M.: Composing Aspects at Shared Join Points. In: Hirschfeld, R., Kowalczyk, R., Polze, A., Weske, M. (eds.) Proc. Net.ObjectDays (NODe 2005). Lect. Notes Informatics, vol. 69, pp. 19–38. Gesellschaft für Informatik (2005)
23. 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`

24. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for Composing Aspect-Oriented Design Class Models. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 75–105. Springer, Heidelberg (2006)
25. Sánchez, P., Fuentes, L., Stein, D., Hanenberg, S., Unland, R.: Aspect-Oriented Model Weaving Beyond Model Composition and Model Transformation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) Proc. 11th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS 2008). LNCS, vol. 5301, pp. 766–781. Springer, Heidelberg (2008)
26. Sommerville, I.: Software Engineering, 8th edn. Addison-Wesley, Reading (2007)
27. Störzer, M., Forster, F., Sterr, R.: Detecting Precedence-Related Advice Interference. In: Proc. 21st IEEE/ACM Int. Conf. Automated Software Engineering (ASE 2006), pp. 317–322. IEEE, Los Alamitos (2006)
28. Vanderperren, W., Suvée, D., Verheecke, B., Cibrán, M.A., Jonckers, V.: Adaptive Programming in JAsCo. In: Mezini, Tarr (eds.) [20], pp. 75–86
29. Whittle, J., Jayaraman, P.K.: MATA: A Tool for Aspect-Oriented Modeling based on Graph Transformation. In: Proc. 11th Int. Wsh. Aspect-Oriented Modeling, AOM@MoDELS 2007 (2007)
30. Whittle, J., Moreira, A., Araújo, J., Jayaraman, P.K., Elkhodary, A.M., Rabbi, R.: An Expressive Aspect Composition Language for UML State Diagrams. In: Engels, et al. (eds.) [11], pp. 514–528
31. Zhang, G.: Towards Aspect-Oriented Class Diagrams. In: Proc. 12th Asia-Pacific Software Engineering Conf (APSEC 2005), pp. 763–768. IEEE, Los Alamitos (2005)
32. Zhang, G.: Aspect-Oriented Modeling of Adaptive Web Applications with HiLA. In: Kotsis, G., Taniar, D., Pardede, E., Khalil, I. (eds.) Proc. 7th Int. Conf. Advances in Mobile Computing & Multimedia (MoMM 2009), pp. 331–335. ACM, New York (2009)
33. Zhang, G., Hölzl, M.: HiLA: High-Level Aspects for UML State Machines. In: 14th Wsh. Aspect-Oriented Modeling (AOM@MoDELS 2009), Denver (2009)
34. Zhang, G., Hölzl, M.M., Knapp, A.: Enhancing UML State Machines with Aspects. In: Engels, et al. (eds.) [11], pp. 529–543
35. Zhang, J., Cottenier, T., van den Berg, A., Gray, J.: Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. Journal of Object Technology 6(7), 89–108 (2007)

A Remaining Use Cases of the CCCMS

A.1 Use Case 2: Capture Witness Report

Main success scenario. Modeled in Fig. 22. 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.

Table 7. Use Case 2: extensions

Ext.	Template	Base	Binding
$1a, 2a$	IntroduceOperation	Fig. 8	C ↦ Crisis Op ↦ ≪signal≫ disconnect
	WhilstOnGoto	Fig. 22	Whilst ↦ s, s ∈ {CollectInfo, WaitForLocationAndType} On ↦ disconnect Goto ↦ Final
$3a$	IntroduceOperation	Fig. 8	C ↦ SurveillanceSystem Op ↦ ≪static≫ covers(loc: Location): Boolean
	IntroduceOperation	Fig. 8	C ↦ Coordinator Op ↦ rcvVideoFeed(vf: VideoFeed): void
	AfterIfDo	Fig. 22	After ↦ CreateChecklist If ↦ SurveillanceSystem.covers(location) Do ↦ System.getCoordinator.rcvVideoFeed(SurveillanceSystem.createVideoFeed(location))
$4a$	IntroduceOperation	Fig. 8	C ↦ Crisis Op ↦ ≪signal≫ disconnect
	WhilstOnGoto	Fig. 22	Whilst ↦ CollectCrisisInfo On ↦ disconnect Goto ↦ CrisisInfoReceived
$5a$	IntroduceOperation	Fig. 8	C ↦ Crisis Op ↦ ≪signal≫ phoneInfoWrong
	WhilstOnGoto	Fig. 22	Whilst ↦ Validate On ↦ phoneInfoWrong Goto ↦ Final
$5b$	IntroduceOperation	Fig. 8	C ↦ Crisis Op ↦ ≪signal≫ s, s ∈ {fake, deny}
	WhilstOnGoto	Fig. 22	Whilst ↦ GatherInfo On ↦ s, s ∈ {fake, deny} Goto ↦ Final

Extensions. The extensions of Use Case 2 are modeled by instantiating the aspect templates given in Figs. 4 and 5. The bindings are given in Table 7. 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 [31]), only one instance of the signal is actually introduced to the class.

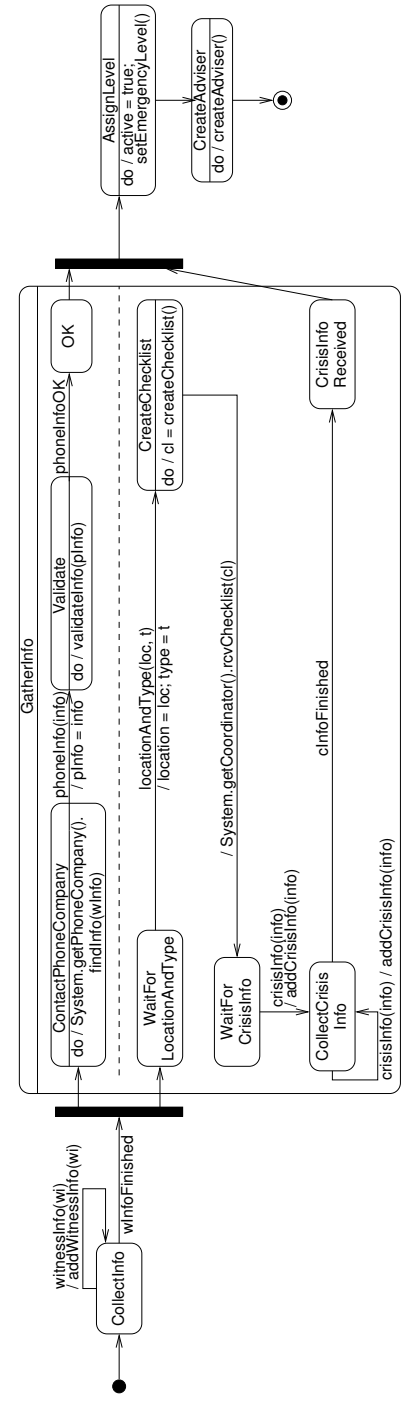
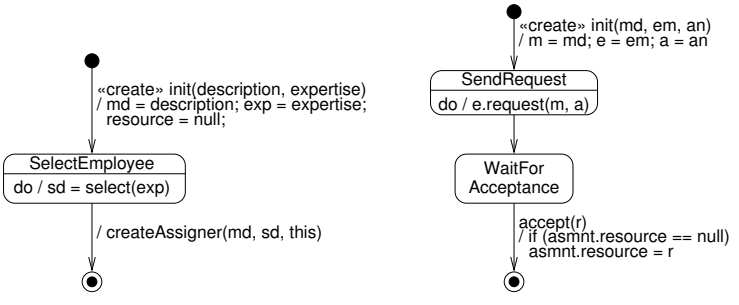


Fig. 22. Class Crisis: base machine

A.2 Use Case 3: Assign Internal Resource

Main success scenario. Modeled in Fig. 23. We assume that all "internal resources" are employees. The Assignment object (see Fig. 23(a)) passes on the mission description, received from the adviser for its own creation, along with references of the most appropriate employee (selected in SelectEmployee), and of this Assignment object itself (this), to an Assigner object, which then (see Fig. 23(b)) sends a request to the employee and waits for acceptance (signal accept).



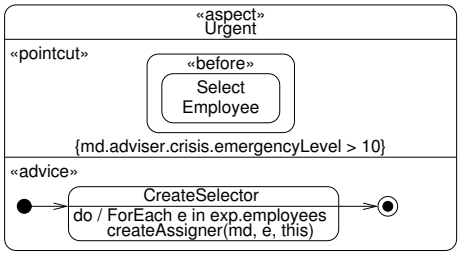
(a) Class Assignment: base machine (b) Class Assigner, base machine

Fig. 23. Classes for allocation of internal resources

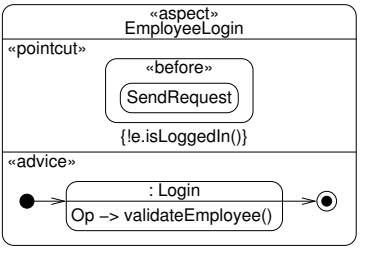
Extensions. The feature "*in very urgent cases, steps 1 and 2 can be performed for several* CMSEmployees *concurrently, until one of the contacted employees accepts the mission*" is not formulated as an extension in [17]. However, we consider this an exceptional feature and decided to model it in a separate aspect.

This feature is modeled by the aspect Urgent (Fig. 24(a)), which alternates 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), we create an Assigner object for each employee with the desired expertise (exp.employees). Each of these assigners then sends in parallel a request to "its" employee.

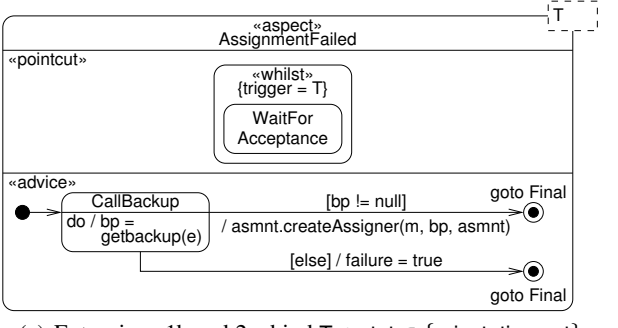
Extension $1a$ is modeled by the aspect in Fig. 24(b). Again, we instantiate the template Login given in Fig. 29 on page 271. The needed operations are introduced by instantiations of aspect templates as given in Table 8. The extensions 1b and 2a are modeled by instantiating the template given in Fig. 24(c) with T -> t, t ∈ {reject, after t time}. When the assigner is waiting for acceptance, and receives one of these two signals, then it tries to find a backup (getBackup) for the resource it wanted. If one can be found (bp != null), then another assigner is created to contact the backup, otherwise the use case ends in failure. The extension of the static structure is also given in Table 8.



(a) Extension 0



(b) Extension 1a



(c) Extensions 1b and 2a: bind T -> t, t ∈ {reject, timeout}

Fig. 24. Use Case 3

Table 8. Introducing static elements for modeling Use Case 3

Ext.	Template	Base	Binding
$1a$	IntroduceProperty	Fig. 9	C ↦ Assigner Prop ↦ u: String
	IntroduceProperty	Fig. 9	C ↦ Assigner Prop ↦ p: String
	IntroduceOperation	Fig. 9	C ↦ Assigner Op ↦ ≪signal≫ input(u: String, i: String)
	IntroduceOperation	Fig. 9	C ↦ Assigner Op ↦ validateEmployee(u: String, i: String): void
	IntroduceOperation	Fig. 9	C ↦ Employee Op ↦ isLoggedIn(): Boolean
$1b, 2a$	IntroduceProperty	Fig. 9	C ↦ Assigner Prop ↦ failure: Boolean = false
	IntroduceOperation	Fig. 9	C ↦ Assigner Op ↦ getBackup(e: Employee): Employee
	IntroduceOperation	Fig. 9	C ↦ Assigner Op ↦ ≪signal≫ reject

A.3 Use Case 4: Request External Resource

Main success scenario. Modeled in Fig. 25. 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.

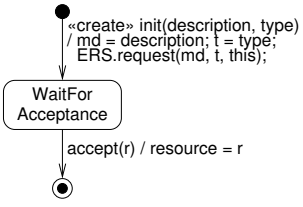


Fig. 25. Class Request: base machine

Extensions. Again, the extensions are modeled with instances of aspect templates (see Figs. 4 and 5) with bindings defined in Table 9. 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.

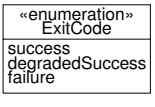


Table 9. Use Case 4: extensions

Ext.	Template	Base	Binding
$2a, 2b$	IntroduceClass	Fig. 9	C $\mapsto$ ExitCode = success
	IntroduceProperty	Fig. 9	C $\mapsto$ Request Prop $\mapsto$ exitCode: ExitCode
	IntroduceOperation	Fig. 9	C $\mapsto$ Assignment Op $\mapsto$ s, s $\in$ { $\ll$signal$\gg$ partialApproval, $\ll$signal$\gg$ denial }
	WhilstOnDoGoto	Fig. 25	Whilst $\mapsto$ WaitForAcceptance On $\mapsto$ partialApproval Do $\mapsto$ exitCode = degradedApproval Goto $\mapsto$ Final
	WhilstOnDoGoto	Fig. 25	Whilst $\mapsto$ WaitForAcceptance On $\mapsto$ denial Do $\mapsto$ exitCode = failure Goto $\mapsto$ Final

A.4 Use Case 5: Execute Mission

Not modeled, for this is an abstract use case.

A.5 Use Case 6: Execute SuperObserver Mission

Main success scenario. Modeled in Fig. 26. In the use case description [17], *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.

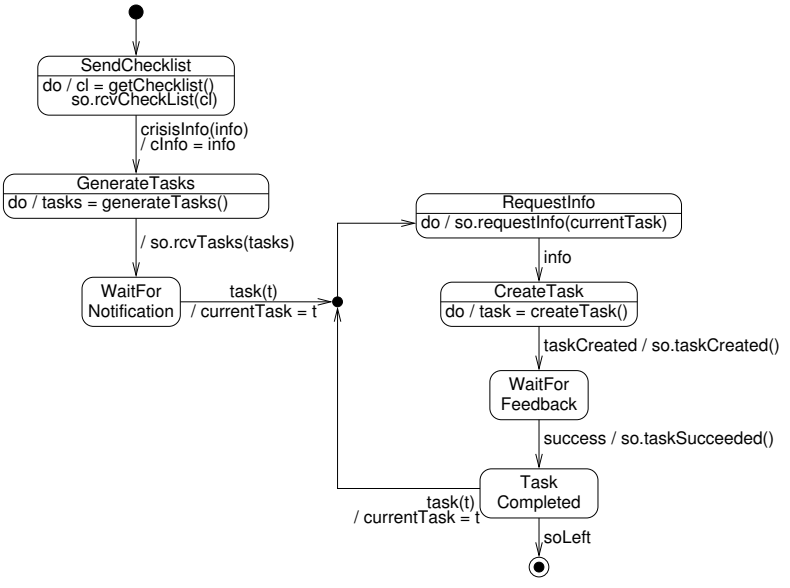


Fig. 26. SuperObserverMission.execute: base machine

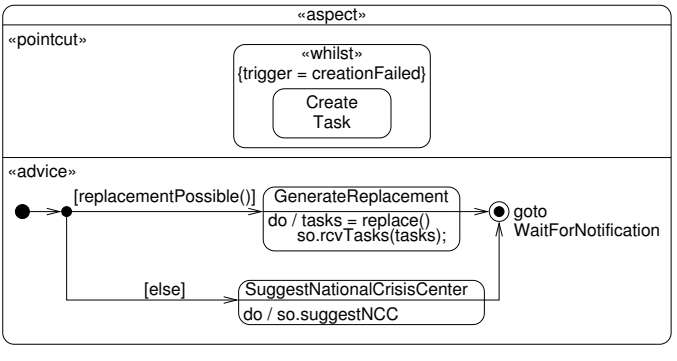


Fig. 27. Use Case 6, extensions 7a/b

Extensions. Extensions 7a and 7b are modeled in Fig. 27 with introduction of static elements as defined in Table 10. Extension 8a is modeled by the instantiations only, also given in Table 10.

Table 10. Use Case 6: Introducing static elements used by the dynamic aspects

Ext.	Template	Base	Binding
$7a, 7b$	IntroduceOperation	Fig. 10	C ↦ SuperObserverMission Op ↦ ≪signal≫ creationFailed()
	IntroduceOperation	Fig. 10	C ↦ SuperObserverMission Op ↦ replacementPossible(): Boolean
	IntroduceOperation	Fig. 10	C ↦ SuperObserverMission Op ↦ replace(): Set<TaskDescription>
	IntroduceOperation	Fig. 10	C ↦ SuperObserver Op ↦ ≪signal≫ suggestNCC
$8a$	IntroduceOperation	Fig. 10	C ↦ SuperObserverMission Op ↦ ≪signal≫ taskFailed(t: Task)
	IntroduceOperation	Fig. 10	C ↦ SuperObserver Op ↦ ≪signal≫ taskFailed(t: Task)
	IntroduceOperation	Fig. 8	C ↦ Coordinator Op ↦ ≪signal≫ taskFailed(t: Task)
	WhilstOnDoGoto	Fig. 26	Whilst ↦ WaitForFeedback On ↦ taskFailed(task) Do ↦ so.taskFailed(task); System.getCoordinator.taskFailed(task) Goto ↦ TaskCompleted

A.6 Use Case 7: Execute Rescue Mission

Main success scenario. Modeled in Fig. 28. 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.

Extensions. The (only) extension is modeled by the instantiations of aspect templates as described in Table 11.

Table 11. Use Case 7, extension 4a

Template	Base	Binding
IntroduceOperation	Fig. 10	C ↦ FirstAidWorker Op ↦ victimHistory(his: History)
IntroduceOperation	Fig. 10	C ↦ RescueMission Op ↦ ≪signal≫ victimHistory(his: History)
WhilstOnDoGoto	Fig. 28	Whilst ↦ FindBestHospital On ↦ victimHistory(his) Do ↦ faw.victimHistory(his) Goto ↦ FindBestHospital

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

Originally not specified in [17], a sample description of the use case and its design are given in Sect. 3.4.

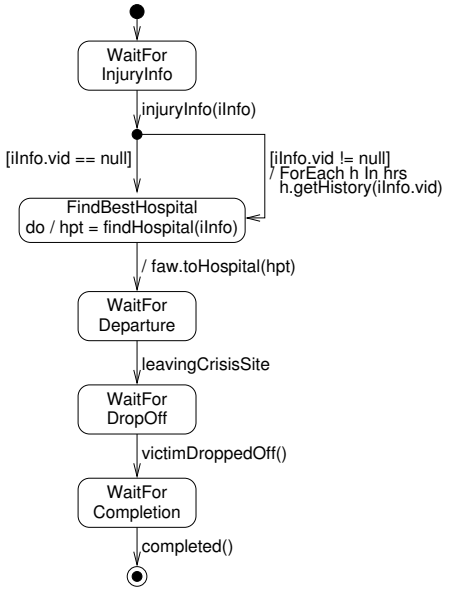


Fig. 28. RescueMission.execute: base machine

A.9 Use Case 10: Authenticate User

The full model of use case 10 (see Fig. 29 and Tab. 12) is similar to the simplified model described in Sect. 2.2. However, as 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. In particular, the extensions 2a and 3a are covered by Figs. 3 and 6.

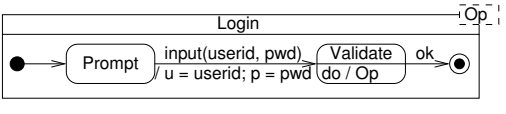


Fig. 29. Base sub-state machine for Login

Table 12. Use Case 10: Introducing new operations to class System

Template	Base	Binding
IntroduceOperation	Fig. 8	C ↦ System Op ↦ ≪signal≫ input(u: String, i: String)
IntroduceOperation	Fig. 8	C ↦ System Op ↦ ≪signal≫ ok