Analysis of Aspect-Oriented Models using Graph Transformation Systems

Katharina Mehner-Heindl[1], Mattia Monga[2], and Gabriele Taentzer[3]

[1] Dep. of Media and Information Engineering, University of Applied Sciences
Offenburg, Germany
katharina.mehner@gmail.com

[2] Dip. di Informatica e Comunicazione, Università degli Studi di Milano, Italy
mattia.monga@unimi.it

[3] Dep. of Computer Science and Mathematics, Philipps-Universität Marburg,
Germany
taentzer@mathematik.uni-marburg.de

Abstract. Aspect-oriented concepts are currently exploited to model systems from the beginning of their development. Aspects capture potentially cross-cutting concerns and make it easier to formulate desirable properties and to understand analysis results than in a tangled system. However, the complexity of interactions among different aspectualized entities may reduce the benefit of aspect-oriented separation of cross-cutting concerns. It is therefore desirable to detect inconsistencies as early as possible.

We propose an approach for analyzing consistency at the level of requirements modeling. We use a variant of UML to model requirements in a use-case driven approach. Activities that are used to refine use cases are the join points to compose cross-cutting concerns. Activities are combined with a specification of pre-and post-conditions into an integrated behavior model. This model is formalized using the theory of graph transformation systems to effectively reason about its consistency. The analysis of an integrated behavior model is performed with the tool ACTIGRA.

1 Introduction

Aspect-oriented programming promises to provide better separation and integration of cross-cutting concerns than plain object-oriented programming. Aspect-oriented concepts have been introduced in all phases of the software development life cycle with the aim of reducing complexity and enhancing maintainability already early on.

On the requirements level, cross-cutting concerns, i.e., concerns that affect many other requirements, cannot be cleanly modularized using object-oriented and view-point-based techniques. Several approaches have been proposed to identify cross-cutting concerns already at the requirements level and to provide means to modularize, represent and compose them using aspect-oriented techniques, e.g., for use-case driven modeling in [1, 2, 3, 4]. A key challenge is to

analyze the interaction and consistency of cross-cutting concerns with each other and with affected requirements. It is in particular the quantifying nature [5] of aspect-oriented composition that makes the detection of interactions and inconsistencies difficult.

Until now, approaches to analyzing the aspectual composition of requirements have been informal [6, 3, 4]. Formal approaches for detecting inconsistencies have been proposed only for the level of aspect-oriented programming, e.g., model checking [7], static analysis [8], and slicing [9, 10]. At the programming level, however, the meta-model considered is pretty different and it takes into account many low-level details. Requirements abstract from these implementation related details, and weaving occurs among the high-level activities which describe the intended behavior of the system.

A commonly used but often informal technique on the requirements level is to describe behavior with pre- and post-conditions, e.g., using intentionally defined states or attributes of a domain entity model. This technique is, for example, used for defining UML [11] use cases, activities, and methods. In order to allow a more rigorous analysis of behavior, this approach has to be formalized and also extended to aspect-oriented units of behavior.

We propose a use-case driven approach with a domain class model. Activity models are used to refine use cases. Object models are used for describing pre- and post-conditions of activities. This integration between structural and functional view is called an *integrated behavior model*. Furthermore, we propose an *aspect-oriented extension*. We model the so called base with use cases and an integrated behavior model. We model aspects as use cases and refine them with an integrated behavior model. During the aspect-oriented composition, we use activities as join points and follow the composition operations suggested by AspectJ [12, 13] and similar languages. An integrated behavior model can be formalized using the theory of graph transformations: Graph transformation rules are used to formalize pre- and post-conditions of activities. Graph transformation sequences are used to capture the semantics of the activity models. A formal analysis can be carried out on integrated behavior models computing favorable and critical signs concerning causalities and conflicts between activities. This analysis can be carried out before and after the aspect-oriented composition in order to understand the behavior of use cases and of aspectual use cases separately and in order to understand the effects of aspects. The new tool ACTI-GRA [14, 15], which itself is based on the well known AGG engine for graph transformations [16, 17], provides this kind of modeling and analysis support. Throughout the paper we use a UML variant that is directly supported by this tool.

The idea of formalizing pre- and post-conditions by graph transformation was presented in [18] first and extended to aspect-oriented models in [19]. The aspect-oriented composition itself was formalized by meta-level graph transformations in [20]. Since then, the theory and the tools for integrated behavior models have been advanced and improved. We demonstrate how they can be used in aspect-oriented modeling.

This chapter is organized as follows. In Sect. 2 we present our aspect-oriented modeling approach and sketch the weaving process. Sect. 3 presents the theory of algebraic graph transformations first, including conflict and causality analysis between transformations. Secondly, we give the formal semantics of activity diagrams augmented by graph transformation rules by means of sequences of graph transformations. Sect. 4 presents the plausibility checks based on the formal semantics. These analysis facilities are applied to our example in Sect. 5. In Sect. 6 we discuss related work. In Sect. 7 we conclude and give an outlook.

2 Aspect-Oriented Modeling with Integrated Behavior Models

Our approach uses *integrated behavior models* and extends them by aspect-oriented features. An integrated behavior model consists of a *domain model* and a set of *activity models*. The domain model provides the types of the domain objects. Each activity is refined by *pre-* and *post-condition*s describing the effect of the activity in terms of domain objects. Typically, an *initial configuration* of the system is provided in terms of domain objects and their relations.

The benefit of an integrated behavior model is an early and better integration of the structural domain model with the functional activity model. Pre- and post-conditions are formalized by the theory of graph transformation systems. This formalization can then be used for a rigorous analysis of integrated behavior models.

In addition to the integrated behavior model, a use case diagram provides a system overview. Each use case is at least specified by a trigger, its actors, pre- and post-conditions and its key scenarios. Scenarios are specified using activity diagrams and use cases are the starting point for the aspect-oriented modeling. We model the so-called *base* of the system with use cases and an integrated behavior model. An *aspect* is modeled as a use case. The *join point* for an aspect is an activity of the base. The *pointcut* of an aspect is specified in terms of the activities of the base. While up to now proposed for modeling techniques like UML, an integrated behavior model is also *suitable* and *beneficial* for aspect-oriented modeling:

- It is well suited for modeling the base of a system at an early stage.
- It can naturally capture the functional and structural description of each aspect. An aspect may share the base domain model or add its own concepts.

Using the formal analysis of integrated behavior models for aspect-oriented modeling is beneficial as well. Each aspect can be analyzed for consistency, and the consistency of the entire system consisting of the base and aspects can be analyzed as well. Analysis is even more crucial for aspect-oriented models:

- Firstly, because of the separated specification of functionality in base and aspects. (Note that separate specification of functionality also exists in complex modular systems.)

- Furthermore, an aspect is specified once but can be used in many different places of the system. (Note that this also bears similarity with modular systems, where a module can be explicitly used by many other modules.)
- Lastly, an aspect is specified on top of and added to modules later on, with modules not necessarily being aware of the aspect. (Note that this is not the case in object-orientation, but is unique to aspect-oriented techniques and similar techniques.)

Because of these three properties, it is difficult to understand and manually analyze functional and data dependencies between base and aspects, and also between aspects. On the other hand, there are well known benefits of this kind of separation of concern, namely for maintenance, reuse, organisation of work etc.

We use ACTIGRA to model the running example before and after the composition, which is carried out manually following the formalization described in [20]. Apart from the use case diagram, all figures have been generated with ACTIGRA.
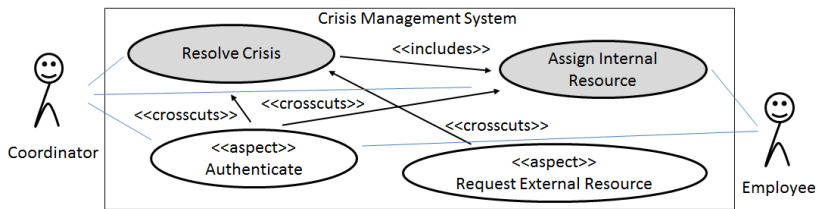

Fig. 1. Crisis Management Use Cases

2.1 The Crisis Management System Example

We present our modeling approach using an example from the Crisis Management Systems Case Study [21]. A crisis management system helps in identifying, assessing, and handling a crisis such as a fire, a flood, or an accident. It orchestrates communication between all parties handling the crisis by managing resources and access to information. Besides informal requirements, the case study contains a wide range of models related to software development. We have adapted a coherent subset of use cases, classes, and activities in the vein of the case study to illustrate our approach. Fig. 1 gives an overview of the chosen use cases. We are using an *«aspect»* stereotype for an aspect use case and a *«crosscuts»* stereotype for the relation of an aspect to the base. Analogous stereotypes have been proposed in [22]. The *«crosscuts»* relation means that the behavior of the aspect is added to a base without refering to the aspect in the base explicitly. It is called *«crosscuts»* because often aspects capture concerns that are broadly scattered. However, it can also be used for adding any other concern without changing the base.

Use Case *ResolveCrisis* The intention of this use case is to resolve a crisis by requesting employees and external resources to execute appropriate missions. An available employee is chosen as the coordinator. First, he or she has to capture the witness' report. With the help of the system, he or she creates the corresponding mission(s). Next, he or she assigns missions to resources and controls their completion.

This use case includes the use case *AssignInternalResource*, indicated by the *«includes»* relationship. When the use case *ResolveCrisis* is refined by an activity diagram it will contain a so called complex activity named *AssignInternalResource*.

Use Case *AssignInternalResource* The intention of this use case is to find, contact, and assign a mission to the most appropriate available employee. Here, appropriateness simply means availability. An available employee is chosen. The employee receives the mission information. Based on it he or she can accept, and is thus assigned to the mission.

When this use case is refined, the refining activity diagram serves as the refinement of the corresponding complex activity *AssignInternalResource*.

Use Case *Authenticate* The actor involved is either a coordinator or an employee. The intention of this use case is to authenticate the actor to the system since authentication is required to use the functions of the system. If the actor is not yet logged on, login id and password are prompted, entered and validated.

This use case is designed into the system upfront as an *«aspect»*. It *«crosscuts»* *ResolveCrisis*, where the coordinating employee has to log on, and *AssignInternalResource*, where all chosen employees have to log on, both, before further activities take place. In a real system, this use case would affect a lot of further use cases. Since the pointcut of this aspect is specified in terms of activities, the complete specification of the composition is given later.

Use Case *RequestExternalResource* The intention of this use case is to request help for a mission from an external resource such as an ambulance service. A request is sent to an external resource. The request is either served or denied.

This use case is added as an *«aspect»* during maintenance because the base system is conceived for one institution and the next version shall allow interaction with other institutions in a distributed system. Using an aspect can evolve the system without changing the base. We are using the same stereotype *«crosscuts»* because technically there is no difference whether an aspect is used once or several times. The aspect shall conditionally replace the use case *AssignInternalResource* if the coordinator whishes to request external resources. The complete specification of the aspectual composition is given later.
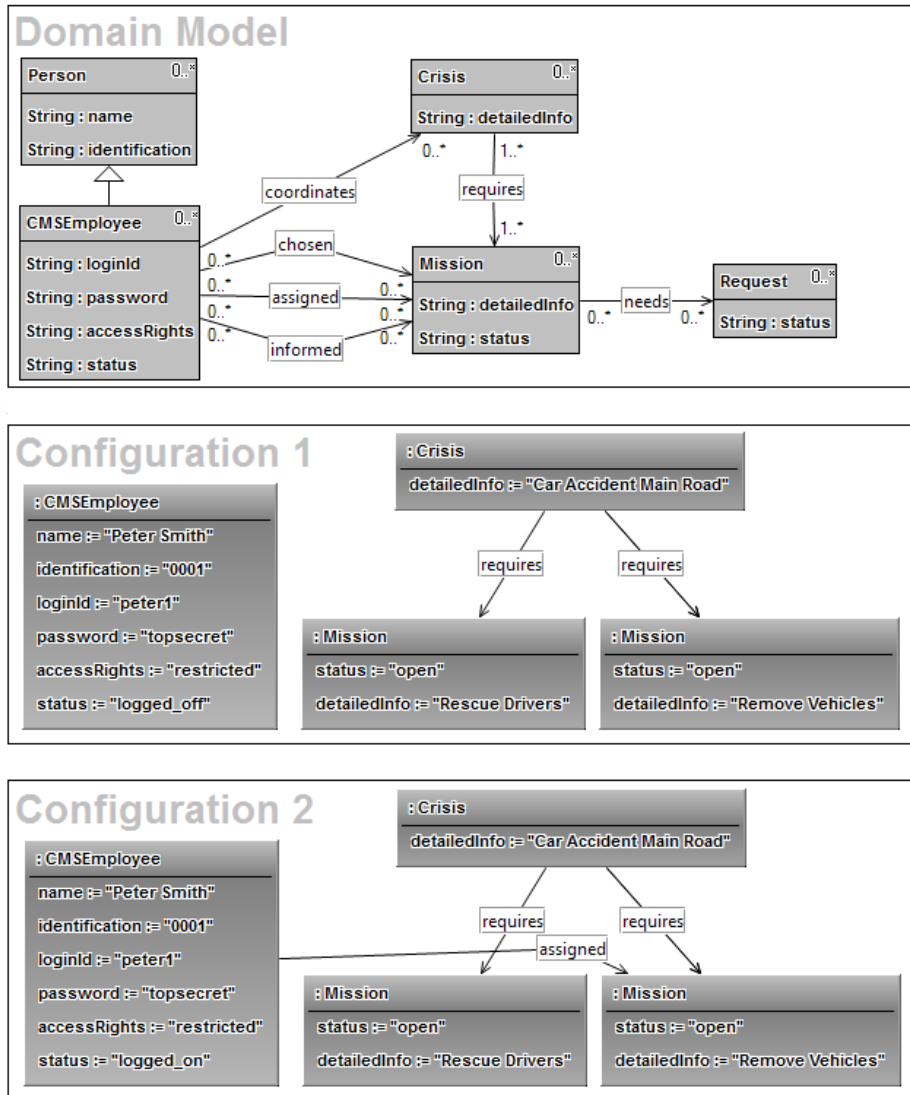
Domain Model

Person 0..*
- String : name
- String : identification

CMSEmployee 0..*
- String : loginId
- String : password
- String : accessRights
- String : status

Crisis 0..*
- String : detailedInfo

Mission 0..*
- String : detailedInfo
- String : status

Request 0..*
- String : status

coordinates
requires
chosen
assigned
informed
needs

Configuration 1

: CMSEmployee
- name := "Peter Smith"
- identification := "0001"
- loginId := "peter1"
- password := "topsecret"
- accessRights := "restricted"
- status := "logged_off"

: Crisis
- detailedInfo := "Car Accident Main Road"

requires   requires

: Mission
- status := "open"
- detailedInfo := "Rescue Drivers"

: Mission
- status := "open"
- detailedInfo := "Remove Vehicles"

Configuration 2

: CMSEmployee
- name := "Peter Smith"
- identification := "0001"
- loginId := "peter1"
- password := "topsecret"
- accessRights := "restricted"
- status := "logged_on"

: Crisis
- detailedInfo := "Car Accident Main Road"

requires   requires
assigned

: Mission
- status := "open"
- detailedInfo := "Rescue Drivers"

: Mission
- status := "open"
- detailedInfo := "Remove Vehicles"

Fig. 2. Type graph (top), initial configuration (middle), simulation result (bottom)

2.2 Integrated Behavior Models for the Base

A part of the domain model of the crisis management system is given in Fig. 2 using the type graph of ActiGra. A "Crisis" "requires" the fulfillment of some "Missions". A "CMSEmployee" "coordinates" a crisis or is "chosen" or "informed" or "assigned" to a mission. The "status" attribute of the employee is either set to "logged on" or "logged off". For a mission that cannot be assigned to an employee, a "Request" "needs" to be generated. Its "status" is either "sent" or "served".

For the subsequent analysis we need a so called *initial configuration* of our system. It contains object instances of the classes defined in the type graph (cf. Fig. 2, middle). Either a valid configuration can be supplied or the system contains graph transformation rules that create corresponding objects. Then the initial configuration is the empty one.

Our well formed activity models (cf. Sect. 3) consist of simple and complex activities, start and end nodes, decisions followed by a merge, and loop nodes. Directed arcs can be labeled by structural constraints ([...]) or interactively evaluated user constraints (<...>).

The use case *ResolveCrisis* is refined by the topmost activity diagram in Fig. 3. Firstly, the coordinating employee is determined who then has to capture the witness report. The first loop generates the required missions. The next loop assigns an employee to each mission using the complex activity *AssignInternal-Resource*. The last loop controls the success of the missions. We have omitted constraints at the loops, since this use case is not presented in more detail. It is used only to illustrate the composition of several aspects.
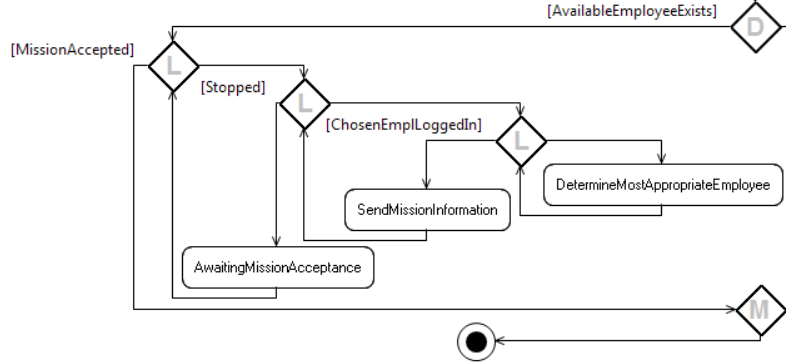
The use case *AssignInternalResource* is refined by the activity diagram in the middle of Fig. 3. The first decision node checks whether the innermost activity *DetermineMostAppropriateEmployee* is applicable. The constraint *[AvailableEmployeeExists]* (cf. Fig. 4) checks whether an employee has not yet been chosen for any mission. The positive pattern "existence of employee" describes parts of a graph that have to exist. The "not chosen" negative application condition (NAC) states that the constraint does not allow this pattern. A constraint can have zero or any number of NACs. This constraint also has a NAC "not informed" and a NAC "not assigned", which are not depicted, expressing that an employee must not be involved in a mission anyhow. Only if the constraint is satisfied, the arc labeled with it can be executed. Each of the following loops are applied until a constraint is satisfied. The innermost loop chooses an employee. Only if the employee is logged on, captured by *[ChosenEmployeeLoggedIn]* (cf. Fig. 4), the enclosing loop is executed which sends mission details to the employee. Only if that is successful, captured by *[Stopped]* (cf. Fig. 4), the system waits for acceptance, in which case, captured by *[MissionAccept]* (cf. Fig. 4) the use case terminates successfully.

In an integrated behavioral model, each activity is refined by a pre- and a post-condition, describing the situation in which the activity can be applied, and the effect. The pre-condition consists of a positive pattern for a graph that has to exist, optionally equipped with negative application conditions (NACs) capturing negative patterns preventing the application. Conditions are presented

ResolveCrisis

FindCoordinator

CaptureWitnessReport

L

CreateMission

L

AssignInternalResource

L

ResourceExecutesMission

AssignInternalResource

[AvailableEmployeeExists]   D

[MissionAccepted]   L

[Stopped]

L

[ChosenEmplLoggedIn]

L

SendMissionInformation

DetermineMostAppropriateEmployee

AwaitingMissionAcceptance

M
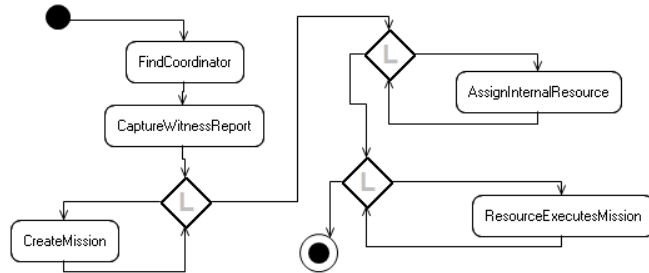
Authenticate_Aspect   RequestExternalResource_Aspect

[RequestLoginIsApplicable]

D

RequestLogin

M

SendRequest

D

DenyRequest   AcceptRequest

M
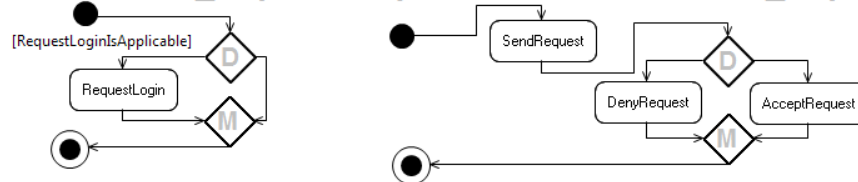
Fig. 3. Crisis Management Activity Diagrams
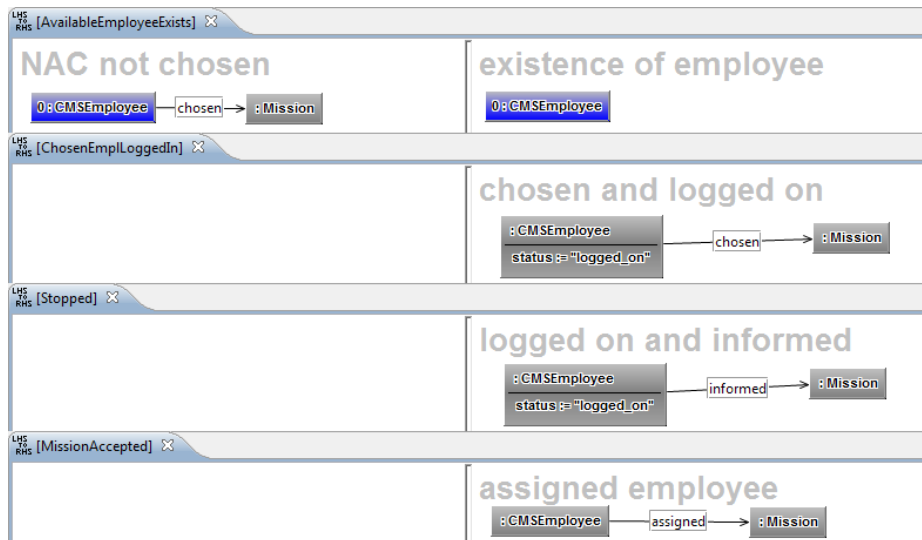
Fig. 4. Structural constraints for activity model *AssignInternalResource*
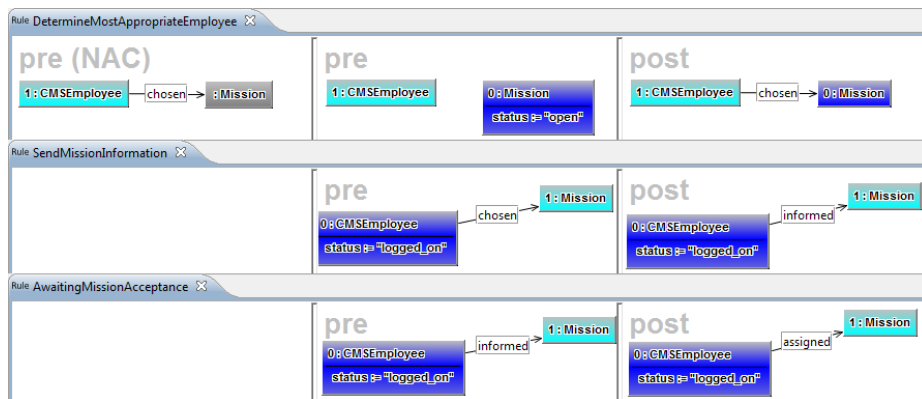
Fig. 5. Pre- and post-conditions of activities of use case *AssignInternalResource*

for the activity model of use case *AssignInternalResource* in Fig. 5. The left column presents NACs, the middle column the positive pre-condition. The right column presents the effect of each activity, i.e. the post-condition. The identity of a node is preserved throughout the three columns by assigning the same instance number to it. The first row states that a employee can be assigned once to an open mission. The second row states that a chosen employee who is logged on can be informed about the mission once. The last row states that an informed employee can be assigned to a mission.

2.3 Aspect Modeling

An aspect is identified on the use case level and subsequently refined with an activity diagram. The use case *Authenticate* is refined by the corresponding activity diagram in Fig. 3 (bottom left). If an employee is not yet logged in, the execution of *RequestLogin* changes the status of the employee from "logged_off" to "logged_on", (cf. also Fig. 6, top).

The use case *RequestExternalResource* is refined by the corresponding activity diagram in Fig. 3 (bottom right). A request is sent to an external resource (cf. Fig. 6, second row). Either the request is accepted (cf. Fig. 6, third row) or denied (cf. Fig. 6, last row). In our example, the decision is not specified further since it comes from an external system. For simulation, an arbitrary arc is chosen and during analysis, both arcs are analyzed.

Fig. 6. Pre- and post-conditions of activities of aspects

Based on the activity models, the aspectual composition can be specified using the following elements:

- The *name* of the aspectual use case is given.
- One of the *modifiers* is given, which describes, how the aspectual use case is composed. Here, we use the modifiers *before*, *after* and *replace* of aspectual

programming languages like AspectJ [12], albeit more complex modifications are conceivable, especially during modeling.

- The *pointcut* specifies, where the aspectual use case is composed, i.e., which join point activities are selected by the pointcut. We assume unique names for activities. Pointcuts can be specified using rather sophisticated intensional languages or by mere enumeration of activities. Here we adopt the latter approach.
- A *condition* specifies under which circumstances the aspect becomes effective. This allows for a flexible composition with the base. If the condition is fulfilled, the aspect is executing. If no condition is given, the aspect will always execute. As conditions we use structural constraints or interactively evaluated conditions.

An aspect is woven in each single join point which matches the pointcut definition. Here, an aspect has only one pointcut, but more complex weaving technologies exist. Regarding order of composition, we simply follow the order of specifications. After a replace composition without a condition, further aspects might not be applicable. Furthermore, we do not consider aspects of aspects in our model. Note that aspects without conditions can simulate aspects with conditions by integrating the condition into the normal control flow of the aspect at the beginning of the aspect.

The composition specification for each *«crosscuts»* relationship is given in Table 1. The *Authenticate* aspect is composed once after the activity *FindCoordinator* (of the use case *ResolveCrisis*), and once after the activity *DetermineMostAppropriateEmployee* (of the use case *AssignInternalResource*). The *Authenticate* aspect has no condition since it shall always be carried out. Also, this aspects checks itself whether an employee is already logged on. Aspect *RequestExternalResource* conditionally replaces the activity *AssignInternalResource* (of the use case *ResolveCrisis*) if the coordinator decides to do so.

Use Case	Modifier	Pointcut (Activity)	Condition
Authenticate	`after`	FindCoordinator, DetermineMostAppropriateEmployee	[empty]
RequestExternal-Resource	`replace`	AssignInternalResource	<Request External Resource?>

Table 1. Aspect-Oriented Composition

Finally, ACTIGRA can be used to execute an activity diagram with its pre- and post-conditions. When applying use case *AssignInternalResource* to the initial *Configuration 1* in the middle of Fig. 2, the simulation is animated on the activity diagram. The execution starts with the innermost loop and executes *DetermineMostAppropriateEmployee* as often as possible but it cannot proceed because the condition *[ChosenEmplLoggedIn]* is never fulfilled. This is due to the absence of an aspect which will be analyzed in more depth later.

2.4 Aspect Weaving

Since its coining, the term aspect-oriented programming has always been a synonym for implementing aspects using *weaving*, i.e., for a transformation of the source code which inserts the aspect code in all places specified by a pointcut. We apply the same concept to the activity model of the aspect-oriented use case, i.e., we weave the aspect activity model into activity models of the base. Weaving is controlled by the composition specifications illustrated in the previous section. The modeling of pre- and post-conditions does not play a specific role during weaving, which is also feasible without, albeit for the subsequent analysis they are mandatory. In [20] we proposed and formalized the model weaving within our approach. Here, we present it informally only and demonstrate the result for the example. The weaving process is as follows. Firstly, the join points have to be determined using the pointcut specifications, i.e., all places where weaving has to take place. The two cases, weaving with conditions and without conditions, have to be combined with the modifiers *before*, *after* and *replace*.

    Weaving without conditions:

- *before*: The aspect activity diagram replaces all incoming arcs to the join point activity specified in the pointcut.
- *after*: The aspect activity diagram replaces the outgoing arcs from the join point activity specified in the pointcut.
- *replace*: The aspect activity diagram replaces the activity. The incoming and outgoing arcs are glued to the first rsp. last activities of the aspect activity.

    Weaving with conditions:

- *before*: The condition is inserted as a decision node into the aspect diagram after the start node with the positive arc linked to the first activity and with the negative arc linked to the end node. A merge node is inserted before the end node and all incoming arcs become incoming arcs of the merge node. The augmented aspect activity diagram replaces all incoming arcs to the join point activity specified in the pointcut.
- *after*: The condition is inserted as a decision node into the aspect diagram after the start node with the positive arc linked to the first activity and with the negative arc linked to the end node. A merge node is inserted before the end node and all incoming arcs become incoming arcs of the merge node. The augmented aspect activity diagram replaces all outgoing arcs from the join point activity specified in the pointcut.
- *replace*: The condition is inserted as a decision node before (see *before* above) the join point activity specified in the pointcut. The positive arc of the branch is linked to the first activity of the aspect. The negative arc is linked to the join point activity. A merge node is inserted after (see *after* above) the join point activity. All incoming arcs of the end node of the aspect become incoming arcs of the merge node.

    In all cases, start and end nodes of the aspect activity diagram are removed and dangling arcs are glued correspondingly. The weaving results of the example
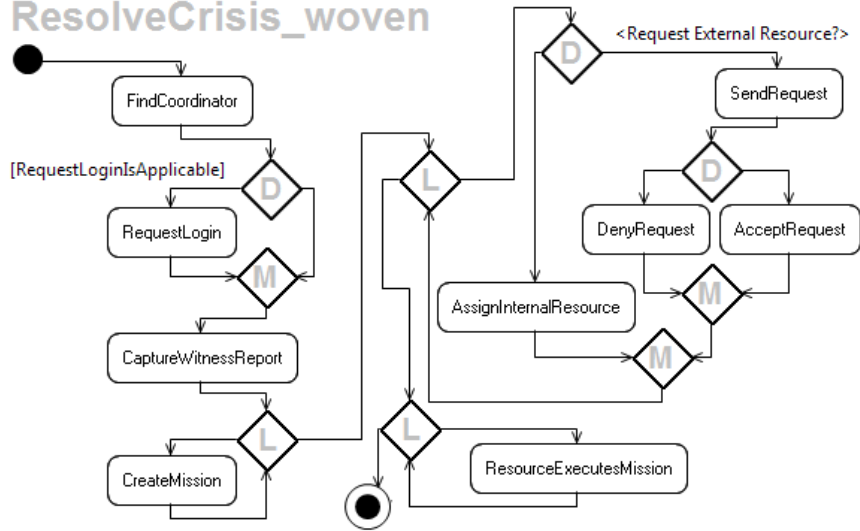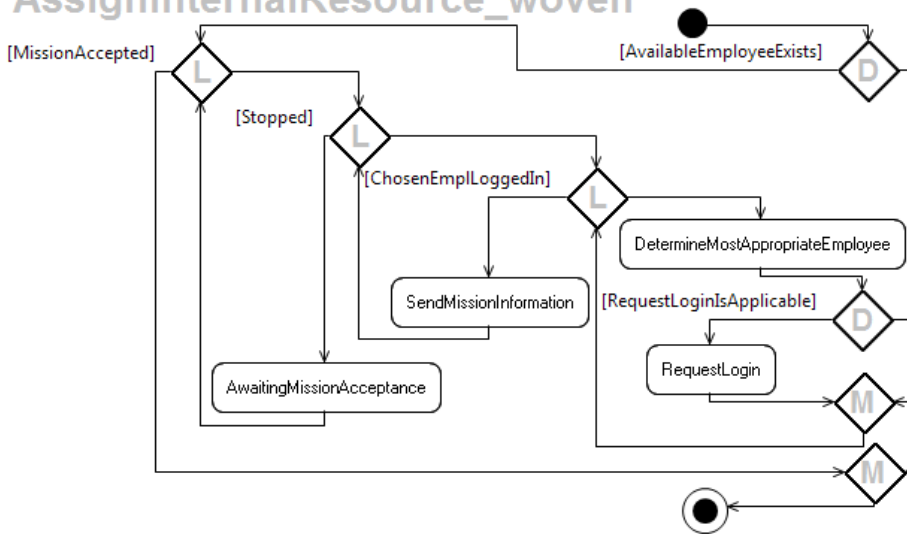
Fig. 7. Use cases with aspects woven

are depicted in Fig.7. The aspect *Authenticate* is woven into the use case *ResolveCrisis* after the join point activity *FindCoordinator*. It is also woven into the use case *AssignInternalResource* after the join point activity *DetermineMostAppropriateEmployee*. The aspect *RequestExternalResource* is woven into the use case *ResolveCrisis*. It is linked via a new decision node to the join point activity *AssignInternalResource*. Note that after weaving the complex activity *AssignInternalResource* is changed but this is not visualized in the activity model for use case *ResolveCrisis*.

Again, ACTIGRA can be used to execute an activity diagram with its pre- and post-conditions. When applying *AssignInternalResource_woven* to the initial configuration in the middle of Fig. 2, the simulation is animated on the activity diagram. It starts with the innermost loop and executes each loop and activity once resulting in *Configuration 2* of Fig. 2, terminating successfully.

3 Formalization of Integrated Behavior Models

Integrated behavior models can be formalized by graph transformation systems. Domain models are formalized by type graphs, while configurations are specified by their instance graphs. Pre- and post-conditions of activities as well as constraints are expressed by graph transformation rules. The control flow of activity models is defined by graph transformation sequences.

Firstly, we present the underlying theory of graph transformation systems, consisting of graphs, transformations, and graph transformation sequences. These systems can be analyzed for conflicts and causalities between transformations. Secondly, we present the semantics of integrated behavior models, which is rooted in graph transformation sequences that are used to simulate the execution of activity models.

3.1 Graph Transformation Systems

Graphs are often used as abstract representation of diagrams. When formalizing object-oriented modeling, graphs occur at two levels: the type level (defined based on class models) and the instance level (given by all valid object models). This idea is described by the concept of *typed graphs*, where a fixed *type graph* TG serves as an abstract representation of the class model. As in object-oriented modeling, types can be structured by a generalization relation. Multiplicities and other annotations are not formalized by type graphs, but have to be expressed by additional graph constraints. Instance graphs of a type graph have a structure-preserving mapping to the type graph.

Graph transformation is the rule-based modification of graphs. Rules are expressed by two graphs (L, R), where L is the left-hand side of the rule and R is the right-hand side, usually overlapping in graph parts. Rule graphs may contain variables for attributes. The left-hand side L represents the pre-conditions of the rule, while the right-hand side R describes the post-conditions. $L \cap R$ (the graph part that is not changed) and the union $L \cup R$ should form a graph again, i.e.,

they must be compatible with source, target and type settings, in order to apply the rule. Graph $L \setminus (L \cap R)$ defines the part that is to be deleted, and graph $R \setminus (L \cap R)$ defines the part to be created. Furthermore, the application of a graph rule may be restricted by so-called *negative application conditions* (NACs) which prohibit the existence of certain graph patterns in the current instance graph. Note that we indicate graph elements common to L and R or common to L and a NAC by equal numbers.

A *direct graph transformation* $G \overset{r,m}{\Longrightarrow} H$ between two instance graphs G and H is defined by first finding a match m of the left-hand side L of rule r in the current instance graph G such that m is structure-preserving and type-compatible and satisfies the NACs (i.e. the forbidden graph patterns are not found in G). We use injective matches only. Attribute variables used in graph object $o \in L$ are bound to concrete attribute values of graph object $m(o)$ in G. The resulting graph H is constructed by (1) deleting all graph items from G that are in L but not also in R; (2) adding all those new graph items that are in R but not also in L; and (3) setting attribute values of preserved and created elements.

A *graph transformation (sequence)* consists of zero or more direct graph transformations. A set of graph rules, together with a type graph, is called a *graph transformation system* (GTS). A GTS may show two kinds of non-determinism: (1) For each rule several matches may exist. (2) Several rules might be applicable to the same instance graph. There are techniques to restrict both kinds of choices. The choice of matches can be restricted by object flow, while the choice of rules can be explicitly defined by control flow on activities.

3.2   Conflicts and Causalities between Transformation Rules

A reason for non-determinism of graph transformation systems is the potential existence of several matches for one rule. If two rules are applicable to the same instance graph, they might be applicable in any order with the same result. In this case they are said to be *parallel independent* otherwise they are in conflict.

Conflict Types. One rule *may disable* the second rule. In this case, the first rule r_1 is also said to be causing a *conflict* with the second rule r_2. The following types of conflicts can occur:

delete/use: Applying r_1 deletes an element used by the application of r_2.
produce/forbid: Applying r_1 produces an element that a NAC of r_2 forbids.
change/use: Applying r_1 changes an attribute value used by the application of r_2.

Causality Types Conversely, one rule *may trigger* the application of another rule. In this case, this sequence of two rules is said to be *causally dependent*. The following types of causalities can occur where rule r_1 *triggers* the application of r_2:

produce/use: Applying r_1 produces an element needed by the application of r_2.

delete/forbid: Applying r_1 deletes an element that a NAC of r_2 forbids.

change/use: Applying r_1 changes an attribute value used by the application of r_2.

Example 1. Figure 8 shows an example of a produce-use dependency between the transformation rules *DetermineMostAppropriateEmployee* and *SendMissionInformation*. While the first rule creates a new relation of type "chosen" between a "CMSEmployee" and a mission, the second rule uses this relation and deletes it.
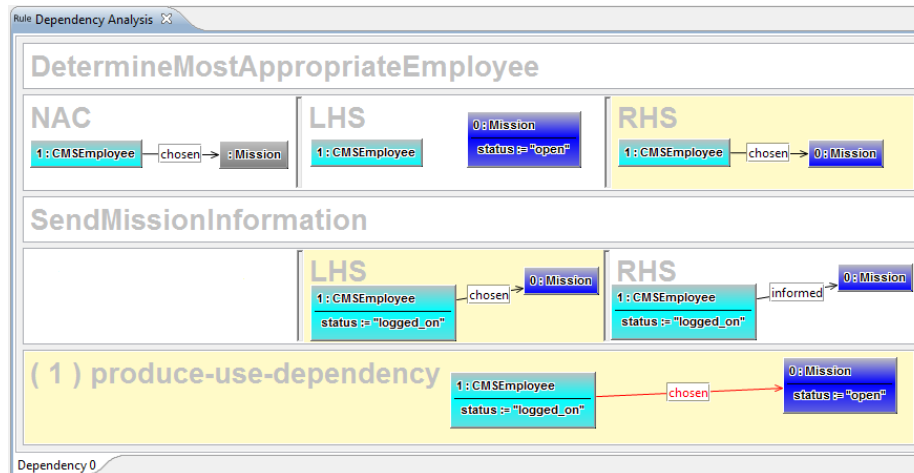

Fig. 8. Produce-use causality example between two transformation rules

3.3 Semantics of Integrated Behavior Models

As in [23], we define integrated behavior models by *well-structured activity models* consisting of a start activity s, an activity block B, and an end activity e such that there is a transition between s and B and another one between B and e. An *activity block* can be a simple activity, a sequence of blocks, a fork-join structure, decision-merge structure, and loop. In addition, we allow complex activities which stand for nested well-structured activity models. In this hierarchy, we forbid nesting cycles. Activity blocks are connected by transitions (directed arcs). Decisions have an explicit *if*-guard and implicit *else*-guard which equals the negated *if*-guard, and loops have a *loop*-guard with corresponding implicit *else*-guard. Guards can be user-defined, i.e. independent of system configurations, or graph constraints checking certain conditions on system configurations.

The semantics of an integrated behavior model is defined by a set of graph transformation rules sequences. Considering the formalization of activities with pre- and post-conditions by graph transformation rules, the sequences represent all possible control flow paths defined by well-structured activity models. In this context, each graph constraint is translated to a rule containing the constraint as left-hand side and an identical right-hand side. The semantics of a simple activity $Sem(A)$ is a set consisting of one sequence with only one rule. The semantics of two subsequent activity blocks A and B contains all sequences beginning with a sequence of $Sem(A)$ and ending with a sequence of $Sem(B)$. For decision blocks we construct the union of sequences of both branches (preceded by the guard rule or its negation, respectively). For loop blocks, we construct sequences containing the body of the loop $0 \leq i \leq n$ times (where each body sequence is preceded by the loop guard rule in case that the loop guard is not user-defined). The semantics of a complex activity is the semantics of the largest block of its contained integrated behavior model.

Example 2. Considering the integrated behavior model of use case *AssignInternalResource*, its semantics contains e.g. sequence *AvailableEmployeeExists*, *NotMissionAccepted*, *NotStopped*, *NotChosenEmpLoggedIn*, *Determine MostAppropriateEmployee*, *NotChosenEmpLoggedIn*, *DetermineMost AppropriateEmployee*, *ChosenEmpLoggedIn*, *SendMissionInformation*, *Stopped*, *AwaitingMissionAcceptance*, *MissionAccepted.*

4 Using Plausibility Checks for Integrated Behavior Models with Aspects

Given the formal semantics of integrated behavioral models as simulation runs, these sequences can be formally analyzed for favorable and critical dependencies and conflicts between the rules in those sequences. The results are captured in different sets of *relations*.

After introducing the checks from [15], we discuss how they can be used specifically in aspect-oriented modeling. The checks are supported by ACTIGRA.

4.1 Plausibility Checks for Integrated Behavior Models

Integrated behavior models combine control flow models with functional behavior specifications. Since two kinds of models are used for this purpose, static analysis of integrated behavior models helps to argue about their consistency. In [15], a variety of so-called plausibility checks are presented that can be used for argumentation. For each check, favorable and critical signs can be determined that support or are opposed to behavior consistency.

1. *Initialization:* The applicability of the first rule in the specified control flow to the initial configuration forms a favorable sign.

2. *Trigger causality along control flow:* This plausibility check computes for each rule in a given control flow which of its predecessor rules may trigger this rule. It favorable that at least one predecessor exists for each rule.
3. *Conflicts along control flow:* This plausibility check computes for each rule in a given control flow which of its successor rules may be disabled this rule. It favorable that there do not exist such successors for any rule.
4. *Trigger causality against control flow:* This plausibility check computes for each rule in a given control flow which of its successor rules may trigger this rule. It favorable that there do not exist such successors for any rule. In case that such a successor exists, the modeler should inspect if it should be shifted before the rule it triggers.

Note that guards are reformulated as non-changing rules and integrated into the plausibility check then.

4.2 Analysis of Aspects with Plausibility Checks

In our modeling approach, plausibility checks will be computed for base and aspect separately, and for the entire woven model. The analysis is therefore applied incrementally in two stages:

1. The consistency of the base and the aspects is checked separately. It is desirable that consistency is achieved separately where feasible.
2. The consistency of the composition of aspect and base is checked. It suffices to analyze the control flow that contains the woven aspect activities. This can be deduced from the pointcut specification (but this inference is not yet implemented in ACTIGRA, and the resulting weaving has to be computed by hands). The problems revealed are directly related to this composition if consistency was achieved before hand. This stage includes checking the consistency between aspects, since their effects on each other can not be generally checked on the stage before. Instead, their specific effect on each other when composed with a base system is considered.

At state 2, in the control flows affected by the aspects, triggers and conflicts between activities of the base may change compared to state 1 if use cases are replaced during the weaving. Conflicts between base activities (including conflicts of an activity with itself) may disappear because an aspect added to a control flow changes the sequence such that a conflict is no longer effective. Newly arising triggers and conflicts at stage 2 have different sources. They may occur between base and aspect or between different aspects. They may also occur between activities of one aspect due to the following reason. After weaving, an aspect becomes part of new control flows. These control flows can have the effect that an aspect is potentially executed several times in a loop. Then its activities are potentially in conflict with themselves and also with each other. If the activities were not part of such loops before weaving, there are new conflicts and triggers after the weaving.

Conflicts and causalities may occur between individual activities rsp. corresponding transformation rules. In general, a potential conflict need not lead to a concrete conflict; this is especially true in the case of change/use conflicts which often indicate that activities use attributes changed by other activities.

- *Conflict between base and aspect:* If a conflict exists between a base activity rsp. its rule r_1 and an aspect activity rsp. its rule r_2, the aspect is disabled by the basis, and vice versa. This is not desirable for before- and after-aspects. For replace aspects it is no problem if the rule r_1 of the basis is completely replaced by the aspect.
- *Conflicts between aspects:* A conflict can exist between two activities rsp. rules stemming from two different aspects. If one aspect disables another aspect and is woven into an activity diagram in the control flow before the other aspect, the conflict is not desirable and has to be examined further.
- *Trigger causality between base and aspect:* If a trigger from base to aspect exists, this is not a problem. If no trigger exists this is also not a problem but then it should be ensured that the aspect still can work.
- *Trigger causality between aspects:* If causalities exist they should be along the control flow of the entire system including aspects. If no trigger causalities between aspect exists, it should be ensured that each aspect can work.

The plausibility checks can be used at *stage one* as follows:

- Initialization is checked for base and aspects separately. At least one base activity model should be applicable to the initial configuration. If an aspect is applicable to the initial configuration this means that it is orthogonal to the base or perhaps conflicting with the base. It is not required that an aspect is applicable to the initial configuration.
- Triggers along control flow inside an activity model are beneficial. Absence has to be checked for consistency.
- Triggers against control flow have to be checked for consistency.
- Conflicts inside an activity model have to be checked for consistency.

At *stage two* plausibility checks can be used as follows:

- An aspect must be applicable to the initial state or needs trigger causalities.
- Trigger causalities along the control flow may stem from the base or from other aspects.
- The check for triggers against control can be used to identify problematic cases. It may be the case that a join point is not well chosen, i.e. too late or too early in a given use case or even in the wrong use case.
- There must not be conflicts newly introduced, i.e., of aspect activities with the (remaining) base or with each other.
- If the base was not consistent without aspect(s) one should check if the entire system becomes consistent after aspect composition.

5 Analysis of the Example

Here we present the plausibility analysis using ACTIGRA of the use case *AssignInternalResource*, the aspect use cases *Authenticate* and *RequestExternalResource*, and the woven use cases *AssignInternalResource* and *ResolveCrisis*.

Analyzing the use case *AssignInternalResource* ACTIGRA visualizes the results of each plausibility check separately in the activity model. For reasons of space, we can not include the figures for all checks.
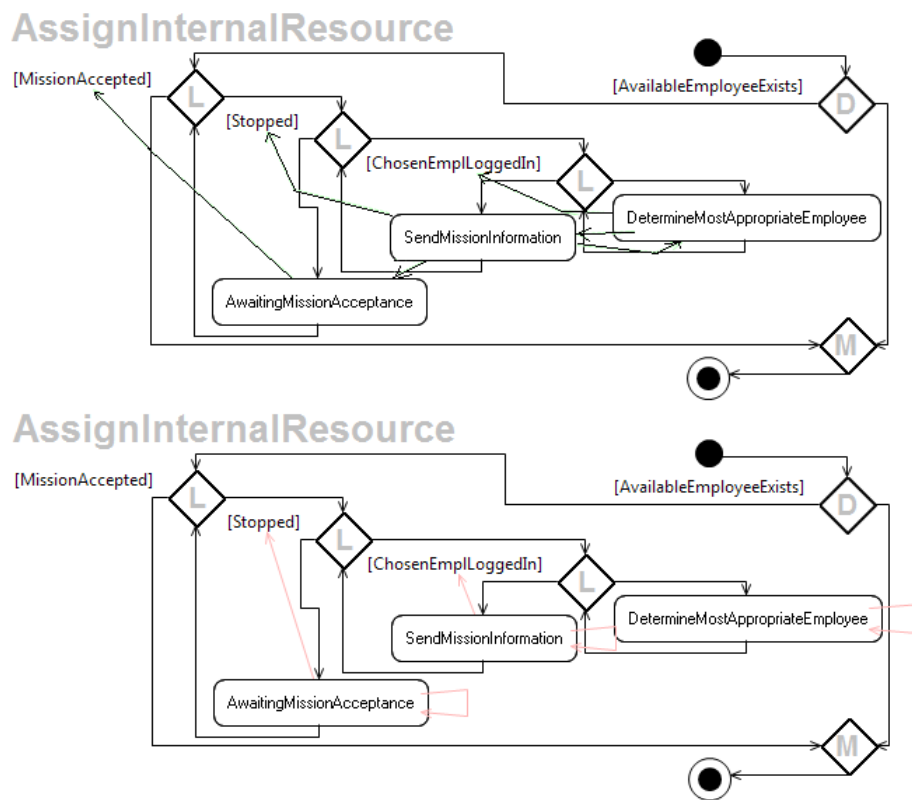
Fig. 9. Trigger and conflict checks for *AssignInternalResource*

1. *Initialization*: The first reachable activity *DetermineMostAppropriateEmployee* is applicable to the initial *Configuration 1*.
2. *Triggers along control flow* (cf. Fig. 9, top): All activities and conditions have triggers. Because of the loops, these triggers are along the control flow.

SendMissionInformation triggers *DetermineMostAppropriateEmployee.* Here the first activity deletes the "chosen" arc which is forbidden by the second activity. The condition *[Stopped]* however avoids this path. Since there is no other trigger for *DetermineMostAppropriateEmployee* and since it is applicable to the initial configuration there is no problem.

DetermineMostAppropriateEmployee triggers *SendMissionInformation* by producing a "chosen" arc which is used by *SendMissionInformation.* However, Fig. 8 reveals that *SendMissionInformation* is not fully enabled by this trigger, since the employee status is not changed. Moreover, there is no other trigger that would change the status. As the employee status in *Configuration 1* (cf. Fig. 2) is "logged_off", the activity model is not executable on this configuration. *SendMissionInformation* triggers *AwaitingMissionAcceptance* by producing the "informed" arc used. More triggers are not needed.

The three triggers for the conditions are producing something used by the conditions and are therefore plausible.

3. *Triggers against control flow*: The triggers are the same as above, only now they are categorized differently. The triggering of conditions is still along the control flow. The mutual triggers between *DetermineMostAppropriateEmployee* and *SendMissionInformation* and the trigger from *SendMissionInformation* to *AwaitingMissionAcceptance* are now considered against the control flow. However, their effects on the entire diagram as discussed above remain the same.

4. *Conflicts along control flow* (cf. Fig. 9, bottom): There are conflicts of each activity with itself. That means that if an activity can occur in the control flow after itself it cannot be applied a second time because it deletes something that is needed or it produces something that is forbidden. This is no problem here.

   Also there is a conflict between *SendMissionInformation* and the condition *[ChosenEmplLoggedIn]* which means that the loop will not be executed a second time which is desirable. The same holds for *[AwaitingMissionAcceptance]* and *[Stopped]*.

Analyzing the aspect *Authenticate* Since this aspect contains only one activity and only a condition that checks the applicability of this activity, only two checks are interesting. We explain them shortly without another figure. Please compare Figure 3. The checks for conflicts and triggers against control flow do not make sense in absence of further activities.

1. *Initialization*: Activity *RequestLogin* is applicable to the initial configuration.
2. *Triggers along control flow*: Obviously the activity *RequestLogin* has no trigger but can be applied to the initial configuration.

Analysing the aspect *RequestExternalResource* In Fig. 10, we visualize the analysis results of the check for triggers. The complete results are as follows:

1. *Initialization*: Activity *SendRequest* is applicable to the initial configuration.

2. *Triggers along control flow*: The activity *SendRequest* is never triggered but applicable to the initial configuration. This activity triggers the activity *AcceptRequest* and *DenyRequest* which is consistent. (cf. Fig. 10).
3. *Triggers against control flow*: There are no triggers against the control flow.
4. *Conflicts along control flow*: There are no conflicts.
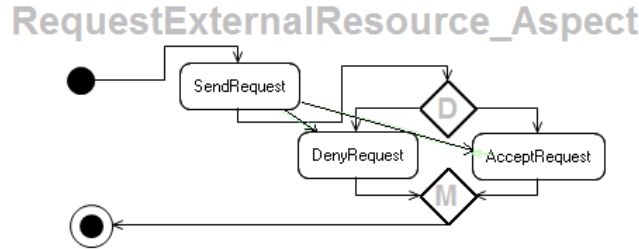
Fig. 10. Triggers *along* the control flow for aspect *RequestExternalResource*

Analyzing the *woven* use case *AssignInternalResource* For the use case *AssignInternalResource_woven*, the results are as follows (cf. Fig. 11.)

1. *Initialization*: *DetermineMostAppropriateEmployee* is still applicable.
2. *Triggers along control flow* (cf. Fig. 11, top): Firstly, there are the same triggers as in the unwoven use case *AssignInternalResource*. Secondly, *RequestLogin* triggers the conditions *[Stopped]* and *[MissionAccepted]* and the two activities *SendMissionInformation* and *AwaitingMissionAcceptance*. This is because *RequestLogin* changes the *status* to "logged_on" which is needed by all of the aforementioned elements.
3. *Triggers against control flow*: Firstly, there are the same triggers as before. Secondly, *RequestLogin* triggers the two activities *SendMissionInformation* and *AwaitingMissionAcceptance*. This is because *RequestLogin* changes the *status* to "logged_on" which is needed by all of the aforementioned elements. Again, because of the loops there are the same triggers along the control flow as against the control flow.
4. *Conflicts along control flow* (cf. Fig. 11, bottom): Now there is one less conflict than in the unwoven use case. The conflict of activity *DetermineMostAppropriateEmployee* with itself does not exist any longer because the overall control flow changed.

The insertion of the aspect into the base makes the woven activity model executable for the given *Configuration 1*. The reason is that the activity *RequestLogin* of the aspect provides the missing trigger for the activity *SendMissionInformation* of the base. Executing *AssignInternalResource_woven* on *Configuration 1* with ACTIGRA also terminated.

Analyzing the *woven* use case *ResolveCrisis* We cannot present the complete analysis of *ResolveCrisis_woven* for reasons of space since this would also require to illustrate all pre- and post-conditions of the involved activities. The interesting question from the aspect-oriented modeling point of view is the analysis of the conflicts and causalities between the aspects involved. This use case has two aspects woven at the top level and one nested aspect woven into its complex activity *AssignInternalResource*. We have to take into account the complete control flow including also all activities of the woven complex activity. There are some noteworthy analysis results (cf. Fig. 7, for the woven use cases):

- Between the two top level aspects *Authenticate* and *RequestExternalResource* there are no conflicts and causalities. It means that the two aspects are independent of each other. This is desirable, especially since the execution of *RequestExternalResource* is conditional.
- The aspect *Authenticate* is also woven into the complex activity *AssignInternalResource*. Here, *Authenticate* does not create conflicts and causalities with the top level aspect *RequestExternalResource*.
- The top level *Authenticate* aspect is the first in the control flow, the nested *Authenticate* aspect is the second in the control flow. The analysis reveals a conflict between the two, since the first occurence of the activity *RequestLogin* changes an attribute used by the second occurence. This is however only a potential conflict, since the first *RequestLogin* takes place for the coordinator and the second takes place for an employee.
- The activities of aspect *RequestExternalResource* are each in conflict with itself, because the aspect is now contained in a loop. Also, *DenyRequest* and *AcceptRequest* are in mutual conflicts since they are now contained in a loop. The same happens with the activity *RequestLogin* nested in the complex activity *AssignInternalResource* after the weaving. It is now in conflict with itself due to the outermost loop in which it is now contained.
- In the analysis of *AssignInternalResource* we identified triggers from *RequestLogin* to other elements. The first occurence of *RequestLogin* triggers now the same activities that are already triggered the second occurence in the nested aspect However, these are potential triggers, since the first *RequestLogin* takes place for the coordinator and the second takes place for an employee.

6   Related Work

The Crisis Management Systems (CMS) case study was proposed in [21] as a benchmark example for comparing aspect-oriented modeling approaches. The paper presents the requirements for a generic CMS informally and details the use cases for a "Car Crash CMS", a system for dealing with car accidents. A non-functional requirement of "Security" states that the CMS shall define access policies for various classes of users. Our analysis introduces an integrated behavior model of the generic CMS and it formalizes the aspects of authentication
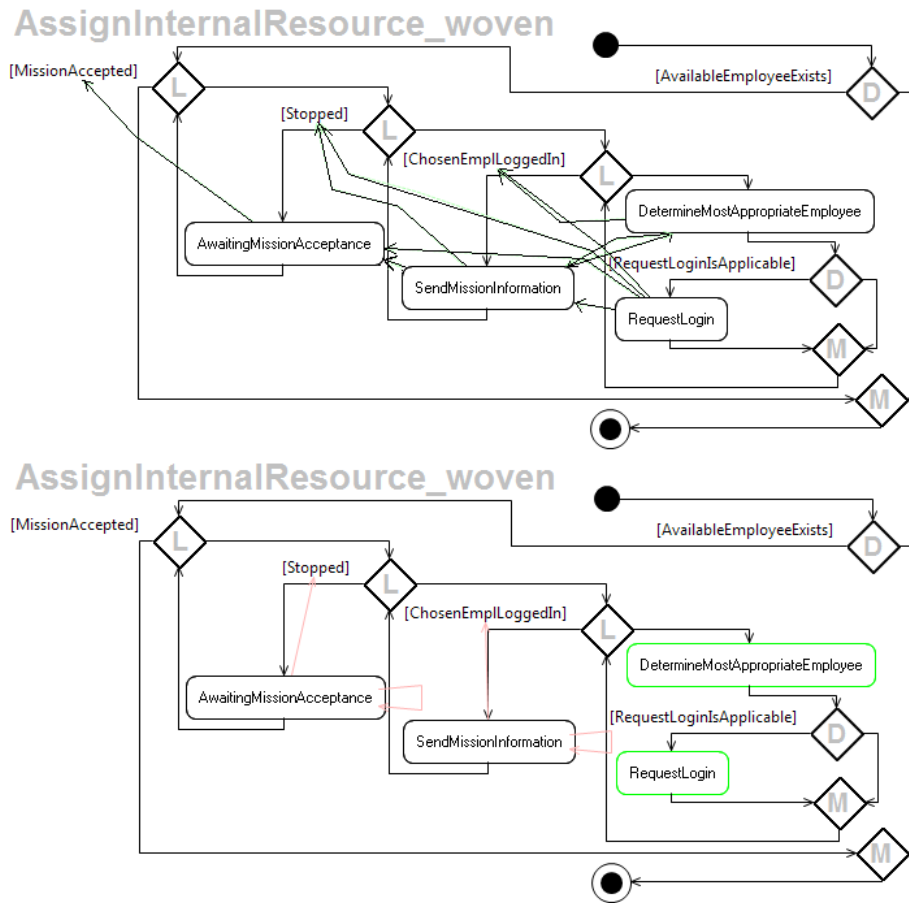
Fig. 11. Trigger and conflict checks for *AssignInternalResource_woven*

(a part of the Security requirement that can be described functionally) and the request of external resources (a functionality which in our modeling crosscuts several parts of the system). Our approach is functional in its nature since it requires that everything relevant in the system is modeled as model elements which create and remove activities.

In our previous work [20] we have used the same conflict and causality (formerly called dependency) definition as here. However, the control flow as given by the activity models was not taken into account, since at that time we could only use AGG [16] to perform the analysis. AGG computes a conflict and dependency matrix and for each two rules all potential conflicts and dependencies. Each conflict is given by graph and two transformations applied to it, while each dependency is given by a transformation sequence of length 2 with its intermediate graph. Given a a control flow, the relevant conflicts and dependencies have to be manually determined in AGG. With ACTIGRA [15], this step is automatized by integrating activity models.

As recalled in the introduction, several researchers have studied the problem of interference among aspects at the coding level. [24] classifies interactions of aspects with a base: an aspect can be considered spectative, regulative, or invasive with respect to the system to which is applied; in [25] the categories are formally described by temporal logic predicates on program states. This classification is useful also at the modeling level we adopted: a *spectative* aspect only gathers information about the system to which it is woven but does not influence the computations of the base otherwise; a *regulative* aspect changes the activation of activities under certain conditions but does not change the base computation further; an *invasive* aspect does change the base system arbitrarily. However, we focus on potential conflicts (and triggers) that may rise when given control flows are woven together.

Other tools for graph transformation systems allow for their specification and controlled simulation according to given activity flows: see for example Fujaba [26], VMTS [27], and GReAT [28]. These tools, however, do not provide support for analyzing conflicts and causalities: ACTIGRA [15] leverages the critical pair analysis implemented by AGG [16] to detect possibly unwanted interactions.

7 Conclusion and Outlook

Activity diagrams are a widely used modeling language for describing the functional behavior of a system at different level of abstractions, ranging from requirements models and work flow descriptions to more coding-oriented specifications like flowcharts. Their semantics however are often semi-formal and vary a lot. Integrated behavior models are one way to give formal semantics to activity models, moreover in a broader context integrated with a domain model by a refined specification of each activity in terms of the domain model. Such a semantics becomes even more useful when supported by a tool. Integrated behavior models are particularly apt for specifying requirements in a use case driven approach using UML.

We have used integrated behavior models which are supported by the ACTI-GRA tool. We modeled aspect-oriented separation of concern at the use case level. Since aspects typically also bear functional behavior, they were straightforward to model as graph transformation rules together with weaving among activities diagrams. Activity diagrams play a key role in the analysis, similar to that of data in dynamic program analysis: in a static program analysis one considers all the possible paths of execution (in fact also several unfeasible ones), while in dynamic analyses the input data are used to narrow the search space. Similarly, activities are used to drive the analysis to a concrete set of interactions, instead of considering all the conceivable ones for a given a set of aspects and a base. At the programming level, this reduction is mostly provided by the base, which gives the control flow in which aspects are intertwined. In our model aspects and base are described as transformation rules on the domain model, thus the activity is the key to reduce the indeterminacy of all the possible weaving actions. By integrating the critical pair ("static" analysis) analysis performed by AGG, with the ACTIGRA support for activities, one has the possibility to see how dependencies might cause problems in the activities of a complex system.

It is an advantage that the analysis is not different across the different modeling concerns, i.e, the base, the aspects, and also the woven system. In the small example presented, we can reveal simple dependencies between base and aspect by using the analysis. The tool also helps in making the example sound and complete by analyzing the base and the aspect separately for flaws. This is an often made observation that models become more sound as soon as a tool for executing or analyzing them is deployed, which is one reason for using tools.

Until now, there is no tool that supports integrated behavior models and transformation of these models on a meta level, which could be used, e.g., for specifying aspect weaving. Using such a tool would even allow to go beyond a set of predefined weaving operations since new activities could be added and tested by the experienced user. Moreover, no dedicated tools for aspect-oriented modeling on top of integrated behavior models exist either, allowing stereotypes and weaving as just mentioned.

In the example it could be studied, how causalities and conflicts established during the separate analysis for base and aspect changed after the weaving had been carried out. It is up to future work to generalize and formally show such effects.

The example was too small to reveal benefits of the modeling approach or the tooling such as major modeling mistakes like overlapping or missing domain concepts or functionality. Here, a more comprehensive case study would be useful. It also remains to implement the example in order to study whether the identified aspects persist in the code at all and whether the analysis has a positive effect on the quality of the code. To this end, empirical studies have to be carried out comparing implementation with and without this particular modeling approach and with and without the tooling support.

References

[1] Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases. Addison Wesley (2005)

[2] Araújo, J., Whittle, J., Kim, D.K.: Modeling and Composing Scenario-Based Requirements with Aspects. In: Proceedings of the 12th IEEE Int. Requirements Eng. Conf., CS-IEEE (2004)

[3] Rashid, A., Sawyer, P., Moreira, A., Araújo, J.: Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In: Proc. IEEE Joint International Conference on Requirements Engineering, IEEE Computer Society Press (2002) 199–202

[4] Rashid, A., Moreira, A., Araújo, J.: Modularisation and Composition of Aspectual Requirements. [29] 11–20

[5] Filman, R., Friedman, D.: Aspect-Oriented Programming is Quantification and Obliviousness. In: Proceedings of OOPSLA 2000 workshop on Advanced Separation of Concerns. (2000)

[6] Sillito, J., Dutchyn, C., Eisenberg, A., DeVolder, K.: Use case level pointcuts. In: Proc. ECOOP 2004, Oslo, Norway (2004)

[7] Katz, S.: Diagnosis of harmful aspects using regression verification. In Leavens, G.T., Lämmel, R., Clifton, C., eds.: Foundations of Aspect-Oriented Languages. (2004)

[8] Rinard, M., Sălcianu, A., Bugrara, S.: A Classification System and Analysis for Aspect-Oriented Programs. In: Proceedings of SIGSOFT'04/FSE-12, Newport Beach, CA, USA, ACM (2004) 147–158

[9] Zhao, J.: Slicing aspect-oriented software. In: Proceedings of the 10th IEEE International Workshop on Programming Comprehension. (2002) 251–260

[10] Balzarotti, D., Castaldo D'Ursi, A., Cavallaro, L., Monga, M.: Slicing AspectJ woven code. In: Proceedings of the Foundations of Aspect-Oriented Languages workshop (FOAL2005), Chicago, IL (USA) (2005)

[11] Object Management Group: UML Specification Version 2.0. Object Management Group (2005) http://www.omg.org.

[12] The Eclipse Foundation: AspectJ Homepage. http://www.eclipse.org/aspectj/ (2011)

[13] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ. In Knudsen, J., ed.: ECOOP 2001 — Object-Oriented Programming. Volume 2072 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2001) 327–354

[14] Ermel, C., Gall, J., Lambers, L., Taentzer, G.: Modeling with plausibility checking: Inspecting favorable and critical signs for consistency between control flow and functional behavior. Technical Report 2011/2, TU Berlin (2011)

[15] Ermel, C., Gall, J., Lambers, L., Taentzer, G.: Modeling with plausibility checking: Inspecting favorable and critical signs for consistency between control flow and functional behavior. In: Proc. Fundamental Aspects of Software Engineering (FASE'11). Volume 6603 of LNCS., Springer (2011) 156–170

[16] Technische Universität Berlin: AGG Homepage. http://tfs.cs.tu-berlin.de/agg (2007)

[17] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in TCS. Springer (2005)

[18] Hausmann, J., Heckel, R., Taentzer, G.: Detection of Conflicting Functional Requirements in a Use Case-Driven Approach. In: Proc. of Int. Conference on Software Engineering 2002, Orlando, USA (2002)

[19] Mehner, K., Monga, M., Taentzer, G.: Interaction Analysis in Aspect-Oriented Models. In: International Conference on Requirements Engineering RE 06. (2006)

[20] Mehner, K., Monga, M., Taentzer, G.: Analysis of Aspect-Oriented Model Weaving. In: Transactions on aspect-oriented software development V. Volume 5490 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2009) 235–263

[21] Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis Management Systems - A Case Study for Aspect-Oriented Modeling. Technical Report SOCS-TR-2009.3, McGill University (2009) Version 1.0.1.

[22] Stein, D., Hanenberg, S., Unland, R.: A UML-based Aspect-oriented Design Notation for AspectJ. [29] 106–112

[23] Jurack, S., Lambers, L., Mehner, K., Taentzer, G., Wierse, G.: Object Flow Definition for Refined Activity Diagrams. In Chechik, M., Wirsing, M., eds.: Proc. Fundamental Approaches to Software Engineering (FASE'09). Volume 5503 of Lecture Notes in Computer Science., Springer (2009) 49–63

[24] Sihman, M., Katz, S.: Superimpositions and aspect-oriented programming. The Computer Journal **46**(5) (2003) 529–541

[25] Katz, S.: Aspect categories and classes of temporal properties. In Rashid, A., Aksit, M., eds.: Transactions on Aspect-Oriented Software Development I. Volume 3880 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 106–134

[26] Nickel, U.A., Niere, J., Wadsack, J.P., Zündorf, A.: Roundtrip engineering with FUJABA. In: Proc of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany. (2000)

[27] Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. Electronic Notes in Theoretical Computer Science **127**(1) (2005) 65–75 Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004).

[28] Agrawal, A.: Graph Rewriting And Transformation (GReAT): A Solution For The Model Integrated Computing (MIC) Bottleneck. International Conference on Automated Software Engineering (2003) 364

[29] Ossher, H., Kiczales, G., eds.: AOSD '02. In Ossher, H., Kiczales, G., eds.: Proceedings of the 1st International Conference on Aspect-oriented Software Development, Enschede, The Netherlands, ACM (2002)