Interaction Analysis in Aspect-Oriented Models

Katharina Mehner[*]
Technical University of Berlin
Germany
mehner@cs.tu-berlin.de

Mattia Monga[†]
Università degli Studi di Milano
Italy
mattia.monga@unimi.it

Gabriele Taentzer[†]
Technical University of Berlin
Germany
gabi@cs.tu-berlin.de

Abstract

Aspect-oriented concepts are currently introduced in all phases of the software development life cycle. However, the complexity of interactions among different aspects and between aspects and base entities may reduce the value of aspect-oriented separation of cross-cutting concerns. Some interactions may be intended or may be emerging behavior, while others are the source of unexpected inconsistencies. It is therefore desirable to detect inconsistencies as early as possible, preferably at the modeling level.
We propose an approach for analyzing interactions and potential inconsistencies 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. The activities and their composition are formalized using the theory of graph transformation systems, which provides analysis support for detecting potential conflicts and dependencies between rule-based transformations. This theory is used to effectively reason about potential interactions and inconsistencies caused by aspect-oriented composition. The analysis is performed with the graph transformation tool AGG. The automatically analyzed conflicts and dependencies also serve as an additional view that helps in better understanding the potential behavior of the composed system.

1 Motivation

Aspect-oriented programming promises to provide better separation and integration of cross-cutting concerns than plain object-oriented programming. Aspect-oriented concepts are currently 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 effect 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 [27, 18, 3, 25, 24].

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 [13] of aspect-oriented composition that makes conquering interactions and inconsistencies difficult. When composing aspect-oriented and object-oriented models, there are two sources of *interactions*, and thus potential inconsistencies.

1. Intended or unintended overlap in concepts between these models can be the source of inconsistencies. Depending on the composition, these inconsistencies become effective or are avoided.

2. Aspect-oriented composition specifies where and when an aspect is applied and how the control flow is augmented or changed. It can be used to solve some of the above-mentioned inconsistencies, e.g. by replacing object-oriented behavior consistently with aspect-oriented behavior. However it might create inconsistencies by accidentally duplicating or suppressing behavior.

It is desirable to identify aspect interactions and potential inconsistencies as early as possible in the life cycle. Not all identified interactions are necessarily inconsistencies. Some of them may be intended or emerging collaborations. Until now, approaches to analyzing the aspectual composition of requirements have been informal [27, 25, 24]. Formal approaches for detecting inconsistencies have been proposed only for the programming level, e.g. model checking [20], static analysis [26], and slicing [32, 5].

[*]This author has been supported by the German Federal Ministry for Education and Research under the grant 01ISC04A (Project TOPPrax).
[†]These authors have been supported by the EU Research Training Network SegraVis.

The programming techniques cannot be used for requirements because they rely on the operational specification of the complete behavior as given by the code, while requirements abstract from these details. On the requirements level, a commonly used, but often informal, technique 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 use cases. 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.

In this paper, we investigate the use of an existing model analysis technique based on *graph transformations* [11] for analyzing the interactions and inconsistencies of an aspect-oriented composition of object and aspect models. The rule-based paradigm of graph transformation can be used as a formal model for behavior specifications with pre- and postconditions. The theory provides results for detecting interactions and potential inconsistencies among behavioral specifications.

We illustrate our approach with a *use case driven* modeling approach using UML [30] use cases, activity, and class diagrams. We specify aspect-oriented compositions for use cases using their refining activities as join points. Activities are rigorously defined with pre- and postconditions using a variant of UML and subsequently analyzed for conflicts and dependencies with the tool AGG [1], an environment for specifying, analyzing, simulating, and executing graph transformation systems. Since the graph transformation system is not aware of aspects, the results have to be interpreted according to the aspect-oriented composition specification.

The paper is organized as follows. In Sect. 2, we describe the formally enhanced use casen driven approach using an example and introduce the notion of conflicting and depending behavioral interactions. In Sect. 3, we introduce graph transformations and their analysis facilities. In Sect. 4, we apply the analysis to the example and interpret results with respect to aspect-oriented composition. In Sect. 5, we discuss related work. Sect. 6 contains our conclusion and outlook.

2 Aspect-Oriented Requirements Modeling

Several authors have proposed extending a use case driven requirements modeling approach with aspects [27, 18, 16, 21]. Aspects represent non-functional or functional cross-cutting requirements. In [2], functional aspects are identified at the level of use case relationships. The *join points*, i.e. the points of aspect-oriented composition, are activities or groups of activities, as in [27]. We present a subset of these techniques in order to demonstrate, how such approaches can be enhanced by (i) a formalization and (ii) a formal analysis.

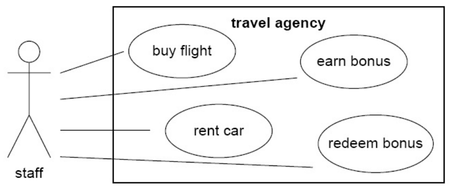


Figure 1: Use Cases in the Travel Agency Example

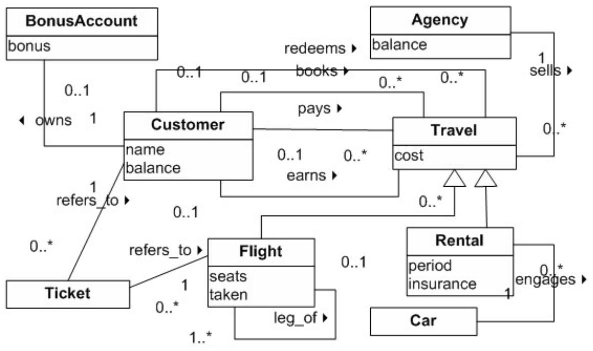


Figure 2: Domain Model Class Diagram

2.1 A Use Case Driven Approach

Central to our approach is the use case diagram, which serves as an overview. In addition, a use case is at least specified by a trigger, an actor, a pre-, a post-condition, main scenario(s), and exceptional scenario(s). Scenarios can be specified with UML activity diagrams. Here, we only present scenarios since they will be formalized. In addition, the domain model class diagram plays an important role because we refer to it in the formalization.

We illustrate the approach using a travel agency offering flights and car rentals and a bonus scheme.

2.1.1 Use Cases

For purchasing travel items, the system offers the use cases "buy flight" and "rent car" (cf. Fig. 1). The use cases "earn bonus" and "redeem bonus" offer a bonus program. A staff member is involved as an actor in all use cases but this does not imply that the actor always triggers the use case.

2.1.2 Domain Model Class Diagram

The class diagram specifies the structure of the domain (cf. Fig. 2). A *Customer* may book and pay for a *Travel* item, either a *Flight* or a *Rental*, sold by an *Agency*. Each travel item can be booked at most once. A *Flight* is composed of one or more legs, denoted by "leg_of". A *Ticket* "refers_to" a *Customer* and a *Flight*. A *Rental* "engages" one *Car*. A *Car* can be engaged in different *Rentals*. A *Customer* who "owns" a *BonusAccount* may earn and re-

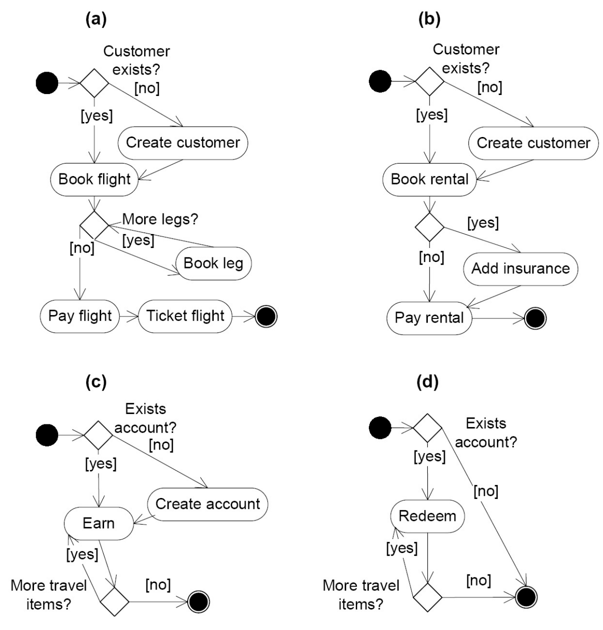


Figure 3: Activity Diagrams

deem bonus for *Travel* items.

2.1.3 Activities

The steps of the main scenario of each use case are described using activity diagrams. The use case "buy flight" is refined in Fig. 3(a). After conditionally creating a customer, the flight and all its legs are booked. Then the flight is paid for and a ticket issued. The use case "rent car" is specified analogously in Fig. 3(b). Bonus use cases are independent of the kind of travel. To earn a bonus, a bonus account must exist. A bonus is earned for all travel items (cf. Fig. 3(c)). For redeeming a travel item, one uses the bonus (cf. Fig. 3(d)).

2.1.4 Pre- and Post-conditions

The domain model can be more tightly integrated with activities by specifying the pre- and post-conditions of each activity using prototypical instances. An object diagram, i.e. the structural part of a UML collaboration diagram, lends itself naturally as a diagrammatic description of such a pre- or post-condition. This has also been advocated by object-oriented methods like Fusion [12] or Catalysis [10].

The pre- and postconditions specify arbitrary but fixed instances. A post-condition can refer to the *same instance* as the pre-condition by referring to the instance identifier, given as a number. *Attributes* can be matched with values. An attribute to be changed in a post-condition has to be instantiated in the pre-condition. *Creation* is specified by introducing a new instance or link in a post-condition. *Deletion* is specified by omitting an instance or

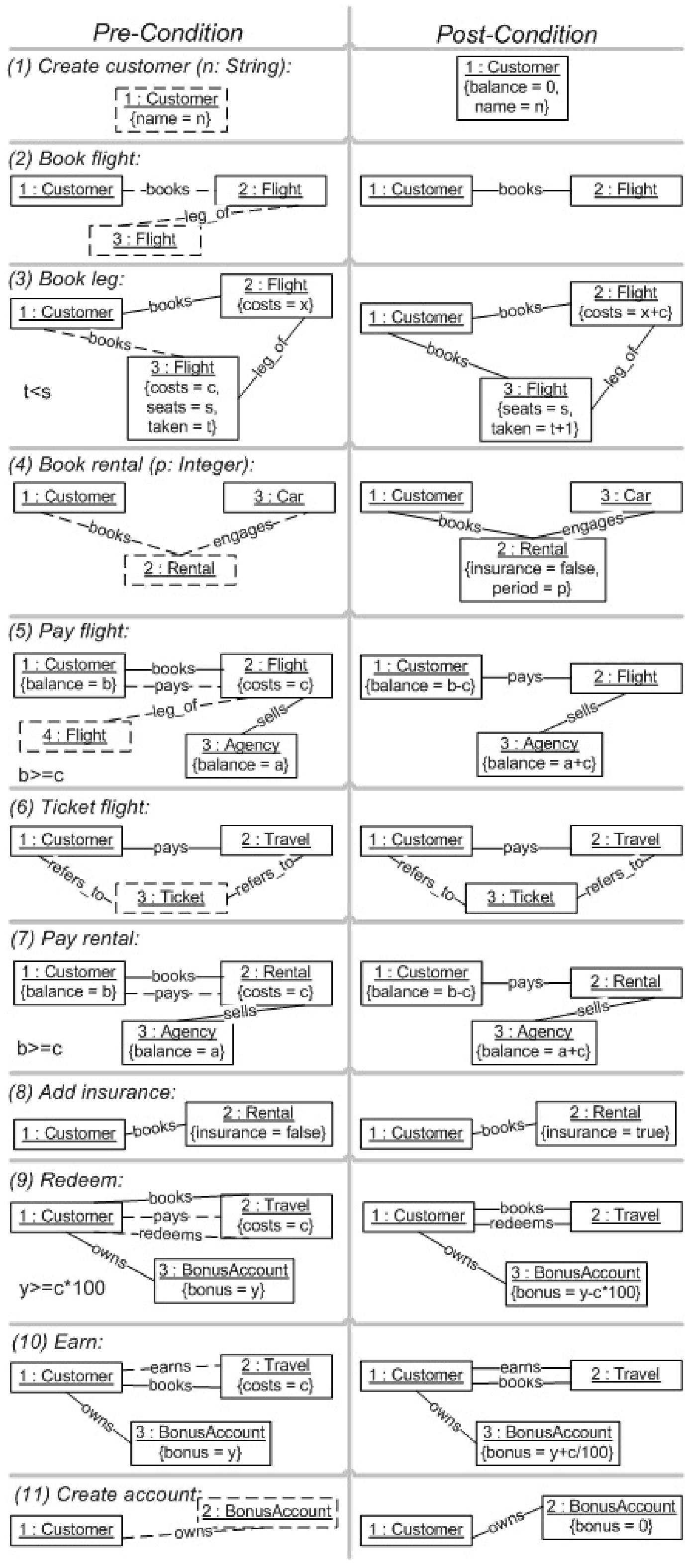


Figure 4: Activity Pre- and Post-Conditions

a link present in a pre-condition in the corresponding post-condition. Pre-conditions can include *negative* conditions, which are often used to specify that a created element does not yet exist. Negative links and instances are drawn using a dashed (out)line with a notation from graph transformations. Several negative elements have AND-semantics. (OR-semantics is possible, but it is not discussed here.)

Fig. 4 gives the pre- and post-conditions of all activities. In (1), the pre-condition checks that a customer named "n" does not exist. The post-condition ensures that this customer is created. In (2), neither "books" exists, nor is the flight a "leg of" another flight. A link "books" is inserted. In (3), attributes are matched with value parameters that are used to calculate the post-condition values. In the pre-condition, a constrained on the values "t<s" is used. The other pre- and post-conditions are constructed in a similar way. When specifying these conditions, the formal analysis of the tool AGG (cf. Sect. 3) already helped in identifying unintended imprecisions.

2.2 Aspect-Oriented Composition

Until now, we have left the specification of aspect-oriented relationships between use cases open. The notion of *aspect-oriented composition* is analogous to AspectJ [4]:

- An *advice* is modeled with a use case and subsequently specified using activity diagrams and pre- and post-conditions.
- The *pointcuts*, i.e. the matching specifications, refer to activities. Partial matching of names is possible. Each activity can thus be a *join point*.
- The modifiers `before`, `after` and `replace` indicate that the advice use case is executed before, after, or instead of each activity matched.

In practice, more complex pointcuts and more sophisticated matching mechanisms may be useful. Pointcuts are statically defined without dynamically evaluated conditions.

In the example of the travel agency use cases (cf. Fig. 1), cross-cutting behavior is exhibited by the use case "earn bonus". It augments the use cases "buy flight" and "rent car". Thus, the activity diagram specifying "earn bonus" is composed with the other activity diagrams. It should take place after booking is completed, i.e. before the following activity "Pay flight" or "Pay rental". Thus, the pointcut matches activities starting with "Pay" (cf. Tab. 1) using the modifier `before`. "Redeem bonus" should take place instead of "pay flight" or "pay rental"(cf. Tab. 1).

2.3 Interactions in Aspect-Oriented Composition

During aspect-oriented modeling, one needs to understand the effects of an aspect model on the model of the rest of the system, i.e. other aspect models and object models, but also how the aspect model is affected by them. The specified aspect-oriented composition should be feasible and should not violate other behavior constraints. This issue has been further analyzed by Katz [19], who distinguishes the following desirable properties of an aspect-oriented composition:

Use Case	Modifier	Pointcut (Activity)
earn bonus	`before`	Pay*
redeem bonus	`replace`	Pay*

Table 1: Aspect-Oriented Composition

- Specified properties of the existing system are preserved (apart from replaced behavior).
- The aspect adds desired new properties.
- Different aspect behaviors do not interfere.

These desirable properties are affected by the two sources of interactions we have already identified in the motivation, those directly between behavior and those that are established through the aspect-oriented composition. We can identify interactions based on the activities specified with pre- and post-conditions. We distinguish conflicts and dependencies.

An activity A2 is in *conflict* with an activity A1, if A2 cannot take place after activity A1 because the pre-conditions of A2 are violated by the post-conditions of A1. An activity A2 *depends* on an activity A1 if A1 produces something needed by the activity A2 or deletes something forbidden by A2. A conflict or dependency can arise between an activity from the object model and between an activity from the aspect model in both directions or between different aspect models.

In the following, we analyze all conflicts and dependencies which become effective with regard to aspect-oriented composition. Some of them may be tolerable or even needed is a function of the application domain, however our formal analysis helps in making them explicit.

Through the aspect-oriented composition two control flows are merged, and activities from different models become direct or indirect successors or predecessors of each other or replace each other. All conflicts and dependencies have to be taken into account in order to determine if the merge is successful, i.e. if the additional behavior is enabled and not prevented by conflicts and if it does not change the existing behavior.

We illustrate the typical scenario with the use case "redeem bonus" (cf. Fig. 1). The aspect-oriented composition (cf. Tab. 1) specifies that its activities (cf. Fig. 3) can replace an activity "Pay$*$" (cf. Fig. 3). To check that the composition can work, we have to compare the pre- and post-conditions of the activities involved in order to establish potential conflicts and dependencies between activities. In the example, one would try to find out whether "Redeem" can occur after "Book flight" and "Book rental". This is possible, because it depends on them.

Manually identifying all conflicts and dependencies from pre- and post-conditions is inefficient and error-prone. In the next section, we describe how the detection of conflicts and dependencies can be automated using existing technologies and tool support. We must therefore introduce the basics of graph transformation theory, which can be used to formalize pre- and post-conditions of behavioral models.

3 Graph Transformation

The UML variant presented in Sect. 2 is a modeling approach for requirements which can be precisely defined by the theory of graph transformation. While class structures are formalized by type graphs, pre- and post conditions of activities are mapped to graph rules. The formalization functions as the necessary basis for analyzing interactions in aspect-oriented composition in precisely. The calculus of graph transformation has a sound background dating back to the early seventies: the interested reader is referred to seminal work in this area [11]. In this paper, we present only as much theoretical background as is needed to understand our approach.

3.1 Attributed Typed Graph

Graphs can be used as an abstract representation of diagrams. A graph is defined by the sets of its vertices and edges as well as two functions, source and target, that map edges on vertices. According to this definition, more than one edge can exist between two given vertices. When formalizing object-oriented modeling, graphs occur at two levels: the type level (defined based on class diagrams) and the instance level (given by all valid object diagrams). This idea is described by the concept of *typed graphs*, where a fixed *type graph* TG serves as an abstract representation of the class diagram. Moreover, both vertices and edges may be decorated by a number of *attributes*, i.e. names with a value and type. As in object-oriented modeling, types can be structured by an inheritance relation. Instances of the type graph are object graphs equipped with a structure-preserving mapping to the type graph, i.e. a mapping that preserves the source and target functions for edges. A class diagram can thus be represented by a type graph, plus a set of constraints over this type graph, expressing multiplicities and perhaps further constraints.

In our running example, the type graph (cf. Fig. 5 (a)) represents the *domain model* of the system, equivalent to the UML class diagram in Fig. 2. However, the inheritance relationship was rendered by *flattening* all the associations of Travel to Flight and Rental. This is necessary because all the edges of a graph should have the same semantics (a relationship between two nodes) to be used consistently during the analysis. Fig. 5 (b) shows an instance graph compliant with the type graph.

3.2 Attributed Typed Graph Transformations

Basically, a *graph transformation* is a rule-based modification of a graph G into a graph H. 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, and a mapping between objects in L and R. 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.

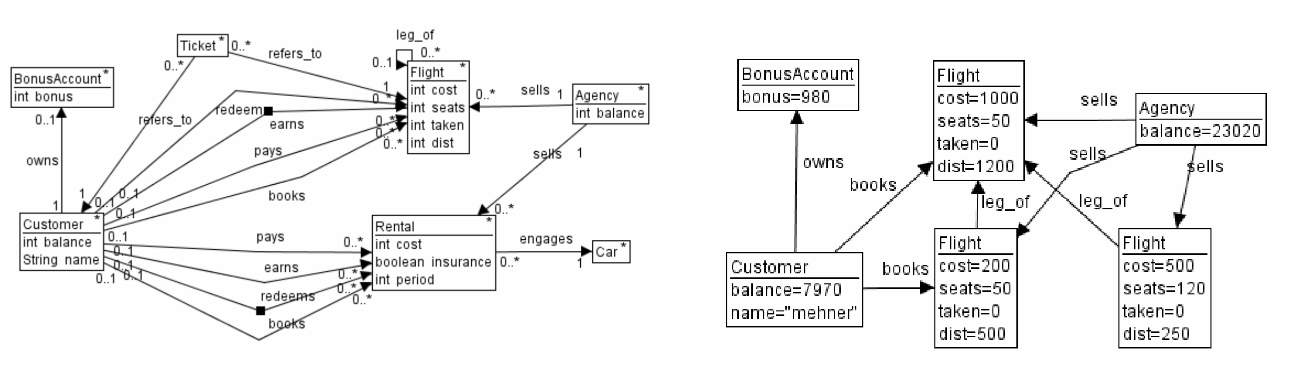
By way of an example, Fig. 4(3) shows pre- and post-conditions of the activity "Book leg", which can be interpreted as a graph rule. The numbers indicate the mapping between left- and right-hand sides. The attribute conditions are interpreted as an instantiation of variables on the left-hand side, and attribute assignment on the right-hand side.

A *graph transformation step* is defined by first finding a match m of the left-hand side L in the current instance graph G such that m is structure-preserving and type-compatible. If a vertex embedded into the context is to be deleted, edges that would not have a source or target vertex after rule application might occur: these are called *dangling edges*. There are mainly two ways to handle this problem: either the rule is not applied at match m, or it is applied and all dangling edges are also deleted. In the sequel we adopt the former strategy.

Performing a graph transformation step that applies rule r at match m, the resulting graph H is constructed in two passes: (1) build $D := G \setminus m(L \setminus (L \cap R))$, i.e., delete all graph items that are to be deleted; (2) $H := D \cup (R \setminus (L \cap R))$, i.e., create all graph items that are to be created. A *graph transformation*, more precisely a graph transformation sequence, consists of zero or more graph transformation steps.

The applicability of a rule can be further restricted by additional application conditions. The left-hand side of a rule formulates some kind of positive condition. In certain cases, *negative application conditions* (NACs), which are pre-conditions prohibiting certain graph parts, are also needed. If several NACs are formulated for one rule, each of them has to be fulfilled. See, for example, rule "Pay flight" in Fig. 4(5), which has two NACs, one forbidding the flight to be paid for to be a leg of another flight, and one checking if the flight to be paid for has already been paid for.

As an example of a graph transformation step, we consider again the rule "Book leg" in Fig. 4(3) and the host graph in Fig. 5(b). There are different ways of matching the rule to the host graph, depending on which flight leg is used. Choosing the left flight leg, the NAC (indicated by dashed lines in Fig. 4) is not fulfilled. Thus, the rule can only be



(a) (b)

Figure 5: Type Graph (a) and a Coherent Instance Graph (b) of the Travel Agency Example

applied to the right flight leg. The result is the host graph in Fig. 5(b) with an additional "books"-edge to the right flight leg.

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. There are techniques to restrict both kinds of choices. The choice of matches can be restricted by either letting the user specify part of the match using, for example, input parameters, or by explicitly defining a control flow over rule application.

The tool environment AGG (Attributed Graph Grammar System) [1] can be used to specify graph transformation systems and analyze their rules. Moreover, the rules can be tested by applying them to possible instance graphs.

3.3 Conflict and Dependency Analysis

One of the main static analysis facilities for GTSs is the check for conflicts and dependencies between rules and transformations, both supported in AGG. The dependency analysis and a critical pair visualization have been added recently, partly motivated by the aspect-oriented composition analysis presented in this paper. The conflict analysis has, for example, been applied to identify conflicts in functional requirements in [15]. In this paper, we argue that the existing theoretical results for graph transformation can be used for analyzing potential conflicts and dependencies in aspect-oriented modeling.

As discussed in the previous section, graph transformation systems can show certain kinds of non-determinism. Considering the case where two graph transformations can be applied to the same host graph, the result might be the same, regardless of the application order. Otherwise, if one of two alternative transformations is not independent of the second, the first will disable the second. In this case, the two rules are in *conflict.* Conversely, two transformations are said to be *parallel independent* if they modify different parts of the host graph. Instead, *sequential independence* guarantees that the order of application in a transformation sequence does not matter, i.e. performing the first transformation does not disable the second.

Analyzing the conflicts and dependencies of graph transformations can be compared to testing a program at run time. The analysis would have greater value if conflicts and dependencies could be detected during compile time, i.e. if it were e a static analysis. A promising approach in this direction is the analysis of potentially conflicting situations by *critical pairs*. A critical pair is a pair of transformation steps $G \overset{p_1,m_1}{\Longrightarrow} H_1$, $G \overset{p_2,m_2}{\Longrightarrow} H_2$ that are in conflict, and host graph G is constructed based on overlapping L_1 and L_2, the left-hand sides of rules p_1 and p_2. The set of critical pairs represents precisely all potential conflicts, i.e., there exists a critical pair as above if and only if p_1 may disable p_2 or, conversely, p_2 may disable p_1. Conflicts can be of the following types:

delete/use : The application of p_1 deletes a graph object that is used by the match of p_2.

produce/forbid : The application of p_1 produces a graph structure that a NAC of p_2 forbids.

change/use : The application of p_1 changes an attribute value of a graph object that is used by the match of p_2.

A delete/use conflict is shown, for example, in Fig. 6. Applying "Pay_flight" to the host graph shown at the bottom of the figure, rule "Redeem_flight" becomes non-applicable, because "Pay_flight" deletes the "books"-edge which is needed for the application of "Redeem_flight".

Another conflict occurs if a Customer has booked both a Flight and a Rental and wants to redeem loyalty points from her/his BonusAccount for both. The rules "Redeem_flight" and "Redeem_rental" change the same attribute "bonus".

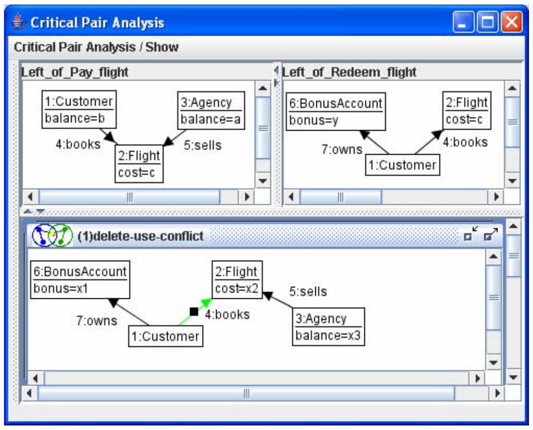
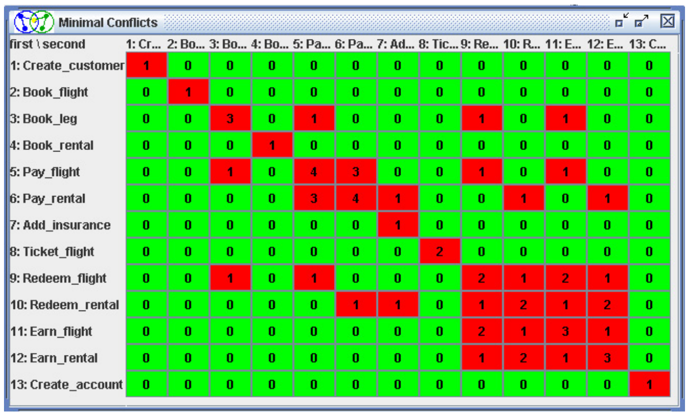


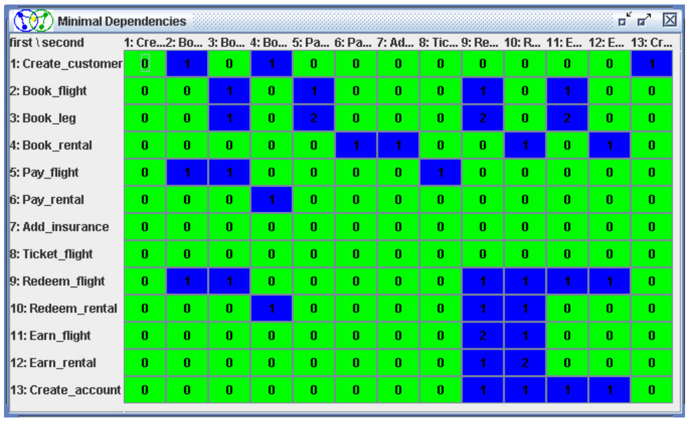
Figure 6: Critical Pair "Pay_flight", "Redeem_flight" in AGG



Minimal Conflicts

first \ second	1: Cr...	2: Bo...	3: Bo...	4: Bo...	5: Pa...	6: Pa...	7: Ad...	8: Tic...	9: Re...	10: R...	11: E...	12: E...	13: C...
1: Create_customer	1	0	0	0	0	0	0	0	0	0	0	0	0
2: Book_flight	0	1	0	0	0	0	0	0	0	0	0	0	0
3: Book_leg	0	0	3	0	1	0	0	0	1	0	1	0	0
4: Book_rental	0	0	0	1	0	0	0	0	0	0	0	0	0
5: Pay_flight	0	0	1	0	4	3	0	0	1	0	1	0	0
6: Pay_rental	0	0	0	0	3	4	1	0	0	1	0	1	0
7: Add_insurance	0	0	0	0	0	0	1	0	0	0	0	0	0
8: Ticket_flight	0	0	0	0	0	0	0	2	0	0	0	0	0
9: Redeem_flight	0	0	1	0	1	0	0	0	2	1	2	1	0
10: Redeem_rental	0	0	0	0	0	1	1	0	1	2	1	2	0
11: Earn_flight	0	0	0	0	0	0	0	0	2	1	3	1	0
12: Earn_rental	0	0	0	0	0	0	0	0	1	2	1	3	0
13: Create_account	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 7: Conflict Matrix in AGG



Minimal Dependencies

first \ second	1: Cre...	2: Bo...	3: Bo...	4: Bo...	5: Pa...	6: Pa...	7: Ad...	8: Tic...	9: Re...	10: R...	11: E...	12: E...	13: Cr...
1: Create_customer	0	1	0	1	0	0	0	0	0	0	0	0	1
2: Book_flight	0	0	1	0	1	0	0	0	1	0	1	0	0
3: Book_leg	0	0	1	0	2	0	0	0	2	0	2	0	0
4: Book_rental	0	0	0	0	0	1	1	0	0	1	0	1	0
5: Pay_flight	0	1	1	0	0	0	0	1	0	0	0	0	0
6: Pay_rental	0	0	0	1	0	0	0	0	0	0	0	0	0
7: Add_insurance	0	0	0	0	0	0	0	0	0	0	0	0	0
8: Ticket_flight	0	0	0	0	0	0	0	0	0	0	0	0	0
9: Redeem_flight	0	1	1	0	0	0	0	0	1	1	1	1	0
10: Redeem_rental	0	0	0	1	0	0	0	0	1	1	0	0	0
11: Earn_flight	0	0	0	0	0	0	0	0	2	1	0	0	0
12: Earn_rental	0	0	0	0	0	0	0	0	1	2	0	0	0
13: Create_account	0	0	0	0	0	0	0	0	1	1	1	1	0

Figure 8: Dependency Matrix in AGG

(See the pre- and post-conditions in Fig. 4(9) and imagine corresponding rules for the instantiation of "Travel" to "Flight" and to "Rental", respectively.)

Critical pair analysis can also be used to find all potential dependencies among rules. In fact, a rule p_1 may enable p_2 if and only if there exists a critical pair between p_1^{-1} and p_2. Consequently, the following dependencies are possible:

produce/use : The application of p_1 produces a graph object that is needed by the match of p_2.

delete/forbid : The application of p_1 deletes a graph objects that a NAC of p_2 forbids.

change/use : The application of p_1 changes an attribute of a graph object that is used by the match of p_2.

In the sequel, we use critical pair analysis to detect conflicts and dependencies among cross-cutting specifications.

4 Analysis of the Travel Agency Example

In the previous section, we introduced graph transformation as the theoretical foundation for detecting conflicts and dependencies between activities specified with pre- and post-conditions.

We computed all potential conflicts and dependencies for the travel agency example. The results are presented with a conflict (cf. Fig. 7) and a dependency (cf. Fig. 8) matrix. The first column and first row contain the list of all rules. The number specifies how many different conflicts/dependencies were found.

- Conflict matrix: a positive entry indicates that column entry A *disables* row entry B; B is in conflict with A.
- Dependency matrix: a positive entry means that column entry A *enables* row entry B; B is dependent on A.

In the conflict matrix, each activity is in at least one conflict with itself, which is typical for changes effected once. They can be ignored in the following. Many conflicts and dependencies which are caused by changes to attributes will not be considered in the following, since they do not cause any effect. The matrices can be used for validating the internal consistency of a single use case. For example, "Add_insurance" is disabled by "Pay_rental" but never occurs after it in use case "Buy flight".

In the sequel, we analyze conflicts and dependencies in the context of the aspect-oriented composition specification (cf. Tab. 1). We apply the following heuristics to check the aspect-oriented composition of two use cases for consistency, thus relationships with (activities from) other use cases are not considered. It is assumed that the composed activities work in the same context, i.e., the pre- and post-conditions overlap in the object graph. In general, a potential conflict might not lead to a concrete conflict.

- For `after`, the aspect activities must not be disabled by the direct predecessor activity, and they must not disable its direct successors.
- For `before`, the aspect activities must not disable the preceded activity and must not be disabled by its direct predecessors.

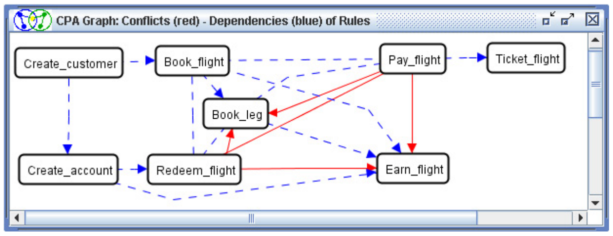


Figure 9: Graph of Critical Pairs in AGG

- For `replace`, the aspect activities must not be disabled by direct predecessors of the replaced activity and must not disable its direct successors. If the replaced activity enables successors, the aspect activities have to provide the same enablements.
- In all cases, the aspect activities may only be disabled by indirect predecessors if other indirect predecessors provide corresponding enablements after disablements. Similarly, aspect activities may disable indirect successors if these are enabled by other successors. Aspect activities do not have to be completely enabled by the use case with which they are composed.

Because of flattening the typing relation, we have to look at four concretely typed compositions. We describe results related to flights; rentals are similar. From the matrices, a *graph* can be generated (cf. Fig. 9), containing a directed edge B to A if B is in conflict (solid red) with or dependent (blue dashed) on A. Undirected edges represent mutual conflicts. In the graph, rules and critical pairs can be hidden, which we used to focus on the relevant conflicts and dependencies.

Before-Composition: Use case "earn_bonus" `before` "Pay$*$". This use case contains "Earn_flight" (flattened) and "Create_account". "Earn_flight" is disabled by "Pay_flight" but never occurs after it. None of the activities following "Book_flight" is disabled by "Earn_flight". "Earn_flight" is disabled by "Redeem_flight", which is however not part of the composed use cases. Such dependencies would have to be looked at in a greater context.

"Earn_flight" is completely enabled by either "Book_flight" or "Book_leg" and occurs after them. Thus, bonus is earned for the flight and each leg. This inconsistency is a kind of jumping aspect problem [6]. A negative condition can prevent redeem from being applied to a leg. "Create_account" is enabled by "Create_customer" that occurs before it.

Replace-Composition: Use case "redeem_bonus" `replaces` "Pay$*$". This use case contains "Redeem_flight" (flattened). "Pay_flight" disables "Redeem_flight" and vice versa. Using the graph, one can easily find activities depending on "Pay_flight". "Pay_flight" enables ticketing but "Redeeming_flight" does not. "Redeem_flight" states its post-condition in its own domain, i.e. it inserts an edge "redeems". To solve the problem, also an edge "pays" could be inserted.

"Redeem_flight" depends on "Book_flight" and "Book_leg". It is completely enabled by each of them because it requires only an edge "books". Thus, the bonus is paid for the flight and each leg, which is undesirable as discussed above. Also, "Redeem_flight" depends on "Create_account", an activity outside both use cases.

Applying the last heuristic can turn out to be a hard problem. The general question is whether the overall activity diagram resulting from the composition conforms with the overall conflict and dependency graph. In fact, conditional branching and cycles in activity diagrams (and in the graph) make it impossible to determine statically every enabling or disabling action. AGG provides formal support to our approximate solution of this hard problem.

5 Related Work

Conflict analysis based on graph transformation has been applied in several contexts in software engineering. The detection of conflicts in functional requirement specifications was investigated in [15]. In this paper, we considered requirement specifications developed with the use case-driven approach. The motivation of this work was the early detection of conflicts within the software engineering process. Another application in this area is the detection of conflicts and dependencies in software evolution, more precisely between several software refactorings [22]. Both investigations use the critical pair analysis of AGG for detection.

A clustering of individual requirements within the specification of the behavioral characteristics of a system is often called a *feature* [28]. The notion of feature, while natural in the "problem domain", is not always present in the "solution domain". In fact, researchers in *feature engineering* propose promoting features as "first-class objects" within the software process in order to bridge the gap between the user needs and design or implementation abstractions. However, in general features are not independent of each other nor necessarily consistent. Finkelstein et al. [14] proposed a framework for tracking relationships among different *viewpoints* of a system, according to the goals pursued by the different stake-holders involved in the system development. By contrast, our analytical approach aims to detect inconsistencies and interactions as early as possible in order to avoid them.

In [3], Araújo et al. describe non-aspectual requirements as sequence diagrams and aspectual requirements with interaction pattern specifications, which are both woven together in state machines that can be simulated. No support for static conflict detection is provided.

Nakajima and Tamai [23] use Alloy [17] to analyze interactions among role models, taking into account object-oriented refinement and aspect-oriented weaving.

Several researchers are working to find interactions at the programming level, normally in AspectJ code. Specific program analysis techniques for AspectJ programs were proposed [32, 5] in order to determine whether two aspects interfere. Clifton and Leavens [7] propose classifying aspects in *observers,* which do not change the system behavior, and *assistants,* which participate actively in the global computation. Similarly, Katz [20] proposed using data-flow analysis to identify *spectative, regulative,* and *invasive* aspects. These techniques can be used to automatically extract models of the code, which can be used to verify that expected properties of the system hold [26, 31, 29, 8]. Douence et al. [9] introduced a generic framework for aspect-oriented programming supporting pointcuts with explicit states, and they provided an abstract formal semantics of their aspect language: this allows the detection of aspect interference.

6 Conclusion

A key problem of aspect-oriented composition is the use of quantification, which makes it more difficult to reason about than in purely object-oriented models. In this paper, we present an approach for detecting conflicts and dependencies in behavioral specifications of use cases refined with activity diagrams. We use pre- and post-conditions for activities to make the modeling more rigorous. Specifying pre- and post-conditions may require an additional effort but pays off if an early formal analysis is required. The number of activities used to refine use cases highly depends on the process and the context in which the software is developed. However, the analysis of pre- and post-conditions is not restricted to activities, but can also be applied to methods and to a wide range of aspect-oriented modeling techniques if they are enhanced by pre- and post-conditions, which are a universally applicable technique.

In our approach, we use graph transformation as a formal technique to give the chosen UML variant a formal semantics and to analyze it rigorously. Detected conflicts and dependencies are potential ones that might occur in the system but do not have to. Nevertheless, the formal technique helps to make the problems explicit. It directs the developer to the problematic parts of a model. It helps in understanding aspect-oriented compositions and it helps in reasoning effectively about the cross-cutting. Graph transformation also allows us reason uniformly about object and aspect models.

Support for analysis of the conflict and dependency graph is definitely needed in order to apply the ideas to larger examples. The analysis of conflicts and dependencies can be carried out with the tool AGG, a tool for specifying and analyzing rule-based transformations of typed attributed graphs. Since the analysis functions are provided with a Java API, AGG is ideal for use with existing UML CASE tools. Furthermore, incremental analysis would be desirable and feasible in such a setting.

The presented approach can be applied in two ways. It can be used to validate an aspect-oriented design by comparing operators with conflicts as demonstrated in this paper. It can also be used to propose aspect-oriented compositions by deriving them from conflicts and dependencies.

The approach presented is not restricted to functional aspects only. So-called non-functional aspects can also be mapped to functional specifications in terms of pre- and post-conditions on the state of a system. This also enables interactions between functional and non-functional aspects to be automatically analyzed.

References

[1] AGG Homepage. `http://tfs.cs.tu-berlin.de/agg`.

[2] J. Araújo and P. Coutinho. Identifying aspectual use cases using a viewpoint-oriented requirements method. In *Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design*, Boston, MA, USA, March 2003.

[3] J. Araújo, J. Whittle, and D.-K. Kim. Modeling and composing scenario-based requirements with aspects. In *Proceedings of the 12th IEEE Int. Requirements Eng. Conf.* CS-IEEE, 2004.

[4] AspectJ Homepage". `http://www.aspectj.org`.

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

[6] J. Brichau, W. De Meuter, and K. De Volder. Jumping aspects. In *Workshop "Aspects and Dimensions of Concerns", ECOOP 2000*, Sophia Antipolis and Cannes, France, June 2000.

[7] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy,. Technical Report TR03-01a, Iowa State University, January 2003. presented at SPLAT 2003.

[8] G. Denaro and M. Monga. An experience on verification of aspect properties. In T. Tamai, M. Aoyama, and K. Bennett, editors, *Proceedings of the International Workshop on Principles of Software Evolution IWPSE 2001*, pages 184–188, Vienna, Austria, September 2001. ACM.

[9] R. Douence, P. Fradet, and M. Südholt. Composition, reuse, and interaction analisys of stateful aspects. In *Proceedings of the 3rd international Conference*

of Aspect-oriented Software Development, Lancaster, UK, March 2004. ACM.

[10] D. D'Souza and A. Wills. *Components and Frameworks with UML: The Catalysis Approach.* Addison Wesley, 1998.

[11] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in TCS. Springer, 2005.

[12] D. Coleman et al. *Object Oriented Development, The Fusion Method.* Prentice Hall, 1994.

[13] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of OOPSLA 2000 workshop on Advanced Separation of Concerns*, 2000.

[14] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in systems development. *International Journal of Software Engineering and Knowledge Engineering*, 1(2):31–58, 1992.

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

[16] S. Herrmann, C. Hundt, and K. Mehner. Mapping Use Case Level Aspects to Object Teams/Java. In A. Moreira et. al, editor, *OOPSLA Workshop on Early Aspects*, 2004.

[17] D. Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[18] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison Wesley, 2005.

[19] S. Katz. A Survey of Verification and Static Analysis for Aspects (AOSD-Europe Network of Excellence. `http://www.aosd-europe.net`.

[20] S. Katz. Diagnosis of harmful aspects using regression verification. In G. T. Leavens, R. Lämmel, and C. Clifton, editors, *Foundations of Aspect-Oriented Languages*, March 2004.

[21] K. Mehner and G. Taentzer. Supporting Aspect-Oriented Modeling with Graph Transformations. In P. Clements et al., editor, *AOSD 05 Workshop on Early Aspects*, 2005.

[22] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts unsing Critical Pair Analysis. In R. Heckel and T. Mens, editors, *Proc. Workshop on Software Evolution through Transformations (SETra'04), Satellite Event of ICGT'04*, ENTCS, Rome, Italy, October 2004. Elsevier.

[23] S. Nakajima and T. Tamai. Lightweight formal analysis of aspect-oriented models. In *UML2004 Workshop on Aspect-Oriented Modeling*, 2004.

[24] A. Rashid, A. Moreira, and J. Araujo. Modularisation and composition of aspectual requirements. In *Proc. AOSD'02*, pages 11–20, Enschede, Netherlands, 2002. ACM Press.

[25] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. Early aspects: A model for aspect-oriented requirements engineering. In *Proc. IEEE Joint International Conference on Requirements Engineering*, pages 199–202. IEEE Computer Society Press, 2002.

[26] M. Rinard, A. Sălcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of SIGSOFT'04/FSE-12*, pages 147–158, Newport Beach, CA, USA, 2004. ACM.

[27] J. Sillito, C. Dutchyn, A. Eisenberg, and K. DeVolder. Use case level pointcuts. In *Proc. ECOOP 2004*, Oslo, Norway, June 2004.

[28] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *The Journal of Systems and Software*, 49(1):3–15, December 1999.

[29] N. Ubayashi and T. Tamai. Aspect-oriented programming with model checking. In *AOSD 2002 (1st International Conference on Aspect-Oriented Software Development) Conference Proceedings*, pages 148–154, Enschede, NL, April 2002.

[30] UML Specification Version 1.5 (formal/03-03-01). Object Management Group. `http://www.omg.org`.

[31] W. Xu and D. Xu. A model-based approach to testing interactions between aspects and classes. In *AOSD'05 Workshop on Testing Aspect-Oriented Programs*, Chicago, 2005.

[32] J. Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th IEEE International Workshop on Programming Comprehension*, pages 251–260, June 2002.