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. Thus, it is 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, which are used to refine use cases, are the join points to compose cross-cutting concerns. The activities and their composition are formalized by using the theory of graph transformation systems, which provides analysis support for the detection of 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 crosscutting concerns than plain object-oriented programming. Aspect-oriented concepts are currently introduced in all phases of the software development life cycle with the expectation to reduce complexity and to enhance maintainabilty already already at early stages.

On the requirements level, crosscutting 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 crosscutting 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 [26, 17, 2, 24, 23].

A key challenge is to analyze the interaction and consistency of crosscutting concerns with each other and with affected requirements. Especially the quantifying nature [12] of aspect-oriented composition makes conquering interactions and inconsistencies difficult. When composing aspect-oriented and object-oriented models, there are two sources of *interactions* and thus potential inconsistencies.

- 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.
- 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 analyse aspectual composition of requirements are still informal [26, 24, 23]. Formal approaches for detecting inconsistencies have been proposed only for the programming level such as model checking [19], static analysis [25], and slicing [32, 4].

The programming techniques cannot be used for require-

[*]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.

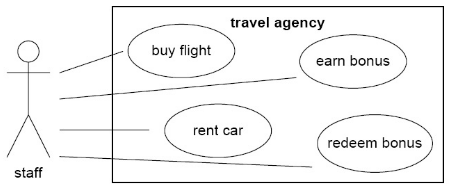


Figure 1: Use cases of travel agency example

ments 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, yet often informal, technique is to describe behavior with pre- and post-conditions, e.g. using intensionally defined states or attributes of a domain entity model. This technique is, e.g. used for defining UML use cases. In order to enable a more rigorous analysis of behavior, this approach has to be formalized and has to be extended also to aspect-oriented units of behavior.

In this paper, we investigate the use of an existing model analysis technique based on *graph transformations* [10] 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 [29] use cases, activity, and class diagrams. We specify aspect-oriented compositions for use cases by using their refining activities as join points. Activities will be 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. As no graph transformation system is aware of aspects, the results have to be interpreted according to the aspect-oriented composition specification.

The paper is structured as follows. In Sect. 2 we describe the formally enhanced use casen driven approach by 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. In Sect. 6 we conclude and give an outlook.

2 Aspect-Oriented Requirements Modeling

Several authors have proposed to extend a use-case driven requirements modeling approach with aspects [26, 17, 15, 20]. Aspects represent non-functional or functional

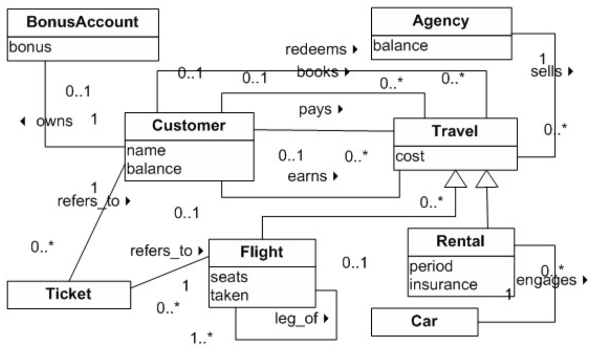


Figure 2: Domain model class diagram

crosscutting requirements. In [3], functional aspects are identified at the level of use case relationships. The *join points*, i.e., the places of aspect-oriented composition, are activities or groups thereof as in [26]. We present a subset of these techniques with the intention to demonstrate, how such approaches can be augmented by (i) a formalization and (ii) a formal analysis.

2.1 A Use Case driven Approach

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

We illustrate the approach with a travel agency software offering flights and car rentals for which bonus subscription is available.

2.1.1 Use Cases

For purchasing travels, the system offers the use cases “buy flight” and “rent car” (see Figure 1). Through the use cases “earn bonus” and “redeem bonus” a bonus program is offered. A staff is involved as 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. A *Customer* may book and pay a *Travel*, either a *Flight* or a *Rental*. Each 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 redeem bonus for *Travels*.

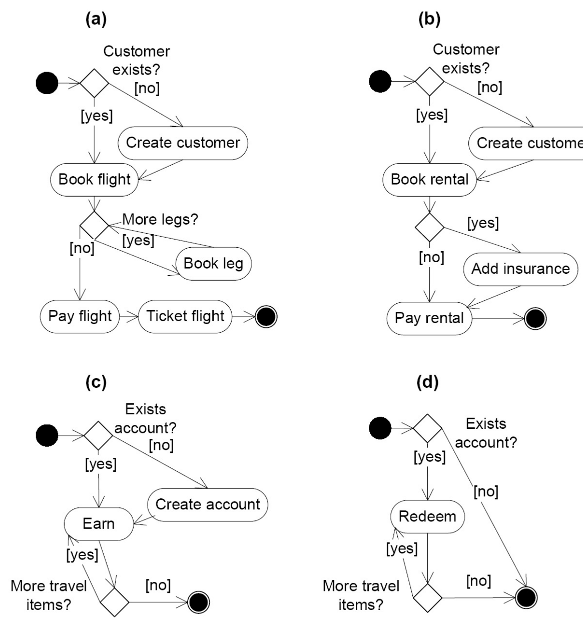


Figure 3: Activity Diagrams

2.1.3 Activities

The steps of a use case are described with 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 payed and a ticket produced. Use case “rent car” is specified analogously in Fig. 3(b). Bonus use cases are independent of the kind of travel. To earn bonus, a bonus account has to exist. Bonus is earned for all items of a travel (cf. Fig. 3(c)). One has to use the bonus for all travel items when redeeming (cf. Fig. 3(d)).

2.1.4 Pre- and Post-conditions

The domain model can be integrated with activities more tightly by specifying the pre- and post-conditions of each activity by 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 [11] or Catalysis [9].

The pre- and postconditions specify arbitrary but fixed instances. A post-condition can refer to the *same instances* as the pre-condition by refering 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. *Deletion* is specified by omitting an instance or a link present in a pre-condition in the corresponding post-condition. Pre-conditions can include *negative* conditions, e.g., that a resulting instance does not exist. Negative links and instances are drawn using a dashed (out)line using a notation from graph transforma-

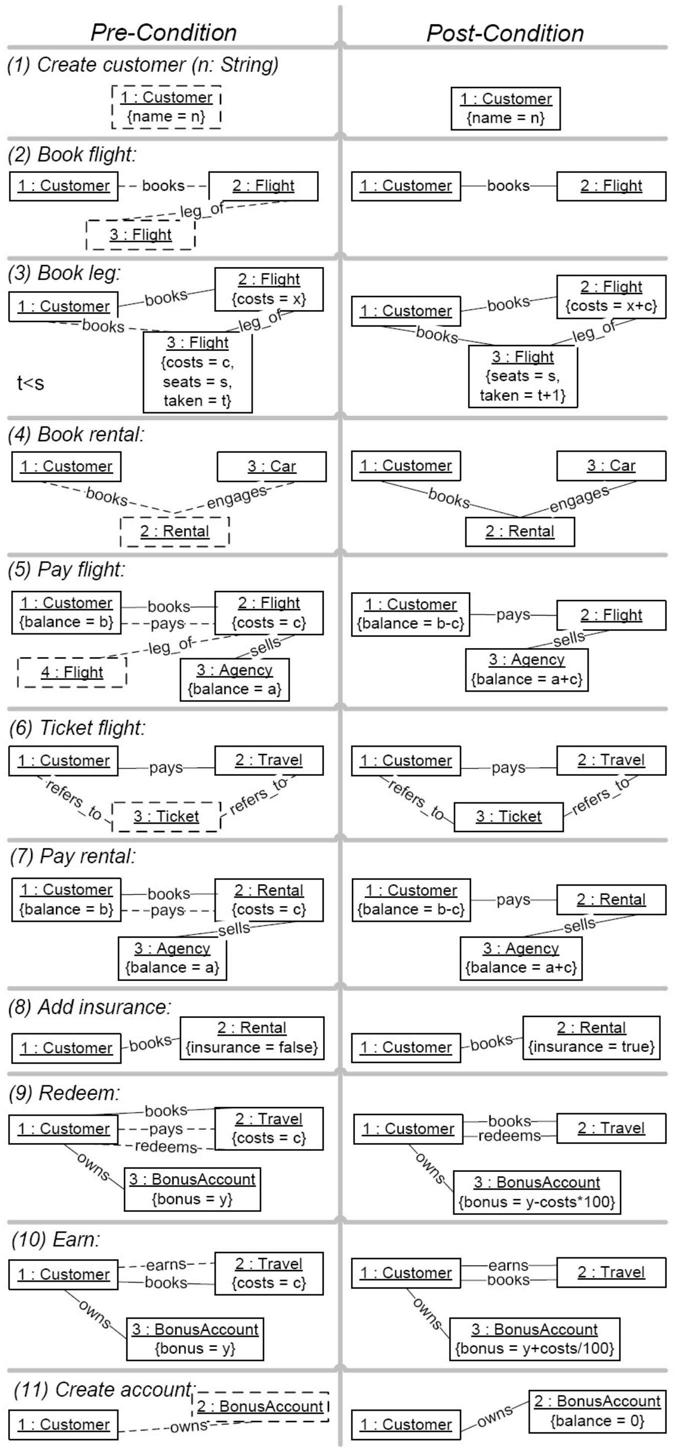


Figure 4: Activity Pre- and Post-Conditions

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

Table 1: Aspect-oriented Composition

tions. Several negative elements have and AND-semantics. (OR-semantics is possible but it will not be discussed here.)

Fig. 4 gives pre- and post-conditions of all activities. In (1), the pre-condition checks that a customer with the parameter name 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 identified with value parameters that are used to calculate the post-condition values. In the pre-condition, a logical condition on the values "t<s" is used. The other pre- and post-conditions are constructed in a similar way.

2.2 Aspect-Oriented Composition

Until now, we left the specification of aspect-oriented relationships between use cases open. The notion of *aspect-oriented composition* is in analogy to AspectJ [30]:

- An *advice* is modeled with a use case, subsequently specified trough 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 (see Fig. 1), crosscutting 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 complex booking behavior is completed, i.e., before starting the following activity "pay flight" or "pay rental". Thus, the pointcut matches activities starting with "pay" (cf. Table 1) using the modifier `before`. "Redeem bonus" should take place instead of "pay flight" or "pay rental"(see Table 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 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 [18] who distinguishes the following desirable properties of an 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, the ones directly between behavior and the ones which 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 preconditions of A2 are violated by the post-conditions of A1. An activity A2 is *depending* 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.

Which conflicts and dependencies can become effective can be analyzed with regard to the aspect-oriented composition. Which conflicts and dependencies are tolerable or needed, is determined by the application domain.

Through the 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 if it is 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. Table 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 conflict and dependencies between activities. In the example, one would try to find out whether "redeem bonus" cannot take place after any activity that is occurring in the control flow before, e.g., "pay_flight". This is not the case, as "redeem bonus" is depending on "book_flight" and "book_rental" and also "pay_flight" is depending on them.

Identifying all conflicts and dependencies from pre- and post-conditions manually is inefficient and error prone. In the next section we will describe, how the detection of conflicts and dependencies can be automated using existing technologies. Therefore, we have to introduce the basics of graph transformation theory, which can be used to formalize pre- and post-conditions of behavioral models. The detection of conflicts and dependencies can be automated by using proper supporting tools.

3 Graph Transformation

The UML variant presented in Section 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 to analyze interactions in aspect-oriented composition in a precise manner. The calculus of graph transformation has a solid background which dates back to the early seventies: the interested reader is referred to seminal work in this area [10]. In this paper we present only as much theoretical background as 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. Formalizing object-oriented modeling, graphs occur at two levels: the type level (defined on the basis of 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 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 maybe further constraints.

In our running example the type graph (see Figure 5 (a)) represents the *domain model* of the system, equivalent to the UML class diagram in Figure 2. However, the inheritance relationship was rendered by *flattening* all the associations of Travel to Flight and Rental: this is needed because all the edges of a graph should have the same semantics (a relationship between two nodes) to be used consistently during the analyses. Figure 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 in 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 which is not changed) and the union $L \cup R$ should form a graph again, i.e., they have to be compatible with source, target and type settings, in order to apply the rule. Graph $L \setminus (L \cap R)$ defines the part which shall be deleted, and graph $R \setminus (L \cap R)$ defines the part to be created.

As one example, Figure 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 instantiation of variables on the left-hand side, and attribute assignment on the right-hand side.

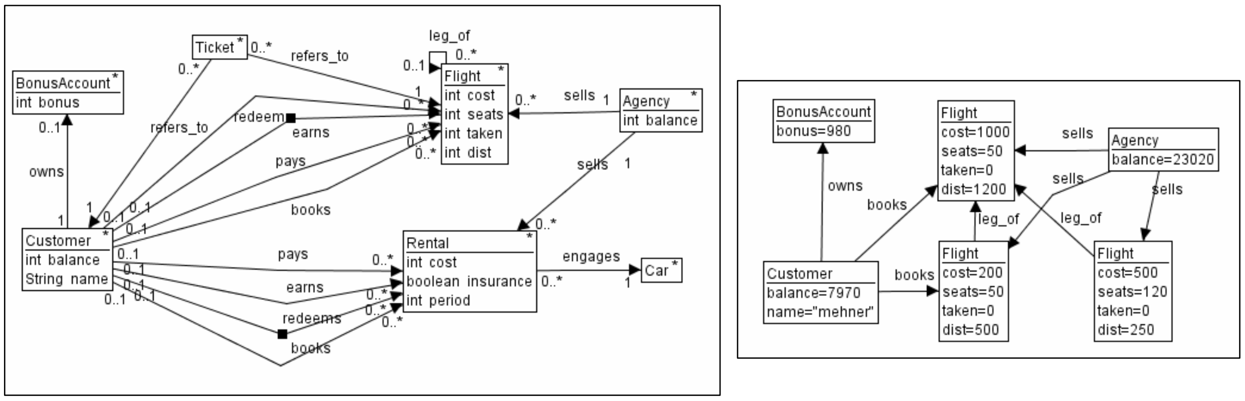
A *graph transformation step* is defined by first finding a match m of the left-hand side L in the current instance graph G such that m is structure-preserving and type compatible. If a vertex embedded into the context shall be deleted, edges which would not have a source or target vertex after rule 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 following we shall adopt the former strategy.

Performing a graph transformation step which 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 which shall be deleted; (2) $H := D \cup (R \setminus (L \cap R))$, i.e. create all graph items which shall 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 also *negative application conditions* (NACs) which are pre-conditions prohibiting certain graph parts, are needed. If several NACs are formulated for one rule, each of them has to be fulfilled. See e.g. rule ”Pay flight” in Fig. 4 which has two NACs, one which forbids the flight to be paid to be a leg of another flight, and one which checks if the flight to be paid is already paid.

As an example for a graph transformation step, we consider again rule ”Book leg” in Fig. 4(3) and the host graph in Fig. 5(b). There are different possibilities to match the rule to the host graph, dependent on which leg flight is used. But choosing the left leg flight, the NAC (indicated by dashed lines in Fig. 4) is not fulfilled. Thus, the rule can only be applied to the right leg flight. The result is the host graph in Fig. 5 with an additional ”books”-edge to the right leg flight.

A set of graph rules, together with a type graph, is called



(a) (b)

Figure 5: Type graph (a) and a coherent instance graph (b) of the travel agency example

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

The choice of matches can be restricted by either letting the user specify part of the match using e.g. 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 analyse its 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. AGG supports the check for conflicts between rule applications. It has been applied to e.g. identify conflicts in functional requirements in [14]. In this paper we argue that the existing theoretical results for graph transformation can advantageously be used for analyzing potential conflicts and dependencies in aspect-oriented modelling.

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, independent of the application order. Otherwise, if one of two alternative transformations is not independent of the second, the first one will disable the second. In this case, the two rules are in *conflict.* Vice versa, 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 one.

Analyzing the conflicts and dependencies of graph transformations can be compared with testing a program at run time. The analysis would have more value, if conflicts and dependencies could be discovered during compile time, i.e. if it would be 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 $G \overset{p_1,m_1}{\Longrightarrow} H_1$, $G \overset{p_2,m_2}{\Longrightarrow} H_2$ which 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, that is, there exists a critical pair like above if, and only if, p_1 may disable p_2 or, vice versa, p_2 may disable p_1. Conflicts can be of the following types:

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

A delete/use-conflict is shown, for example, in Figure 6. Applying "Pay_flight" to the host graph shown at the bottom of the figure, rule "Redeem_flight" becomes non-applicable, since the application of "Pay_flight" delete 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 if s/he wants to redeem loyalty points from her/his BonusAccount for both, the "Redeem_flight" and "Redeem_rental" rules change the same

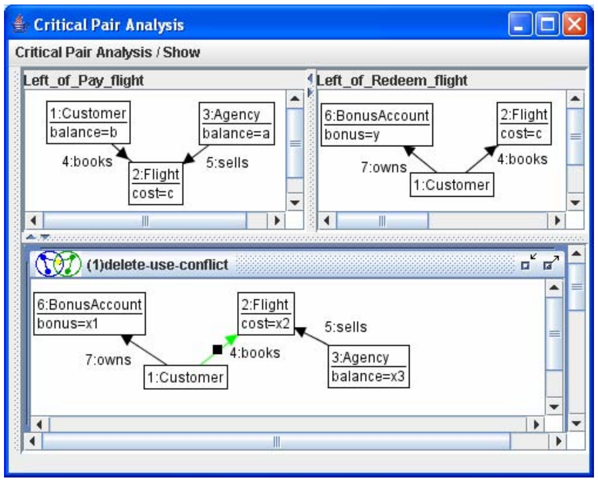


Figure 6: Critical pair "Pay_flight", "Redeem_flight"

attribute "bonus". (See the pre- and post-conditions in Figure 4(8) 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 which 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 which is used also by the match of p_2.

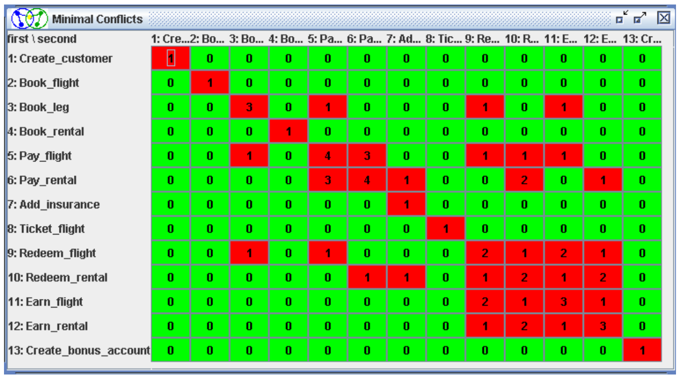
In the following we use critical pair analysis to detect conflicts and dependencies among cross-cutting specifications.

4 Analysis of Travel Agency Example

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

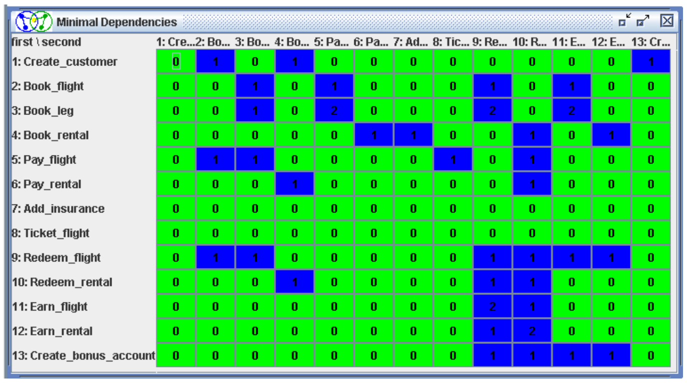
We computed all potential conflicts and dependencies for travel agency example. The results are presented with a conflict (see Fig. 7) and a dependency (see Fig. 8) matrix. The first column and first row contain the list of all activities. The number specifies how many different conflicts/dependencies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


Minimal Conflicts

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	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	1	1	0	0
6: Pay_rental	0	0	0	0	3	4	1	0	0	2	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	1	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_bonus_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	1	0	0	0
6: Pay_rental	0	0	0	1	0	0	0	0	0	1	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_bonus_account	0	0	0	0	0	0	0	0	1	1	1	1	0

Figure 8: Dependency Matrix in AGG

From the matrices, a *graph* is generated (see Fig. 9), containing a directed edge B to A if B is conflict(red) with or dependent(blue) on A. The graph captures *chains*.

In the conflict matrix, each activity is in at least one conflict with itself, which is typical for changes effected once. These conflict are not of interest for our further analysis. Most of the depicted conflicts and dependencies are caused by attributes. They point to problems, when a pre-condition requires a special value as in"Book_leg", and can be ignored otherwise.

In the following, we discuss selected conflicts and dependencies in the context of the aspect-oriented composition specification. We compare the composed control flow (cf. Table 1) with the conflicts/dependencies of the composed activities. Internal validation of activity diagrams is not our focus here. Because of the flattening, we have to look at four compositions instead of two. We describe results related to flights; rentals are similar.

Composition: Use case "Earn_bonus" `before` "Pay$*$". This use case contains activities "Earn_flight" (flattened) and "Create_account". For each enablement or disablement, one has to decide whether it is desirable and whether it has an effect when also taking into account the control flow.

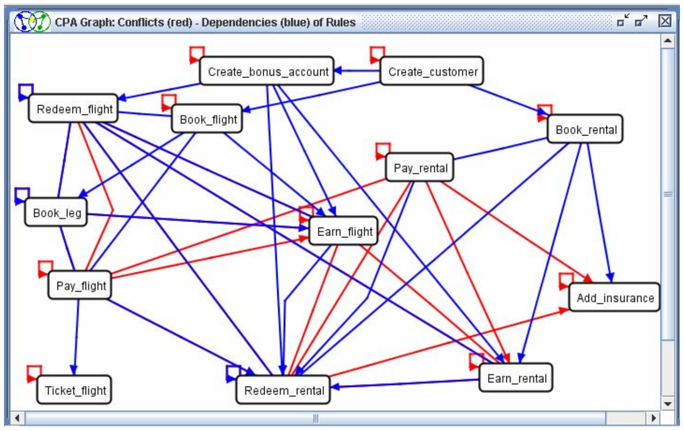


Figure 9: Graph of Critical Pairs in AGG

Here, we can give only some example.

"Earn_flight" is dependent on "Book_flight" but taking place after it. One has to look at the kind of dependency to identify whether "Earn_flight" is completely enabled by "Book_flight". "Book_flight" inserts an edge "books" on which 'Earn_flight" depends. "Earn_flight" is also dependent on "Book_leg". The bonus is earned for the flight and each leg. This inconsistency is a kind of jumping aspect problem [5]. The legs composed by a flight should not enable aspects. A negative condition can prevent that redeem is applied to a leg. All attribute dependencies can be ignored.

"Earn_flight" is disabled by "Pay_flight". As this is an activity occuring in the control flow after the composition, this is no problem. None of the activities following "Book_flight" is disabled by "Earn_flight".

Composition: Use case "Redeem_bonus" `replaces` "Pay$*$". This use case contains the activity "Redeem_flight" (flattened). "Redeem_flight" is depending on "Book_flight" and "Book_leg". It is completely enabled by each of them because it requires only an edge "books". Thus, bonus is payed for the flight and each leg which is undesirable as above.

"Pay_flight" disables "Redeem_flight" and vice versa. One has to analyze via the chains which activities depending on "Pay_flight". "Pay_flight" enables ticketing but "Redeeming_flight" does not. This is not desirable because a ticket should be printed in both cases. "Redeem_flight" does state its post-condition in its own domain, i.e., it inserts an edge "redeems". To solve the problem, an edge "pays" could be inserted, because it does not make sense to change all activities depending on "Pay_flight".

In the best practice demonstrated with the examples, we can identify generalizable heuristics: (1) For `after`, the aspect activities must not be disabled by the affected activity and they must not disable its direct successors. (2) For `before`, the aspect activities must not disable the affected activity and must not be disabled by its direct predecessors. (3) For `replace`, the aspect activities must not be disabled by direct predecessors of the replaced activity and must not disable its direct successors.

It is difficult to generalize over the required and forbidden conflicts and dependencies for activities that are not immediate predecessors and successors of the affected activity. The general question is, whether the overall activity diagram resulting from the composition is conform with the overall conflict and dependency graph. Because of potential cycles in both, activity diagrams and in the graph, and because of conditional branching in activity diagrams, this problem can only be solved in an approximative way. Thus, in its presented state, the use of AGG provides a formal aid in supporting the solving of a hard problem.

5 Related Work

Conflict analysis based on graph transformation has been applied in several contexts within the software engineering area. The detection of conflicts in functional requirement specifications was investigated in [14]. In this work, 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 precisley between several software refactorings [21]. Both investigations are based on graph transformation and use the critical pair analysis of AGG for detection. In addition, we discuss for possible conflicts between refactorings how they can be resolved.

A clustering of individual requirements within the specification of the behavioural characteristics of a system is often called *feature* [27]. The notion of feature, while natural in the "problem domain", it is not always present in the "solution domain". In fact, researchers in *feature engineering* propose to promote 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 one from each other nor necessarily consistent. Finkelstein et. al [13] proposed a framework for tracking relationships among different *viewpoints* of a system, according to the goals that the different stake-holders involved in the development of the system are pursuing. Our analytical approach, instead, is aimed at discovering inconsistencies and interactions as early as possible, in order to avoid them.

In [2], Araújo et al. describe non-aspectual requirements as sequence diagrams and aspectual requirements with interaction pattern specifications, then they are both woven together in state machines that can be simulated. No support for static conflict detection is provided.

Nakajima and Tamai [22] use Alloy [16] to analyze interactions among role models, by taking into account object-oriented refinement and aspect-oriented weaving.

Several researchers work on finding interactions at the programming level, normally in AspectJ code. Specific program analysis techniques for AspectJ programs were proposed [32, 4] in order to analyze if two aspects interfere. Clifton and Leavens [6] propose to classify aspects in *observers,* that do not change the system behavior, and *assistants,* that participate actively in the global computation. Similarly Katz [19] proposed to use 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 [25, 31, 28, 7]. Douence et al. [8] introduced a generic framework for aspect-oriented programming supporting pointcuts with explicit states and they provide an abstract formal semantics of their aspect language: this allows for detection of aspect interference.

6 Conclusion

A key problem of the aspect-oriented composition is the use of quantification which makes it more difficult to reason about it than in purely object-oriented models. In this paper, we presented an approach for detecting conflicts and dependencies in behavioral specifications of use cases refined with activity diagrams. The approach uses formal aid to analyze systematically an semi-formal specifications.

The analysis of conflicts and dependencies was carried with the tool AGG, a tool for specifying and analysing rule-based transformations of typed attributed graphs. The tool was used for specifying the behavior of aspects and objects in terms of pre- and post-conditions and for analysing conflicts and dependencies between them. The tool computed the necessary input in form of conflicts and dependencies which were then compared with the specified composition. Found conflicts and dependencies are potential conflicts (since they can be interactions) and not every possible conflict since it depends on the accurateness of the pre- and post-conditions. Nevertheless, the formal technique helps in making 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 crosscutting. Graph transformation also allow to reason uniformly about object and aspect models.

Besides an editor for specifying the rules, the tool also provides all analysis functions as an API. Rules can be read from an XML file. Therefore, AGG is ideal to be used with existing UML CASE Tools.

The presented approach can be applied in two ways. It can (i) be used, as demonstrated here, to validate an aspect-oriented design by comparing operators with conflicts. It can also (ii) be used to propose feasible aspect-oriented compositions by deriving them from the conflicts and dependencies found. The latter case will however not be the major application area for this.

We feel that pre- and post-conditions are an essential counterpart for an informal language like the UML, making modeling more rigorous. The analysis of pre- and post-conditions is not restricted to activity diagrams, which are essentially not yet object-oriented. It can also be applied to pre- and postconditions of 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.

The approach is not restricted to functional aspects, as presented here. Also so called non-functional aspects can be mapped to functional specifications in terms of pre- and post-conditions on the state of a system. Thereby, also interactions between functional and non-functional aspects are automatically covered.

7 Outlook

Support of analysis of the conflict and dependency graph is definitely needed to put the ideas to work also with larger examples. The AGG tool can also be used for testing through its simulation facility. This is needed in the absence of a completely automated analysis of the composition.

While pecifying the pre- and postconditions with object diagrams over the domain classes has been advocated by many object-oriented methods [11, 9, 14], the UML primarily proposes another solution. Conditions over instances of a model can be specified using OCL [29]. They could be analysed for conflicts and dependencies as proposed here. Either they have to be transformed into graph-based conditions readable by AGG or OCL tools are extended with critical pair analysis which they do not possess yet. Currently, AGG is integrating object-orientation also into the analysis facilities. So far, it has not yet integrated aspect-oriented facilities. One feature which could be extremely useful for reasoning about models is unification of types. Often, a reusable aspect model does use its own types which are not necessarily the same as those used in the domain class diagram.

References

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

[2] Jo ao Araújo, Jon Whittle, and Dae-Kyoo Kim. Modeling and composing scenario-based requirements with aspects. In *Proceedings of the 12th IEEE Int. Requirements Eng. Conf.* CS-IEEE, 2004.

[3] 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.

[4] Davide Balzarotti, Antonio Castaldo D'Ursi, Luca Cavallaro, and Mattia Monga. Slicing AspectJ woven code. In *Proceedings of the Foundations of Aspect-Oriented Languages workshop (FOAL2005)*, Chicago, IL (USA), March 2005.

[5] J. Brichau, W. De Meuter, and K. De Volder. Jumping aspects. position paper at the workshop "Aspects and Dimensions of Concerns", ECOOP 2000, June 2000.

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

[7] Giovanni Denaro and Mattia 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.

[8] Remi Douence, Pascal Fradet, and Mario 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.

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] Shmuel Katz. Diagnosis of harmful aspects using regression verification. In Gary T. Leavens, Ralf Lämmel, and Curtis Clifton, editors, *Foundations of Aspect-Oriented Languages*, March 2004.

[20] 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.

[21] 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: Model-based vs. Implementation-level Solutions (SETra'04), Satellite Event of ICGT'04)*, ENTCS, Rome, Italy, October 2004. Elsevier.

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

[23] 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.

[24] 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.

[25] Martin Rinard, Alexandru Sălcianu, and Suhabe 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.

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

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

[28] 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.

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

[30] Xerox Corporation. *AspectJ Programming Guide*. available from `http://eclipse.org/aspectj`.

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

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