Supporting Aspect-Oriented Modeling with Graph Transformations

Katharina Mehner*   Gabriele Taentzer
Technical University of Berlin
{mehner, gabi}@cs.tu-berlin.de

ABSTRACT

Aspect-oriented software development aims at supporting separation of crosscutting concerns throughout the full software life cycle. In this contribution we focus on life cycle support for crosscutting concerns which are given as rule-based knowledge. We propose support for aspect-oriented development for the requirements specification, analysis and design. In particular, we describe a novel approach which uses graph transformation systems in an aspect-oriented way.

1. MOTIVATION

In an object-oriented software development process the assumption is made that the requirements for the system to be developed can be captured in a straightforward way under the object-oriented paradigm. While the part of the knowledge describing sequences of behavior of objects can be easily captured, there is often a part of the knowledge which is *rule-based*. This knowledge is difficult to transform into the object-oriented paradigm.

Thus, instead of an early transformation of rules into object-oriented concepts, it is sensible to consider a heterogeneous requirements engineering process combining standard object-oriented techniques such as the *Rational Unified Process* [11] with rule-based specification techniques.

This approach has the following advantages. The rules can be modeled more adequately and intuitively. They are naturally linked to the domain model of the object-oriented part of the requirements, because they express conditions and conclusions in terms of the domain entities. Keeping the rules as such follows the principal of separation of concern and allows for independent and hence improved maintenance of both, rule-based and object-oriented models.

If rule-based knowledge contains conditions or conclusions which refer not only to values but to relationships between objects, *graph rules* are an adequate means for capturing these requirements [14]. To the contrary, business rules [3] are intended to capture value-based rules, such as a rule describing that the bill is reduced by 3 per cent if the number of items is bigger than 100.

Furthermore, graph rules are not only visually appealing but are also a language with formal semantics supporting a

range of analysis techniques [4]. This can be used to identify errors already on the analysis level. Furthermore, there are many tools for executing graph rules, which can be used for simulation or even for implementation of the rules.

Proposing an integration of the paradigm of graph rules with the object-oriented paradigm raises the question how their behavioral models can be linked. On the one hand, there is the imperative flow-controlled execution in an object-oriented program, on the other hand, there is the application of rules in a forward chaining manner, i.e., rules are triggered by the assertion of a new condition, as opposed to backward chaining where an explict query of a conclusion may generate new facts. The control over rule application can take different forms, ranging from complete nondeterminism to deterministically controlled application via rule orders or explicit rule selection.

Typically, by their declarative nature graph rules crosscut the domain entities. Instead of imperatively invoking rules from the OO model, i.e., explicitly triggering a single rule when a set of concrete conditions is fulfilled, we advocate to keep the quantifying nature of a rule-based system, i.e., we want to keep the idea that rules are triggered automatically when facts have changed. Therefore, we propose to *combine rules in an aspect-oriented way [10] with the object-oriented model*. This means that we propose to identify join points in the object-oriented model model which trigger rule applications.

Existing approaches have already considered the aspect-oriented integration of business rules with OO code [12]. This approach proposes a hybrid aspect model where rules trigger imperative models in an aspect-oriented way and where OO models trigger rules in an aspect-oriented way. Existing approaches integrating graph rules with OO have dealt only with invoking graph rules directly and explicitly from within object-oriented methods [6]. Such graph rules can also invoke object-oriented methods as side-effects.

Furthermore, there are several conceivable ways how to derive a *design* from an OO model and a forward chaining rule-base. We shortly sketch design issues and introduce our design approach using the aspect-oriented programming model Object Teams [13, 8].

This paper is structured as follows. Sect. 2 provides background on graph transformation systems. Sect. 3 describes

the extensions to a standard object-oriented development process with respect to requirements specification, analysis and design. Sect. 4 summarizes the key ideas and identifies future work.

2. GRAPH TRANSFORMATION

In this paper we present the concepts of graph transformation quite informally and give only as much theoretical background as needed. The interested reader is referred to seminal work in this area [14, 4, 5].

2.1 Typed Graph Transformations

Graphs are often used as abstract representation of diagrams. The transformation concepts described below are largely independent of the notion of graph, which can be chosen to reflect as closely as possible the concepts of the modelling language. In the following we use typed, attributed graphs.

In object-oriented modelling, 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 can be described more generally by the concept of *typed graphs* [2], where a fixed *type graph* TG serves as abstract representation of the class diagram. As in object-oriented modelling, types can be structured by an inheritance relation [1]. Instances of a type graph are object graphs equipped with a structure-preserving mapping to the type graph. 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.

A *graph transformation rule* $r : L \rightarrow R$ consists of a pair of TG-typed graphs L, R such that the union $L \cup R$ is defined. In this case, $L \cup R$ forms a graph again, i.e. the union is compatible with source, target and type settings. The left-hand side L represents the pre-conditions of the rule, while the right-hand side R describes the post-conditions. $L \cap R$ defines a graph part which has to exist to apply the rule, but which is not changed. $L \setminus (L \cap R)$ defines the part which shall be deleted, and $R \setminus (L \cap R)$ defines the part to be created. To make sure that newly created items are not already in the graph, we have to generate new vertex and edges identifiers whenever a rule is applied. Formally, for each application a new rule instance is created.

A *graph transformation step* is defined by first finding a match m of the left-hand side L in the current object graph G such that m is structure-preserving and type compatible. If a vertex embedded into the context, shall be deleted, dangling edges can occur. These are edges which would not have a source or target vertex after rule application. 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.

The applicability of a rule can be further restricted, if additional application conditions have to be satisfied. A special kind of application conditions are *negative application conditions* which are pre-conditions prohibiting certain graph parts.

Performing a graph transformation step with rule r at match m, all the vertices and edges which are matched by $L \setminus (L \cap R)$ are removed from G. The removed part is not a graph in general, but the remaining structure $D := G \setminus m(L \setminus (L \cap R))$ still has to be a legal graph, i.e., no edges should left dangling. This means if dangling edges occur during a rule application, they have to be deleted in addition. In the second step of a graph transformation, graph D is glued with $R \setminus (L \cap R)$ to obtain the derived graph H. Since L and R can overlap in a common graph, its match occurs in the original graph G and is not deleted in the first step, i.e. it also occurs in the intermediate graph D. For gluing newly created vertices and edges into D, graph $L \cap R$ is used. It defines the gluing items at which R is inserted into D. A *graph transformation*, more precisely a graph transformation sequence, consists of zero or more graph transformation steps.

The class diagram shown in Fig. 1 can be considered as type graph if all multiplicity constraints are ignored. It will be explained in more depth in the following section. Examples of rules are presented in Fig. 5. In each rule, the left and the right-hand sides are separated by an arrow. Numbers denote instance identifiers. A number occurring on both sides identifies equal instances occurring in both, the left and the right-hand sides, i.e. these are the items of $L \cap R$. Dashed nodes and edges in L denote graph parts which are prohibited, i.e. graph parts which belong to a negative application condition. The meaning of the sample rules will be explained in more depth in the next section.

A tool environment like AGG [15] can be used to specify and analyse rules over a certain type graph. Moreover, the rules can be tested by applying them to graphs which are compatible to the given type graph.

Given a host graph and a set of graph rules, two kinds of non-determinism can occur: first several rules might be applicable and one of them is chosen arbitrarily, and second given a certain rule several matches might be possible and one them has to be chosen. There are techniques to restrict both kinds of choices. Some kind of control flow on rules can be defined by applying them in a certain order or using explicit control constructs, priorities, etc. Moreover, the choice of matches can be restricted by specifying partial matches using input parameters. A common form of controlled rule application is the following one: One rule is selected from outside (e.g. the user) and triggers the application of a number of other rules which become applicable after the first rule has been applied.

Due to the formal foundation of graph transformation [14], rule dependencies can be analysed. One of the main static analysis facilities for graph transformation is the check for conflicts and dependencies between rules and transformations, resp. The check for conflicts between rule applications is available in AGG and has already been applied to identify conflicts in functional requirements in [7]. It depends very much on the application if conflicts can be tolerated and are even expected (as in models for concurrent systems) or have to be resolved (to get a functional behaviour). It remains future work to investigate how far the existing theoretical results for graph transformation can advantageously be used for aspect-oriented modelling.
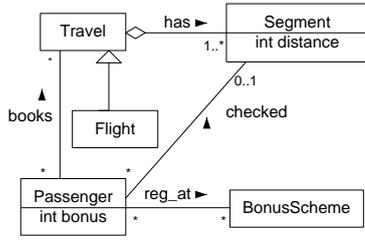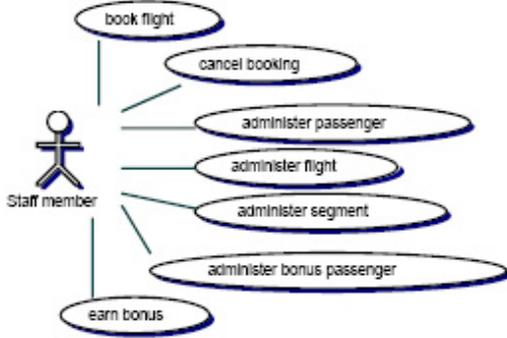
Figure 1: Domain classes flight booking and bonus

Figure 2: Flight booking use cases

In order to use graph rules in an object-oriented development process, the following issues have to be considered. Based on a requirement specification where certain use cases which heavily rely on rule-based knowledge are identified, an analysis model is created where these uses cases are modelled by graph rules and others by activity diagrams. Here the question arises how rules are integrated into an object-oriented model. When are which rules triggered? And where is a certain rule applied? These questions are discussed in the next section.

3. APPROACH

We view the development process as a chain of stepwise refinements gradually making the transition from requirements to a platform specific design. For a clear presentation we follow a standard object-oriented development process, namely the Rational Unified Process (RUP) [11], and distinguish between *requirements specification*, platform-independent *object-oriented analysis*, and *object-oriented design* which may contain platform-specific parts. We demonstrate how this process can be adopted to cover also rule-based modeling with graph transformations.

3.1 Requirements Specification

In an object-oriented requirements specification the main usage scenarios are captured with *use cases*. Each such use case has to be described in more detail, either textually or by means of *activity diagrams*. Also, during requirements specification a structural model of the problem domain is captured with a *class diagram*.

As an example, consider a simple flight booking system. Passengers can book flights consisting of individual flight
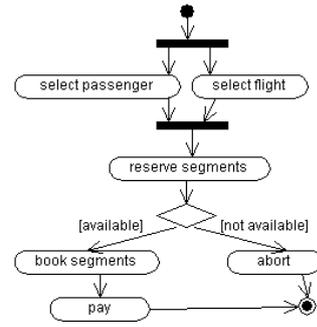

Figure 3: Activity diagram of use case book flight

segments. Bookings can be cancelled. Passengers are registered in the system and can be unregistered as well. Both is conducted by staff members on behalf of passengers. Also, flights and flight segments are administered by staff members. Passengers can register for a bonus scheme which enables them to collect bonus miles for flights they book depending on the lengths of flight segments and the previously collected bonus.

The domain classes are given in Fig. 1. A passenger can book a number of travels each of which consists of segments. Here, we have already integrated one generalisation using inheritance with the intention that the bonus collection can be applied to different kinds of traveling, not only to flights. Typically, this would be done during analysis but for simplicity we have already integrated it here. A passenger can register at a bonus scheme and collects bonus in the bonus field. When a passenger has received the bonus for a segment, there exists a link to the segment.

Use case	book flight
Actor	staff member
Trigger	passenger orders booking
Precondition	flight exists
Postcondition	each segment of the flight is booked
Main scenario	1. select flight
	2. select passenger
	3. reserve segments
	4. book segments
	5. pay

Table 1: Description of use case book flight

An overview of typical use cases is presented in Fig. 2. It comprises use cases for administration of passenger data, flight and segment data. For the main usage scenarios it has use cases for booking and canceling flights, for registering a bonus passenger, and for earning bonus for a flight booked. In the following, we will focus on the use cases *book flight* and *earn bonus*.

The steps, pre-, and post-conditions of use case *book flight* are described in Tab. 1. The steps are also presented in the activity diagram in Fig. 3. It is conceivable, that the use case *earn bonus* is expressed as rules such as in Tab. 2, which might be used to inform passengers about the bonus scheme. We further assume that the bonus can not only be

Bonus Scheme
1. Only a *registered* passenger can collect bonus miles.
2. A passenger can collect bonus miles only for *booked* segments.
3. A passenger can collect bonus miles only *once* for each booked segment.
4. A passenger with a bonus under 100.000 miles is in the *silver* bonus scheme.
5. A passenger with a bonus over 100.000 miles is in the *gold* bonus scheme.
6. A silver passenger collects one bonus mile per one segment mile.
7. A gold passenger collects two bonus miles per one segment mile.

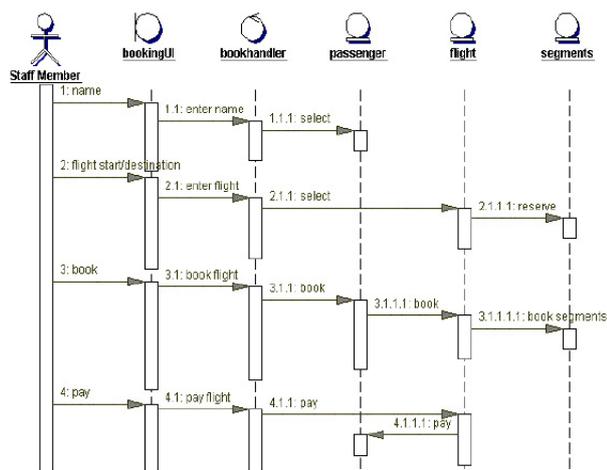**Table 2: Bonus Scheme given as Rules**

Figure 4: Sequence diagram book flight

applied to flights but to any kind of travels.

3.2 Analysis

In object-oriented analysis the use cases are refined with *sequence diagrams*. In addition, we show how to refine the rules which where used to describe the use case earn bonus. Based on the following two refinements we will show how object-oriented analysis and graph rules can be related to each other.

3.2.1 Refining Use Cases with Sequence Diagrams

The sequence diagram for the use case *book flight* is given in Fig. 4. For the explanation of the stereotypes used, the reader is referred to [9, 11]. The sequence diagram serves to identify additional classes and methods which are not yet part of the class diagram. In the refinement of the use case book flight we use of course an instance of Flight.

3.2.2 Refining Textual Rules with Graph Rules

Here we describe, how the textual rule base can be refined. The textual rules refer to entities modelled in the class diagram and to behavior modelled in other use cases. We see an advantage in graph rules because they can match and operate not only on the values of objects but also on the link structure. Also note, that here, we have rules which
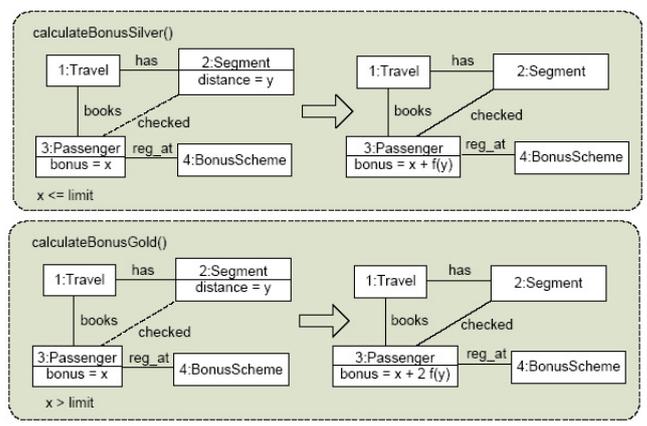

Figure 5: Bonus Rules

produce effects such as that a bonus is added, and not only rules, that constrain the application of some object-oriented behavior. Therefore, it is straight forward to use production or forward chaining rules which is the case for graph rules as presented here.

The textual rules contain rules common to all bonus application and rules different for different passengers. Here, we introduce two rules (see Fig. 5), one for the silver bonus scheme and one for the gold bonus scheme, each covering also the textual rules 1-3 which are common to all bonus applications.

The left hand side and the right hand side of each rule is specified as an instance diagram which is compatible with the class diagram. For both rules, the precondition is that a passenger is registered for the bonus scheme and that the passenger has booked a travel, e.g., a flight. Also, the precondition is, that the passenger has no link to a segment for which a bonus is earned. This is an example for a negative application condition which is depicted by a dashed link. In order to improve maintainability, we have not specified fixed values in the rules such as the limit of 100.000 miles or the assignment of one bonus mile for one segment. Instead we provide a variable *limit* and a function *f*. The silver rule has the precondition that a passenger must have a bonus x under a certain limit. In the gold rule, the bonus x must be over this limit.

Only if all preconditions are met, the rule can be applied. The silver rule computes the bonus with a function f of the distance and adds the result (cf. rule *calculateBonusSilver()*). The gold rule adds the calculated bonus twice (cf. *calculateBonusGold()*). Both rules have as post condition that a link between the passenger and the segment for which it has calculated the bonus is added. Note that this prevents a rule to be applied at the same match again.

We have already pointed out that a set of graph rules can be formally analysed for conflicts and dependencies. Some of the identified conflicts will lead to changes in the rule bases while others are inherent in the domain and will only have to be dealt with in the design. The inherent non-determinism might be desirable for a simulation software but undesirable

4

in other software where it has to be eliminated. Here, the rules given are mutually exclusive.

3.2.3 Integrating Object-Oriented Analysis and Rules

Until now, the coupling between graph rules and OOA is only via the structural model, i.e., the class diagram. That means that the graph rules will conceptually be executed whenever they find a match in the object graph of the running system.

This can make the system model difficult to understand and to reason about it. Also, it is not sufficient information to derive an efficient implementation, as it will be inefficient to check after each statement or method whether a match can be found. We therefore propose to make the implicit coupling of rules and object-oriented concepts more explicit.

Other parts of the requirements might already contain specifications when these rules should be applied. They could be a source to link the rule application more tightly with the object-oriented behaviour of the system. In the absence of a specification a more technical approach can be taken. The idea is to identify which behaviour will make changes to the object graph which can be in the match of a rule, i.e. the applicability of rules need not be checked after each change but only after certain changes. For our example (and many examples alike) it is adequate, because our rules are influenced by the OO behaviour but not vice versa. Here, we choose methods as a suitable granularity for identifying changes because they are the finest level of granularity for specifying behaviour in OOA and they are used in specifying sequence diagrams.

Also, when combining OO behavior and rules it has to be decided whether this can be done in parallel and if not, when the control flow is transfered between the two. Here, we consider a sequential paradigm.

We illustrate the ideas with our example. We aim at linking the graph rules from the example to selected triggers in the object-oriented behaviour. In the sequence diagram in Fig. 4 we want to identify a call which has the effect, that a link "book" is established in the object graph between a Passenger and a Travel object. We want to identify such a call on a domain object, not on boundary or control objects, because the rules are expressed in terms of the domain. The call book is the enclosing call on the domain objects involved in the booking, therefore, after this call, the link will be established. This method call can serve as a trigger for both of our rules.

The results of the analysis is captured by assigning the trigger rsp. join point to the rule and by defining the modus of the triggering, i.e., if the rule is to be executed before or after the method call, see also Tab. 3. Remember that the advice

Join point (Class and Method Call)	Modus	Rule (Advice)
Passenger.book()	after	calculateBonusSilver()
Passenger.book()	after	calculateBonusGold()

Table 3: Rules as aspects

is still the right hand side of the rule, the left hand side can be seen as additional conditions to be checked on the join point. Hence, a rule becomes an aspect in the AspectJ sense [10]. In this example, the join points are concrete calls, but also a set of calls is admissible, e.g., if the bonus is used with a system where different kinds of travels are purchased using different methods. Here, we have only considered the after modus which is in analogy to a rule being applied whenever a new match is found after something in the graph has changed.

Sequence diagrams are not necessarily the best place to identify which calls change which preconditions. Preconditions can only be determined by a more rigorous OOA. In [7] it was proposed to specify pre- and postconditions of activities with collaboration diagrams. This would allow to compare the conditions of a rule with the pre- and postconditions of an activity automatically. Once activities are mapped to methods, methods can be subsequently identified.

Defining triggers for rules replaces the search for an applicable rule. Still, the places *where* to apply the chosen rule are not determined, i.e., when a call triggers a rule, the rule is matched again the entire object graph. Only if one decides to restrict rules to the context of a method call, this avoids having to search the graph and allows to provide already a partial match for rule application.

The advantage of coupling rules with the object-oriented code in an aspect-oriented ways is threefold. First of all, the object-oriented model does not have to be changed to include triggers. It is oblivious to these enhancements. Also, the places in the behavior where a rule has to be triggered do not have to be identified and enhanced on a case-by-case basis but the enhancement can be specified once for all these places by using quantification similar to the pattern matching quantification already present in the rule base. Keeping the rules as such greatly improves their maintenance.

3.3 Design and Implementation

Deriving a design and an implemenation is still subject to further research. Here we give only a short discussion and present briefly our approach.

There are a number of options to derive a design from the platform independent analysis model.

- OOD for use with a graph transformation engine
- semi-automated transformation into OOD
- semi-automated transformation into OOD and AOD

The choice has different implications. The first choice aims at using a graph transformation system at runtime. This might incur more overhead than object-oriented or aspect-oriented solutions but can be an option for rapid prototyping.

The latter two will require, that the rules are finally translated into the object-oriented behavior, which is not a trivial task. It requires to implement the check of preconditions and the effect of the rules. For this task, we think that aspect-oriented design models have an advantage over a pure OO

design. They can better separate the rules and improve traceability and maintenance.

Furthermore it is desirable that the checks for preconditions and the effected transformation can be clearly encapsulated in the aspect model. Both only require access to a dedicated part of the object graph.

An aspect-oriented model which supports the encapsulation of a collaboration of objects is the design and programming model Object Teams [13]. This model decorates classes with *roles* which are special classes. A decorated class is called *base*. An object and its role instance can be seen as an aggregated entity. A role instance can interact with its decorated object in two ways: It can intercept method calls to the decorated object. This is called a *callin*. Callins can be of type *before*, *after*, or *replace*. A role can call methods of the decorated object without specifying the object. This is called a *callout*. *Teams* are a structuring concept to encapsulate collaborating roles. Teams have class features. A base object can have only one role instance per role class and per team instance. A callin definition can be compared to a join point interception by a pointcut in aspect-oriented programming [10]. The code executed on behalf of the role after the interception can be compared to an advice. Roles and their enclosing team thereby can be used to encapsulate crosscutting behavior and can be seen as an aspect.

Object Teams has already successfully been used previously for deriving a design of aspects with a complex internal structure and rich behaviour in [9].

4. CONCLUSION

We have proposed to integrate rule specifications with object-oriented models in an aspect-oriented way.

Graph transformation systems provide means to specify rule-based aspects directly as rules. The advantage of using graph transformation lies in their formal support to detect conflicts.

The advantage of the aspect-oriented integration of graph transformation into the object-oriented control flow is that the coupling is not specified on a case-by-case basis but for several cases at once. Furthermore, capturing the rules as separate entities allows easier maintenance of the rules and the traceability of rules in the code is kept.

The ideas outlined here form one part of our goal to develop a methodology for developing Object Teams programs. In the future, we want to investigate the relationship between graph transformation and aspect-oriented languages further.

5. REFERENCES

[1] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In M. Wermelinger and T. Margaria-Steffens, editors, *Proc. Fundamental Aspects of Software Engineering 2004*, volume 2984. Springer LNCS, 2004.

[2] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.

[3] C. Date. *What not how: The Business Rules Approach to Application Development*. Addison-Wesley, 2000.

[4] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[5] H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors. *Proc. 2nd International Conference on Graph Transformations (ICGT 2004), Rome, Italy, September/October 2004*, 2004.

[6] Fujaba homepage. `http://www.fujaba.de`.

[7] 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.

[8] S. Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In M. Aksit, M. Mezini, and R. Unland, editors, *Proc. Net Object Days 2002*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2002.

[9] 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.

[10] G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, and J. Palm. An overview of AspectJ. In *Proc. of 15th ECOOP*, number 2072 in LNCS, pages 327–353. Springer–Verlag, 2001.

[11] P. Kruchten. *The Rational Unified Process*. Addison Wesley, 2000.

[12] M.D'Hondt and V. Jonckers. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In *Proceedings of the Third International Conference on Aspect-Oriented Software Development*. ACM, 2004.

[13] Object Teams home page. `http://www.ObjectTeams.org`.

[14] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.

[15] G. Taentzer, C. Ermel, and M. Rudolf. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999.