# A Guidance for Model Composition

Kleinner S. F. Oliveira and Toacy Cavalcante de Oliveira
Informatics Faculty
Pontifical Catholic University of Rio Grande do Sul
Ipiranga Avenue 6681 - Building 32 - ZIP 90619-900
Porto Alegre - Brazil
{ksoliveira,toacy}@inf.pucrs.br

***Abstract*—With the advent of the Model Driven Development (MDD), models are replacing code as the major artifact in software development. These models are typically specified using OMG's Unified Modeling Language (UML) . In MDD, model transformation and model composition are two essential model management tasks. Model transformation has been extensively researched while model composition still needs further investigation. The goal of this paper is to propose a guidance for model composition (specifically for class diagrams composition) based on match rules, model transformation rules and model composition strategy. Moreover, we present UML Profile for Model Composition and make an analysis about UML's composition mechanism**

## I. Introduction

Advances in software development and information processing technologies have resulted in attempts to build more complex software systems. These systems have highlighted the inadequacies of the abstractions provided by modern high-level programming languages. This has led to a demand for languages, methods, and technologies that raise the abstraction level at which software systems are conceived, built, and evolved [1].

A current trend in software engineering consists of managing programs at the level of their concepts (using modeling languages) in order to simplify their design and maintenance and to increase their robustness against the rapid change of technologies. This trend proposes to raise the level of abstraction at which programs are designed and developed [2]. One reference to this new trend is Model Driven Architecture (MDA) [3], an approach to MDD from Object Management Group (OMG).

However, with the emergence of MDA, the role of model transformation and model composition[1] becomes more and more important. Model transformation has been extensively researched, documented and achieved an important advance, while model composition still needs more investigation and efforts to address significant problems. Model composition is an emerging research field, based on related work on database integration [4], aspect oriented modeling [5] and model transformation [6].

Model composition consists of combining two (or more) input models to generate an output model that combines the present content in both models. However, there are many open questions along with model composition, such as: If we need to compose two input models then what activities will we execute? UML was adopted as a standard modeling language, is UML's composition mechanisms so good as we think? In this paper the composition of UML's class diagram is considered as a merging of model elements of the same type. For example, class must be composed with class, association with association, attribute with attribute and as forth. The main contribution of this paper is a guidance for the specification of class diagram composition based on match rules, model transformation rules and model composition strategy. Moreover, we present UML Profile for Model Composition and make an analysis about UML's composition mechanism

In order to briefly answer these questions (this work does not claim completeness), this paper is organized as follows. Section 2 describes what a model composition is. In Section 3 we propose an initial guidance for model composition explaining the activities defined in our approach. In Section 4 we present the transformation rules. Section 5 presents an example of model composition. Section 6 presents some conclusions.

[1]Model composition and model merging are considered corresponding terms in this work.

## II. Model Composition in a Nutshell

An object oriented design consists of a set of models and in which each model consists of a number of different kinds of UML's class diagrams. In order to have an integrate view of the system, the models must be composed. Model composition is defined by composition operation, a special type of transformation, that takes two models $M_A$ and $M_B$ as input and combines their elements into a model $M_{AB}$. Model composition is a generic operation that varies from application to application. Which elements from $M_A$ and $M_B$ are combined and in which way depends on the operation implementation [7].

### A. Model Composition Semantic

When is needed to merge two models, they must be identified in order to avoid wrong interpretation and to define the role of each model in model composition mechanism. Therefore, we now present a semantic for model composition. This semantic is based on composition semantic defined in [8]. In the Figure 1 are defined:

- **merged model**: the first operand of the merge. It is merged into the receiving model.
- **receiving model**: the second operand of the merge. It is the central element during the composition and is the source of the merge arrow in the diagrams.
- **resulting model**: this term is used to specify the model obtained after model composition have been performed.
- **merged element**: refers to a model element that exits in the merged model.
- **receiving element**: is a model element in the receiving model.
- **resulting element**: is a model element in the receiving model.

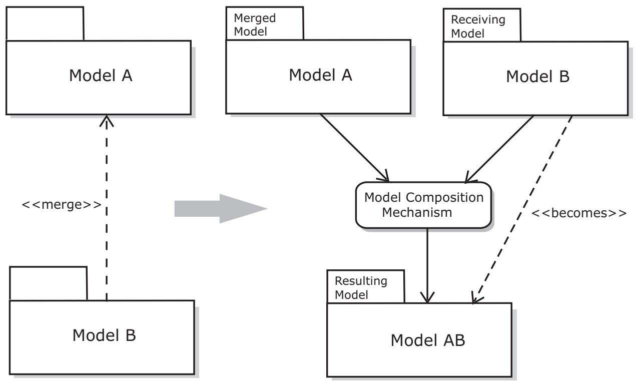

Figure 1. Model composition semantic

In order to provide a clear view of the model composition mechanism we present our approach based on four phases described below, know as: initial phase, comparison phase, merge phase and post-composition phase.

### B. Initial Phase

In this phase, the input models are analyzed in order to know each type of model (i.e. class, association, etc). The models are separated and grouped according to its types. For example, Class and Association are identified and grouped. The goals of this phase is to know the input models.

### C. Comparison Phase

The goal of this phase is to find equivalents model elements based on its signatures. The model element's signature is defined in terms of its syntactic properties, where a syntactic property of a model element defines its structure. The signature is a collection of values for a subset of syntactic properties in model element's metamodel class. For example, *isAbstract* is a syntactic property defined in the metamodel class called *Class*. If an instance of a *Class* is an abstract class then *isAbstract = true* for the class, otherwise the instance is a concrete class, *isAbstract = false*. The set of syntactic properties used to determine a model element's signature is called *signature type*, as defined in [9]. A signature that consists of all syntactic properties associated with a model element is called *complete signature type* and the signature only based on name is called *default signature type*.

The signature is structured in comparison levels organized hierarchically . For instance, in Figure 3, a possible definition of levels for the class *P1.Student* would be: *P1.Student* (level 1) with *P1.Student.name* (level 2) and *P1.Student.age* (level 2), and *P1.Student.getName()* (level 3). This indicate that *P1.Student* is the first syntactic property used to compare this class with another. For each model element type is defined one signature.

In order to check if two model elements are equivalents (same signature), we defined *match rules* and *match operator*. This operator is responsible to execute the match rules. Match rule defines when two model elements are equivalents. if a match rule fails then the models are not possible to compose themselves. Otherwise, models will be composed. In order to group the equivalent models two files were created: *Match Model Package* (MMP) to group the equivalent models, and *No-Match Model Package* to group the no equivalent models, (NoMMP). The main goal of grouping equivalents model elements in MMP is facility the model comparison process. The end of comparison phase is reached as soon as all models are located either MMP or NoMMP.

*1) Match rules:* Merged model and receiving model are corresponds if, and only if, they satisfy all match rules applied to merged element and receiving element. We present a short description of the these rules, as follows:

*Class match rule:*
MatchClass(Class mrgd,Class rcv) → mrgd.name = rcv.name

*Association match rule:*
MatchAssociation(Association mrgd, Association rcv) → (mrgd.name = rcv.name) AND (mrgd.memberEnds = rcv.memberEnds)

*Attribute match rule:*
MatchAttribute(Class mrgd, Class rcv) →
(mrgd.ownedAttribute.name = rcv.ownedAttribute.name)
AND (mrgd.ownedAttribute.TypedElement = rcv.
ownedAttribute.TypedElement)

*Operation match rule:*
MatchOperation(Class mrgd, Class rcv) → (mrgd.
ownedOperation.name = rcv.ownedOperation.name) AND
(mrgd.ownedOperation.ownedParameter.length = rcv.
ownedOperation.ownedParameter.length) AND (∀x(mrgd.
ownedOperation.ownedParameter[x] = rcv.ownedOperation.
ownedParameter[x])

### D. Merge Phase

There is little agreement on requirements, activities and steps are need to follow in order to accomplish the composition of two simple input models, and even less on good practices to avoid problems during model composition. Several approaches [10] [11] have been proposed to resolve the problems found in model composition, but none, as yet, was defined as standard. In [8], the UML's Model Composition Mechanisms (PackageMerge) does not present a tasks flow to merge UML models, does not present a good documentation, and, moreover, does not defines how model merge is accomplished.

As a preliminary requirements for this phase, MMP and NoMMP have already been defined once they are input models (source models) for this phase. The models must be properly composed in order to obtain an integrate view of input models. Our approach is based on *merging strategy* as well as [10]. *Strategies* are pluggable algorithms that can be attached to merging phase to implement specific functionality of model composition, such as: *override*, *integration* and *union*. For each strategy there is an *operator* responsible to accomplish the activities defined in this strategy. For instance, *union operator* accomplishes all activities specified in *union strategy*.

Merged elements and receiving elements with same signature and syntactic type are merged to form a single element in composed model. As in [12], we assume that model elements of the same syntactic type and with same name represent different and consistent view of the same concept. However, in some case, this may not be the case if model elements are developed independently.

In order to work at the abstract level in a model composition, it is necessary to have a language capable of modeling this domain. To this end, we present a simple UML Profile for Model Composition (UPMC) to allow tailoring UML to fit the needs of model composition domain. Using this profile is easier to identify a merged and an unchangeable model. Figure 2 defines an early version of UPMC. Table I specifies a description for each stereotype of UPMC.

*1) UML's Package Merge Mechanism:* The UML has been used as standard modeling language, but it does not contain enough elements to model the composition. Model composition mechanism is defined by PackageMerge. A PackageMerge defines how the contents of one package are extended by the contents of another package. As UML metamodel is organized into packages, according to function and complexity, the package merge is used extensively. However, package merge does not have a solid theoretical background, noting written about it in the literature, outside of the UML specification itself and in [13].

Based on analysis of UML 2.1 specification and of [13], some problems in package merge were found, such as: (i) the definition of package merge in the UML specification is incomplete, ambiguous and inconsistent; (ii) the semantics of package merge is not well defined; (iii) most popular UML modeling tools do not implement package merge.

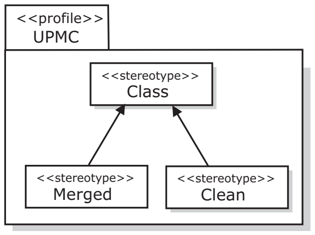

Figure 2. UML Profile for Model Composition in early version

## E. Post-Composition Phase

The goals of this phase is to uncover design errors, undesirable properties and conflicts. All models from composition phase are verified against *well-formedness rules* in order to identify badly formed models, such as: class with elevated number of attribute, operation with excessive number of parameters. For each uncovered problems are identified an applicable *Model Transformation Rules* in order to resolve the problems and conflicts. These *rules* are based on [9] and its definition will present on Section IV.

## III. Guidance for Model Composition

Based on the Section 2, we now present an activity flow in order to provide a description of how model composition activities are accomplished. Figure 4 outlines our approach to model composition by an activity flow.

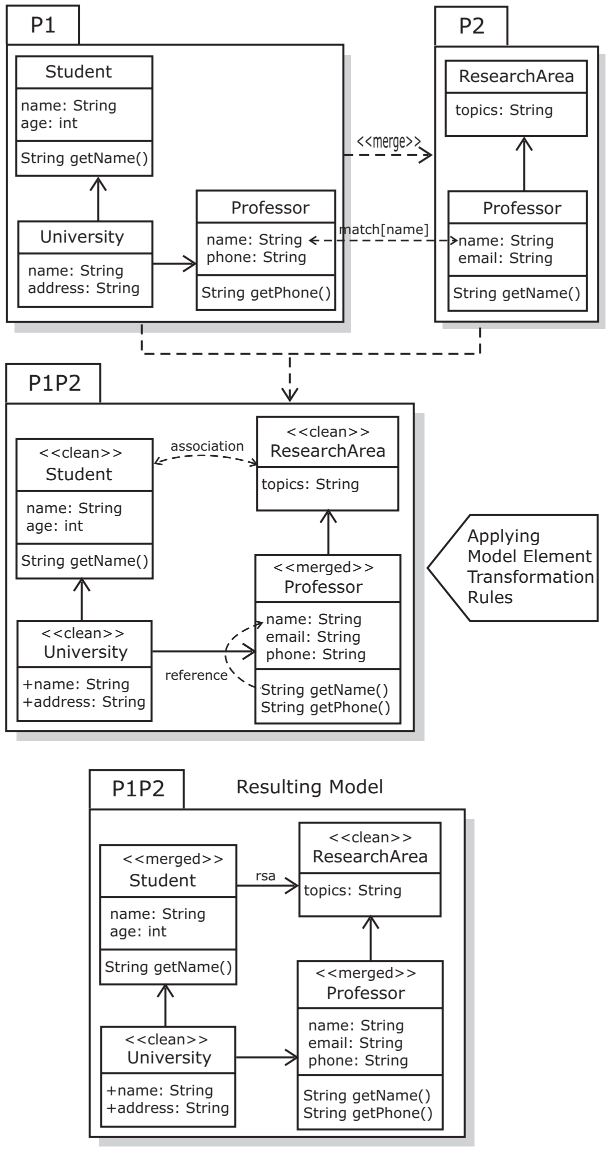

Figure 3. A composition with corresponding Classes and Attributes

Table I
CANDIDATES STEREOTYPES FOR MODEL COMPOSITION

| UPMC | Applies To | Stereotype | Description |
|---|---|---|---|
| Merged | Class | ≪Merged≫ | Merged stereotype is used to identify a composed model. Model class branded by this stereotype implying that one merged strategy was executed. |
| Clean | Class | ≪Clean≫ | Clean stereotype is used to identify an unchangeable model. Model class branded by this stereotype implying that no merged strategy was executed. |

## IV. TRANSFORMATION RULES

In order to produce the resulting model with desired features and without conflicts is necessary to apply some transformation rules. These rules are also used to ensure well-formedness rules. For example, a class must not have an operation with five or more parameters, an attribute must not have a name so large (i.e. with twenty character). The *bad-formedness conflicts* occurs when a well-formedness rules is not respected. Model Element Transformation Rules (METR) are the transformation rules applied to class elements. With METR is possible:

1) Creating and deleting models.
2) Add to and remove from a package.
3) Rename model elements.
4) Changing references to a model element.
5) Merging two models according to a merge strategy

The transformation rules are described following a standard, as follows:
**Name**: define the name of the transformation rule.
**Description**: used to make a description of how a transformation rules must be applied.
**Context**: define the context in which transformation rule is used.
**Syntax**: specify the syntax of the transformation rule.
**Pre-condition**: specify the conditions must be satisfied in order to transformation rule can be executed.
**Post-condition**: define the conditions must be obtained after transformation rule has been performed.

### A. Model element transformation rules

TR1. Creating new model element
**Name**: create
**Description**: this transformation rule is used for creating new ModelElement. This rule is applied as a *factory* and is able to create any model element. To specify this rule was used *reflection* defined in *CMOF::Reflection* [14]. Basically, we provide the type of element need to create and its arguments (when need).
**Context**: to create a new model element
**Syntax**:
newElement = **create**[ModelElement.type](arg: Argument[*]){

Factory.createElement([ModelElement.type], arg) }

**Pre-condition**: check if the arguments are valid properties of the correct type and if that values are supplied for all mandatory properties with no default.
**Post-condition**: this rule must have created a model correctly. Moreover, it is possible to make a reference to well-formedness rules in order to certify *bad-formedness conflict*.

TR2. Deleting a model element
**Name**: delete
**Description**: we can use this transformation rule when a class's model element have to be deleted. It has two operands: (i) the ModelElement to be deleted; and (ii) the Package in which ModelElement is inserted.
**Context**: to delete a model element
**Syntax**:
**delete**[ModelElement.type] name :ModelElement
**from** owner :Package
**Pre-condition**: check if first the namespace exist then verify the name of ModelElement. All operands must be correctly specified.
**Post-condition**: this rule must have deleted a model correctly. Moreover, it is possible to make a reference to well-formedness rules in order to certify *bad-formedness conflict*. For instance, if a class was deleted and another class makes reference to her then a conflict arise.

TR3. Inserting a model into a Package
**Name**: insert
**Description**: All model must be located in a Package. Therefore, the goal of this rule is to insert a model in a Package. It has two operands: (i) the ModelElement (i.e. class or association) to be inserted; and (ii) the Package specification in which ModelElement will be inserted.
**Context**: to insert a model into a package
**Syntax**:
**insert**[ModelElement.type] name :ModelElement
**into** owner :Package
**Pre-condition**: check if first the Package exists then verify the name of ModelElement. All operands must be correctly specified.
**Post-condition**: this rule must have inserted a model correctly in a Package.

TR4. Renaming a model into a Package
**Name**: rename
**Description**: The goal of this rule is rename model elements avoid *name conflict*. For instance, two attribute of different class have same name and they must be inserted in resulting model. This set a *name conflict*. Therefore, one of these attributes must be renamed. Rename rule has tree operands:

(i) a model (i.e. Class, attribute, operation, association, etc.) to be renamed; (ii) the model's Namespace and (iii) a new name is used to assign to rename the model.
**Context**: to rename a model into a Package
**Syntax**:
**rename**[ModelElement.type] name :ModelElement
**from** owner :Package
**to** newName :ModelElement.name
**Pre-condition**: a model must be defined in a Package.
**Post-condition**: this rule must have inserted a model correctly in a Package.

TR5. Merging two models according to a merge strategy
**Name**: merge
**Description**: The goal of this rule is merge two model elements according to a merge strategy. Each merge strategy will produce a different kind of output model. This rule has five operands: (i) two model elements and theirs Package specification. (ii) a merge strategy definition.
**Context**: to merge two models
**Syntax**:
**merge**[ModelElement.type]
name :ModelElement **in** owner :Package **with**
name :ModelElement **in** owner :Package
**by** strategy :MergeStrategy
**Pre-condition**: the MergeStrategy must be valid.
**Post-condition**: this rule must correctly merge two model.

TR6. Creating an association between two models
**Name**: createAssociation
**Context**: To create an association between two models.
**Description**: This rule creates an association from two supplied Elements that is an instance of the supplied Association. The first element is associated with the second element and must conform to its type. To specify this rule was used *reflection* defined in *CMOF::Reflection*. Basically, we use Fatory class to create a Link between two models.
**Syntax**:
```
createAssociation(ass :Association, arg1 :AssociationEnd,
arg2 :AssociationEnd){
    Factory.createLink(ass, arg1, arg2)
}
```
**Pre-condition**: the arguments must be correctly specified.
**Post-condition**: a association between two model must be correctly provided.

TR6. Replacing reference to a ModelElement
**Name**: replaceReferences
**Context**: To replace an association between two models.
**Description**: Removing a ModelElement may lead to invalid references that refer to a non-existent ModelElement, a *reference conflict*. Adopted from [12], we make use of replaceReferences directive in order to change these references.
**Syntax**:
**replaceReferences** originalName :Name
**with** replacementName :Name **in** owner :Package
**Pre-condition**: the arguments must be correctly specified.
**Post-condition**: a association between two model must be correctly changed.

## V. Example of Model Composition

One specific application of model composition is when separate models contains specifications for different requirements of a software system. For example, in the context of global software development the system development effort is shared with design teams may work on different requirements concurrently. In determined moment, compose the models is necessary. Thus, the model composition is applied. Based on the previous sections, we now show one simple example of model composition depicted in Figure 3. A desired feature is defined and it must be found in output model: a *Student* must have a research area as well as *Professor*.

In order to compose P1 and P2 the guidance defined in Section III is followed:

1) **Initial Phase**: the classes *P1.Student*, *P1.University*, *P1.Professor*, *P2.Professor* and *P2.ResearchArea* are grouped because are same types.
2) **Comparison Phase**: comparison process is based on *default signature*, therefore the classes *P1.Professor* and *P2.Professor* are equivalences by merging relationship between *P1* and *P2*. These classes are inserted into MMP and the remainder classes into NoMMP.
3) **Merge Phase**: In this example, merging strategy applied is *integration strategy*. Therefore, corresponding classes, attributes and operation appear once in the composed model. Since *P1.Professor* and *P2.Professor* are corresponding, *P1.Professor* only appears once in the composed model. In the same way, *P1.Professor.name* and *P2.Professor.name* appear once in *P1P2.Professor*.
4) **Post-Composition Phase**: composed model does not result in a well-formed model, since there may be references to the *P2.Professor.name* attribute in *P1P2.Professor.getName()* (reference conflict). In order to solve this conflict are executed some activities, as follows:
   **Step 1**: All references to *P2.Professor.name* attribute must be changed to *P1P2.Professor.name*, as follows:
   **replaceReferences** *P2.Professor.name*
   **with** *P1P2.Professor.name* **in** P1P2
   **Step 2**: the *P1P2.Student* class must have an association to P1P2.ResearchArea, therefore, a new association between *P1P2.Student* and P1P2.ResearchArea must be created, as follows:
   ```
   createAssociation( rsa, Student , ResearchArea){
     Factory.createLink(rsa, Student, ResearchArea)
   }
   ```

   The resulting model is showed in Figure 3.

## VI. Conclusion

The main contributions of this paper are a definition and a organization of model composition activities in phases

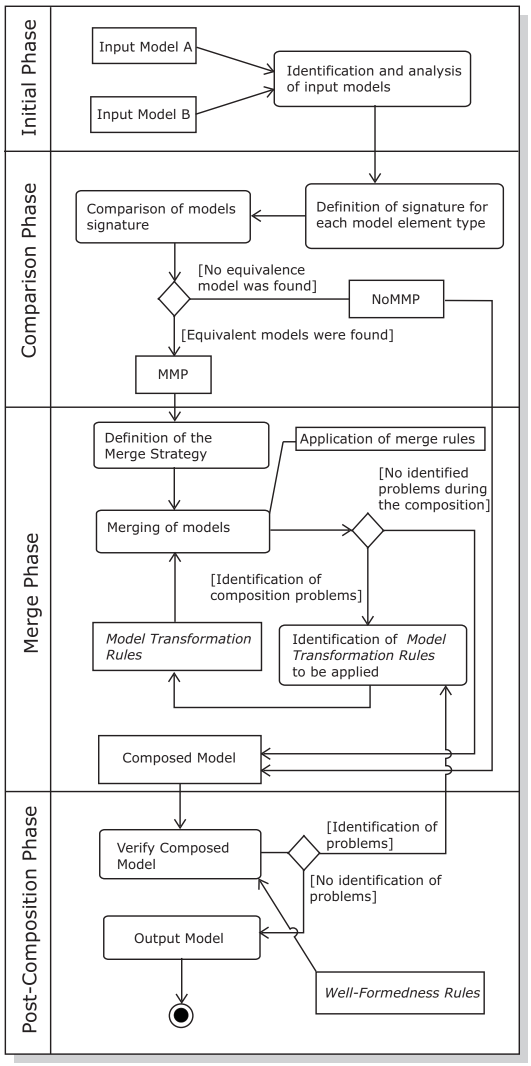
Initial Phase
Comparison Phase
Merge Phase
Post-Composition Phase
Input Model A
Input Model B
Identification and analysis of input models
Definition of signature for each model element type
Comparison of models signature
[No equivalence model was found]
NoMMP
[Equivalent models were found]
MMP
Definition of the Merge Strategy
Application of merge rules
[No identified problems during the composition]
Merging of models
[Identification of composition problems]
Model Transformation Rules
Identification of Model Transformation Rules to be applied
Composed Model
Verify Composed Model
[Identification of problems]
[No identification of problems]
Output Model
Well-Formedness Rules

Figure 4. Model composition activity flow

and a specification of activity flow of model composition, moreover a definition of transformation rules. We presented a guidance for defining how the activities are distributed among the phases. Moreover, UML model composition mechanism (Package Merge) was analyzed and identified some its problems. In order to elevate the abstract level in a model composition and make easier to identify the merged model and no merged model was defined an UML Profile for model composition that have been evolved. The definition of model composition semantics, models transformation rules and UML Profile for model composition provide a set of terms are used during model composition description.

Future work will concentrate on clearly defining of merging strategy and its operators. The model element transformation rules are in the initial definition stage so that they can be used to resolve identified problems in composed model. Empirical evaluation is needed to validate this approach in real world design settings of model composition. Finally, we observe that model composition is in initial stage and its improvement is absolutely necessary to model engineering evolution.

## References

[1] Robert B. France, S. Ghosh, and T. Dinh-Trong, "Model Driven Development Using UML 2.0: Promises and Pitfalls," *IEEE Computer Society*, vol. 39, no. 2, pp. 59–66, February 2006.

[2] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, September/October 2003.

[3] Object Management Group, *MDA Guide Version 1.0.1*, 2003, http://www.omg.org/docs/omg/03-06-01.pdf.

[4] C. Batini, M. Lenzerini, and S. B. Navathe, "A Comparative Analysis of Metodologies for Database Schema Integration," *ACM Computing Surveys*, vol. 18, no. 4, pp. 323–364, December 1986.

[5] T. Cotternier, A. van den Berg, and T. Elrad, "Modeling Aspect-Oriented Compositions," in *Proceedings of Workshop on Aspect-Oriented Modeling co-located with MODELS 2005*, October 2005.

[6] F. Jouault and I. Kurtev, "On the Architectural Alignment of ATL and QVT," in *Proceedings of Symposium on Applied Computing (SAC 06).* ACM Press, April 2006.

[7] M. D. D. Fabro, J. Bzivin, and P. Valduriez, "Weaving Models with the Eclipse AMW Plugin," in *Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany*, 2006.

[8] OMG, *Unified Modeling Language: Infrastructure version 2.0*, Object Management Group, February 2007, http://www.omg.org/cgi-bin/apps/doc?formal/07-02-03.pdf.

[9] Y. Reddy, R. France, G. Straw, N. M. J. Bieman, E. Song, and G. Georg, "Directives for Composing Aspect-Oriented Design Class Models," in *Transactions of Aspect-Oriented Software Development*, vol. 1, no. 1, 2006.

[10] D. Kolovos, R. Paige, and F. Polack, "Merging Models with the Epsilon Merging Language (EML)," in *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MODELS/UML 2006)*, Genova, Italy, October 2006.

[11] R. F. Benoit Baudry, Franck Fleury and R. Reddy, "Exploring the Relationship Between Model composition and Model Transformation," in *Proceedings of Aspect Oriented Modeling Workshop, in conjution with MoDELS'05*, 2005.

[12] G. Straw, G. Georg, E. Song, S. Ghosh, R. France, and J. Bieman, "Model Composition Directives," in *Proceedings 7th International Conference on UML Modelling Languages and Applications*, ser. LNCS, A. M. Thomas Baar, Alfred Strohmeier and Stephen J. Mellor, Eds. Springer-Verlag, Outubro 2004, pp. 84–97.

[13] A. P. Zito, "UML's Package Extension Mechanism: Taking a Closer Look at Package Merge," Master's thesis, School of Computing, Quenn's University Kingston, Ontario, Canada, September 2006.

[14] OMG, *Meta-Object Facility Core Specification Version 2.0*, Object Management Group, 2002, http://www.omg.org.