Towards a generic approach for model composition

Adil Anwar[1,3], Sophie Ebersold[1] , Mahmoud Nassar[2] , Bernard Coulette[1] and Abdelaziz Kriouile[2]
[1]*University of Toulouse*
IRIT, UT2, 5 allées A. Machado, F-31058 Toulouse, France
anwar@univ-tlse2.fr, ebersold@ univ-tlse2.fr, coulette@ univ-tlse2.fr
[2] *SI2M, ENSIAS, BP 713 Agdal, Rabat, Maroc*
nassar@ensias.ma, kriouile@ensias.ma
[3] *LRIMIARF, Université Mohammed V-Agdal, Rabat, Maroc*

Abstract

Model composition is considered as a crucial activity in Model Driven Software Development (MDSD). It is a common feature when adopting a multi-modeling approach to analyze and design software systems. Similar modular approaches are known under several names such as viewpoints, aspects, subjects, etc. In previous works, we proposed a View-based UML profile called VUML. In this paper, we describe a conceptual framework whose goal is to separate the generic composition part from the specific part dedicated to a given modeling domain. We apply our approach to the composition of UML class diagrams into one VUML class diagram.

1. Introduction

Several approaches adopted by the software engineering community rely on the principle of multi-modeling which allows modeling a system as a set of less complex software units. This principle has been introduced in several programming approaches such as subject-oriented programming [19] or aspect-oriented programming [13]. At the model level, comparable approaches use concepts such as Views/viewpoints [9], Subject oriented development [6], and Aspect oriented Modeling [2] [11].

Our work in this area [16] has resulted in the definition of the VUML language (View based Unified Modeling Language). VUML provides a formalism and a methodology to carry out a view-based modeling approach from analysis to coding. The VUML approach makes possible to model a software system according to each user's viewpoint. First, actors of system are identified and each of them is associated to a unique viewpoint. Then, for each viewpoint, we describe, in an iterative way, use cases, scenarios as well as related classes. The result is a class diagram (called also partial model) in the UML formalism. Secondly, we produce a VUML model by composing the partial models. The composition technique is based on a set of transformation rules [1]. More precisely, we define the composition of static UML models as a set of transformation rules available as correspondence, merging and translation rules. These rules allow first to establish correspondences between input models, then to merge these partial models into a global VUML model.

The main contribution of this paper with regard to our previous work in [1], is the focus on the reusability of the composition process. For this purpose, we propose a two levels composition approach: a generic level independent from any modeling domain, and a specific level which depends on a given modeling domain. The generic level is defined by a generic composition metamodel. This metamodel is independent from any specific transformation language and provides means to express the key features necessary to automatically compose models. The generic composition language comprises a relationship metamodel, a transformation rule metamodel and a generic transformation strategy metamodel. For a given modeling domain (e.g. source models in UML and target model in VUML), we extend the generic metamodel by (i) specializing the correspondence relationships, (ii) specializing transformation rules and (iii) specifying transformation strategies so as to generate a set of enactable transformation rules.

The present paper is organized as follows: Section 2 describes the main steps of our composition process according to a rule driven approach. Section 3 is devoted to the description of the generic composition metamodel. In Section 4 we show how this generic

metamodel can be specialized for a given modeling domain, and in Section 5, we apply this specialization process to the VUML modeling domain. Section 6 considers related works. We conclude this paper in section 7.

2. Model Composition process

Several studies [14] [4] [10] have demonstrated that automating the model composition can be done through two separate operators: a correspondence operator and a merging operator. To allow the verification of the composed model, we propose a composition process which is made of three main steps called correspondence, merging and analysis (Figure 1).

The Correspondence step consists of identifying the set of links between models to be composed (to make things easy we consider only two source models here). It is governed by correspondence rules which implement the comparison strategy between model elements. The comparison of elements is based on internal properties defined at the metamodel level. For example, a sub set of internal properties of an UML class may be represented by the set {*name*, *isAbstract*, *ownedAttribute*, *ownedOperation*} that are properties of the metaclass Class in the UML metamodel [18]. A correspondence rule, applied to two elements describing the same concept in different source models, creates a correspondence relationship between those elements. This relationship will be stored in a correspondence model.

The Merging step depends on the target metamodel. In our application context, this merging step produces a VUML model which elements are stereotyped according to the VUML profile [16]. VUML elements are created by applying both merging and translation rules. The merging strategy is mainly depending on link type. In fact, merging rules are applied to the elements that are related to each other through correspondence relationships. Elements which have no correspondent in the opposite model are translated into the VUML model with respect to the translation rules.

The Analysis step aims to uncover composition errors. The produced composed model can be formally analyzed against desired properties or by verifying its compliance with the well-formedness rules (W.F.R). While a rule is violated an error is detected and it creates a problem element with respect to the structure of a problem model which conforms to a problem metamodel [5]. The problem model can be analyzed and then imported into a model refactoring tool dedicated to resolve problems. Activities related to problem fixing are not in the scope of this paper.

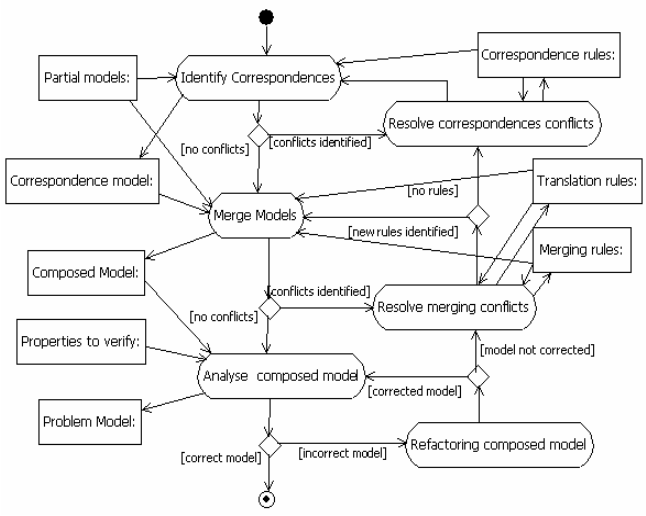

Figure 1. Process composition in VUML approach

3. A generic framework for model composition

A model composition framework should provide means to support common features for building a generic composition operator. The survey presented in [4] summarizes a core set of minimal requirements for a model composition framework. We propose to define this generic composition operator through three components: links, transformations and strategies. The first component allows defining and capturing relationships between model elements. We have defined a generic correspondence metamodel which describes the basic constructors for link management and supports feature extension. The second component provides means to carry out transformations between involved models. To this end, we have defined a rule-based transformation metamodel which allows the implementation the three fundamental operations: correspondence identifying, merging and translation. Finally, the third component provides the means to define transformation strategies. Strategies specify an operational semantics for transformation rules. We have defined a metamodel which summarizes the common practical strategies and studied the possibility to extend the default behavior of defined strategies to cover specific user's needs.

3.1. A core correspondence metamodel

The correspondence metamodel defines the different kinds of relationships between model elements (

Figure 2). To this end, we have extended the core weaving metamodel proposed in [8] to support composition requirements and to handle new relationship types. The correspondence metamodel defines generic constructs needed to model the correspondence concepts. Models that conform to this metamodel should contain relationships that establish mappings between elements of the input models. We present in the following the key elements of the correspondence metamodel.

- **CorrespondenceRelationship:** this metaclass defines the relationship between elements of the source models. The definition of a new relationship is made through a specialization of this metaclass; this allows semantic definition of each relationship. The parameter *TypeElement* is substituted by the type of model element related by an instance of this metaclass.
- **CorrespondanceRelationshipEnd**: this construct represents the end of a correspondence relationship. It serves as a reference to elements linked by the relationship.
- **CorrespondingElementRef**: this metaclass models the concepts of reference. It contains an attribute *name* that represents the name of the referenced element, and an attribute *ref* which acts as a persistent model-element identifier.
- **ReferencePackage:** this metaclass is a container for reference elements. An instance of this metaclass contains all references of connected elements.

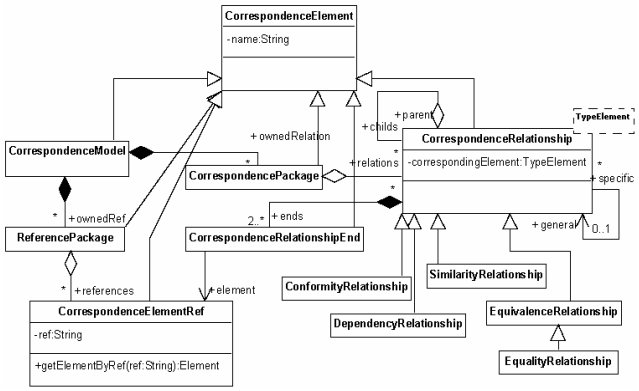

Figure 2. The correspondence metamodel.

The generic *CorrespondenceRelationship* metaclass must be extended to create different relationship types. Defining extensions is a practical solution to represent a specific semantic for each correspondence relationship. For instance, *SimilarityRelationship* indicates that the linked elements represent the same concept but differ in some properties (e.g. two classes with the same name and different sets of attributes), the *DependencyRelationship* denotes dependency information between two elements, and the *ConformityRelationship* defines the link type between elements which are two consistent views of the same concept.

3.2. Transformation rules metamodel

Our composition process is governed by a set of transformation rules. Indeed, the composition operation can be specified using the model transformation principles. We have defined three categories of transformation rules: correspondence, merging and translation (Figure 3). This enables us firstly to establish correspondences between input models, and secondly to merge these models to produce the global model. Finally, elements that have no corresponding element in the opposite model are transformed according to translation rules. We state that transformation strategies depend on each specific domain, so we do not consider them in the generic part of our metamodel.

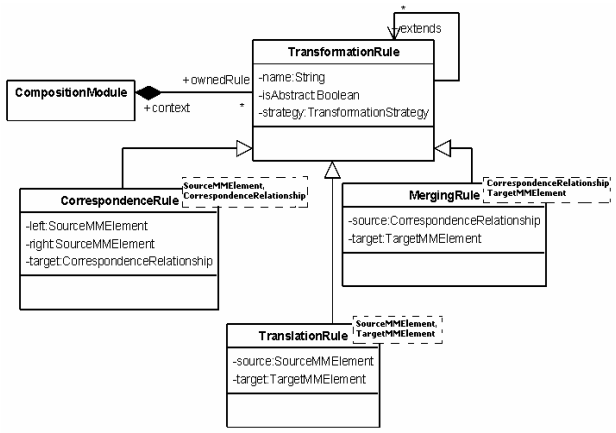

Figure 3. Transformation rules metamodel

The transformation rules metamodel is a generic description of transformation rules. Therefore, we use the UML template diagram to describe this metamodel. In the next section we show how this metamodel allows define (when instantiated) the specific transformation rules as a declarative approach to compose models in a particular application domain. This solution improves reusability of the approach. Indeed, we can use this approach for merging models of diverse metamodels by instantiating generic rules multiple times, this is determined by a set of bindings, and each binding represents an association between a template parameter and an application-specific value.

3.3. Transformation strategies metamodel

So far, we have introduced the structural aspects of our composition operator. Yet, those structural aspects are not sufficient to provide a comprehensive definition of such an operator, as one must define the behavioral aspect of the operator as well. Our approach uses the transformation strategies to specify the behavioral aspect of each transformation rule (Figure 4). Strategies are defined in [14] as pluggable algorithms that can be attached to a transformation rule to implement a recursive and reusable functionality which may be inferred by the metamodel structure. Using strategies has the advantage of minimizing the manual intervention of the developer.

Correspondence strategies define comparison logic between model elements. We distinguish three types of correspondence strategies. The first type of strategy is based on the signature type as described in [20]. The signature type of an element is described by a set of internal properties (name, type, cardinality, etc) defined in the metamodel. For elements of type class, this strategy depends mainly on the meta-properties values of the metaclass Class defined in the metamodel UML. For example, if one decides to define the signature type of the class element by the couple (*name*, *isAbstract*), then comparing two classes defined in two different models is reduced to compare the values of these two properties. Correspondence strategies may be also based on structural relationships between elements such as inheritance or containment; in this case, the correspondence strategy depends on the information about the neighbors of each element in the models.

Unlike correspondence strategies, merging strategies depend mainly on the type and the semantics associated to correspondence relationship that link source elements, and on the structure and semantics of the elements in the target metamodel. Most merging strategies such as *UnionMergingStrategy*, *PartialMergingStrategy* or *TotalMergingStrategy* have been implemented. The partial merging strategy is used for merging two elements to create two or more elements in the target model. In Section 5, we give an example of this strategy.

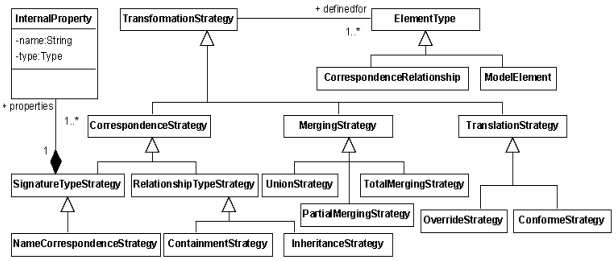

Figure 4. Transformation strategies metamodel

4. Specialization process of the generic framework

This section describes how the generic composition framework can be specialized to create a particular composition operator for a specific application domain. Figure 5 gives an overview of the specialization process which results in the enactable specific transformation rules.

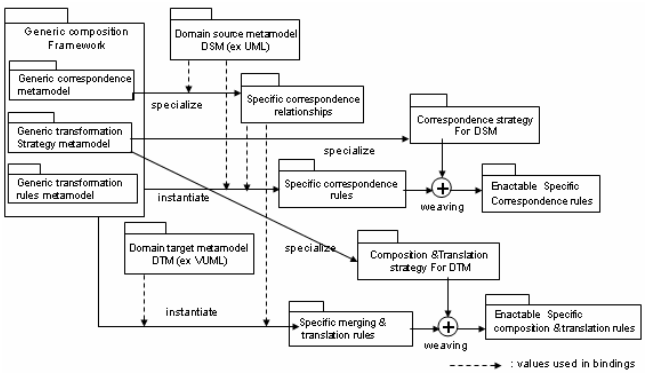

Figure 5. Specializing the generic framework for composing UML diagrams.

This process can be divided into three steps: The first step consists of extending the generic correspondence metamodel for a given domain specific modeling language (DSML). This task consists of defining specific link types with respect to the underlying domain. For example, if we consider the UML2 metamodel as source language, to express the similarity between classes, we define a new link type called *ClassSimilarityRelationship* by binding the template parameter *TypeElement* to value *Class*. Figure 6 shows an example of this specialization case.

The whole set of specific correspondence relationships constitutes the specific part of the *CorrespondenceMetamodel*.

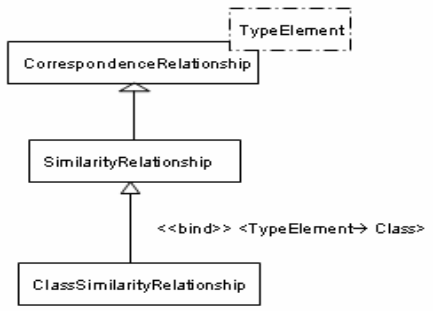

Figure 6. Example of instantiating a CorrespondenceRelationship template class.

The second step consists of specializing the transformation rules metamodel. The generic *CorrespondenceRule* metaclass is instantiated by binding template parameters to specific correspondence relationships and to the source models elements type. This is achieved when the parameters (*sourceMMElement*, and *Relationship*) are substituted respectively by elements of the source metamodel and elements of the specific correspondence metamodel. It is, afterwards, necessary to define the correspondence strategies for the source metamodel. These correspondence strategies specify the comparison logic between elements. A correspondence strategy can be designed as a model by defining an instance of the transformation strategy metamodel (Figure 4). The specific strategy model should be woven with the specific correspondence rules in order to produce a set of enactable specific correspondence rules, which can be performed to produce the correspondence model. The weaving operation is outside the scope of this paper.

The last step of the process considers a particular target metamodel, and produces a set of specific merging and translation rules. For the first case, the parameterized merging rule metaclass defined in the generic rules metamodel (Section 3.2) is instantiated by a specific correspondence relationship that bind *Relationship* parameter and a target metamodel element that bind *TargetMMElement* parameter. For translation rules the instantiation operation is achieved by the specification of both source and target metamodel elements. We show in Figure 7 an example of two different instantiation cases of the merging rule template. In Figure 7a, the generic merging rule is instantiated respectively with the *ClassSimilarityRelationship* and with the *MultiViewClass* element. In Figure 7b the same generic class is instantiated with other values. In this case, the parameter *source* gets the value *ClassConformityRelationship* and the parameter *target* gets the value *Class*.

As for the correspondence rule, both specific merging and translation rules can be woven with an appropriate strategy. This operation results in a set of enactable rules which can be used to perform the composition operation.

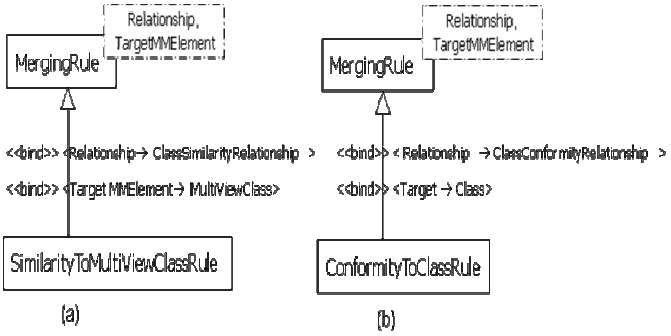

Figure 7. Two Examples of instantiating a merging rule template

The merging strategy defines how elements being related by a specific correspondence relationship are merged so as to create an element of the target metamodel. The translation strategy specifies how an element which has non corresponding elements in the opposite model is transformed in the composed model. By default, elements which verify this constraint may be deeply copied into target model. The translation strategy may be overridden in order to cover specific cases.

5. An example for composing UML class models

To illustrate our approach, we describe in this section how the generic composition framework can be specialized to compose models (class diagram) that conform to the UML2 metamodel [18]. The result of the composition is a VUML model which conforms to the VUML profile [16]. As mentioned previously, the VUML profile was developed to meet the needs of modeling complex information systems with UML according to various viewpoints. A point of view (called *viewpoint* in VUML) on the system represents an actor's requirement and rights. Such types of viewpoint may be considered as functional aspects. The main concept of VUML language is the multiview class, which is composed of a base class (shared by all viewpoints), and a set of view classes (extensions of the base class) where each view class is specific to a given viewpoint. Figure 11 illustrates some examples of multiview classes. Such a class is composed of a base class (stereotyped by <<base>>) that has the same

name as the corresponding UML classes, and a set of views (stereotyped by <<view>> or <<abstract view>>))) that are related to the base through a dependency relation (stereotyped by <<viewExtension>>)).

To define the specific correspondence relationships for UML2 class diagram concepts, we have specialized the generic correspondence metamodel according to the specializing process discussed above. For example, for class elements, we have identified two correspondence relationships: ClassConformity and ClassSimilarity. Conformity holds when two classes appear with the same name in different view point models and are semantically equivalent (represents two views of the same concept in the VUML terminology), and when they have the same properties (attributes, operations, associations, etc.). Similarity holds when two classes appear with the same name but are not conform. We have also defined a set of transformation rules for composing Structural UML2 diagrams into VUML.

To validate our approach, we consider an example for modeling a Shared Medical File Management System (SMFMS). In this application, we merge two UML2.0 models (class diagrams) that have been developed independently according to two different actors a patient and a doctor (Figure 9, Figure 10). We have implemented the transformation rules in ATL (Atlas transformation language) [12] which is considered as a standard component of Eclipse for model transformation, and is now integrated into the M2M project [8]. Figure 8 illustrates an example of transformation rules implemented in ATL language. Due to space restrictions, only a sub set of rules will be discussed. Note that this composition has been implemented with two transformation modules: the first module implements correspondence rules and generates a correspondence metamodel while the second module implements both merging and translation rules and has as a result the VUML target model.

```
rule ClassConformityRule extends
       ModelElementCorrespondenceRule{
   from e1 : UML2!Class, e2 : UML2!Class
              (e1.conform(e2))
   to r : MMC!ConformityRelationship
}
rule ConformityRelationship2Class{
  from r : MMC!ConformityRelationship
  to c : UML2!Class(
    name <-
thisModule.getRefElement(r).name,
    visibility <-
thisModule.getRefElement(r).
                getElement.visibility,
    isAbstract <-
thisModule.getRefElement(r).
                getElement.isAbstract,
  ownedAttribute<-
thisModule.getRefElement(r).
                ownedAttribute,
  ownedOperation <-
hisModule.getRefElement(r).
                ownedOperation
       )
}
rule Class2Class{
  from c1 : UML2!Class(
        c1.isNotMultiviewClass)

  to c2 : UML2!Class(
       name <- c1.name,
       isAbstract <- c1.isAbstract,
       visibility <- c1.visibility,
       ownedAttribute<-c1.
ownedAttribute,
       ownedOperation<-c1. ownedOperation
       )}
```

Figure 8. UML2VUML Transformation specification

Rule *ClassConformityRule* states that two class elements will be related by a *ConformityRelationship* if they are conform. The helper *conform* implements the conformity strategy between classes. The rule *ClassConformityRule* is declared as an extension of *ModelElementCorrespondenceRule* which is defined as an abstract rule. This means that it can only be invoked through the rules that extend it. The inheritance mechanism has the advantage of factorizing common code between several transformation rules.

Rule *ConformityRelationship2Class* specifies that for each defined *conformityRelationship* an UML2 Class element is created in the composed model. This rule implements the *Total MergingStrategy*. Finally, the rule *Class2Class* implements the conform translation Strategy which stipulates that a class that does not have any corresponding class in the opposite model may be translated as it is in the target model.

Figure 11shows the VUML model resulting from the merging of two UML source models for (SMFMS).

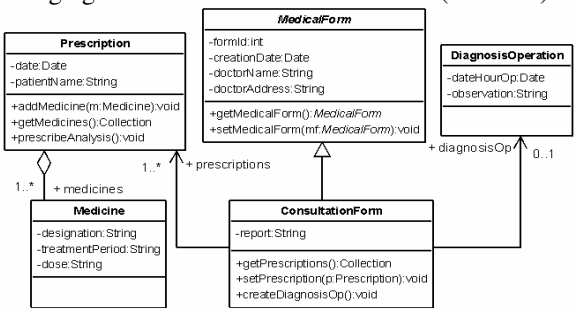

Figure 9. Doctor model

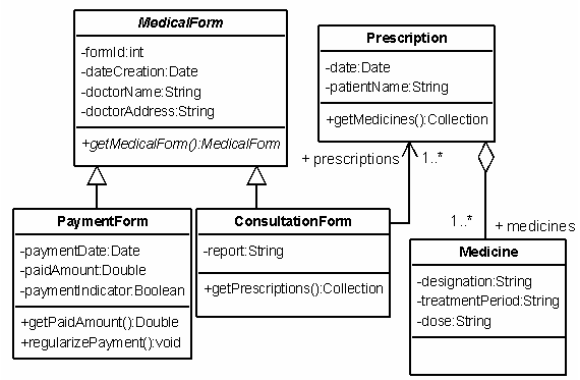

Figure 10. Patient model

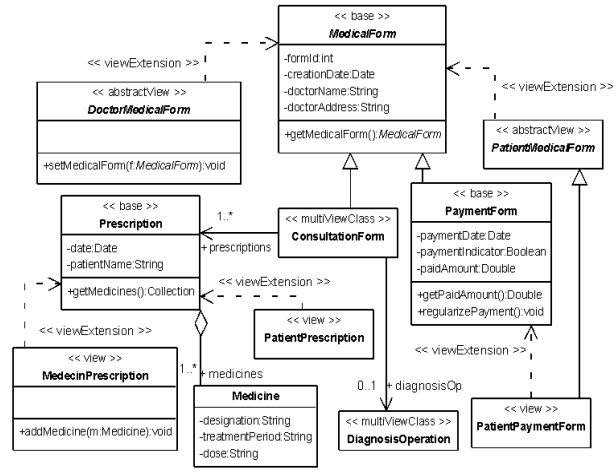

Figure 11. VUML model of SMFMS system

6. Related Work

Many researchers have developed model composition approaches in different application domains. We will focus in this section on works that are close to ours.

The Atlas Model Weaver (AMW) [7] is a model composition framework that uses model weaving and model transformation to define and execute composition operation. The tool support is available as as an Eclipse pluging. The composition operation is divided into two phases. First, a weaving model captures the links between the input model elements according to a weaving metamodel. Second, the weaving model is used to generate a transformation that takes two input models and produces the composed model. This technique has the advantage of being generic and flexible thanks to the extension mechanism of the weaving metamodel; however, the manual definition of the links between model elements is a tedious work.

The EML language (Epsilon Merging Language) proposed by [14] is a rule based language for merging models. EML belongs to the Epsilon platform, which is a model driven framework for developing integrated languages for model management tasks such as model comparison, model transformation, model validation, etc. Close to our work, this approach proposes to merge models of diverse metamodels through three categories of rules: MatchRule, MergeRule and TransformRule. Our correspondence rules produce a set of links between model elements, whereas in the EML approach this information is stored at the run time in a temporary memory called 'MatchTrace'. We consider that it is more convenient to separate the comparison and the merging steps in a reuse objective. So we generate a correspondence model at the end of the comparison step which is exploited during the merging step (MergingRule). This correspondence model may be used for other aims such as managing dependencies between views in order to ensure the system consistency.

The work presented in [3] discusses some similarities between model composition and model transformation. Thus, a number of approaches to implement composition as transformations are analyzed. The comparison criteria between each approach are based on the degree of generality, ease of use and ease of implementation. The authors have explored the possibility of composing a set of models based on crosscutting concerns (aspects), with a primary base model (which represent the core functionality of an application). By varying level of knowledge about aspect models, signatures and bindings, a number of composition-oriented transformations have been identified. In the continuity of this work [3], the model composition approach presented recently in [10] offers a generic framework which is independent from any modeling language. In this approach, the authors propose a generic metamodel, describing structural and behavioral features of a composition operator. This metamodel supports the composition directives concept introduced in [20]. These directives are specified by a domain-independent language (generic metamodel) and the concrete syntax is supported by the Kermeta language [15]. This approach can be depicted as imperative because it describes the operation of composition in an algorithmic way. Therefore, it is not easily compatible with our approach which is based on declarative rules that rather specify what should be transformed than how it should be done.

7. Conclusion

In this paper we have presented a generic approach for model composition, and a specialization process to apply the generic composition operator to specific modeling domains (Domain Specific Languages). This approach combines the use of metamodeling and model transformation techniques.

The core composition operator is generic: it allows designing the composition requirements at a high level of abstraction. The core metamodel is composed of three separate metamodels: (1) a relationship metamodel that contains abstract constructs for defining the correspondences between model elements, (2) a rules metamodel that contains generic transformation rules in order to implement the composition operator, and (3) a strategy metamodel that defines common transformation strategies. To validate our approach, we have implemented the transformation rules in ATL and we are working on the development of a model composition environment as an Eclipse plugging for developing transformations-oriented framework for model composition.

In the works to come, we intend to automate the weaving activity of the specialization process. Thus, we will apply the technique of model transformation to generate the enactibale transformation rules in a specific transformation language. This mechanism is known as HOT (Higher Order Transformation). So, the manual programming step of the transformation will be dramatically reduced. Another interesting issue is to compose behavioral diagrams of UML (mainly state charts and sequence diagrams).

Acknowledgments. This work has been partially supported by the COMPUS Project MA/06/152, PAI VOLUBILIS 2006.

8. References

[1] Anwar, A., Nassar, M., Ebersold, S., Coulette, B., Kriouile, A., "A QVT- Based Approach For Model Composition: Application to the VUML Profile". In Proc of ICEIS 2008, pp 360-367. Barcelona, Spain, Juin 2008.

[2] Baniassad, E., Clarke, S.: "Theme: An approach for aspect-oriented analysis and design", In Proc.of ICSE'04. (2004), pp 158 -167.

[3] Baudry, B., Fleurey, F., France, R., Reddy, R., Exploring Relationship between Model Composition and Model transformation. In Proceedings of Aspect Oriented Modeling (AOM) Workshop associated to MoDELS'05. Montego Bay, Jamaica, October 2005

[4]Bézivin J., Bouzitouna S., Del Fabro M.D., Gervais M., Jouault F., Kolovos D., Kurtev I., Paige R., "A Canonical Scheme for Model Composition", Proc. of ECMDA-FA, LNCS 4066, Springer-Verlag, 2006, p. 346-360

[5] Bézivin J., Jouault F., « Using ATL for Checking Models », In proc. of the International Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia. 2005.

[6] Clarke, S.: "Extending Standard UML with Model Composition Semantics". Science of Computer Programming, 44 (2002) 71.100

[7]Didonet Del Fabro M., Bézivin J., Jouault F., Breton E., Gueltas G., "AMW: a generic model weaver", Actes IDM'05. Paris, France, juin 2005, p. 105-114

[8]Eclipse/M2M Project Web Page. http://www.eclipse.org/m2m/, 2007

[9] Finkelstein, A., Kramer, J., Goedicke, M., "Viewpoint Oriented Software Development". In Software Eng. and Applications Conference. Toulouse, France, pages 337-351, 1990

[10] Fleurey F., Baudrey B., France R., Ghosh S., "A Generic Approach For Automatic Model Composition", Proceedings of the Aspect Oriented Modeling, Workshop at Models 2007, Nashville USA 2007

[11] France, R.B., Ray, I., Georg, G., Ghosh, S.: "An aspect-oriented approach to design modeling". IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design 151 (2004) 173.185

[12] Jouault F., Kurtev I., "Transforming Models with ATL", Proceedings of the Model Transformations in Practice, Workshop at Models 2005, Montego Bay, Jamaica 2005

[13] Kiczales, G., Lampng, J., Mendhekar, A., Maeda, C., Videira, L.C., "Aspect-Oriented Programming". In ECOOP'97. Springer-Verlag LNCS 1241. Finland, 1997

[14] Kolovos, DS., Paige, RF., Polack, FAC., 2006. Merging Models with the Epsilon Merging Language (EML). In Proc. ACM/IEEE 9th International Conference on Models/UML), Genova, Italy, October

[15] Muller P.-A, Fleurey, F. and Jézéquel, J.M. "Weaving executability into object-oriented meta-languages". In Proceedings of MoDELS'05, p. 264 - 278. Montego Bay, Jamaica, October 2005

[16] Nassar, M., Coulette, B., Crégut, X., Marcaillou, S and Kriouile., A. "Towards a View based Unified Modeling Language". In ICEIS'03, Angers, France, 2003

[17] OMG 2002, OMG/MOF Meta Object Facility (MOF) 1.4. Final Adopted Specification Document. Formal/02-04-03, 2002

[18] OMG 2003, UML 2.0 Superstructure Final Adopted specification, Document-ptc/03-08-02

[19] Ossher, H., Kaplan, M., Katz, A., Harrison, W., Kruskal, V.: "Specifying subject-oriented composition". Theory and Practice of Object Systems, Wiley and Sons 2 (1996)

[20] Reddy Y. R., Ghosh S., France R. B., Straw G., Bieman J. M., McEachen N., Song E., Georg G., «Directives for Composing Aspect-Oriented Design Class Models ». Transactions of Aspect-Oriented Software Development, Vol.1, No. 1, LNCS 3880, p75-105, Springer.