# A Semi-automatic Approach for Bridging DSLs with UML[⋆]

Manuel Wimmer[1], Andrea Schauerhuber[1⋆⋆], Michael Strommer[1],
Wieland Schwinger[2], and Gerti Kappel[1]

[1] Institute of Software Technology and Interactive Systems
Vienna University of Technology, Austria
{wimmer|schauerhuber|strommer|gerti}@big.tuwien.ac.at
[2] Department of Telecooperation
Johannes Kepler University Linz, Austria
wieland.schwinger@jku.ac.at

**Abstract.** The definition of modeling languages is a key-prerequisite for model-driven engineering (MDE). In this respect, domain-specific languages (DSL) defined in terms of metamodels and UML profiles are often considered as two alternatives. For interoperability reasons, however, the need arises to bridge modeling languages originally defined as DSLs to UML profiles by defining (1) a specific UML profile to represent the domain-specific modeling concepts in UML and (2) model transformations for transforming DSL models to UML models and vice versa. A manual definition of a UML profile typically is a tedious and error-prone task, but contains a high potential for automation. The contribution of this paper is to integrate the so far competing worlds of DSLs and UML. We report on our semi-automatic approach based on the manual mapping of domain-specific metamodels and UML using a dedicated bridging language as well as the automatic generation of UML profiles and model transformations. We present our ideas within a case study for bridging ComputerAssociate's DSL of the *AllFusion Gen* CASE tool with IBM's *Rational Software Modeler* for UML.

**Key words:** UML Profiles, Metamodels, Interoperability, Model Exchange

## 1 Introduction

**Motivation**. In software engineering in general and in model-driven engineering (MDE) in particular, there is a movement from general-purpose languages (GPL) to domain-specific languages (DSL). For defining DSLs in the field of MDE *metamodels* and *UML profiles* are the proposed options. While metamodels, mostly based on the Meta Object Facility (MOF) [7], allow the definition of DSLs from scratch, UML profiles are used to extend UML with domain-specific concepts.

In this work we focus on the interoperability between these two approaches, because the need to bridge modeling languages originally defined as DSLs to UML often arises in practice. For example in the ModelCVS project [3] our industry partner is using the ComputerAssociate's CASE tool *AllFusion Gen*[3] which supports a DSL for designing data-intensive applications and provides sophisticated code generation facilities. Due to modernization of the IT department and the search for an exit strategy (if tool support is no longer guaranteed), the need arises to extract models from the legacy tool and import them into UML modeling tools while at the same time the code generation of AllFusion Gen should in the future be usable for UML models as well. Besides these typical tool integration and interoperability issues, when buidling UML profiles from scratch, the modeling concepts are often defined in metamodels before creating the actual UML profiles, as it was done in [6].

**Related Work**. The integration of DSLs and UML has been already addressed in previous research. In Abouzahra et al. [1], the integration process starts with an already existing, probably manually defined, UML profile and one has to define the mappings between the DSL metamodel and the UML profile elements. Subsequently, the model transformations between the DSL and UML models are automatically derived from these mappings. Our work is different in that we do not assume that an UML profile is available yet. Instead, we assume that the whole bridge, i.e., the transformations and the UML profile, has to be developed.

**Ingredients for a DSL/UML bridge**. First of all, it has to be mentioned that the current practice in bridge development is an ad-hoc manner focused on implementation tasks as illustrated in Figure 1(a). First, one has

[⋆] This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806.0

[⋆⋆] This research has been partly funded by the Austrian Federal Ministry for Education, Science, and Culture, and the European Social Fund (ESF) under grant 31.963/46-VII/9/2002.

[3] http://ca.com/us/products/product.aspx?ID=256

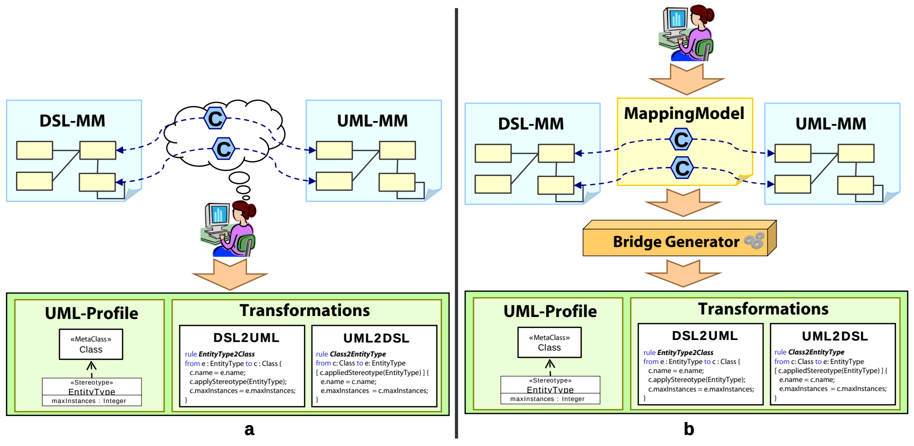

**Fig. 1.** DSL/UML integration. a. Ad-hoc approach. b. Systematic approach.

to reason about correspondences between DSL and UML metamodel elements. Then, a profile must be defined for the DSL metamodel in which stereotypes for each metaclass of the DSL are added as well as tagged values for DSL features which are not directly representable in UML. Afterwards, one has to define the transformations from DSL to UML and back again, whereas the transformations are based on the UML profile definition, e.g., for assigning stereotypes and tagged values to UML model elements.

**Drawbacks of ad-hoc implementation**. The described approach for bridging DSLs to UML has several drawbacks. In particular, these drawbacks are responsible for making the bridging task tedious and error-prone, resulting in low productivity and maintainability.

1. **Transformations and profiles are highly coupled**. A change in the profile definition triggers changes in the model transformation code.
2. **Bridging covers many repetitive tasks**. For each DSL modeling concept nearly the same tasks have to be fulfilled in the integration process.
3. **No guidelines**. Users are not familiar with the integration task since it is typically a one-time job.
4. **No explicit correspondences between DSL and UML elements**. No high-level specification artifacts are defined, instead one has to start with an implementation of the UML profile and the transformation rules directly.

## 2 A semi-automatic Approach for Bridging DSLs with UML

We propose a semi-automatic approach and introduce two additional facilities for the integration process, as can be seen in Figure 1(b). First, we propose the use of an explicit and formal *mapping model* which is built manually. Second, we provide a *Bridge Genenerator* component which is capable of generating the required profiles and transformations from the mapping models. The integration process then is as follows: First, the user defines the correspondences between the DSL metamodel and the UML metamodel in terms of a mapping model in an interactive mapping environment. The mapping model is expressed with a dedicated metamodel bridging language which enables the automatic processing. After finalizing the mapping model, the Bridge Generator automatically generates the UML profile and in the case that an uni-directional model transformation language is used, also the transformations from the DSL to UML and back again.

**Benefits of systematic integration approach**. Our approach tackles the four mentioned drawbacks of the ad-hoc implementation approach explained in Section 1.

1. **Mapping model is single source of information**. In our approach, it is sufficient that one modifies the mapping model, only. The changes made in the mapping model are automatically propagated to the UML profile and to the transformation definitions. Hence, the high coupling between the UML profiles and transformations is transparent for the user.
2. **Repetitive tasks are automated by model transformations**. The only manual task is the definition of the mapping model. Subsequently, the implementation artifacts, i.e., the UML profile and the transformations are automatically produced by the Bridge Generator component. Furthermore, the Bridge Generator component

ensures that the profiles and transformations are always developed in the same manner for the same kind of integration problem, leading directly to the next benefit.

3. **Guidelines support for systematic integration**. The ad-hoc integration does not ensure a systematic integration. This drawback is resolved due to the following two reasons. First, the mapping model is built with a specific mapping language, and second, there is an explicit integration method for overlapping (semantic matches) and distinct elements (semantic mismatches) of DSLs and UML implemented in the Bridge Generator component.
4. **Explicit representation of correspondences**. Our approach supports an explicit mapping model which represents the whole integration specification in a single conceptual model. Furthermore, currently used documentations such as mapping tables can be automatically generated out of the mapping models.

## 2.1 Bridging Language

In this subsection we describe the abstract syntax of the language used to describe the mappings between DSL and UML metamodels. As can be seen in Figure 2, we reuse the core weaving language of the ATLAS Model Weaver [2], displayed in the *mwcore* package in Figure 2, which defines abstract concepts for namespaces such as *ModelRef* and *ElementRef* and linking semantics, such as a *WModel*, *WLink*, and *WLinkEnd*. Note that each concept is defined abstract. Our dedicated bridging language, defined in the package *BridgingLanguage*, consequently refines them with concrete concepts. In particular, we defined a metamodel integration language which allows homogenous mappings, meaning mappings between elements which are instances of the same metaclass. In the following, we briefly describe each concrete subclass of the class *WLink*, which represents the mapping operators of the bridging language.

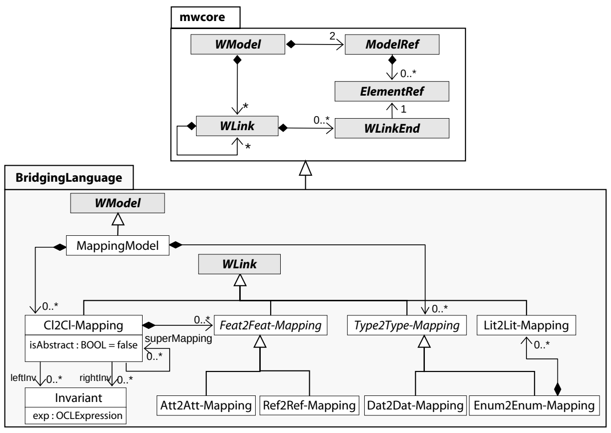

**Fig. 2.** DSL/UML Bridging Language.

- **Cl2Cl-Mapping**. This kind of mapping operator allows the user to map classes of the DSL metamodel with classes of the UML metamodel. One speciality of *Cl2Cl* mappings is, that they can have *superMappings*, i.e., the feature mappings from super mappings are inherited to sub mappings. This allows for reuse of already provided mapping information. Furthermore, the user can mark *Cl2Cl* mappings as abstract, meaning that they have to be refined with concrete sub mappings. Finally, *Cl2Cl* mappings can have invariants expressed in Object Constraint Language (OCL) [8] for defining conditions when a mapping should be executed.
- **Att2Att-Mapping**. This kind of mapping allows to map corresponding attributes between DSLs and UML metamodels. In order to derive executable transformations for each *Att2Att* mapping the *Cl2Cl* mapping that contains the *Att2Att* mapping has to be specified.
- **Ref2Ref-Mapping**. This kind of mapping is similar to *Att2Att* mappings with the difference that corresponding references are mapped instead of attributes.
- **Enum2Enum-Mapping**. In order to transform attribute values correctly between DSL and UML models, corresponding enumerations must be mapped. Therefore, we introduce *Enum2Enum* mappings to allow the definition of equivalent enumerations.

- **Lit2Lit-Mapping**. This kind of mapping is used to define equivalences between literals of enumerations and consequently, each *Lit2Lit* mapping is owned by an *Enum2Enum* mapping. Note that only Enumerations which have totally corresponding literals can be mapped, i.e., each literal must be mapped to exactly one literal.
- **Dat2Dat-Mapping**. This kind of mapping is used to define equivalences between data types of DSLs and UML. It is required because (1) data types which are semantically identical are sometimes named differently and (2) DSLs often support specific data types which are not available in plain UML but can additionally be defined in profiles.

## 2.2 Automatic Generation Process

The overall idea of using a mapping model is that sematic equivalent metamodel elements of DSLs and UML are mapped. Consequently, some metamodel elements remain unmapped. The distinction between mapped and unmapped elements is the main driver for the bridge generation process which is discussed in this subsection.

**UML Profile Generation** It is required that each DSL metaclass should be mapped in the final mapping model. Unmapped DSL metaclasses are only allowed in prototypical mappings, since they are not included in the generation process and consequently no complete bridge can be generated. For each *Cl2Cl* mapping a stereotype is generated which extends the UML metaclass being referenced by the right end of the mapping. If the *Cl2Cl* mapping has super mappings, the generated stereotype is a sub stereotype of the stereotypes generated from super mappings.

The next step concerns the generation of the tagged values from *Feat2Feat* mappings, i.e., *Att2Att* and *Ref2Ref* mappings. Assume the following *Cl2Cl* mapping: *DSL::Class* $\leftrightsquigarrow$ *UML::Class*. If we want to derive the tagged values for the stereotype which is generated for the DSL class, we must analyze the features of the DSL class and whether they are mapped to corresponding UML features or not. We can distinguish three distinct cases which are described in the following.

- **DSL::Class.features $\cap$ UML::Class.features**. Features which are available in the DSL and in UML metamodel should be linked in the mapping model via feature mappings, i.e., the values of the DSL features are directly representable with UML features.
- **DSL::Class.features $\setminus$ UML::Class.features**. Features which are only available in the DSL must not be mapped via feature mappings to UML features. Furthermore, this means that the values of these features are not representable in plain UML. Therefore, for each DSL feature which is not mapped to an UML feature a tagged value is generated. In particular, if the feature is an attribute then a tagged value having as type a data type is generated. If the feature is a reference, a tagged value is generated which can link to the stereotype actually representing the referenced type in the DSL metamodel.
- **UML::Class.features $\setminus$ DSL::Class.features**. Features which are only available in UML must be specially treated. Though not relevant for profile generation, this case is relevant for generating model transformations. The problem is that the values of these kind of features cannot be set with values of the DSL models. In order to produce valid UML models, we must distinguish between optional and mandatory features. The first case is that the feature is optional leading to no problems because a *null* value can be assigned. The second case is more problematic, namely if the UML attribute is mandatory. We can check if a default value is available for this attribute. If no default value is defined for the feature, the user must specify a value in the mapping model which is automatically assigned for this particular feature.

The last step concerns type mappings, i.e., (un)mapped *Enumerations*, *Literals*, and *DataTypes*. If an enumeration of the DSL metamodel fully corresponds to an enumeration in the UML metamodel then they are mapped with an *Enum2Enum* mapping. This *Enum2Enum* mapping further requires that each literal is mapped to exactly one literal with a *Lit2Lit* mapping. No further definitions are required in the UML profile. Otherwise, an additional enumeration with corresponding literals must be generated in the profile. The treatment of *DataTypes* is similar to that of enumerations.

**Model Transformation Generation** After derivation of the UML profile, the model transformations can be generated. Due to brevity reasons we will not discuss the model transformations generation in detail, but give an overview of the main characteristics. We generate model transformation rules for mapped classes which transform instances of the source class to instances of the target class. If the target model is a UML model we must also apply the profile and assign the corresponding stereotypes to the generated objects. This is required in order to set tagged values for unmapped DSL features. When transforming attribute values we have to take care that enumerations are treated correctly, i.e., if the enumeration is not mapped then the generated literal has to be assigned, else the mapped literal has to be assigned.

## 3 Case Study

In this section we present our approach within a case study for bridging parts of the ComputerAssociate's DSL of the *AllFusion Gen* CASE tool and IBM's *Rational Software Modeler* which implements the UML 2.0 standard. First we briefly describe the involved metamodels, then an overview of the mappings, and finally the details for one particular *Cl2Cl* mapping. Further details of the case study can be found on our project site[4] including the details for all *Cl2Cl* mappings.

### 3.1 AllFusion Gen's Data Model

The metamodel for the data model of AllFusion Gen is illustrated in the package *DataModel* of Figure 3 and contains concepts that allow modeling the data used by the applications. Since AllFusion Gen's data model is based on the ER model, it supports ER modelling concepts like *EntityTypes*, *Attributes*, and *Relationships*. In addition to the ER modeling concepts, *SubjectAreas* can be used that contain *EntityTypes* as well as further *SubjectAreas*. *EntityTypes* can have zero-or-one super type. Furthermore two concrete sub types of the abstract *EntityType* concept can be distinguished, namely *AnalysisEntityType* and *DesignEntityType*. AllFusion Gen is typically used for modeling data intensive applications which make excessive use of database technologies. Therefore, the data model allows the definition of platform specific information typically usable for generating optimized database code, e.g., *EntityTypes* have special occurrence configurations.

### 3.2 UML Class Diagram

It is obvious that the corresponding UML model type for AllFusion Gen's data model is the class diagram. In this work we only present the part of the UML metamodel which is relevant for integration purposes. The metamodel excerpt is shown in Figure 3 in the package *ClassModel*. In UML *Packages* can contain further *Packages* as well as *Classes*. *Classes* can be defined as either abstract or concrete and can have properties as well as arbitrary superclasses. *Properties* represent attributes if the opposite property is not set, or role ends if the opposite property is set.

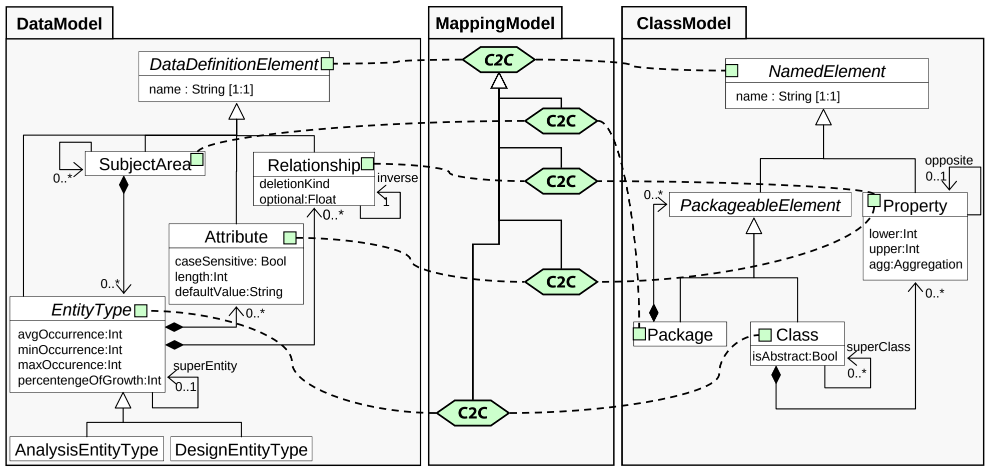

**Fig. 3.** Cl2Cl Mappings at a Glance.

### 3.3 Overview on the Mapping Model

In this subsection we present an overview of the mappings (cf. package *MappingModel* in Figure 3), covering only the *Cl2Cl* mappings. Both metamodels make use of inheritance which results in abstract superclasses only containing the *name* attribute. In order to allow for reuse of mapping information in sub mappings and to minimize

[4] www.modelcvs.org/prototypes/

the number of feature mappings, the abstract classes are mapped with an abstract *Cl2Cl* mapping which is used as super mapping for all other *Cl2Cl* mappings. *SubjectArea*, *EntityType*, *Attribute*, and *Relationship* are mapped to *Package*, *Class*, *Property*, and *Property*, respectively. While the first two mappings are obvious, the last two mappings have both the same target class. This is due the fact that UML does not distinguish explicitly between attributes and relationships in the metamodel. *AnalysisEntityType* and *DesignEntityType* are not mapped to UML metamodel elements, because both concepts would be mapped to *Class* in UML. Hence we decided to reuse the mapping of the abstract *EntityType* class in order to infer that both sub concepts should be also mapped to *Class*.

In Figure 4 the resulting stereotypes for the *Cl2Cl* mappings are shown. Besides concrete stereotypes, two abstract stereotypes have been produced: ≪DataDefinitionElement≫ is the super stereotype for all others and ≪EntityType≫ has two concrete sub stereotypes corresponding to the subclasses of the metaclass *EntityType*.

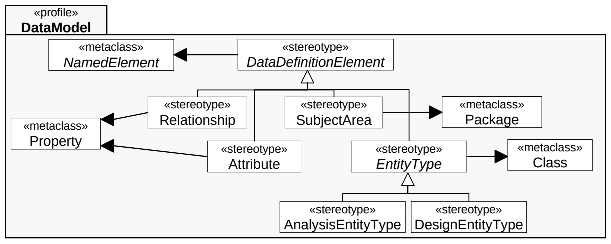

**Fig. 4.** Stereotypes for Cl2Cl mappings.

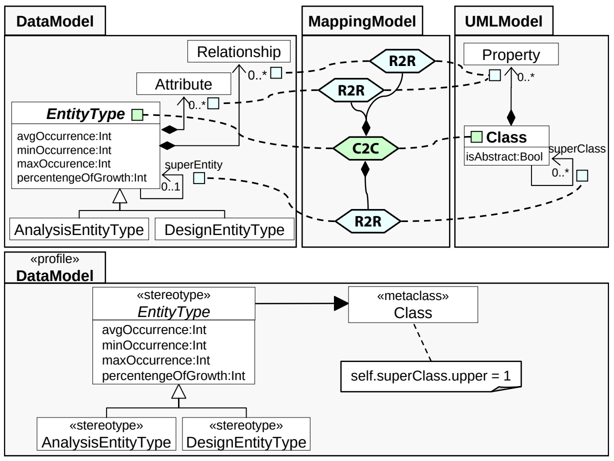

**Fig. 5.** The EntityType_2_Class Mapping.

## 3.4 Details on the EntityType_2_Class Mapping

The mapping details for the *Cl2Cl* mapping between EntityType and Class are illustrated in the upper half of Figure 5. Both concepts support inheritance, but with the difference that *EntityTypes* only support single inheritance and *Classes* allow multiple inheritance. Despite this difference, the two references are mapped with a *Ref2Ref* mapping because they support a similar feature. Furthermore, *EntityTypes* can have attributes and references, in contrast UML classes can have properties which can be divided into two distinct subsets. As discriminator the opposite reference is used whereas the property is either an attribute if the opposite value is null or a reference if the opposite value is not null. Thus, we can map the attribute and relationship references to the property reference with two *Ref2Ref* mappings with specific OCL conditions.

**Profile generation** The profile details resulting from the *EntityType_2_Class* mapping is shown in the lower half of Figure 5. An abstract stereotype ≪EntityType≫ is generated for the abstract metaclass *EntityType*. Furthermore, the metaclass EntityType has 4 platform specific attributes which are not available in UML. These attributes are represented in the ≪EntityType≫ stereotype as tagged values. Finally, the multiple inheritance mechanism of UML classes is restricted by assigning a special OCL constraint to the metaclass *Class*.

**Transformation generation** A simplified excerpt of the resulting ATL code is shown in Listing 1.1. An abstract transformation rule (cf. first rule in Listing 1) with three reference mappings is generated. However, the current ATL version does not allow to define *do* blocks for super rules, thus, *feature to tagged value* mappings must be defined in concrete sub rules. In fact, two concrete sub rules are generated, one for transforming *AnalysisEntityTypes* (cf. second rule in Listing 1.1) and one for *DesignEntityTypes*, which implement the *feature to tagged value* mappings - which are also defined for unmapped features of the superclasses, e.g., the *avgOccurrence* attribute.

**Listing 1.1.** Resulting ATL code.

```
module DSL2UML; create OUT:UML from IN:DSL;

abstract rule EntityType_2_Class {
   from s : DSL!EntityType
   to t : UML!Class (
       property <- s.attribute,
       property <- s.relationship,
       superClass <- s.superEntity
   )
}

rule AnalysisEntityType_2_Class extends
    EntityType_2_Class{
   from s : DSL!AnalysisEntityType
   to t : UML!Class
   do{
    t.assignStereotype(AnalysisEntityType);
    t.setTaggedValue("avgOccurrence",
        s.avgOccurrence);
    ...
    }
}
```

## 4 Current Limitations

In this section we discuss current limitations of our approach which should be resolved in the future. In particular, most of the limitations are related to typical database, XML schema, and ontology integration issues which have been extensively reported in the literature [4], [5]. These reported issues are more general, nevertheless, they have to be tackled when providing a special integration mechanism, in our case for integrating DSLs with UML.

**Challenge 1: Multiple Correspondences**. In this work we have focused on 1:1 correspondences, only. However, in practice often 1:n, n:1, and n:m correspondences are needed to describe the integration. From a data transformation point of view the question arises how to combine multiple input values to produce multiple output values. From the profile generation point of view it seems challenging how to produce stereotypes and tagged values for n:m correspondences.

**Challenge 2: Lost information during round-trip**. The goal of supporting full round-trip transformations is currently not always achieved due to heterogeneity issues which have been first identified database integration [4]. These issues lead to value transformations that go beyond simple copying of values. Hence, some transformations can be inverted and some transformations can't. We are confronted with this challenge when there is a mismatch in *abstraction*, *aggregation*, or *precision* between the source and the target model elements.

**Challenge 3: 0:1 Mappings**. This kind of mapping represents the problem when UML features have no corresponding DSL features. In round-trip, the information of UML features which cannot be mapped to DSL features are lost. One possible solution would be to use an annotation mechanism of the DSL if provided. If no annotation mechanism is provided, the UML specific information cannot be captured within the DSL model, instead this information has to be carried in a separate file and merged with the UML model which is produced from the DSL model.

**Challenge 4: Validation Support**. One of the main goals of the generation process is that meaningful UML profiles and working model transformation code is produced, also for prototypical mappings. Thus, the mapping models must be validated. Validation support should cover at least reasoning on a set of mappings, compatible data and object types, and mandatory features. The result of the validation process should be a list of mapping errors as well as warnings.

## 5 Conclusion and Future Work

In this paper we have introduced a semi-automatic approach for bridging DSLs with UML. We developed a dedicated metamodel mapping language between the DSL and UML metamodels and a component for automatically generating UML profiles and model transformations from mapping models. This approach allows for faster development and a higher maintainability of bridges between DSLs and UML. Furthermore, we developed tool support for our bridging approach which is built on existing components available on the Eclipse platform. Further information about tool support can be found on our project site.
For future work we strive to reduce the aforementioned current limitations. In addition, we are looking for automation techniques for building the mapping model by applying matching techniques similar to proposed techniques in the research field of ontology and schema matching [9].

## 6 Acknowledgments

We thank Arikan Productivity Group (`www.arikan.at`) for their cooperation in the ModelCVS project in general and specially for their work on the AllFusion Gen metamodel.

## References

1. A. Abouzahra, J. Bzivin, M. D. D. Fabro, and F. Jouault. A Practical Approach to Bridging Domain Specific Languages with UML profiles. In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05*, San Diego, California, USA, 2005.
2. M. D. D. Fabro, J. Bzivin, F. Jouault, E. Breton, and G. Gueltas. AMW: a generic model weaver. In *Proceedings of the 1re Journe sur l'Ingnierie Dirige par les Modles (IDM05)*, 2005.
3. G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer. On Models and Ontologies - A Semantic Infrastructure Supporting Model Integration. In H. C. Mayr and R. Breu, editors, *Modellierung*, volume 82 of *LNI*, pages 11–27. GI, 2006.
4. V. Kashyap and A. P. Sheth. Semantic and Schematic Similarities Between Database Objects: A Context-Based Approach. *VLDB J.*, 5(4):276–304, 1996.
5. F. Legler and F. Naumann. A Classification of Schema Mappings and Analysis of Mapping Tools. In A. Kemper, H. Schöning, T. Rose, M. Jarke, T. Seidl, C. Quix, and C. Brochhaus, editors, *BTW*, volume 103 of *LNI*, pages 449–464. GI, 2007.
6. N. Moreno, P. Fraternali, and A. Vallecillo. WebML modeling in UML. *IET Software Journal*, 2007.
7. OMG. *Meta Object Facility Core Specification*, version 2.0 formal/2006-01-01 edition, 2006.
8. OMG. *Object Constraint Language (OCL)*, version 2.0 formal/2006-05-01 edition, 2006.
9. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.